MSL C Reference
Version 10

Revised: 23 January 2007

freescale semiconductor

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

This reference contains a description of the ANSI library and extended libraries bundled with Main Standard Library for C.

Organization of Files

The C header files are organized alphabetically. Items within a header file are also listed in alphabetical order. Whenever possible, sample code has been included to demonstrate the use of each function.

"Introduction to Multi-Threading" on page 7 covers multi-threading and thread-safeness of the Main Standard C Library functions.

"Overview of alloca.h" on page 25 covers the non-ANSI `alloca()` function for dynamic allocation from the stack.

"Overview of assert.h" on page 27 covers the ANSI C exception handling macro `assert()`.

"Overview of conio.h" on page 29 covers the Windows console input and output routines.

"Overview of console.h" on page 41 covers Macintosh console routines.

"Overview of crtl.h" on page 49,covers Win32 console routines.

"Overview of ctype.h" on page 53 covers the ANSI character facilities.

"Overview of direct.h" on page 65 covers Win32x86 directory facilities.

"Overview of dirent.h" on page 69 covers various POSIX directory functions.

"Overview of div_t.h" on page 73 covers two arrays for math routines.

"Overview of errno.h" on page 75 covers ANSI global error variables.

"Overview of extras.h" on page 79 covers additional non standard functions included with the MSL library.

"Overview of fcntl.h" on page 117 covers non-ANSI control of files.

"Overview of fenv.h" on page 125, covers floating-point environment facilities.

"Overview of float.h" on page 139 covers ANSI floating point type limits.

"Overview of FSp_fopen.h" on page 141 contains Macintosh file opening routines.

"Overview of inttypes.h" on page 145 contains greatest-width integer routines and macros.

"Overview of io.h" on page 157, contains common Windows stream input and output routines.

"Overview of iso646.h" on page 161 contains operator symbol alternative words.

"Overview of limits.h" on page 163 covers ANSI integral type limits.

"Overview of locale.h" on page 165 covers ANSI character sets, numeric and monetary formats.

"Overview of malloc.h" on page 169, covers the alloca function for Windows.

"Overview of math.h" on page 171 covers ANSI floating point math facilities.

"Overview of Process.h" on page 239, covers Windows thread process routines.

"Overview of setjmp.h" on page 243 covers ANSI means used for saving and restoring a processor state.

"Overview of signal.h" on page 249 covers ANSI software interrupt specifications.

"Overview of SIOUX" on page 255 covers CodeWarrior™ SIOUX console emulation for Macintosh.

"Overview of stat.h" on page 269 covers non-ANSI file statistics and facilities.

"Overview of stdarg.h" on page 279 covers ANSI custom variable argument facilities.

"Overview of stddef.h" on page 287 covers the ANSI Standard Definitions.

"Overview of stdint.h" on page 289 covers the latest integer types macros.

"Overview of stdio.h" on page 297 covers ANSI standard input and output routines.

"Overview of stdlib.h" on page 403 covers common ANSI library facilities.

"Overview of string.h" on page 453 covers ANSI null terminated character array facilities.

"Overview of tgmath.h" on page 483 lists type-generic math macros.

"Overview of time.h" on page 485 covers ANSI clock, date and time conversion and formatting facilities.

"Overview of timeb.h" on page 507 allows for programmer allocation of a buffer to store time information.

"Overview of unistd.h" on page 511 covers many of the common non-ANSI facilities.

"Overview of unix.h" on page 539 covers some CodeWarrior non-ANSI facilities.

"Overview of utime.h" on page 543 covers non-ANSI file access time facilities.

"Overview of utsname.h" on page 549 covers the non-ANSI equipment naming facilities.

"Overview of wchar.h" on page 553 covers the wide character set for single and array facilities.

"Overview of wctype.h" on page 615 covers the wide character set type comparison facilities.

"Overview of WinSIOUX" on page 627 covers CodeWarrior SIOUX console emulation for Windows.

"Overview of the MSL Switches, Flags and Defines" on page 633 covers the various switches and flags, and defines used to customize the MSL C library.

ANSI C Standard

The ANSI C Standard Library included with Freescale CodeWarrior follows the specifications in the ANSI: Programming Language C / X3.159.1989 document together with extensions according to ISO/IEC 9899:1999 (known for some time as "C99"). The functions, variables and macros available in this library can be used transparently by both C and C++ programs.

The Main Standard Library implements internal macros, _MSL_C99, that separate those parts of the C library that were added to the first version of the ANSI Standard for the C programming language (ANSI/ISO 9899-1990) by the second version (ISO/IEC 9899-1999(E)). If _MSL_C99 is defined in a prefix file to have the value 0 before building the MSL C library, only those parts of the library that were defined in ANSI/ISO 9899-1990 are compiled, yielding a smaller library. If _MSL_C99 is defined to have a non-zero value before building the library the full MSL C library is compiled in accord with ISO/IEC 9899-1999(E).

The ANSI C Library and Apple Macintosh

Some functions in the ANSI C Library are not fully operational on the Macintosh environment because they are meant to be used in a character-based user interface instead of the Macintosh computer's graphical user interface. While these functions are available, they may not work as you expect them to. Such inconsistencies between the ANSI C Standard and the MSL C implementation are noted in a function's description.

Except where noted, ANSI C Library functions use C character strings, not Pascal character strings.

MSL Extras Library

The MSL Extras Library contains functions macros and types that are not included in the ANSI/ISO C Standard as well as POSIX functions. These are included for Windows and UNIX compatibility. See the description of "_MSL_NEEDS_EXTRAS" on page 635, for access of these functions when including a C standard header.

The functions, procedures and types in the headers listed in Table 1.1 are included in the MSL Extras Library.

Table 1.1 MSL Extras Library Headers

dirent.h	extras.h	fcntl.h	stat.h
unistd.h	unix.h	utime.h	utsname.h
Mac OS Only	Console.h		
Windows Only	conio.h	direct.h	io.h

Windows Only

On Windows, the MSL extras functions are enabled using the same name with a leading underscore.

POSIX Functionality

The Main Standard Libraries include some, but not all, `POSIX` functions and types.

Console I/O and the Macintosh

The ANSI Standard Library assumes interactive console I/O (the `stdin`, `stderr`, and `stdout` streams) is always open. Many of the functions in this library were originally designed to be used on a character-oriented user interface, not the graphical user interface of a Macintosh computer. These header files contain functions that help you run character-oriented programs on a Macintosh:

- `console.h` declares `ccommand()`, which displays a dialog that lets you enter command-line arguments
- `SIOUX.h` is part of the SIOUX package, which creates a window that's much like a dumb terminal or TTY. Your program uses that window whenever your program refers to `stdin`, `stdout`, `stderr`, `cin`, `cout`, or `cerr`.

Console I/O and Windows

The ANSI Standard Library assumes interactive console I/O (the stdin, stderr, and stdout streams) is always open. This command line interface is provided by the Windows console applications. You may want to check the headers io.h, crtl.h and process.h for specific Windows console routines. In addition, WinSIOUX.h and the WinSIOUX libraries provide a the Graphical User Interface consisting of a window that is much like a dumb terminal or TTY but with scrolling and cut and paste facilities.

Using Mac OS X and the Extras Library

On OS X, the functions which would previously have come from MSL_Extras are available from the System.framework using headers from the {OS X Volume}:usr:include:access path. Therefore Mach-O on Mac OS X requires access path settings so as to not use the MSL Extras library.

Do not use an access path to the top level {Compiler}:MSL: directory as that will bring in files from the new MSL_Extras.

Compatibility

Parts of the Main Standard Library, including both the ISO Standard C library and POSIX conforming procedures and definitions, as well as MSL library extensions, may not be implemented on all platforms. Furthermore, information about your target may not appear in this version of the printed documentation. You should consult the electronic documentation or release notes for your product to determine whether a particular function is compatible with your target platform.

Intrinsic Functions

Intrinsic functions generate in-line assembly code to perform the library routine. Generally these exist to allow direct access to machine functions which are not easily expressed directly in C. In some cases these map to single assembler instructions.

Some examples of intrinsics are as follows:

```
long __labs(long);
double __fabs(double);
```

and

```
void *__memcpy(void *, const void *, size_t);
```

where `__memcpy()` provides access to the block move in the code generator to do the block move inline.

2

MSL C and Multi-Threading

This reference contains a description of multi-threading and thread safety in the Main C library.

Introduction to Multi-Threading

In programming, the term thread is used to refer to the smallest amount of processor context state necessary to encapsulate a computation. A thread consists of a register set and a stack, together with the address space where data is stored. Some parts of this address space are private to the thread while other parts may be shared with other threads. Variables that belong to the storage class auto and that are instantiated by the thread are private to the thread and cannot be accessed by other threads. This is in contrast to variables that belong to the storage class static, which may be accessed by other threads in the same process. All threads also have access to the standard files, stdin, stdout, and stderr. In addition, a multi-threading implementation may provide for data with the same kind of lifetime as data in the static storage class but where access is restricted to the thread that owns it. Such data is known as thread-local data.

Most current operating systems are said to be multi-threaded because they allow the creation of additional threads within a process beyond the one that begins the execution of the process. Thus, in a multi-threaded operating system, a process may consist of more than one thread, all of them executing simultaneously.

Unless there is more than one processor, the threads are not really executing simultaneously; the operating system is giving the impression of simultaneity by multiplexing between the threads. The operating system determines which thread is to have control of the processor at any particular time. There are two models for operating a multi-threaded process — the cooperative model and the preemptive model. In the cooperative model, the threads signal their willingness to relinquish their control of the processor through a system call and the operating system then decides which thread will gain control. Thus, in this model, the execution of a thread can only be interrupted at points that are convenient to the algorithm.

In the preemptive model, the operating system switches between the threads at arbitrary and unpredictable times and points in the code being executed, Such a thread switch may occur between any two machine instructions and not coincide with a boundary between two C statements; it may even be part-way through the evaluation of an expression. One important result of this is that, since switching between threads happens unpredictably, if more than one thread is changing the value of a shared variable, the results of an execution

are likely to differ from one run to another. This lack of repeatability makes debugging and validation difficult. In what follows, we shall be assuming a preemptive model of multi-threading.

Multi-threading calls for mechanisms to protect against such uncertainty. Various methods exist for protecting segments of code from being executed by two threads at the same time. A program that is suitably protected against errors in the presence of multi-threading is said to be thread-safe.

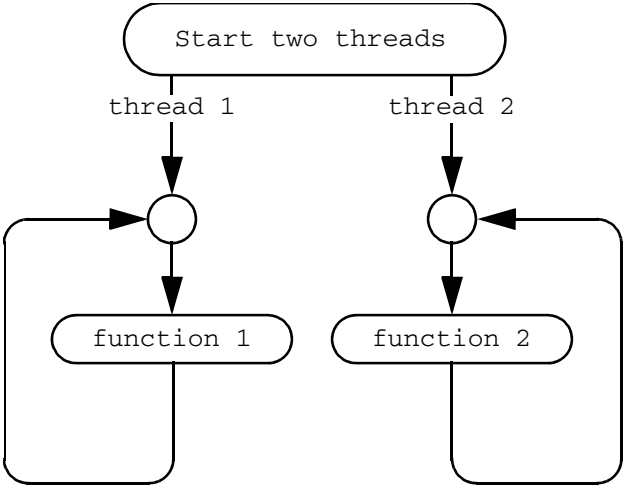
Definitions

Essentially, there are no standards for implementing multi-threading. In particular, neither the C99 Standard nor the POSIX Standard makes reference to threads. For the MSL C library, an MSL C Library function is said to be “thread-safe” if a single invocation of the function can be viewed as a single uninterruptable (i.e., atomic) operation. That is, it is possible to have two simultaneously executing threads in a single process, both using the function without danger of mutual interference.

For most functions in the library, the meaning of this is fairly clear. Some functions, such as `rand()` or `strtok()` are defined in C99 to maintain internal state variables and would appear, by definition, to be not thread-safe. However, in MSL on Windows, functions make use of thread-local data to ensure their thread-safety.

Figure 2.1 is a model for a test program that demonstrates the mutual thread safety of two library functions.

Figure 2.1 Thread Safety Test



- Function 1 and function 2 can be the same or different library functions.
- Each loop has to iterate many times.

For these two functions are said to be mutually threadsafe, the results produced by thread 1 must be exactly the same as they would be if thread 2 did not exist.

Thread safety problems only arise if the two threads are sharing data, for example, if function 1 and function 2 are both writing to the same file or if they both use a function that maintains its own internal state without protecting it in thread-local data, as is done in `strtok()`.

The MSL C library functions listed in Table 2.1 have special precautions to make them thread-safe:

Table 2.1 Functions with Special Thread Precautions

asctime	atexit	calloc	ctime	exit	fgetc
fgetpos	fgets	fgetwc	fgetws	fopen	fprintf
fputc	fputs	fputwc	fputws	fread	free
fscanf	fseek	fsetpos	ftell	fwprintf	fwrite
fwscanf	getc	getchar	gets	getwc	getwchar
gmtime	localeconv	localtime	malloc	printf	putc
putchar	puts	putwc	putwchar	raise	rand
realloc	scanf	setbuf	setlocale	setvbuf	signal
srand	strtok	tmpfile	tmpnam	ungetc	ungetwc
vfprintf	vfscanf	vfwprintf	vfwscanf	vprintf	vscanf
vwprintf	vwscanf	wcrtomb	wcsrtombs	wctomb	wprintf
wscanf					

The remaining MSL C functions are intrinsically thread-safe.

Reentrancy Functions

In the most recent versions of MSL C, great pains have been taken to make sure that every function works properly in a multi-threaded environment. The ANSI C standard makes no provisions at all for thread safety. Unlike some other library vendors, MSL C takes the

standpoint that all functions should be thread safe, providing that the _MSL_THREADSAFE macro is defined to 1.

When the _MSL_THREADSAFE macro is 0, many of the MSL C library functions lose their thread safe attributes. However, it may be useful to leave the _MSL_THREADSAFE macro as 0 even on a multi-threaded system for the reasons of speed. The library functions will be faster if they do not have to wait for thread synchronization. Since many programs are written using only a single thread, it is often advantageous to provide an efficient single-threaded library.

GCC and other library vendors provide an assortment of helper functions, all with a _r suffix, to indicate they are naturally thread safe.

Table 2.2 is a list of enhanced header files and the new "_r" functions they implement:

Table 2.2 Reentrant Functions in Standard Headers

dirent.h	stdlib.h	string.h	time.h
readdir_r	rand_r	strerror_r	asctime_r
			ctime_r
			gmtime_r
			localtime_r

3

Configuring MSL C

This chapter describes how to configure the Main Standard Library for C programming on your system. The topics are as follows:

Configuring Memory Management

MSL has a flexible memory management system. In most cases, the default configuration should be sufficient. If the platform operating system has its own memory management, simply complete the `__sys_alloc()`, `__sys_free()`, and `__sys_pointer_size()` routines in the `pool_alloc_xxx.c` file, where xxx represents the operating system, such as Mac or Win.

The `__sys_alloc()` routine is called with the size of a desired memory block whenever MSL needs more memory from the system to satisfy malloc(). If the request succeeds, a pointer is returned to the requested memory. The `__sys_free()` routine is called with a pointer to a previously allocated memory block obtained from `__sys_alloc()` when MSL no longer needs system memory. The memory should be returned to the operating system.

The `__sys_pointer_size()` routine is called with a pointer to a previously allocated memory block obtained from `__sys_alloc()` when MSL needs to determine the size of the memory block (the value of `size` passed to the `__sys_alloc()` call that obtained the memory).

If the platform does not have an operating system, or the OS does not support its own memory management, MSL can still accommodate malloc() by using a block of RAM in the program heap as a memory pool. The `_MSL_OS_ALLOC_SUPPORT` macro must be turned off in the platform prefix file. The `_MSL_HEAP_EXTERN_PROTOTYPES`, `_MSL_HEAP_START`, and `_MSL_HEAP_SIZE` macros must also be defined in order for MSL to find the heap block to use as a memory pool. Generally the pool is provided from the linker, or it could also be provided in the user program in the form of a large array.

The macros listed in (Table 3.1) are used to configure the MSL memory management system:

Table 3.1 MSL Memory Management Macros

Macro	Details
_MSL_MALLOC_IS_ALTIVEC_ALIGNED	Defined to 1 if MSL returns a 16 byte aligned memory block for malloc() requests. Defined to 0 if MSL returns less than a 16 byte aligned memory block for malloc() requests. Having `_MSL_MALLOC_IS_ALTIVEC_ALIGNED` may decrease code side on AltiVec machines because it does not have to have a separate implementation of vec_malloc(), but instead the existing malloc() is sufficient to satisfy all memory allocations.
_MSL_MALLOC_0_RETURNS_NON_NULL	Defined to 1 if MSL returns a non-NULL value for zero sized malloc() requests. Defined to 0 if MSL returns NULL for zero sized malloc() requests.
_MSL_OS_DIRECT_MALLOC	Defined to 1 if MSL is to ignore its own memory pools and make direct OS requests for memory every time malloc() is called. Defined to 0 if MSL uses its internal memory pools to satisfy malloc() requests. Generally it is preferred for MSL to use its own memory pooling, however using `_MSL_OS_DIRECT_MALLOC` can sometimes provide some help for debugging. `_MSL_OS_ALLOC_SUPPORT` must be on in order for `_MSL_OS_DIRECT_MALLOC` to work.
_MSL_CLASSIC_MALLOC(old name: _MSL_PRO4_MALLOC)	Defined if MSL is to use the Pro 4 memory pool scheme. Left undefined if MSL uses its more modern pooling scheme.
_MSL_ALLOCATE_SIZE	Defined to the routine name that returns the size of an allocated memory block. Default routine name is __allocate_size
_MSL_ALLOCATE	Defined to the internal MSL routine name that allocates a memory block, used only with the modern memory pooling scheme. Default routine name is __allocate

Table 3.1 MSL Memory Management Macros (*continued*)

Macro	Details
_MSL_ALLOCATE_RESIZE	Defined to the internal MSL routine name that changes the size of an allocated memory block, used only with the modern memory pooling scheme. Default routine name is __allocate_resize
_MSL_ALLOCATE_EXPAND	Defined to the internal MSL routine name that tries to expand the size of an allocated memory block, used only with the modern memory pooling scheme. Default routine name is __allocate_resize
_MSL_OS_ALLOC_SUPPORT(old name: _No_Alloc_OS_Support)	Defined to 1 if the MSL platform supports memory allocation. Defined to 0 if the MSL platform does not support memory allocation. When defined to 1, the MSL platform must supply both __sys_alloc(), __sys_free(), and __sys_pointer_size() functions in its pool_alloc_xxx.c file. When defined to 0, the MSL platform must define the `_MSL_HEAP_EXTERN_PROTOTYPES`, `_MSL_HEAP_START`, and `_MSL_HEAP_SIZE` macros, and there must be writable space provided at link time for MSL to use as a memory pool.
_MSL_HEAP_EXTERN_PROTOTYPES	When `_MSL_OS_ALLOC_SUPPORT` is off, the MSL alloc.c file must be able to access external symbols in order to get access to the start of the writable memory pool area and determine the memory pool size. The platform prefix file must define `_MSL_HEAP_EXTERN_PROTOTYPES` so it expands to appropriate external prototypes.
_MSL_POOL_ALIGNMENT	Specifies the alignment requirements of **malloc/ free** only when using the “Classic” allocator. The 'alignment' is a mask used to ensure that blocks allocated always have sizes that are multiples of a given power-of-two, in this case: four. Other values are possible, but for internal reasons the alignment factor must be a multiple of four and must also be a multiple of `sizeof(long)`.

Table 3.1 MSL Memory Management Macros (*continued*)

Macro	Details
_MSL_USE_FIX_MALLOC_POOLS	For tiny allocations, fixed sized pools help significantly speed allocation/deallocation, used only with the modern memory pooling scheme. You can reserve a pool for a small range of sizes. The use of fixed sized pools can be disabled by setting `_MSL_USE_FIX_MALLOC_POOLS` to 0. The default value is 1. Use of fixed size pools requires further configuration. The current shipping configuration is: 1. Each pool will handle approximately 4000 bytes worth of requests before asking for more memory. 2. There are 4 pool types. Each type is responsible for a different range of requests: a. 0 - 12 bytes b. 13 - 20 bytes c. 21 - 36 bytes d. 37 - 68 bytes Requests for greater than 68 bytes go to the variable size pools. The number of types of pools is configurable below. The range of requests for each type is also configurable.
_MSL_HEAP_EXTERN_PROTOTYPES	When `_MSL_OS_ALLOC_SUPPORT` is off, the MSL alloc.c file must be able to access external symbols in order to get access to the start of the writable memory pool area and determine the memory pool size. The platform prefix file must define `_MSL_HEAP_EXTERN_PROTOTYPES` so it expands to appropriate external prototypes.
_MSL_HEAP_START	When `_MSL_OS_ALLOC_SUPPORT` is off, the MSL alloc.c file must be able to find the start of the writable memory pool area. The `_MSL_HEAP_START` macro must be defined in the platform prefix file to expand to a memory location signifying the start of the writable memory pool area.

Table 3.1 MSL Memory Management Macros (*continued*)

Macro	Details
_MSL_HEAP_SIZE	When `_MSL_OS_ALLOC_SUPPORT` is off, the MSL alloc.c file must be able to determine the size of the writable memory pool. The `_MSL_HEAP_SIZE` macro must be defined in the platform prefix file to expand to the size of the writable memory pool.
__CALLOC	If `__CALLOC` is undefined, the name of the MSL calloc() routine is simply calloc. Otherwise, if `__CALLOC` is defined, the MSL calloc() routine is named whatever the `__CALLOC` macro is defined to. This is useful in case a platform has its own system implementation of calloc and the MSL implementation conflicts over the same name.
__FREE	If `__FREE` is undefined, the name of the MSL free() routine is simply free. Otherwise, if `__FREE` is defined, the MSL free() routine is named whatever the `__FREE` macro is defined to. This is useful in case a platform has its own system implementation of free and the MSL implementation conflicts over the same name.
__MALLOC	If `__MALLOC` is undefined, the name of the MSL malloc() routine is simply malloc. Otherwise, if `__MALLOC` is defined, the MSL malloc() routine is named whatever the `__MALLOC` macro is defined to. This is useful in case a platform has its own system implementation of malloc and the MSL implementation conflicts over the same name.
__REALLOC	If `__REALLOC` is undefined, the name of the MSL realloc() routine is simply realloc. Otherwise, if `__REALLOC` is defined, the MSL realloc() routine is named whatever the `__REALLOC` macro is defined to. This is useful in case a platform has its own system implementation of realloc and the MSL implementation conflicts over the same name.

Configuring Time and Clock

MSL comes configured by default to support a platform having the ability to determine the time of day and also the ability to return an internal clock tick. The support does not come for free. Each platform must define four simple functions to provide MSL with low-level information from the platform hardware or operating system. Time and clock stub functions are in the `time_xxx.c` platform file, where xxx is the platform, such as Mac or Win.

For systems that support an internal clock tick, the `__get_clock()` function must obtain the current clock tick and return its value to MSL. If the clock tick information is not obtainable, return the value -1.

For systems that support the ability to determine the time of day, the `__get_time()` function must obtain the current time of day and return its `time_t` equivalent value to MSL. Depending on the value of `_MSL_TIME_T_IS_LOCALTIME`, the current time is either the “local time” time of day, or it is “Universal Time Coordinated” (UTC), which was formerly called “Greenwich Mean Time” (GMT).

If the current time of day is not obtainable, return the value -1. The `__to_gm_time()` function must take a “local time” time of day `time_t` value and convert it into a “global mean” `time_t` value. If the conversion takes place properly, return 1 to MSL. If the conversion fails, return 0 to MSL.

The `__to_local_time()` function must take a UTC time of day `time_t` value and convert it into a “local time” `time_t` value. If the conversion takes place properly, return 1 to MSL. If the conversion fails, return 0 to MSL. The `__to_local_time()` function is only used when `_MSL_TIME_T_IS_LOCALTIME` is off.

Finally, the `__isdst()` function must try to determine whether or not daylight savings time is in effect. If daylight savings time is not in effect, return 0. If daylight savings time is in effect, return 1. If daylight savings time information is not obtainable, return -1.

The macros listed in Table 3.2 are used to configure the time and clock support:

Table 3.2 MSL Time and Clock Macros

Macros	Details
_MSL_OS_TIME_SUPPORT	Defined to 1 if the MSL platform supports retrieving the time. Defined to 0 if the MSL platform does not support retrieving the time.
_MSL_CLOCK_T_AVAILABLE	Defined to 1 if the MSL platform supports the `clock_t` type. Defined to 0 if the MSL platform does not support the `clock_t` type. The `_MSL_OS_TIME_SUPPORT` macro must be on before the `_MSL_CLOCK_T_AVAILABLE` macro is recognized.
_MSL_CLOCK_T_DEFINED	Defined to 1 if the MSL platform defined the `clock_t` type. Defined to 0 if the MSL platform does not define the `clock_t` type.
_MSL_CLOCK_T	Set to the `clock_t` type. Default value is unsigned long.
_MSL_TIME_T_AVAILABLE	Defined to 1 if the MSL platform supports the `time_t` type. Defined to 0 if the MSL platform does not support the `time_t` type. The `_MSL_OS_TIME_SUPPORT` macro must be on before the `_MSL_TIME_T_AVAILABLE` macro is recognized.
_MSL_TIME_T_DEFINED	Defined to 1 if the MSL platform defined the `time_t` type. Defined to 0 if the MSL platform does not define the `time_t` type.
_MSL_TIME_T_IS_LOCALTIME	Defined to 1 if the MSL platform value for `time_t` represents local time, preadjusted for any time zone and offset from UTC (GMT). Defined to 0 if the MSL platform value for `time_t` represents Universal Time Coordinated. The default value is 1.
_MSL_CLOCKS_PER_SEC	Set to the number of clock ticks per second. The default value is 60.

Configuring File I/O

Setting up MSL to handle file I/O is a fairly intensive task. It requires many platform-specific routines to be written. The easiest way to configure file I/O is to simply have MSL not know about it by defining `_MSL_OS_DISK_FILE_SUPPORT` to 0. In that mode,

MSL does not know about any routines requiring file manipulation such as fopen(), fread(), fwrite(), fclose(), etc.

When _MSL_OS_DISK_FILE_SUPPORT is defined to 1, many low-level file routines need to be written, and several supporting macros also need to be defined properly. First, make sure that _MSL_FILENAME_MAX properly reflects the maximum size of a filename. Also, if the default internal MSL file buffer size is not appropriate, choose a new value for _MSL_BUFSIZ. Once all the macros are properly defined, the following routines in file_io_xxx.c (where xxx represents platform, such as Mac or Win) must be completed.

Routines

The __open_file routine is perhaps the most complicated of all the low level file I/O routines. It takes a filename and a bunch of mode flags, opens the file, and returns a handle to the file. A file handle is a platform-specific identifier uniquely identifying the open file. The mode flags specify if the file is to be opened in read-only, write-only, or read-write mode. The flags also specify if the file must previously exist for an open to be successful, if the file can be opened whether or not it previously existed, or if the file is truncated (position and end of file marker set to 0) once it is open. If the file is opened successfully, return __no_io_error. If there was an error opening the file, return __io_error.

The __open_temp_file routine in the file_io_Starter.c file is mostly platform independent. It may be customized if there are more efficient ways to perform the task. The __open_temp_file routine is called by tmpfile() to perform the low level work of creating a new temporary file (which is automatically deleted when closed).

The __read_file routine takes a file handle, a buffer, and the size of the buffer and should read information from the file described by the file handle into the buffer. The buffer does not have to be completely filled, but at least one character should be read. The number of characters successfully read is returned in the *count parameter. If an end of file is reached after more than one character has been read, simply return the number of characters read. The subsequent read call should then return zero characters read and a result code of __io_EOF. If the read was successful, return __no_io_error. If the read failed, return __io_error.

The __write_file routine takes a file handle, a buffer, and the size of the buffer. It should then write information to the file described by the file handle from the buffer. The number of characters successfully written is returned in the *count parameter. If the write was successful, return __no_io_error. If the write failed, return __io_error.

The __position_file routine takes a file handle, a position displacement, and a positioning mode and should set the current position in the file described by the file handle based on the displacement and mode. The displacement value is passed as an unsigned long due to certain internal constraints of MSL. The value should actually be treated as signed long. The mode specifies if displacement is an absolute position in the file (treat as position of 0 + displacement), a change from the current position (treat as current position

+ displacement), or an offset from the end of file mark (treat as EOF position + displacement). If the positioning was successful, return __no_io_error. If the positioning failed, return __io_error.

The __close_file routine closes the specified file. If the file was created by __open_temp_file, it should additionally be deleted. If the close was successful, return __no_io_error. If the close failed, return __io_error.

The __temp_file_name routine creates a new unique filename suitable for a temporary file. It is called by tmpnam() to perform the low-level work.

The __delete_file routine deletes an existing file, given its filename. If the delete was successful, return __no_io_error. If the delete failed, return __io_error.

The __rename_file routine renames an existing file, given its existing filename and a desired new filename. If the rename was successful, return __no_io_error. If the rename failed, return __io_error.

Finally, if the platform wants to provide some additional nonstandard file I/O routines that are common to the Windows operating system, make sure _MSL_WIDE_CHAR is on, then also define _MSL_WFILEIO_AVAILABLE to 1. The following stub routines must also be completed in the file_io_xxx.c source file. All routines take wchar_t* parameters instead of char*: __wopen_file (same function as __open_file), __wtemp_file_name (same function as __temp_file_name), __wdelete_file (same function as __delete_file), and __wrename_file (same function as __rename_file).

The macros listed in Table 3.3 are used to configure the MSL file I/O system:

Table 3.3 MSL File I/O Macros

Macros	Details
_MSL_OS_DISK_FILE_SUPPORT	Defined to 1 if the MSL platform supports disk file I/O. Defined to 0 if the MSL platform does not support disk file I/O.
_MSL_FILENAME_MAX	Set to the maximum number of characters allowed in a filename. The default value is 256.

Table 3.3 MSL File I/O Macros (*continued*)

Macros	Details
_MSL_BUFSIZ	Set to the file I/O buffer size in bytes used to set the `BUFSIZ` macro. The default value is 4096.
_MSL_WFILEIO_AVAILABLE	Defined to 1 if MSL has `wchar_t` extensions for file I/O calls needing filenames, such as `_wfopen()`. Defined to 0 if MSL only has traditional C file I/O calls. It is an error to have `_MSL_WIDE_CHAR` off and `_MSL_WFILEIO_AVAILABLE` on.

Configuring Console I/O

Console I/O for stdin, stdout, and stderr can be configured in many different ways. The easiest way to configure console I/O is to have it turned off completely. When `_MSL_CONSOLE_SUPPORT` is off, MSL does not know about stdin, stdout, or stderr. Calls such as printf() are not placed in the standard C library.

When `_MSL_CONSOLE_SUPPORT` is on, there are essentially three ways in which to provide console access.

- The first way is to have MSL automatically throw away all items read and written to the console by turning on `_MSL_NULL_CONSOLE_ROUTINES`.
- The second way is to have MSL treat all console I/O as if it were file I/O by turning on `_MSL_FILE_CONSOLE_ROUTINES`. Treating the console as file I/O requires configuring the file I/O portion of MSL as described in the previous section. Input and output go through the `__read_file`, `__write_file`, `__close_file` bottlenecks instead of `__read_console`, `__write_console`, and `__close_console`.
- The third way to provide console access is to simply turn on `_MSL_CONSOLE_SUPPORT` and leave the remainder of the `_MSL_CONSOLE_FILE_IS_DISK_FILE` and `_MSL_FILE_CONSOLE_ROUTINES` flags in their default (off) state. MSL will then call `__read_console` when it needs input (for example from scanf()), `__write_console` when it wants to send output (for example from printf()), and `__close_console` when the console is no longer needed. The three routines should be provided in the `console_io_xxx.c` file.

The macros listed in Table 3.4 are used to configure the MSL console I/O:

Table 3.4 MSL Console I/O Macros

Macro	Details
_MSL_CONSOLE_SUPPORT	Defined to 1 if the MSL platform supports console I/O. Defined to 0 if the MSL platform does not support console I/O. Default value is 1.
_MSL_BUFFERED_CONSOLE	Defined to 1 if the MSL platform console I/O is buffered. Defined to 0 if the MSL platform console I/O is unbuffered. Default value is 1.
_MSL_CONSOLE_FILE_IS_DISK_FILE	Defined to 1 if the MSL platform has console I/O, but it is really in a file. Defined to 0 if the MSL platform has traditional console I/O. Default value is 0.
_MSL_NULL_CONSOLE_ROUTINES	Defined to 1 if the MSL platform does not perform console I/O. Defined to 0 if the MSL platform performs console I/O. This flag may be set independently of `_MSL_CONSOLE_SUPPORT`; however, when `_MSL_CONSOLE_SUPPORT` is off, `_MSL_NULL_CONSOLE_ROUTINES` must always be on. When `_MSL_CONSOLE_SUPPORT` is on and `_MSL_NULL_CONSOLE_ROUTINES` is also on, all console I/O is essentially ignored. Default value is 0.
_MSL_FILE_CONSOLE_ROUTINES	Defined to 1 if the MSL platform uses the file I/O read/write/close routines for console I/O instead of using the special console read/write/close routines. Define to 0 if the MSL platform uses the special console read/write/close routines. When `_MSL_CONSOLE_FILE_IS_DISK_FILE` is on, `_MSL_FILE_CONSOLE_ROUTINES` must always be on. Default value is 0.

Configuring Threads

MSL is highly adaptive when it comes to multi-threading. The C standard makes no mention of threading, and it even has points where global data is accessed directly, for example `errno`, `asctime()`, etc. MSL can be configured to know about multithreaded

systems and be completely reentrant. There are essentially three ways to configure the MSL thread support:

- single thread (not reentrant at all)
- multithreaded with global data (mostly reentrant)
- and multithreaded with thread local data (completely reentrant).

With `_MSL_THREADSAFE` off, MSL is setup to operate in a single thread environment. There are no critical regions to synchronize operations between threads. It is sometimes advantageous to configure MSL for a single threaded environment. Operations such as file I/O and memory allocations will be quicker, since there is no need to ask for or wait for critical regions. Many simple programs do not make use of threads, thus there may be no need for the additional overhead.

With `_MSL_THREADSAFE` on, MSL is setup to synchronize operations between threads properly by the use of critical regions. Critical regions are supplied either by POSIX pthread functions or by the use of platform specific calls. If the platform has an underlying POSIX layer supporting pthreads, simply turning on `_MSL_THREADSAFE` and `_MSL_PTHREADS` is enough for MSL to fully operate. No other custom code is necessary.

With `_MSL_THREADSAFE` on and `_MSL_PTHREADS` off, the platform must provide its own critical region code. This is generally done by first providing an array of critical region identifiers in the `critical_regions_xxx.c` file, and then by completing the four critical region functions in the `critical_regions_xxx.h` header file (where xxx represents operating system, such as Mac or Win). The compiler runtime library must make a call to `__init_critical_regions()` before calling `main()`.

With `_MSL_THREADSAFE` on, the `_MSL_LOCALDATA_AVAILABLE` flag controls whether or not the MSL library is completely reentrant or not. When `_MSL_LOCALDATA_AVAILABLE` is off, the MSL library uses global and static variables, and is therefore not completely reentrant for such items as `errno`, the random number seed used by rand(), strtok() state information, etc. When `_MSL_LOCALDATA_AVAILABLE` is on, the MSL library uses thread local storage to maintain state information. Each thread has its own copy of some dynamic memory that gets used.

With `_MSL_LOCALDATA_AVAILABLE` on and `_MSL_PTHREADS` on, simply adding the following line to the platform prefix file is enough to fully support complete reentrancy:

```
#define _MSL_LOCALDATA(_a) __msl_GetThreadLocalData()->_a
```

With `_MSL_LOCALDATA_AVAILABLE` on and `_MSL_PTHREADS` off, the platform must completely supply its own routines to maintain and access thread local storage. The `thread_local_data_xxx.h` and `thread_local_data_xxx.c` files are used to provide the necessary functionality. Also, the common MSL header (`msl_thread_local_data.h`) must be modified to include the platform header

(thread_local_data_xxx.h) based on its __dest_os value. The _MSL_LOCALDATA macro is used to access items in thread local storage. So, for example, if the random number seed needs to be obtained, the MSL code will say _MSL_LOCALDATA(random_next) to get the random number seed. The macro must expand to an l-value expression.

At times, it may be easier to turn on _MSL_PTHREADS even if the underlying platform does not have built-in pthread support. Instead of writing custom code to support the MSL threading model, it may be easier to turn on _MSL_PTHREADS and then write comparable pthread routines. When _MSL_PTHREADS is on and _MSL_THREADSAFE is on, four pthread routines in the pthread_xxx.c file are used by MSL to implement critical regions.

pthread Routines

The pthread_mutex_init routine creates a single mutex. MSL will always pass NULL as the second attr argument, which means to use default mutex attributes. Return zero upon success, return an error code upon failure.

The pthread_mutex_destroy routine disposes of a single mutex. Return zero upon success, return an error code upon failure.

The pthread_mutex_lock routine acquires a lock on a single mutex. If the mutex is already locked when pthread_mutex_lock is called, the routine blocks execution of the current thread until the mutex is available. Return zero upon success, return an error code upon failure.

The pthread_mutex_unlock routine releases a lock on a single mutex. Return zero upon success, return an error code upon failure.

Additionally, when _MSL_LOCALDATA_AVAILABLE is on, four more pthread routines in the pthread_xxx.c file are used by MSL to implement thread local data:

- The pthread_key_create routine creates a new thread local data identifier. Each thread can then access its own individual data elements through the identifier. When a thread terminates, the destructor routine is called with the single argument of pthread_getspecific() to clean up any necessary thread local data. Return zero upon success, return an error code upon failure.
- The pthread_key_delete routine disposes of a thread local data identifier. Return zero upon success, return an error code upon failure.
- The pthread_setspecific routine associates a value to a previously created thread local data identifier. The value is specific to the currently executing thread. Return zero upon success, return an error code upon failure.
- The pthread_getspecific routine retrieves a value associated with a thread local data identifier. The value is specific to the currently executing thread. If no

value has been associated with the thread local data identifier, return NULL. Otherwise, return the previously associated value.

The macros listed in Table 3.5 are used to configure and use the MSL threading support:

Table 3.5 MSL Thread Support Macros

Macros	Details
_MSL_THREADSAFE	Defined to 0 if there is no multi-thread support in MSL. Defined to 1 if there should be multi-thread support in MSL. When defined to 1, many internal aspects of MSL are guarded by critical regions. Having critical regions inside MSL will slow down the execution time for the trade-off of working correctly on a multi-threaded system. Also, many MSL functions will use thread local storage for maintaining state information.
_MSL_PTHREADS	Defined to 1 if the MSL platform supports the POSIX threading model. Defined to 0 if the MSL platform does not support the POSIX threading model. It is an error to define `_MSL_PTHREADS` to 1 and `_MSL_THREADSAFE` to 0. MSL has generic support for the POSIX thread model, so turning on `_MSL_THREADSAFE` and `_MSL_THREADS` is enough to properly support a multithreaded system without the need to write any additional support code.
_MSL_LOCALDATA	Internal MSL flag for accessing thread local data when `_MSL_THREADSAFE` is 1. Accesses static global data when `_MSL_THREADSAFE` is 0.
_MSL_LOCALDATA_AVAILABLE	Defined to 1 if the MSL platform supports thread local data, accessible using the `_MSL_LOCALDATA` macro. Defined to 0 if the MSL platform does not support thread local data.

4

alloca.h

This header defines one function, alloca, which lets you allocate memory quickly using the stack.

Overview of alloca.h

The alloca.h header file consists of "alloca", which allocates memory from the stack.

---

alloca

Allocates memory quickly on the stack.

```
#include <alloca.h>
void *alloca(size_t nbytes);
```

Table 4.1 alloca

nbytes	size_t	number of bytes of allocation

Remarks

This function returns a pointer to a block of memory that is `nbytes` long. The block is on the function's stack. This function works quickly since it decrements the current stack pointer. When your function exits, it automatically releases the storage.

NOTE The AltiVec version of `alloca()` allocates memory on a 16 byte alignment.

If you use `alloca()` to allocate a lot of storage, be sure to increase the Stack Size for your project in the Project preferences panel.

If it is successful, `alloca()` returns a pointer to a block of memory. If it encounters an error, `alloca()` returns `NULL`.

This function may not be implemented on all platforms.

See Also

"calloc" on page 417, "free" on page 422

"malloc" on page 427, "realloc" on page 434)

5

assert.h

The `assert.h` header file provides a debugging macro, `assert`, that outputs a diagnostic message and stops the program if a test fails.

Overview of assert.h

The `assert.h` header file provides a debugging macro, "assert", which outputs a diagnostic message and stops the program if a test fails.

assert

Abort a program if a test is false.

```
#include <assert.h>
void assert(int expression);
```

Table 5.1 assert

expression	int	A boolean expression being evaluated

Remarks

If `expression` is false the `assert()` macro outputs a diagnostic message to `stderr` and calls `abort()`. The diagnostic message has the form

```
file: line test -- assertion failed
abort -- terminating
```

where file is the source file, line is the line number, and test is the failed expression.

To turn off the `assert()` macros, place a `#define NDEBUG` (no debugging) directive before the `#include <assert.h>` directive.

This macro may not be implemented on all platforms.

See Also

"abort" on page 405

Listing 5.1 Example of assert() Usage

```
#undef NDEBUG
   /* Make sure that assert() is enabled */
#include <assert.h>
#include <stdio.h>

int main(void)
{
   int x = 100, y = 5;
   printf("assert test.\n");

/*This assert will output a message and abort the program */
   assert(x > 1000);
   printf("This will not execute if NDEBUG is undefined\n");
   return 0;
}
```

```
Output:
assert test.
foo.c:12 x > 1000 -- assertion failed
abort -- terminating
```

6

conio.h

The `conio.h` header file consist of various runtime declarations that pertain to the Win32 x86 targets for console input and output.

Overview of conio.h

This header file defines the facilities as follows:

- “ clrscr” on page 30 clears the console window.
- “getch” on page 30 reads a char from the input screen.
- “getche” on page 31 reads a char and echoes to the screen.
- “ gotoxy” on page 31 places the cursor on a console window.
- “ initscr” on page 32 sets up a console in a GUI application.
- “inp” on page 32 reads a byte from an input port.
- “inpd” on page 33 reads a double word from and input port.
- “inpw” on page 33 reads a word from an input port.
- “kbhit” on page 34 returns true if a key is pressed.
- “outp” on page 34 outputs a byte to a port.
- “outpd” on page 35outputs a double word to a port.
- “outpw” on page 36 outputs a word to a port.
- “ textattr” on page 36 sets the text attributes.
- “ textbackground” on page 37 sets the text background color.
- “ textcolor” on page 37 sets the text color.
- “ wherex” on page 38 returns the horizontal coordinate.
- “ wherey” on page 38 returns the vertical coordinate.

_clrscr

Clears the standard output screen.

```
#include <conio.h>
void _clrscr(void);
```

Remarks

This facility has no parameters.

No value is returned in this implementation.

This function is Windows only when declared from this header.

getch

This function reads a single char from the standard input device and does not echo it to the output.

```
#include <conio.h>
int getch(void);
int _getch(void);
```

Remarks

This facility has no parameters.

The function `getch` returns the `char` read.

This function is Windows only when declared from this header.

See Also

"getc" on page 348

"getchar" on page 349

getche

This function reads a single char from the standard input device and echoes it to the output without the need of pressing the enter key.

```
#include <conio.h>
int getche(void);
int _getche(void);
```

Remarks

This facility has no parameters.

The function `getche` returns the `char` read.

This function is Windows only when declared from this header.

See Also

"getc" on page 348

"getchar" on page 349

_gotoxy

Moves the cursor to the horizontal and vertical coordinates in a standard output device.

```
#include <conio.h>
void _gotoxy(int x, int y);
```

Table 6.1 _gotoxy

x	int	vertical screen coordinate
y	int	horizontal screen coordinate

Remarks

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

“ wherex” on page 38
“ wherey” on page 38

_initscr

Sets up standard 80x25 console with no scrolling region.

```
#include <conio.h>
void _initscr(void);
```

Remarks

This facility has no parameters.

There is no need to call `_initscr()` unless you have a GUI application (subsystem Windows GUI), where no console is available at runtime.

No value is returned in this implementation.

This function is Windows only when declared from this header.

inp

Reads a `byte` from the specified port

```
#include <conio.h>
unsigned char inp(unsigned short port);
unsigned char _inp(unsigned short port);
```

Table 6.2 inp

port	unsigned short	a port specified by number

Remarks

The value as a `byte` read is returned.

This function is Windows only when declared from this header.

See Also

“inpd” on page 33

"inpw" on page 33

inpd

Reads a `double word` from the specified port

```
#include <conio.h>
unsigned long inpd(unsigned short port);
unsigned long _inpd(unsigned short port);
```

Table 6.3 inpd

port	unsigned short	a port specified by number

Remarks

The value as a `long` read is returned.

This function is Windows only when declared from this header.

See Also

"inp" on page 32

"inpw" on page 33

inpw

Reads a `word` from the specified port

```
#include <conio.h>
unsigned short inpw(unsigned short port);
unsigned short _inpw(unsigned short port);
```

Table 6.4 inpw

port	unsigned short	a port specified by number

Remarks

The value as a `word` read is returned.

This function is Windows only when declared from this header.

See Also

"inp" on page 32

"inpd" on page 33

kbhit

This function is used in a loop to detect an immediate keyboard key press.

```
#include <conio.h>
int kbhit(void);
int _kbhit(void);
```

This facility has no parameters.

Remarks

The `kbhit` function has the side effect of discarding any non-keyboard input records in the console input queue

If the keyboard is pressed `kbhit()` returns a non zero value otherwise it return zero.

This function is Windows only when declared from this header.

See Also

"getch" on page 30

"getche" on page 31

outp

Outputs a `byte` to a specified port.

```
#include <conio.h>
void outp(unsigned short pt, unsigned char out);
void _outp(unsigned short pt, unsigned char out);
```

Table 6.5 outp

pt	unsigned short	a port specified by number
out	unsigned char	a byte value sent to the output

Remarks

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

"outpd" on page 35

"outpw" on page 36

outpd

Sends a double word to the output port specified.

```
#include <conio.h>
void outpd(unsigned short pt, unsigned long out);
void _outpd(unsigned short pt, unsigned long out);
```

Table 6.6 outpd

pt	unsigned short	a port specified by number
out	unsigned long	a double word value sent to the output

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

"outp" on page 34

"outpw" on page 36

outpw

Sends a word to the output port specified.

```
#include <conio.h>
void outpw(unsigned short pt, unsigned short out);
void _outpw(unsigned short pt, unsigned short out);
```

Table 6.7 outpw

pt	unsigned short	a port specified by number
out	unsigned short	a word value sent to the output

Remarks

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

"outp" on page 34

"outpd" on page 35

_textattr

This function sets the attributes of the console text.

```
#include <conio.h>
void _textattr(int newattr);
```

Table 6.8 _textattr

newattr	int	the text attributes to be set

Remarks

The function `_textattr` allows you to set both the foreground and background attributes with the variable `newattr`. The attributes available for this function are defined in the header file `wincon.h`.

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

“ textbackground” on page 37

“ textcolor” on page 37

_textbackground

This function sets the console’s background color.

```
#include <conio.h>
void _textbackground(int newcolor);
```

Table 6.9 _textbackground

newcolor	int	the background color to be set

Remarks

The attributes available for this function are defined in the header file `wincon.h`.

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

“ textattr” on page 36

“ textcolor” on page 37

_textcolor

This function sets the console’s text color.

```
#include <conio.h>
void _textcolor(int newcolor);
```

Table 6.10 _textcolor

newcolor	int	the text color to be set

Remarks

The attributes available for this function are defined in the header file wincon.h.

No value is returned in this implementation.

This function is Windows only when declared from this header.

See Also

"_textattr" on page 36

"_textbackground" on page 37

_wherex

Determines the horizontal coordinate of the cursor in a console window.

```
#include <conio.h>
int _wherex(void);
```

Remarks

This facility has no parameters.

Returns the horizontal coordinate.

This function is Windows only when declared from this header.

See Also

" wherey" on page 38

" gotoxy" on page 31

_wherey

Determines the vertical coordinate of the cursor in a console window.

```
#include <conio.h>
int _wherey(void);
```

Remarks

This facility has no parameters.

Returns the vertical coordinate.

This function is Windows only when declared from this header.

See Also

“_gotoxy” on page 31

“_wherex” on page 38

7

console.h

This header file contains procedures and types, which help you port a program that was written for a command-line/console interface to the Macintosh operating system.

The console.h header file consist of various runtime declarations that pertain to the Classic and Carbon Macintosh interfaces.

Overview of console.h

This header file defines the facilities as follows:

- "ccommand" on page 41 helps you port a program that relies on command-line arguments.
- "clrscr" on page 43 clears the SIOUX window and flushes the buffer.
- "getch" on page 43 returns the keyboard character pressed when an ascii key is pressed.
- "InstallConsole" on page 44 installs the Console package.
- "kbhit" on page 44 returns true if any keyboard key is pressed without retrieving the key.
- "ReadCharsFromConsole" on page 45 reads from the Console into a buffer.
- "RemoveConsole" on page 45 removes the console package.
- "__ttyname" on page 46 returns the name of the terminal associated with the file id. The unix.h function ttyname calls this function.
- "WriteCharsToConsole" on page 46 writes a stream of output to the Console window.

ccommand

Lets you enter command-line arguments for a SIOUX program.

```
#include <console.h>
int ccommand(char ***argv);
```

Table 7.1 ccommand

argv	char ***	The address of the second parameter of your command line

Remarks

The function ccommand() must be the first code generated in your program. It must directly follow any variable declarations in the main function.

This function displays a dialog that lets you enter arguments and redirect standard input and output. Please refer to "Overview of SIOUX" on page 255, for information on customizing SIOUX, or setting console options.

Only `stdin, stdout, cin, and cout` are redirected. Standard error reporting methods `stderr, cerr,` and `clog` are not redirected.

The maximum number of arguments that can be entered is determined by the value of MAX_ARGS defined in ccommand.c and is set to 25. Any arguments in excess of this number are ignored.

Enter the command-line arguments in the Argument field. Choose where your program directs standard input and output with the buttons below the field: the buttons on the left are for standard input and the buttons on the right are for standard output. If you choose Console, the program reads from or write to a SIOUX window. If you choose File, `ccommand()` displays a standard file dialog which lets you choose a file to read from or write to. After you choose a file, its name replaces the word *File*.

The function `ccommand()` returns an integer and takes one parameter which is a pointer to an array of strings. It fills the array with the arguments you entered in the dialog and returns the number of arguments you entered. As in UNIX or DOS, the first argument, the argument in element 0, is the name of the program. Listing 7.1 has an example of command line usage.

This function returns the number of arguments you entered.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"Customizing SIOUX" on page 258

Listing 7.1 Example of ccommand() Usage

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

```
int main(int argc, char *argv[])
{
  int i;

  argc = ccommand(&argv);

  for (i = 0; i < argc; i++)
    printf("%d. %s\n", i, argv[i]);
  return 0;
}
```

clrscr

Clears the console window and flushes the buffers;

```
#include <console.h>
```

```
void clrscr(void);
```

Remarks

This function is used to select all and clear the screen and buffer by calling SIOUXclrscr from SIOUX.h.

Macintosh only—this function may not be implemented on all Mac OS versions.

getch

Returns the keyboard character pressed when an ascii key is pressed

```
#include <console.h>
```

```
int getch(void);
```

Remarks

This function is used for console style menu selections for immediate actions.

Returns the keyboard character pressed when an ascii key is pressed.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"kbhit" on page 44

InstallConsole

Installs the Console package.

```
#include <console.h>
extern short InstallConsole(short fd);
```

Table 7.2 InstallConsole

fd	short	A file descriptor for standard i/o

Remarks

Installs the Console package, this function will be called right before any read or write to one of the standard streams.

This function returns any error.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"RemoveConsole" on page 45

kbhit

Returns true if any keyboard key is pressed.

```
#include <console.h>
int kbhit(void);
```

Remarks

Returns true if any keyboard key is pressed without retrieving the key used for stopping a loop by pressing any key

This function returns non zero when any keyboard key is pressed.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"getch" on page 43

ReadCharsFromConsole

Reads from the Console into a buffer.

```
#include <console.h>
extern long ReadCharsFromConsole
  (char *buffer, long n);
```

Table 7.3 ReadCharsFromConsole

buffer	char *	A stream buffer
n	long	Number of char to read

Remarks

Reads from the Console into a buffer. This function is called by read.

Any errors encountered are returned.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"WriteCharsToConsole" on page 46

RemoveConsole

Removes the console package.

```
#include <console.h>
extern void RemoveConsole(void);
```

Remarks

Removes the console package. It is called after all other streams are closed and exit functions (installed by either atexit or __atexit) have been called.

Since there is no way to recover from an error, this function doesn't need to return any.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"InstallConsole" on page 44

__ttyname

Returns the name of the terminal associated with the file id.

```
#include <console.h>
extern char *__ttyname(long fildes);
```

Table 7.4 _ttyname

fildes	long	The file descriptor

Remarks

Returns the name of the terminal associated with the file id. The unix.h function ttyname calls this function (we need to map the int to a long for size of int variance).

Returns the name of the terminal associated with the file id.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"ttyname" on page 534

WriteCharsToConsole

Writes a stream of output to the Console window.

```
#include <console.h>
extern long WriteCharsToConsole(char *buffer, long n);
```

Table 7.5 WriteCharsToConsole

buffer	char *	A stream buffer
n	long	Number of char to write

Remarks

Writes a stream of output to the Console window. This function is called by write.

Any errors encountered are returned.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"ReadCharsFromConsole" on page 45

8

crtl.h

The `crtl.h` header file consist of various runtime declarations that pertain to the Win32 x86 targets.

Overview of crtl.h

This header file defines the facilities as follows:

- “Argc” on page 49 is the argument list count.
- “Argv” on page 50 is the argument list variables.
- “ DllTerminate” on page 50 shows when a DLL is running terminate code.
- “environ” on page 50 is the environment pointers.
- “ HandleTable” on page 51 is a structure allocated for each ed file handle.
- “ CRTStartup” on page 51 initializes the C Runtime start-up routines.
- “ RunInit” on page 52 initializes the runtime, static classes and variables.
- “ SetupArgs” on page 52 sets up the command line arguments.

Argc

The argument count variable

```
#include <crtl.h>
extern int __argc;
```

Remarks

Used for command line argument count.

This function may not be implemented on all platforms.

Argv

The argument command variables.

```
#include <crtl.h>
extern char **__argv;
```

Remarks

The command line arguments.

This function may not be implemented on all platforms.

_DllTerminate

A flag to determine when a DLL is running terminate code.

```
#include <crtl.h>
extern int _DllTerminate;
```

Remarks

This flag is set when a DLL is running terminate code.

This function may not be implemented on all platforms.

environ

The environment pointers

```
#include <crtl.h>
extern char *(*environ);
```

Remarks

This is a pointer to the environment.

This function may not be implemented on all platforms.

_HandleTable

FileStruct is a structure allocated for each file handle

```
#include <crtl.h>
typedef struct
{
   void *handle;
   char translate;
   char append;
} FileStruct;

extern FileStruct *_HandleTable[NUM_HANDLES];
extern int _HandPtr;
```

Remarks

The variable `_HandPtr` is a pointer to a table of handles.

The variable `NUM_HANDLES` lists the number of possible handles.

This function may not be implemented on all platforms.

_CRTStartup

The function _CRTStartup is the C Runtime start-up routine.

```
#include <crtl.h>
extern void _CRTStartup();
```

Remarks

This function may not be implemented on all platforms.

_RunInit

The function `_RunInit` initializes the runtime, all static classes and variables.

```
#include <crtl.h>
extern void _RunInit();
```

Remarks

This function may not be implemented on all platforms.

_SetupArgs

The function _SetupArgs sets up the command line arguments.

```
#include <crtl.h>
extern void _SetupArgs();
```

Remarks

This function may not be implemented on all platforms.

9

ctype.h

The `ctype.h` header file supplies macros and functions for testing and manipulation of character type.

Overview of ctype.h

Character Testing and Case Conversion

The `ctype.h` header file supplies macros for testing character type and for converting alphabetic characters to uppercase or lowercase. The `ctype.h` macros support ASCII characters (`0x00` to `0x7F`), and the `EOF` value. These macros are not defined for the Apple Macintosh Extended character set (`0x80` to `0xFF`).

This header file defines the facilities as follows:

- "isalnum" on page 55 tests for alphabetical and numerical characters.
- "isalpha" on page 57 tests for alphabetical characters.
- "isblank" on page 57 tests for a space between words and is dependent upon the locale usage.
- "iscntrl" on page 58 tests for control characters.
- "isdigit" on page 58 tests for digit characters.
- "isgraph" on page 59 tests for graphical characters.
- "islower" on page 59 tests for lower case characters.
- "isprint" on page 60 tests for printable characters.
- "ispunct" on page 60 tests for punctuation characters.
- "isspace" on page 61 tests for white space characters.
- "isupper" on page 61 tests for upper case characters.
- "isxdigit" on page 62 tests for hexadecimal characters.
- "tolower" on page 62 changes from uppercase to lowercase.
- "toupper" on page 63 changes from lower case to uppercase.

Character Sets Supported

A Main Standard Library character tests the ASCII character set. Testing of extended character sets is undefined and may or may not work for any specific system. See Table 9.1 for return values.

Table 9.1 Character Testing Functions

This function	Returns true if c is
isalnum(c)	Alphanumeric: `[a-z]`, `[A-Z]`, `[0-9]`
`isalpha(c)`	Alphabetic: `[a-z]`, `[A-Z]`.
isblank(c)	A blankspace between words based on locale
`iscntrl(c)`	The delete character (`0x7F`) or an ordinary control character from `0x00` to `0x1F`.
isdigit(c)	A numeric character: `[0-9]`.
isgraph(c)	A non-space printing character from the exclamation (`0x21`) to the tilde (`0x7E`).
islower(c)	A lowercase letter: `[a-z]`.
isprint(c)	A printable character from space (`0x20`) to tilde (`0x7E`).
ispunct(c)	A punctuation character. A punctuation character is neither a control nor an alphanumeric character.
isspace(c)	A space, tab, return, new line, vertical tab, or form feed.
isupper(c)	An uppercase letter: `[A-Z]`.
isxdigit(c)	A hexadecimal digit `[0-9]`, `[A-F]`, or `[a-f]`.

isalnum

Determine character type.

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

Table 9.2 isalnum

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of c. For example usage, see Listing 9.1.

In the "C" locale, isalnum returns true only for alphabetical or numerical characters.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

See Also

"tolower" on page 62

"toupper" on page 63

Listing 9.1 Example of Character Testing Functions Usage

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char *test = "Fb6# 9,";

   isalnum(test[0]) ?
      printf("%c is alpha numberical\n", test[0]) :
      printf("%c is not alpha numberical\n", test[0]);
   isalpha(test[0]) ?
      printf("%c is alphabetical\n", test[0]) :
      printf("%c is not alphabetical\n", test[0]);
   isblank(test[4]) ?
      printf("%c is a blank sapce\n", test[4]) :
      printf("%c is not a blank space\n", test[4]);
   iscntrl(test[0]) ?
```

```
      printf("%c is a control character\n", test[0]) :
      printf("%c is not a control character\n", test[0]);
   isdigit(test[2]) ?
      printf("%c is a digit\n", test[2]) :
      printf("%c is not a digit\n", test[2]) ;
   isgraph(test[0]) ?
      printf("%c is graphical \n", test[0]) :
      printf("%c is not graphical\n", test[0]);
   islower(test[1]) ?
      printf("%c is lower case \n", test[1]) :
      printf("%c is not lower case\n", test[1]);
   isprint(test[3]) ?
      printf("%c is printable\n", test[3]) :
      printf("%c is not printable\n", test[3]);
   ispunct(test[6]) ?
      printf("%c is a punctuation mark\n", test[6]) :
      printf("%c is not punctuation mark\n", test[6]);
   isspace(test[4]) ?
      printf("%c is a space\n", test[4]) :
      printf("%c is not a space\n", test[4]);
   isupper(test[0]) ?
      printf("%c is upper case \n", test[1]) :
      printf("%c is not upper case\n", test[1]);
   isxdigit(test[5]) ?
      printf("%c is a hex digit\n", test[5]) :
      printf("%c is not a hex digit\n", test[5]);
   return 0;
}
```

```
Output:
F is alpha numberical
F is alphabetical
  is a blank sapce
F is not a control character
6 is a digit
F is graphical
b is lower case
# is printable
, is a punctuation mark
  is a space
b is upper case
9 is a hex digit
```

isalpha

Determine character type.

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

Table 9.3 isalpha

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isblank

Tests for a blank space or a word separator dependent upon the locale usage.

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

Table 9.4 isblank

c	int	character being evaluated

Remarks

This function determines if a character is a blank space or tab or if the character is in a locale specific set of characters for which `isspace` is true and is used to separate words in text.

In the “C” locale, `isblank` returns true only for the space and tab characters.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

See Also

"isspace" on page 61

iscntrl

Determine character type.

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

Table 9.5 iscntrl

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isdigit

Determine character type.

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

Table 9.6 isdigit

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isgraph

Determine character type.

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

Table 9.7 isgraph

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

islower

Determine character type.

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

Table 9.8 islower

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

In the `"C"` locale, `islower` returns true only for the lowercase characters.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isprint

Determine character type.

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

Table 9.9 isprint

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

ispunct

Determine character type.

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

Table 9.10 ispunct

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

In the “C” locale, `ispunct` returns true for every printing character for which neither `isspace` nor `isalnum` is `true`.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isspace

Determine character type.

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

Table 9.11 isspace

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

In the `"C"` locale, `isspace` returns `true` only for the standard white-space characters.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isupper

Determine character type.

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

Table 9.12 isupper

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

In the `"C"` locale, `isupper` returns `true` only for the uppercase characters.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

isxdigit

Determine hexidecimal type.

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

Table 9.13 isxdigit

c	int	character being evaluated

Remarks

This macro returns nonzero for true, zero for false, depending on the integer value of `c`. For example usage, see Listing 9.1.

Table 9.1 describes what the character testing functions return.

This function may not be implemented on all platforms.

tolower

Character conversion macro. For example usage see Listing 9.2.

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

Table 9.14 tolower

c	int	character being evaluated

`tolower()` returns the lowercase equivalent of uppercase letters and returns all other characters unchanged.

This function may not be implemented on all platforms.

See Also

"isalpha" on page 57

"toupper" on page 63.

Listing 9.2 Example of tolower(), toupper() Usage

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

int main(void)
{
   static char s[] =
            "** DELICIOUS! lovely? delightful **";
   int i;

   for (i = 0; s[i]; i++)
      putchar(tolower(s[i]));
   putchar('\n');

   for (i = 0; s[i]; i++)
      putchar(toupper(s[i]));
   putchar('\n');

   return 0;
}
```

```
Output:
** delicious! lovely? delightful **
** DELICIOUS! LOVELY? DELIGHTFUL **
```

toupper

Character conversion macro.

```
#include <ctype.h>

int toupper(int c);
```

Table 9.15 toupper

c	int	character being evaluated

Remarks

`The function toupper()` returns the uppercase equivalent of a lowercase letter and returns all other characters unchanged. For example usage, see Listing 9.1.

This function may not be implemented on all platforms.

See Also

"isalpha" on page 57

"tolower" on page 62

10

direct.h

The `direct.h` header file consist of various runtime declarations that pertain to the Win32 x86 targets for reading and manipulation of directories/folders.

Overview of direct.h

This header file defines the facilities as follows:

- “ getdcwd” on page 65 gets the current working directory.
- “ getdiskfree” on page 66 gets the amount of free disk space.
- “ getdrives” on page 66 gets drives available.

_getdcwd

Determines the current working directory information.

```
#include <direct.h>
char * _getdcwd(int drive, char *path, int len);
```

Table 10.1 _getdcwd

drive	int	The current drive as a number
path	char *	The current working directory path
len	int	The buffer size

Remarks

If the full path exceeds the buffers length unexpected results may occur.

The current working directory is returned if successful or NULL if failure occurs.

This function is Windows only when declared from this header.

See Also

“ chdrive” on page 81

_getdiskfree

Determines the free disk space.

```
#include <direct.h>
unsigned _getdiskfree(unsigned int drive, struct _diskfree_t
     * dfree);
```

Table 10.2 _getdiskfree

drive	unsigned int	The current drive as a number
dfree	`_diskfree_t *`	A structure that holds the disk information

Remarks

The structure _diskfree_t holds the disk attributes.

Zero is returned on success, true value is returned on failure.

This function is Windows only when declared from this header.

See Also

“ getdrive” on page 85

_getdrives

Determines the logical drive information.

```
#include <direct.h>
unsigned long _getdrives(void);
```

Remarks

There is no parameter for this function.

Returns the logical drives as a long integer value. Bit 0 is drive A, bit 1 is drive B, bit 2 is drive C and so forth.

This function is Windows only when declared from this header.

See Also

“ getdrive” on page 85

11

dirent.h

The header `dirent.h` defines several file directory functions for reading directories.

Overview of dirent.h

This header file defines the facilities as follows:

- "opendir" on page 69 opens a directory stream.
- "readdir" on page 70 reads a directory stream.
- "readdir_r" on page 70 reentrant read a directory stream.
- "rewinddir" on page 71 rewinds a directory stream.
- "closedir" on page 72 closes a directory stream.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

Table 1.1 for information on `POSIX` naming conventions.

opendir

This function opens the directory named as an argument and returns a stream pointer of type `DIR`.

```
#include <dirent.h>
DIR * opendir(const char *path);
```

Table 11.1 opendir

path	const char *	The path of the directory to be opened

Remarks

This function returns `NULL` if the directory can not be opened. If successful a directory stream pointer of type `DIR *` is returned.

This function may not be implemented on all platforms.

See Also

"closedir" on page 72

readdir

This function is used read the next directory entry.

```
#include <dirent.h>
struct dirent * readdir(DIR *dp);
```

Table 11.2 readdir

dp	DIR *	The stream being read

Remarks

The data pointed to by `readdir()` may be overwritten by another call to `readdir()`.

The function `readdir` returns the next directory entry from the stream `dp` as a pointer of struct `dirent`.

This function may not be implemented on all platforms.

See Also

"rewinddir" on page 71

readdir_r

This function is the reentrant version to read the next directory entry.

```
#include <dirent.h>
int readdir_r(DIR *ref, struct dirent *entry,
struct dirent ** result);
```

Table 11.3 readdir_r

dp	DIR *	The stream being read
entry	struct dirent *	Storage for the directory entry
result	struct dirent **	The next directory entry

Remarks

The `readdir_r()` function provides the same service as "readdir" on page 70. The difference is that `readdir()` would return a pointer to the next directory entry, and that pointer was internal to the library implementation. For `readdir_r()`, the caller provides the storage for the `dirent struct`.

On a successful call to `readdir_r()`, the function result is zero, the storage for entry is filled with the next directory entry, and the result pointer contains a pointer to entry. If the end of the directory is reached, the function result is zero and the result pointer is `NULL`. If any error occurs, the function result is an error code.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"rewinddir" on page 71

"readdir" on page 70

rewinddir

This function `rewinddir` resets the directory stream to the original position.

```
#include <dirent.h>
void rewinddir(DIR * dp);
```

Table 11.4 rewinddir

dp	DIR *	The stream being rewound

Remarks

There is no return.

This function may not be implemented on all platforms.

See Also

"readdir" on page 70

closedir

The function `closedir()` ends the directory reading process. It deallocates the directory stream pointer and frees it for future use.

```
#include <dirent.h>
int closedir(DIR * dp);
```

Table 11.5 closedir

dp	DIR *	The directory stream pointer to be closed

Remarks

The function `closedir()` returns zero on success and the value of `-1` on failure.

This function may not be implemented on all platforms.

See Also

"opendir" on page 69

12

div_t.h

The div.h.h header defines two structures used for math computations.

Overview of div_t.h

This header file defines the facilities as follows:

- "div_t" on page 73 stores remainder and quotient variables.
- "ldiv_t" on page 74 stores remainder and quotient variables.
- "lldiv_t" on page 74 stores remainder and quotient variables.

---

div_t

Stores the remainder and quotient from the `div` function.

```
#include <div_t.h>
typedef struct {
int quot;
int rem;
} div_t;
```

Remarks

This function may not be implemented on all platforms.

See Also

"div" on page 419

ldiv_t

Stores the remainder and quotient from the `ldiv` function.

```
#include <div_t.h>
typedef struct {
int   quot;
int   rem;
} ldiv_t;
```

Remarks

This function may not be implemented on all platforms.

See Also

"ldiv" on page 425

lldiv_t

Stores the remainder and quotient from the `lldiv` function.

```
#include <div_t.h>
typedef struct {
long long    quot;
long long    rem;
} lldiv_t;
```

Remarks

This function may not be implemented on all platforms.

See Also

"ldiv" on page 425

13

errno.h

The `errno.h` header file provides the global error code variable extern errno.

Overview of errno.h

There is one global declared in errno.h: "errno" on page 75

---

errno

The `errno.h` header file provides the global error code variable `errno`.

```
#include <errno.h>

extern int errno;
```

The math library used for Mac OS and Windows (when optimized) is not fully compliant with the 1990 ANSI C standard in that none of the math functions set errno. The MSL math libraries provide better means of error detection. Using fpclassify (which is C99 portable) provides a better error reporting mechanism. The setting of errno is considered an obsolete mechanism because it is inefficient as well as uninformative.

Most functions in the standard library return a special value when an error occurs. Often the programmer needs to know about the nature of the error. Some functions provide detailed error information by assigning a value to the global variable `errno`. The `errno` variable is declared in the `errno.h` header file. See Table 13.1

The `errno` variable is not cleared when a function call is successful; its value is changed only when a function that uses `errno` returns its own error value. It is the programmer's responsibility to assign 0 to `errno` before calling a function that uses it. For example:

Table 13.1 lists the error number macros defined in MSL. Not all these values are used in MSL but are defined in POSIX and other systems and are therefore defined in MSL to facilitate the compilation of codes being ported from other platforms.

This macro may not be implemented on all platforms.

Table 13.1 Error Number Definitions

Error Value	Description
E2BIG	Argument list too long
EACCES	Permission denied
EAGAIN	Resource temporarily unavailable
EBUSY	Device busy
ECHILD	No child processes
EDEADLK	Resource deadlock avoided
EDOM	Numerical argument out of domain
EEXIST	File already exists
EFAULT	Bad address
EFBIG	File too large
EFPOS	File Position Error
EILSEQ	Wide character encoding error
EINTR	Interrupted system call
EIO	Input/output error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENAMETOOLONG	File name too long
ENFILE	Too many open files in system
ENODEV	Operation not supported by device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOLCK	No locks available
ENOSPC	No space left on device

Table 13.1 Error Number Definitions (*continued*)

Error Value	Description
E2BIG	Argument list too long
ENOSYS	Function not implemented
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Inappropriate ioctl for device
ENXIO	Device not configured
ERANGE	Range error. The function cannot return a value
EPERM	Operation not permitted
EPIPE	Broken pipe
EROFS	Read-only file system
ESIGPARM	Signal error
ESPIPE	Illegal seek
ESRCH	No such process
EUNKNOWN	Unknown error
EXDEV	Cross-device link
Platform Assigned	
EBADF	Win32 assigned only, Bad file descriptor
EINVAL	Win32 assigned only, Invalid argument
ENOERR	Win32 assigned only, Bad file number
ENOMEM	Win32 assigned only, No error detected
EMACOSERR	Mac OS assigned only, the value is assigned to the global variable __MacOSErrNo

Listing 13.1 Example of errno Usage

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

int main(void)
{
   char *num = "5000000000";
   long result;
   result = strtol( num, 0, 10);

   if (errno == ERANGE)
      printf("Range error!\n");
   else
      printf("The string as a long is %ld", result);

   return 0;
}
```

```
Output:
Range error!
```

14

extras.h

The header extras.h defines several CodeWarrior provided non standard console functions.

Overview of extras.h

The extras.h header file consists of several functions which may be indirectly included by other headers or included for use directly through extras.h. This header file defines the facilities as follows:

- “strcmpi” on page 94 performs case-insensitive string compare.
- “strdate” on page 94 stores a date in a string.
- “strdup” on page 95 duplicates a string.
- “stricmp” on page 96 performs string comparison ignoring case.
- “stricoll” on page 96 performs case-insensitive locale collating string comparison.
- “strlwr” on page 97 converts string to lower case.
- “strncasecmp” on page 97 converts both strings to lower case before comparing them.
- “strncmpi” on page 98 performs case-insensitive string compare.
- “strnicmp” on page 100 performs string comparison ignoring case but specifying the comparison length.
- “strncoll” on page 99 performs length-limited locale collating string comparison.
- “strnicoll” on page 100 performs case-insensitive locale collating string comparison with a length limitation.
- “strnset” on page 101 sets a number of characters in a string.
- “strrev” on page 102 reverses characters in a string.
- “strset” on page 102 sets characters in a set.
- “strspnp” on page 103 finds a string in another string.
- “strupr” on page 104 converts string to upper case.
- “tell” on page 104 gets the file indicator position.
- “ultoa” on page 105 converts unsigned long to string.
- “ ultow” on page 106 converts unsigned long to wide character string.
- “wcsdup” on page 107 creates a wide character string duplicate.
- “wcsicoll” on page 108 performs case-insensitive string comparison collated by locale setting.
- “wcsicmp” on page 108 performs wide character string comparison ignoring case.
- “wcslwr” on page 109 converts wide character string to lower case.
- “wcsncoll” on page 110 performs length-limited locale collating wide string comparison.
- “wcsnicmp” on page 111 performs wide character string comparison ignoring case but specifying the comparison length.
- “wcsnicoll” on page 110 performs length-limited locale collating wide string case insensitive comparison.
- “wcsnset” on page 112 sets a number of characters in a wide character string.

- "wcsrev" on page 113 reverses a wide character string.
- "wcsset" on page 113 sets characters in a wide character string.
- "wcsspnp" on page 114 finds a wide character string in another wide character string.
- "wstrrev" on page 115 reverses wide character string.
- "wcsupr" on page 114 converts wide string to upper case.
- "wtoi" on page 115 converts wide string to integer.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4 for information on `POSIX` naming conventions.

_chdrive

Changes the current drive by number.

```
#include <extras.h>
_chdrive(int drive);
```

Table 14.1 _chdrive

drive	int	The drive as a number

Remarks

The drive is listed as a number, 1 for A, 2 for B, 3 for C, and so forth.

Zero is returned on success and negative one on failure.

This function may not be implemented on all platforms.

See Also

"_getdrive" on page 85

chsize

This function is used to change a files size.

```
#include <extras.h>
int chsize(int handle, long size);
int _chsize(int handle, long size);
```

Table 14.2 chsize

handle	int	The handle of the file being changed
size	long	The size to change

Remarks

If a file is truncated all data beyond the new end of file is lost.

This function returns zero on success and a negative one if a failure occurs.

See Also

"GetHandle" on page 86

filelength

Retrieves the file length based on a file handle.

```
#include <extras.h>
int filelength(int fileno);
int _filelength(int fileno);
```

Table 14.3 filelength

fileno	int	The file as a handle

The filelength as an int value is returned on success. A negative one is returned on failure.

This function may not be implemented on all platforms.

See Also

"GetHandle" on page 86

fileno

Obtains the file descriptor associated with a stream.

```
#include <extras.h>
int fileno(FILE *stream);
```

Table 14.4 fileno

stream	FILE *	A pointer to a FILE stream

Remarks

This function obtains the file descriptor for the stream. You can use the file descriptor with other functions in `unix.h`, such as `read()` and `write()`.

For the standard I/O streams `stdin`, `stdout`, and `stderr`, `fileno()` returns the values listed in Table 14.5.

Table 14.5 Fileno Return Value for Standard Streams

This function call…	Returns this file descriptor…
fileno(stdin)	0
fileno(stdout)	1
fileno(stderr)	2

If it is successful, `fileno()` returns a file descriptor. If it encounters an error, it returns `-1` and sets `errno`.

This function may not be implemented on all platforms.

See Also

"fdopen" on page 306

"open, wopen" on page 121

Figure 14.1 Example of fileno() usage.

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

int main(void)
{
   printf("The handle for the standard input device is: %d",
             fileno(stdin) );

   return 0;
}
```

```
Reult
The handle for the standard input device is: 0
```

_fullpath

Converts a relative path name to a full path name.

```
#include <extras.h>
char *_fullpath(char * absPath,
   const char * relPath, size_t maxLength);
```

Table 14.6 _fullpath

absPath	char *	The full absolute path
relPath	const char *	The relative path
maxLength	size_t	The maximum path length

Remarks

If the `maxLength` is `NULL` a path of up to `MAX_PATH` is used.

A pointer to the `absPath` is returned on success.On failure `NULL` is returned.

This function may not be implemented on all platforms.

gcvt

This function converts a floating point value to a null terminated char string.

```
#include <extras.h>
char *gcvt(double value, int digits, char *buffer);
char *_gcvt(double value, int digits, char *buffer);
```

Table 14.7 gcvt

value	double	The floating point value to be converted into a string
digits	int	The number of significant digits to converted
buffer	char *	The string to hold the converted floating point value

Remarks

The resultant string includes the decimal point and sign of the original value.

This function returns a pointer to the `buffer` argument.

This function may not be implemented on all platforms.

See Also

"atof" on page 410

_getdrive

Finds the current drive as a number.

```
#include <extras.h>
int _getdrive();
```

This facility has no parameters.

The current drive number is returned.

This function may not be implemented on all platforms.

See Also

“ chdrive” on page 81

“GetHandle” on page 86

GetHandle

GetHandle retrieves the current objects handle.

```
#include <extras.h>
int GetHandle();
```

This facility has no parameters.

The device handle.is returned on success. A negative one is returned on failure.

This function may not be implemented on all platforms.

See Also

“fileno” on page 83

_get_osfhandle

Retrieve an operating system file handle.

```
#include <extras.h>
long _get_osfhandle(int filehandle);
```

Table 14.8 _get_osfhandle

filehandle	int	An operating system file handle

An operating system file handle as opposed to C file handle is returned if successful otherwise sets `errno` and returns `NULL`.

This function may not be implemented on all platforms.

See Also

“GetHandle” on page 86

“ open_osfhandle” on page 90.

heapmin

This function releases the heap memory back to the system.

```
#include <extras.h>
int heapmin(void);
int _heapmin(void);
```

This facility has no parameters.

`Heapmin` returns zero if successful otherwise sets errno to `ENOSYS` and returns -1;

itoa

This function converts an int value to a null terminated char array.

```
#include <extras.h>
char * itoa(int val, char *str, int radix);
char * _itoa(int val, char *str, int radix);
```

Table 14.9 itoa

val	int	The integer value to convert
str	char *	The string to store the converted value
radix	int	The numeric base of the number to be converted

Remarks

The radix is the base of the number in a range of 2 to 36.

A pointer to the converted string is returned.

This function may not be implemented on all platforms.

See Also

"atoi" on page 411

"ltoa" on page 88

itow

This function converts an int value to a null terminated wide char array.

```
#include <extras.h>
wchar_t* itow(int val, wchar_t *str, int radix);
wchar_t* _itow(int val, wchar_t *str, int radix);
```

Table 14.10 itow

val	int	The integer value to convert
str	wchar_t *	The string to store the converted value
radix	int	The numeric base of the number to be converted

Remarks

The radix is the base of the number in a range of 2 to 36.

A pointer to the converted wide character string is returned.

This function may not be implemented on all platforms.

See Also

"itoa" on page 87

ltoa

This function converts a long int value to a null terminated char array.

```
#include <extras.h>
#define ltoa(x, y, z) _itoa(x, y, z);
#define _ltoa(x, y, z) _itoa(x, y, z);
```

Remarks

This function simply redefines _itoa for 32 bit systems.

A pointer to the converted wide character string is returned.

This function may not be implemented on all platforms.

See Also

"itoa" on page 87

"atol" on page 412

_ltow

This function converts an long value to a null terminated wide char array.

```
#include <extras.h>
wchar_t *_ltow(unsigned long val, wchar_t *str, int radix);
```

Table 14.11 _ltow

val	unsigned long	The long value to convert
val	wchar_t *	The wide character string to store the converted value
radix	int	The numeric base of the number to be converted

Remarks

The radix is the base of the number in a range of 2 to 36.

A pointer to the converted wide character string is returned.

This function may not be implemented on all platforms.

See Also

"itow" on page 88.

"ltoa" on page 88.

makepath

Makepath is used to create a path.

```
#include <extras.h>
void makepath(char *path, const char *drive,
   const char *dir, const char *fname, const char *ext);
void _makepath(char *path, const char *drive,
   const char *dir, const char *fname, const char *ext);
```

Table 14.12 makepath

path	char *	String to receive the created path
drive	const char *	String containing the drive component
dir	const char *	String containing the directory component
fname	const char *	String containing the file name component
ext	const char *	String containing the file extension component

There is no return value.

This function may not be implemented on all platforms.

See Also

“ chdrive” on page 81

_open_osfhandle

Opens an operating system handle.

```
#include <extras.h>
int _open_osfhandle(long ofshandle, int flags);
```

Table 14.13 _open_osfhandle

ofshandle	long	The file as an operating system handle
flags	int	file opening flags

Remarks

This function opens an operating system handle as a C file handle.

The file handle value is returned on success. A negative one is returned on failure.

This function may not be implemented on all platforms.

See Also

“ _get_osfhandle” on page 86.

putenv

Adds a string to the environmental variable.

```
#include <extras.h>
int putenv(const char * inVarName)
int _putenv(const char * inVarName)
```

Table 14.14 putenv

inVarName	char *	The string to add to the environmental variable

Remarks

May also be used to modify or delete an existing string. The string must be a global value.

The environment is restored at the program termination.

Zero is returned on success or negative one on failure.

This function may not be implemented on all platforms.

_searchenv

Searches the environmental path for a file.

```
#include <extras.h>
void _searchenv(const char *filename,
   const char *varname, char *pathname);
```

Table 14.15 _searchenv

filename	const char *	The file to search for
varname	const char *	The environmental variable name
pathname	char *	The path name of the file

Remarks

The current drive is searched first then a specified environmental variable path is searched.

There is no return value.

This function may not be implemented on all platforms.

splitpath

This function takes a path and returns pointers to each of the components of the path.

```
#include <extras.h>
void splitpath(const char *path, char *drive,
   char *dir, char *fname, char *ext)
void _splitpath(const char *path, char *drive,
   char *dir, char *fname, char *ext)
```

Table 14.16 splitpath

path	const char *	The file path to be split
drive	char *	The string to receive the drive component
dir	char *	The string to receive the directory component
fname	char *	The string to receive the filename component
ext	char *	The string to receive the file extension component

Remarks

The programmer must provide arrays large enough to hold the various components

This function may not be implemented on all platforms.

See Also

"chdir" on page 513

strcasecmp

A functioin for ignore case string comparison

```
#include <extras.h>
int strcasecmp(const char *str1, const char *str2);
```

Table 14.17 strcasecmp

str1	const char *	String being compared
str2	const char *	Comparison string

Remarks

The function converts both strings to lower case before comparing them.

Strcasecmp returns greater than zero if str1 is larger than str2 and less than zero if str2 is larger than str 1. If they are equal returns zero.

This function may not be implemented on all platforms.

See Also

"strncasecmp" on page 97

"stricmp" on page 96,

strcmpi

A case insensitive string compare.

```
#include <extras.h>
int strcmpi(const char *s1, const char *s2);
int _strcmpi(const char *s1, const char *s2);
```

Table 14.18 strcmpi

s1	const `char` *	The first string to compare
s2	const `char` *	The comparison string

An integer value of less than zero if the first argument is less than the second in a case insensitive string comparison. A positive value if the first argument is greater than the second argument in a case insensitive string comparison. Zero is returned if both case insensitive strings are the same.

This function may not be implemented on all platforms.

See Also

"strcasecmp" on page 93

"strnicmp" on page 100

strdate

The strdate function stores a date in a buffer provided.

```
#include <extras.h>
char * strdate(char *str);
char * _strdate(char *str);
```

Table 14.19 strdate

str	char *	A char string to store the date

The function returns a pointer to the `str` argument

Remarks

This function stores a date in the buffer in the string format of mm/dd/yy where the buffer must be at least 9 characters.

This function may not be implemented on all platforms.

See Also

"strftime" on page 499

strdup

Creates a duplicate string in memory.

```
#include <extras.h>
char * strdup(const char *str);
char * _strdup(const char *str);
```

Table 14.20 strdup

str	const char *	The string to be copied

A pointer to the storage location or NULL if unsuccessful.

This function may not be implemented on all platforms.

See Also

"memcpy" on page 457

stricmp

A function for string comparison ignoring case.

```
#include <extras.h>
int stricmp(const char *s1,const char *s2);
int _stricmp(const char *s1,const char *s2);
```

Table 14.21 stricmp

s1	const char *	The string being compared
s2	const char *	The comparison string

Stricmp returns greater than zero if str1 is larger than str2 and less than zero if str2 is larger than str 1. If they are equal returns zero.

This function may not be implemented on all platforms.

See Also

"strcmp" on page 461

"strncmp" on page 470

stricoll

A case insensitive locale collating string comparison.

```
#include <extras.h>
int stricoll(const char *s1, const char *s2);
int _stricoll(const char *s1, const char *s2);
```

Table 14.22 stricoll

s1	const char *	The string to compare
s2	const char *	A comparison string

Remarks

The comparison is done according to a collating sequence specified by the `LC_COLLATE` component of the current locale.

If the first string is less than the second a negative number is returned. If the first string is greater than the second then a positive number is returned. If both strings are equal then zero is returned.

This function may not be implemented on all platforms.

See Also

"strncoll" on page 99

"wcsicoll" on page 108

strlwr

This function converts a string to lowercase.

```
#include <extras.h>
char * strlwr(char *str);
char * _strlwr(char *str);
```

Table 14.23 strlwr

str	char	The string being converted

A pointer to the converted string is returned.

This function may not be implemented on all platforms.

See Also

"strupr" on page 104

"tolower" on page 62

strncasecmp

Ignore case string comparison function with length specified.

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

Table 14.24 strncasecmp

str1	const char *	String being compared
str2	const char *	Comparison string
n	unsigned int	Length of comparison

Remarks

The function converts both strings to lower case before comparing them.

Strncasecmp returns greater than zero if str1 is larger than str2 and less than zero if str2 is larger than str 1. If they are equal returns zero.

This function may not be implemented on all platforms.

See Also

"strcasecmp" on page 93

"strnicmp" on page 100.

strncmpi

A length limited case insensitive comparison of one string to another string.

```
#include <extras.h>
int strncmpi(const char *s1, const char *s2, size_t n);
int _strncmpi(const char *s1, const char *s2, size_t n);
```

Table 14.25 strncmpi

s1	const char *	A string to compare
s2	const char *	A comparison string
n	size_t	number of characters to compare

Remarks

Starting at the beginning of the string characters of the strings are compared until a difference is found or until `n` characters have been compared.

If the first argument is less than the second argument in a case insensitive comparison a negative integer is returned. If the first argument is greater than the second argument in a case insensitive comparison then a positive integer is returned. If they are equal then zero is returned.

This function may not be implemented on all platforms.

See Also

"stricmp" on page 96

"strncasecmp" on page 97

"wcsicmp" on page 108

strncoll

A length limited locale collating string comparison.

```
#include <extras.h>
int strncoll(const char *s1, const char *s2, size_t n);
int _strncoll(const char *s1, const char *s2, size_t sz);
```

Table 14.26 strncoll

s1	const char *	The string to compare
s2	const char *	A comparison string
sz	size_t	Number of characters to collate

Remarks

The comparison is done according to a collating sequence specified by the `LC_COLLATE` component of the current locale.

If the first string is less than the second a negative number is returned. If the first string is greater than the second then a positive number is returned. If both strings are equal then zero is returned.

This function may not be implemented on all platforms.

See Also

"strnicoll" on page 100

"wcsncoll" on page 110

strnicmp

A function for string comparison ignoring case but specifying the comparison length.

```
#include <extras.h>
iint strnicmp(const char *s1,const char *s2,int n);
nt _strnicmp(const char *s1,const char *s2,int n);
```

Table 14.27 strnicmp

s1	const char *	The string being compared
s2	const char *	The comparison string
n	int	Maximum comparison length

The function strnicmp returns greater than zero if s1 is larger than s2 and less than zero if s2 is larger than s1. If they are equal returns zero.

This function may not be implemented on all platforms.

See Also

"strcmp" on page 461

"strncmp" on page 470

strnicoll

A case insensitive locale collating string comparison with a length limitation.

```
#include <extras.h>
int strnicoll(const char *s1, const char *s2,size_t sz);
int _strnicoll(const char *s1, const char *s2, size_t sz);
```

Table 14.28 strnicoll

s1	const char *	The string to compare

Table 14.28 strnicoll (*continued*)

s1	const char *	A comparison string
sz	size_t	

Remarks

The comparison is done according to a collating sequence specified by the `LC_COLLATE` component of the current locale.

If the first string is less than the second a negative number is returned. If the first string is greater than the second then a positive number is returned. If both strings are equal then zero is returned.

See Also

"stricoll" on page 96

"wcsicoll" on page 108

strnset

This function sets the first n characters of string to a character.

```
#include <extras.h>
char * strnset(char *str, int c, size_t n)
char * _strnset(char *str, int c, size_t n)
```

Table 14.29 strnset

str	char *	The string to be modified
c	int	The char to be set
n	size_t	The number of characters of str to be set.to the value of c

Remarks

If the number of characters exceeds the length of the string all characters are set except for the terminating character.

A pointer to the altered string is returned.

This function may not be implemented on all platforms.

See Also

"strset" on page 102

strrev

This function reverses a string.

```
#include <extras.h>
char * _strrev(char *str);
char * _strrev(char *str);
```

Table 14.30 _strrev

str	char *	The string to be reversed

A pointer to the reversed string.

This function may not be implemented on all platforms.

See Also

"strcpy" on page 464

strset

This function sets characters of string to a character

```
#include <extras.h>
char * strset(char *str, int c)
char * _strset(char *str, int c)
```

Table 14.31 strset

str	char *	The string to be modified
c	int	The char to be set

A pointer to the altered string is returned.

This function may not be implemented on all platforms.

See Also

"strnset" on page 101

strspnp

This function returns pointer to first character in s1 that isn't in s2

```
#include <extras.h>
char * strspnp(char *s1, const char *s2)
char * _strspnp(char *s1, const char *s2)
```

Table 14.32 strspnp

s1	char *	The string being checked
s2	const char *	The search string as a char set.

Remarks

This function determines the position in the string being searched is not one of the chars in the string set.

A pointer to the first character in s1 that is not in `s2` or `NULL` is returned if all the characters `of s1` are in `s2.`

This function may not be implemented on all platforms.

See Also

"strcspn" on page 465

"strspn" on page 475

strupr

The function `strupr` converts a string to uppercase.

```
#include <extras.h>
char * strupr(char *str);
char * _strupr(char *str);
```

Table 14.33 strupr

str	char	The string being converted

A pointer to the converted string is returned.

This function may not be implemented on all platforms.

See Also

"toupper" on page 63

"strlwr" on page 97

tell

Returns the current offset for a file.

```
#include <extras.h>
long tell(int fildes);
```

Table 14.34 tell

fildes	int	The file descriptor

Remarks

This function returns the current offset for the file associated with the file descriptor fildes. The value is the number of bytes from the file's beginning.

If it is successful, `tell()` returns the offset. If it encounters an error, `tell()` returns `-1L`

This function may not be implemented on all platforms.

See Also

"ftell" on page 344

"lseek" on page 527

Listing 14.1 Example of tell() Usage

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

int main(void)
{
   int fd;
   long int pos;

   fd = open("mytest", O_RDWR | O_CREAT | O_TRUNC);
   write(fd, "Hello world!\n", 13);
   write(fd, "How are you doing?\n", 19);

   pos = tell(fd);

   printf("You're at position %ld.", pos);

   close(fd);

   return 0;
}
```

```
Result
This program prints the following to standard output:
You're at position 32.
```

ultoa

This function converts an unsigned long value to a null terminated char array.

```
#include <extras.h>
char * ultoa(unsigned long val,char *str, int radix);
char * _ultoa(unsigned long val, char *str, int radix);
```

Table 14.35 ultoa

val	unsigned long	The integer value to convert
str	char *	The string to store the converted value
radix	int	The numeric base of the number to be converted

Remarks

The radix is the base of the number in a range of 2 to 36. This function is the converse of `strtoul()` and uses the number representation described there.

A pointer to the converted string is returned.

This function may not be implemented on all platforms.

See Also

"ltoa" on page 88

"itoa" on page 87

_ultow

This function converts an unsigned long value to a null terminated wide character array.

```
#include <extras.h>
wchar_t *_ultow(unsigned long val, wchar_t *str, int radix);
```

Table 14.36 _ultow

val	unsigned long	The value to be converted
str	wchar_t *	A buffer large enough to hold the converted value
radix	int	The base of the number being converted

Remarks

The radix is the base of the number in a range of 2 to 36. This function is the wide character equivalent of `strtoul()` and uses the number representation described there.

A pointer to the converted wide character string is returned.

This function may not be implemented on all platforms.

See Also

"strtoul" on page 443

"itow" on page 88

"ltow" on page 89

wcsdup

Creates a duplicate wide character string in memory.

```
#include <extras.h>
wchar_t * wcsdup(const wchar_t *str)
wchar_t * _wcsdup(const wchar_t *str)
```

Table 14.37 wcsdup

str	const wchar_t *	The string to be copied

A pointer to the storage location is returned upon success or NULL is returned if unsuccessful.

This function may not be implemented on all platforms.

See Also

"strdup" on page 95

wcsicmp

A function for wide character string comparison ignoring case.

```
#include <extras.h>
int wcsicmp(const wchar_t *s1, const wchar_t *s2)
int _wcsicmp(const wchar_t *s1, const wchar_t *s2)
```

Table 14.38 wcsicmp

s1	const wchar_t *	The string being compared
s2	const wchar_t *	The comparison string

The function _wcsicmp returns greater than zero if str1 is larger than str2 and less than zero if str2 is larger than str 1. If they are equal returns zero.

This function may not be implemented on all platforms.

See Also

"strcmp" on page 461

"strncmp" on page 470

wcsicoll

A case insensitive locale collating wide character string comparison.

```
#include <extras.h>
int wcsicoll(const wchar_t *s1, const wchar_t *s2);
int _wcsicoll(const wchar_t *s1,const wchar_t *s2);
```

Table 14.39 wcsicoll

s1	const wchar_t *	The wide string to compare
s2	const wchar_t *	A comparison wide character string

Remarks

The comparison is done according to a collating sequence specified by the `LC_COLLATE` component of the current locale.

If the first string is less than the second a negative number is returned. If the first string is greater than the second then a positive number is returned. If both strings are equal then zero is returned.

This function may not be implemented on all platforms.

See Also

"wcsncoll" on page 110

"stricoll" on page 96

wcslwr

This function converts a string to lowercase.

```
#include <extras.h>
wchar_t *wcslwr(wchar_t *str);
wchar_t *_wcslwr(wchar_t *str);
```

Table 14.40 wcslwr

str	wchar_t	The string being converted

A pointer to the converted string is returned.

This function may not be implemented on all platforms.

See Also

"strupr" on page 104

"strlwr" on page 97

wcsncoll

A length limited locale collating wide string comparison.

```
#include <extras.h>
int wcsncoll(const wchar_t *s1, const wchar_t *s2, size_t sz);
int _wcsncoll(const wchar_t *s1, const wchar_t *s2, size_t
     sz);
```

Table 14.41 wcsncoll

s1	const wchar_t *	The wide string to compare
s2	const wchar_t *	A comparison wide character string
sz	size_t	The number of wide characters to compare

Remarks

The comparison is done according to a collating sequence specified by the `LC_COLLATE` component of the current locale.

If the first string is less than the second a negative number is returned. If the first string is greater than the second then a positive number is returned. If both strings are equal then zero is returned.

This function may not be implemented on all platforms.

See Also

"stricoll" on page 96

"wcsnicoll" on page 110

wcsnicoll

A length limited locale collating wide string case insensitive comparison.

```
#include <extras.h>
int wcsnicoll(const wchar_t *s1, const wchar_t *s2, size_t
```

```
     sz);
int _wcsnicoll(const wchar_t *s1, const wchar_t *s2, size_t
     sz);
```

Table 14.42 wcsnicoll

s1	const wchar_t *	The wide string to compare
s2	const wchar_t *	A comparison wide character string
sz		

Remarks

The comparison is done according to a collating sequence specified by the `LC_COLLATE` component of the current locale.

If the first string is less than the second a negative number is returned. If the first string is greater than the second then a positive number is returned. If both strings are equal then zero is returned.

This function may not be implemented on all platforms.

See Also

"wcsncoll" on page 110

"strnicoll" on page 100

wcsnicmp

A function for a wide character string comparison ignoring case but specifying the comparison length.

```
#include <extras.h>
iint wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t n);
iint _wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t
     n);
```

Table 14.43 wcsnicmp

s1	const `wchar_t` *	The string being compared

Table 14.43 wcsnicmp (*continued*)

s2	const `wchar_t` *	The comparison string
n	int	Maximum comparison length

The function `_wcsnicmp` returns greater than zero if str1 is larger than str2 and less than zero if str2 is larger than str 1. If they are equal returns zero.

This function may not be implemented on all platforms.

See Also

"stricmp" on page 96

"strncmp" on page 470

wcsnset

This function sets the first n characters of wide character string to a character.

```
#include <extras.h>
wchar_t *wcsnset(wchar_t *str, wchar_t wc, size_t n);
wchar_t *_wcsnset(wchar_t *str, wchar_t wc, size_t n);
```

Table 14.44 wcsnset

str	wchar_t *	The string to be modified
wc	wchar_t	The char to be set
n	size_t	The number of characters in the string to set.

Remarks

If the number of characters exceeds the length of the string all characters are set except for the terminating character.

A pointer to the altered string is returned.

This function may not be implemented on all platforms.

See Also

"strset" on page 102

wcsrev

This function reverses a wide character string.

```
#include <extras.h>
wchar_t * wcsrev(wchar_t *str);
wchar_t * _wcsrev(wchar_t *str);
```

Table 14.45 wcsrev

str	`wchar_t *`	The string to be reversed

A pointer to the reversed string is returned.

This function may not be implemented on all platforms.

See Also

"wstrrev" on page 115

wcsset

This function sets characters of a wide character string to a wide character.

```
#include <extras.h>
wchar_t * wcsset(wchar_t *str, wchar_t wc);
wchar_t *_wcsset(wchar_t *str, wchar_t wc);
```

Table 14.46 wcsset

str	`wchar_t *`	The string to be modified
c	wchar_t	The char to be set

A pointer to the altered string is returned.

See Also

"strnset" on page 101

wcsspnp

This function returns pointer to first character in s1 that isn't in s2

```
#include <extras.h>
wchar_t *wcsspnp(const wchar_t *s1, const wchar_t *s2);
wchar_t *_wcsspnp(const wchar_t *s1, const wchar_t *s2);
```

Table 14.47 wcsspnp

s1	const wchar_t *	The string being checked
s2	const wchar_t *	The search string as a char set.

Remarks

This function determines the position in the string being searched is not one of the chars in the string set.

A pointer to the first character in s1 that is not in s2 or NULL if all the characters of s1 are in s2.

This function may not be implemented on all platforms.

See Also

"strspnp" on page 103

"strspn" on page 475

wcsupr

The function `_wcsupr` converts a wide character string to uppercase.

```
#include <extras.h>
wchar_t * wcsupr(wchar_t *str);
wchar_t *_wcsupr(wchar_t *str);
```

Table 14.48 wcsupr

str	`wchar_t *`	The string being converted

A pointer to the converted string is returned.

This function may not be implemented on all platforms.

See Also

"strupr" on page 104

"strlwr" on page 97

wstrrev

This function reverses a wide character string.

```
#include <string.h>
wchar_t * _wstrrev(wchar_t * str);
wchar_t * _wstrrev(wchar_t * str);
```

Table 14.49 _wstrrev

str	`wchar_t *`	The string to be reversed

A pointer to the reversed string is returned.

This function may not be implemented on all platforms.

See Also

"strrev" on page 102

wtoi

This function converts a null terminated wide char array to an int value.

```
#include <extras.h>
int wtoi(const wchar_t *str);
int _wtoi(const wchar_t *str);
```

Table 14.50 wtoi

str	const wchar_t *	The string to be converted to an int.

Remarks

The `_wtoi()` function converts the character array pointed to by `str` to an integer value.

This function skips leading white space characters.

This function sets the global variable `errno` to `ERANGE` if the converted value cannot be expressed as a value of type int.

The function `_wtoi()` returns an the converted integer value.

This function may not be implemented on all platforms.

See Also

"atoi" on page 411

15

fcntl.h

The header file `fcntl.h` contains several file control functions that are useful for porting a program from UNIX.

Overview of fcntl.h

This header file defines the facilities as follows:

- "creat, wcreate" on page 118 creates a file.
- "fcntl" on page 119 manipulates a file descriptor.
- "open, wopen" on page 121 opens a file and returns its ID.

fcntl.h and UNIX Compatibility

The header file `fcntl.h.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4 for information on `POSIX` naming conventions.

creat, _wcreate

Create a new file or overwrite an existing file and a wide character variant.

```
#include <fcntl.h>
int creat(const char *filename, int mode);
int _creat(const char *filename, int mode);
int _wcreat(const wchar_t *wfilename, int mode);
```

Table 15.1 creat, _wcreate

filename	char *	The name of the file being created
wfilename	wchar_t *	The name of the file being created
mode	int	The open mode

Remarks

This function creates a file named `filename` you can write to. If the file does not exist, `creat()` creates it. If the file already exists, `creat()` overwrites it. The function ignores the argument mode.

This function call:

```
creat(path, mode);
```

is equivalent to this function call:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

If it's successful, `creat()` returns the file description for the created file. If it encounters an error, it returns `-1`.

This function may not be implemented on all platforms.

See Also

"fopen" on page 317

"fdopen" on page 306

"close" on page 515.

Listing 15.1 Example of creat() Usage

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

int main(void)
{
   int fd;

   fd = creat("Jeff:Documents:mytest", 0);
      /* Creates a new file named mytest in the folder
         Documents on the volume Akbar. */

   write(fd, "Hello world!\n", 13);
   close(fd);
   return 0;
}
```

fcntl

Manipulates a file descriptor.

```
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
int _fcntl(int fildes, int cmd, ...);
```

Table 15.2 fcntl

fildes	int	The file descriptor
cmd	int	A command to the file system
...		A variable argument list

Remarks

This function performs the command specified in `cmd` on the file descriptor `fildes`.

In the CodeWarrior ANSI library, `fcntl()` can perform only one command, `F_DUPFD`. This command returns a duplicate file descriptor for the file that `fildes` refers to. You must include a third argument in the function call. The new

file descriptor is the lowest available file descriptor that is greater than or equal to the third argument.

Table 15.3 Floating Point Characteristics

Mode	Description
F_DUPFD	Return a duplicate file descriptor.

If it is successful, `fcntl()` returns a file descriptor. If it encounters an error, `fcntl()` returns `-1`.

This function may not be implemented on all platforms.

See Also

"fileno" on page 83

"open, wopen" on page 121

"fdopen" on page 306.

Listing 15.2 Example of fcntl() Usage

```
#include <unix.h>

int main(void)
{
   int fd1, fd2;

   fd1 = open("mytest", O_WRONLY | O_CREAT);

   write(fd1, "Hello world!\n", 13);
      /* Write to the original file descriptor.          */

   fd2 = fcntl(fd1, F_DUPFD, 0);
      /* Create a duplicate file descriptor.             */

   write(fd2, "How are you doing?\n", 19);
      /* Write to the duplicate file descriptor.         */

   close(fd2);

   return 0;
}
```

```
Result After you run this program,
```

```
the file mytest contains the following:
Hello world!
How are you doing?
```

open, _wopen

Opens a file and returns its ID and a wide character variant.

```
#include <fcntl.h>
int open(const char *path, int oflag);
int _open(const char *path, int oflag);
```

Table 15.4 open

path	char *	The file path as a string
oflag	int	The open mode

Remarks

The function `open()` opens a file for system level input and output. and is used with the UNIX style functions `read`() and `write`().

Table 15.5 Floating Point Characteristics

Mode	Description
O_RDWR	Open the file for both read and write.
O_RDONLY	Open the file for read only.
O_WRONLY	Open the file for write only.
O_APPEND	Open the file at the end of file for appending.
O_CREAT	Create the file if it doesn't exist.
O_EXCL	Do not create the file if the file already exists.
O_TRUNC	Truncate the file after opening it.
O_NRESOLVE	Don't resolve any aliases.

Table 15.5 Floating Point Characteristics (*continued*)

Mode	Description
O_ALIAS	Open alias file (if the file is an alias).
O_RSRC	Open the resource fork.
O_BINARY	Open the file in binary mode (default is text mode).

open() returns the file id as an integer value.

This function may not be implemented on all platforms.

See Also

"close" on page 515

"lseek" on page 527

"read" on page 528

"write" on page 536

Listing 15.3 Example of open() Usage

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>


#define SIZE FILENAME_MAX
#define MAX 1024

char fname[SIZE] = "DonQ.txt";


int main(void)
{
   int fdes;
   char temp[MAX];
   char *Don = "In a certain corner of la Mancha, the name of\n\
which I do not choose to remember,...";
   char *Quixote = "there lived\none of those country\
 gentlemen, who adorn their\nhalls with rusty lance\
```

```
and worm-eaten targets.";

  /* NULL terminate temp array for printf */
  memset(temp, '\0', MAX);

  /* open a file */
  if((fdes = open(fname, O_RDWR | O_CREAT ))== -1)
  {
     perror("Error ");
     printf("Can not open %s", fname);
     exit( EXIT_FAILURE);
  }

  /* write to a file  */
  if( write(fdes, Don, strlen(Don)) == -1)
  {
     printf("%s Write Error\n", fname);
     exit( EXIT_FAILURE );
  }

  /* move back to over write ... characters */
  if( lseek( fdes, -3L, SEEK_CUR ) == -1L)
  {
     printf("Seek Error");
     exit( EXIT_FAILURE );
  }

  /* write to a file  */
  if( write(fdes, Quixote, strlen(Quixote)) == -1)
  {
     printf("Write Error");
     exit( EXIT_FAILURE );
  }

  /* move to beginning of file for read */
  if( lseek( fdes, 0L, SEEK_SET ) == -1L)
  {
     printf("Seek Error");
     exit( EXIT_FAILURE );
  }

  /* read the file  */
  if( read( fdes, temp, MAX ) == 0)
  {
     printf("Read Error");
     exit( EXIT_FAILURE);
  }
```

```
   /* close the file */
   if(close(fdes))
   {
      printf("File Closing Error");
      exit( EXIT_FAILURE );
   }

    puts(temp);

   return 0;
}
```

```
In a certain corner of la Mancha, the name of
which I do not choose to remember,there lived
one of those country gentlemen, who adorn their
halls with rusty lance and worm-eaten targets.
```

16

fenv.h

The <fenv.h> header file prototypes and defines C99 data types, macros and functions that allow interaction with the floating-point environment.

Overview of fenv.h

Using the data types, macros, and functions in fenv.h, programmers are able to test and change rounding direction as well as test, set, and clear exception flags. Both the rounding direction and exception flags can be saved and restored as a single entity.

Data Types

There are two data types defined in fenv.h:

- "fenv_t" is the floating point environment type.
- "fexcept_t" is the floating point exception type.

fenv_t

This type represents the entire floating-point environment.

fexcept_t

The type represents the floating-point exception flags collectively.

Macros

Each macro correlates to a unique bit position in the floating-point control register and a bitwise OR of any combination of macros results in distinct values. Macro values are platform dependent. there are three distinct macro types.

- "Floating-point exceptions" on page 127,

- "Rounding Directions" on page 126.
- "Environment" on page 127.

Floating-Point Exception Flags

Table 16.1 Floating-Point Exception Flags

Macro	Description
FE_DIVBYZERO	Divide by zero
FE_INEXACT	Inexact value
FE_INVALID	Invalid value
FE_OVERFLOW	Overflow value
FE_UNDERFLOW	Underflow value
FE_ALL_EXCEPT	The result of a bitwise OR of all the floating-point exception macros

Rounding Directions

Table 16.2 Rounding Directions

Macro	Description
FE_DOWNWARD	Rounded downwards
FE_TONEAREST	Rounded to nearest
FE_TOWARDZERO	Rounded to zero
FE_UPWARD	Rounded upwards

Environment

Table 16.3 Environment

Macro	Description
FE_DFL_ENV	is a pointer to the default floating-point environment defined at the start of program execution.

Pragmas

The header fenv.h requires one pragma "FENV_ACC" on page 127, which must be set in order for floating point flags to be tested.

FENV_ACC

`FENV_ACCESS` must be set to the on position in order for the floating-point flags to be tested. Whether this pragma is on or off by default is implementation dependent.

This pragma may not be implemented on all platforms.

```
#pragma STDC FENV_ACCESS on|off|default
```

Floating-point exceptions

The header `fenv.h` includes several floating point exception flags manipulators.

- "feclearexcept" on page 128,
- "fegetexceptflag" on page 129,
- "feraiseexcept" on page 130,
- "fesetexceptflag" on page 131,
- "fetestexcept" on page 132,

feclearexcept

The feclearexcept clears one or more floating-point exception flag indicated by its argument. The argument represents the floating-point exception flags to be cleared.

```
#include <fenv.h>
```

```
void feclearexcept(int excepts);
```

Table 16.4 feclearexcept

excepts	int	Determines which floating-point exception flags to clear

Remarks

The following example illustrates how programmers might want to “overlook” a floating-point exception. In this case two division operations are taking place. The first uses real numbers, the second imaginary. Suppose that even if the first operation fails we would still like to use the results from the second operation and act like no exceptions have occurred.

There is no return value.

This function may not be implemented on all platforms.

Listing 16.1 Example or feclearexcept Usage

```
void ie_feclearexcept(void)
{
   float complex1[2] = {0,1};
   float complex2[2] = {0,2};
   float result[2]   = {0,0};
   feclearexcept(FE_ALL_EXCEPT);

   /* CALCULATE */
result[0] = complex1[0] / complex2[0]; /* OOPs 0/0, sets the FE_INVALID
flag */
   result[1] = complex1[1] / complex2[1];   /* = some # */
   /* CHECK IF @ LEAST 1 OPERATION WAS SUCCESS */
   if ( (result[0] != 0) || (result[1] != 0) )
      feclearexcept(FE_INVALID);
      /* clear flag */
   else
      cout << "what ever\n";
```

```
   cout <<" Rest of code ... \n";
}
```

fegetexceptflag

The fegetexceptflag function stores a representation of the states of the floating-point exception flags in the object pointed to by the argument flag. Which exception flags to save is indicated by a second argument.

```
#include <fenv.h>
```

```
void fegetexceptflag(fexcept_t *flagp, int excepts);
```

Table 16.5 fegetexceptflag

flagp	fexcept_t	Pointer to floating-point exception flags
excepts	int	Determines which floating-point exception flags to save

Remarks

The purpose of this function is to save specific floating-point exception flags to memory. In the case of the example below the saved values determine the output of the program.

Which exception flags to save is indicated by the argument excepts.

There is no return value.

This function may not be implemented on all platforms.

Listing 16.2 Example of fegetexceptflag Usage

```
void ie_fegetexceptflag()
{
   int result = 0;
   fexcept_t flag;
   feclearexcept(FE_ALL_EXCEPT);

   /* SOME OPERATION TAKES PLACE */
   result = 1+1;
   /* NEED TO KNOW IF OPERATION WAS SUCCESSFUL */
   fegetexceptflag(&flag, FE_INVALID | FE_OVERFLOW);
```

```
   /* NOW CHECK OBJECT POINTED 2 BY flag */
   if (flag == FE_INVALID)
      cout << "The operation was invalid!\n";
   if (flag == FE_OVERFLOW)
      cout << "The operation overflowed!\n";
   if (flag == FE_INVALID | FE_OVERFLOW)
      cout << "A failure occured\n";
   else
      cout << "success!\n";
}
```

feraiseexcept

The feraiseexcept function raises the floating-point exceptions represented by its argument.

```
#include <fenv.h>

void feraiseexcept(int excepts);
```

Table 16.6 feraiseexcept

excepts	int	determines which floating-point exception flags to raise

Remarks

The difference between feraiseexcept and fesetexceptflag is what value the floating-point exception is raised to. feraiseexcept simply raises the exception, raise meaning setting to a logical one. On the other hand fesetexceptflag looks at the value of a stored floating-point exception flag and raises the current flag to the level of the stored value. If the stored exception flag is zero, the current exception flag is changed to a zero.

There is no return value.

This function may not be implemented on all platforms.

Listing 16.3 Example of feraiseexcept Usage

```
void ie_feraiseexcept(void)
{
   feclearexcept(FE_ALL_EXCEPT); /* all exception flags = 0 */
   /* RAISE SPECIFIC FP EXCEPTION FLAGS */
```

```
   feraiseexcept(FE_INVALID | FE_OVERFLOW);
}
```

fesetexceptflag

The fesetexceptflag function will set the floating-point exception flags indicated by the second argument to the states stored in the object pointed to in the first argument.

```
#include <fenv.h>

void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

Table 16.7 fesetexceptflag

flag	const fexcept_t *	Constant pointer to floating-point exception flags
excepts	int	Determines which floating-point exception flags to raise

Remarks

The example below illustrates how this function can be used to restore floating-point exception flags after a function call.

There is no return value.

This function may not be implemented on all platforms.

Listing 16.4 Example of fesetexceptflag Usage

```
void ie_fesetexceptflag(void)
{
   float result = 0;
   fexcept_t flag = 0;   /* fp exception flags saved here */
   feclearexcept(FE_ALL_EXCEPT);

   fegetexceptflag(&flag,FE_INVALID | FE_OVERFLOW); /* save some flags
*/
   /* SOME FUNCTION CALL */
   result = 0/0; /* OOPS caused an exception */
   for (int i = 0; i < 100; i++)
      cout << i << endl;
```

```
   /* NOW WE'RE BACK & WANT ORIGINAL VALUES OF FLAGS */
   fesetexceptflag(&flag, FE_INVALID | FE_OVERFLOW); /* restore flags
*/
}
```

fetestexcept

Use the fetestexceptflag function to find out if a floating-point exception has occurred. The argument determines which exceptions to check for.

```
#include <fenv.h>

int fetestexcept(int excepts);
```

Table 16.8 fetestexcept

excepts	int	Determines which floating-point exception flags to check

Remarks

The example below is similar to the fegetexceptflag example. This time though fetestexcept provides a direct interface to the exception flags so there is no need to save and then operate on the exception flag values.

Returns true if floating point exception has occurred.

This function may not be implemented on all platforms.

Listing 16.5 Example of fetestexcept Usage

```
void ie_fetestexcept(void)
{
   int result = 0;
   feclearexcept(FE_ALL_EXCEPT);

   /* SOME OPERATION TAKES PLACE */
   result = 1+1;
   /* NEED TO KNOW IF OPERATION WAS SUCCESSFUL */
   /* but instead of getting flags, check them directly */
   if (fetestexcept(FE_INVALID) == 1)
      cout << "The operation was invalid!\n";
   if (fetestexcept(FE_OVERFLOW) == 1)
      cout << "The operation overflowed!\n";
```

```
   if (fetestexcept(FE_INVALID | FE_OVERFLOW) == 1)
         cout << "A failure occured\n";
   else
      cout << "success!\n";
}
```

Rounding

The header `fenv.h` includes two functions for determining the rounding direction.

- "fegetround" on page 133.
- "fesetround" on page 134.

fegetround

The fegetround function returns the value of the rounding direction macro representing the current rounding direction.

```
#include <fenv.h>

int fegetround(void);
```

This facility has no parameters.

Remarks

The example that follows changes the rounding direction to compute an upper bound for the expression, then restores the previous rounding direction. This example illustrates the functionality of all the rounding functions.

The function fegetround returns the current rounding direction as an integer value.

This function may not be implemented on all platforms.

Listing 16.6 Example of fegetround Usage

```
void ie_fegetround(void)
{
int direction = 0;
double x_up = 0.0, a = 5.0, b = 2.0, c = 3.0,
   d = 6.0, f = 2.5, g = 0.5, ubound = 0;
   feclearexcept(FE_ALL_EXCEPT);

   /* CALCULATE DENOMINATOR */
   fesetround(FE_DOWNWARD);
```

```
   x_up = f + g;
      /* calculate denominator */
      /* CALCULATE EXPRESSION */
   direction = fegetround();
      /* save rounding direction */
   fesetround(FE_UPWARD);
      /* change rounding direction */
   ubound = (a * b + c * d) / x_up;
      /* result should = 9.3333 */
   fesetround(direction);
       /* return original state */
   cout << " (a * b + c * d) / (f + g) = " << ubound << endl;
}
```

fesetround

The fesetround function sets the rounding direction specified by its argument, round. If the value of round does not match any of the rounding macros, the function returns 0 and the rounding direction is not changed.

```
#include <fenv.h>

int fesetround(int round);
```

Table 16.9 fesetround

round	int	Determines the direction result are rounded.

Remarks

For a function to be reentrant the function must save the rounding direction in a local variable with fegetround( ) and restore with fesetround( ) upon exit.

Zero is returned if and only if the argument is not equal.

This function may not be implemented on all platforms.

For an example of fesetround, see Listing 16.6.

Environment

The header `fenv.h` includes several functions for determining the floating-point environment information.

fegetenv

The fegetenv stores the current floating-point environment in the object pointed to by the argument.

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

Table 16.10 fegetenv

envp	fenv_t *	Pointer to floating-point environment

Remarks

This function is used when a programmer wants to save the current floating-point environment, that is the state of all the floating-point exception flags and rounding direction. In the example that follows the stored environment is used to hide any floating-point exceptions raised during an interim calculation.

There is no return value.

This function may not be implemented on all platforms.

Listing 16.7 Example of fegetenv Usage

```
long double ie_fegetenv(void)
{
   float x=0, y=0;
   fenv_t env1 =0, env2 = 0;
   feclearexcept(FE_ALL_EXCEPT);

   fesetenv(FE_DFL_ENV); /* set 2 default */
   x = x +y; /* fp op, may raise exception */
   fegetenv(&env1);
   y = y * x;/* fp op, may raise exception */
   fegetenv(&env2);
}
```

feholdexcept

Saves the current floating-point environment and then clears all exception flags. This function does not affect the rounding direction and is the same as calling:

```
fegetenv(envp);
feclearexcept(FE_ALL_EXCEPT);
```

```
#include <fenv.h>
int feholdexcept(fenv_t *envp)
```

Table 16.11 feholdexcept

envp	fenv_t	Pointer to floating-point environment

There is no return value.

This function may not be implemented on all platforms.

Listing 16.8 Example of feholdexcept Usage

```
void ie_feholdexcept(void)
{   /* This function signals underflow if its result is
       denormalized, overerflow if its result is infinity,
       and inexact always, but hides spurious exceptions
       occurring from internal computations. */

   fenv_t local_env;
   int c;
      /* can be FP_NAN, FP_INFINITE , FP_ZERO,
            FP_NORMAL, or FP_SUBNORMAL */
   long double A=4, B=3, result=0;
   feclearexcept(FE_ALL_EXCEPT);

   feholdexcept(&local_env);
         /* save fp environment */
         /* INTERNAL COMPUTATION */
   result = pow(A,B);
         c = fpclassify(result);
         /* inquire about result */
      feclearexcept(FE_ALL_EXCEPT);
         /* hides spurious exceptions */
      feraiseexcept(FE_INEXACT);
         /* always raise */
```

```
        if (c==FP_INFINITE)
                    feraiseexcept(FE_OVERFLOW);
        else if (c==FP_SUBNORMAL)
                    feraiseexcept(FE_UNDERFLOW);
        /* RESTORE LOCAL ENV W/ CHNGS */
        feupdateenv(&local_env);
}
```

fesetenv

The fesetenv function establishes the floating-point environment represented by the object pointed to by envp. This object will have been set by a previous call to the functions fegetenv, feholdexcept or can use `FE_DEFAULT_ENV`.

```
#include <fenv.h>

void fesetenv(constfenv_t *envp);
```

Table 16.12 fesetenv

envp	const fenv_t	Pointer to floating-point environment

There is no return value.

This function may not be implemented on all platforms.

For an example of fesetenv, see Listing 16.7.

feupdateenv

This is a multi-step function that takes a saved floating-point environment and does the following:

1. Save the current floating-point environment into temporary storage. This value will be used as a mask to determine which signals to raise.
2. Restore the floating-point environment pointed to by the argument envp.
3. Raise signals in newly restored floating-point environment using values saved in step one.

```
#include <fenv.h>

void feupdateenv(const fenv_t *envp);
```

Table 16.13 feupdateenv

envp	const fenv_t	Constant pointer to floating-point environment

This function may not be implemented on all platforms.

There is no return value.

For an example of feupdateenv, see Listing 16.8.

17

float.h

The `float.h` header file macros specify the "Floating Point Number Characteristics" on page 139 for `float`, `double`, and `long double` types.

Overview of float.h

The `float.h` header file consists of macros that specify the characteristics of floating point number representation for `float`, `double`, and `long double` types.

Floating Point Number Characteristics

These macros are listed in Table 17.1. Macros beginning with `FLT` apply to the `float` type; `DBL`, the `double` type; and `LDBL`, the `long double` type.

The `FLT_RADIX` macro specifies the radix of exponent representation.

The `FLT_ROUNDS` specifies the rounding mode. CodeWarrior C rounds to nearest.

Table 17.1 Floating Point Number Characteristics

Macro	Description
FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG	The number of base `FLT_RADIX` digits in the significant.
FLT_DIG, DBL_DIG, LDBL_DIG	The decimal digit precision.
FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP	The smallest negative integer exponent that `FLT_RADIX` can be raised to and still be expressible.
FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP	The smallest negative integer exponent that 10 can be raised to and still be expressible.
FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP	The largest positive integer exponent that `FLT_RADIX` can be raised to and still be expressible.

Table 17.1 Floating Point Number Characteristics (*continued*)

Macro	Description
FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP	The largest positive integer exponent that 10 can be raised to and still be expressible.
FLT_MIN, DBL_MIN, LDBL_MIN	The smallest positive floating point value.
FLT_MAX, DBL_MAX, LDBL_MAX	The largest floating point value.
FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON	The smallest fraction expressible.

18

FSp_fopen.h

The FSp_fopen.h header defines functions to open a file based on the Macintosh file system.

Overview of FSp_fopen.h

This header file defines the facilities as follows:

- "FSp_fopen" on page 141 opens a Macintosh file for fopen.
- "FSRef_fopen" on page 142 opens a Macintosh file for fopen when files exist.
- "FSRefParentAndFilename_fopen" on page 143 opens Macintosh files for existing and non-existing files that can be created.

FSp_fopen

The function FSp_fopen opens a file with the Macintosh Toolbox FSpec function and returns a FILE pointer.

```
#include <Fsp_fopen.h>

FILE * FSp_fopen(ConstFSSpecPtr spec, const char *
     open_mode);
```

Table 18.1 FSp_fopen

spec	ConstFSSpecPtr	A toolbox file pointer
open_mode	char *	The open mode

Remarks

This function requires the programmer to include the associated FSp_fopen.c source file in their project. It is not included in the MSL C library.

The FSp_fopen facility opens a file with the Macintosh Toolbox `FSRefPtr` function and return a FILE pointer.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"fopen" on page 317

FSRef_fopen

Opens an existing file with FSRef and returns a FILE pointer.

```
#include <Fsp_fopen.h>
FILE * FSRef_fopen(FSRefPtr spec, const char * open_mode);
```

Table 18.2 FSRef_fopen

spec	FSRefPtr	An FSRef pointer for an existing file.
open_mode	char *	The open mode

Remarks

This function requires the programmer to include the associated FSp_fopen.c source file in their project as it is not included in the MSL C library. Also, there are three libraries that may need to be weak linked in a non-Carbon target in order to build when you use the new file system APIs.

- `TextCommon`
- `UnicodeConverter`
- `UTCUtils`

The `FSRefPtr` facility returns a `FILE` pointer

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"fopen" on page 317

"FSRefParentAndFilename_fopen" on page 143

FSRefParentAndFilename_fopen

FSRefParentAndFilename_fopen works on both files that already exist as well as nonexistent files that can be created by the call.

```
#include <Fsp_fopen.h>
FILE *  FSRefParentAndFilename_fopen(
    const FSRefPtr theParentRef,
    ConstHFSUniStr255Param theName,
    const char *open_mode);
```

Table 18.3 FSRefParentAndFilename_fopen

theParentRef	FSRefPtr	A Carbon file pointer to the existing parent FSRef
theName	ConstHFSUniStr255Param	The unicode name for the file to open
open_mode	char *	The open mode

Remarks

This function requires the programmer to include the associated FSp_fopen.c source file in their project as it is not included in the MSL C library. Also, there are three libraries that may need to be weak linked in a non-Carbon target in order to build when you use the new file system APIs.

- `TextCommon`
- `UnicodeConverter`
- `UTCUtils`

The `FSRefParentAndFilename_fopen` facility returns a `FILE` pointer

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"fopen" on page 317

"FSRef_fopen" on page 142

19

inttypes.h

The header `inttypes.h` defines integer type equivalent symbols and keywords.

Overview of inttypes.h

The `inttypes.h` header file consists of functions for manipulation of greatest-width integers and for converting numeric character string to greatest-width integers. It includes various types and macros to support these manipulations. It also includes formatting symbols for formatted input and output functions.

- "Greatest-Width Integer Types" on page 145, a type used for long long division
- "Greatest-Width Format Specifier Macros" on page 146, formatting specifiers for printf and scanf family functions.
- "Greatest-Width Integer Functions" on page 150, functions for manipulation and conversion

The header `inttype.h` includes several functions for greatest-width integer manipulation and conversions.

- "imaxabs" on page 150 computes the absolute value of a greatest-width integer.
- "imaxdiv" on page 150 computes the quotient and remainder.
- "strtoimax" on page 151 converts string to greatest-width integer.
- "strtoumax" on page 152 converts string to greatest-width unsigned integer.
- "wcstoimax" on page 153 converts wide character string to greatest-width integer.
- "wcstoumax" on page 155 converts wide character string to greatest-width unsigned integer.

Greatest-Width Integer Types

One type is defined for greatest-width integer types used for long long division manipulation.

imaxdiv_t

A structure type that can store the value returned by the imaxdiv function as described in Table 19.1.

Table 19.1 imaxdiv_t Structure Elements

quot	Defined for the appropriate int, long or long long type
rem	Defined for the appropriate int, long or long long type

Greatest-Width Format Specifier Macros

The `inttypes.h` header includes object-like macros that expand to a conversion specifier suitable for use with formatted input and output functions.

Table 19.2 shows output formatting macros

Table 19.3 shows input formatting macros.

Table 19.2 Fprintf Greatest-Width Format Specifiers

Macro Substitution	Specifier	Macro Substitution	Specifier	Macro Substitution	Specifier
PRId8	d	PRId16	hd	PRId32	ld
PRId64	lld	PRIdLEAST8	d	PRIdLEAST16	hd
PRIdLEAST32	ld	PRIdLEAST64	lld	PRIdFAST8	d
PRIdFAST16	hd	PRIdFAST32	ld	PRIdFAST64	lld
PRIdMAX	lld	PRIdPTR	lld	PRIi8	i
PRIi16	hi	PRIi32	li	PRIi64	lli
PRIiLEAST8	i	PRIiLEAST16	hi	PRIiLEAST32	li

Table 19.2 Fprintf Greatest-Width Format Specifiers (*continued*)

Macro Substitution	Specifier	Macro Substitution	Specifier	Macro Substitution	Specifier
PRIiLEAST64	lli	PRIiFAST8	i	PRIiFAST16	hi
PRIiFAST32	li	PRIiFAST64	lli	PRIiMAX	lli
PRIiPTR	li	PRIo8	o	PRIo16	ho
PRIo32	lo	PRIo64	llo	PRIoLEAST8	o
PRIoLEAST16	ho	PRIoLEAST32	lo	PRIoLEAST64	llo
PRIoFAST8	o	PRIoFAST16	ho	PRIoFAST32	lo
PRIoFAST64	llo	PRIoMAX	llo	PRIoPTR	lo
PRIu8	u	PRIu16	hu	PRIu32	lu
PRIu64	llu	PRIuLEAST8	u	PRIuLEAST16	hu
PRIuLEAST32	lu	PRIuLEAST64	llu	PRIuFAST8	u
PRIuFAST16	hu	PRIuFAST32	lu	PRIuFAST64	llu
PRIuMAX	llu	PRIuPTR	lu	PRIx8	x
PRIx16	hx	PRIx32	lx	PRIx64	llx
PRIxLEAST8	x	PRIxLEAST16	hx	PRIxLEAST32	lx
PRIxLEAST64	llx	PRIxFAST8	x	PRIxFAST16	hx
PRIxFAST32	lx	PRIxFAST64	llx	PRIxMAX	llx
PRIxPTR	lx	PRIX8	X	PRIX16	hX

Table 19.2 Fprintf Greatest-Width Format Specifiers (*continued*)

Macro Substitution	Specifier	Macro Substitution	Specifier	Macro Substitution	Specifier
PRIX32	lX	PRIX64	llX	PRIXLEAST8	X
PRIXLEAST16	hX	PRIXLEAST32	lX	PRIXLEAST64	llX
PRIXFAST8	X	PRIXFAST16	hX	PRIXFAST32	lX
PRIXFAST64	llX	PRIXMAX	llX	PRIXPTR	lX

NOTE Separate macros are used with input and output functions because different format specifiers are generally required for the `fprintf` and `fscanf` family of functions.

Table 19.3 Fscanf Greatest-Width Format Specifiers

Macro Substitution	Specifier	Macro Substitution	Specifier	Macro Substitution	Specifier
SCNd8	hhd	SCNd16	hd	SCNd32	ld
SCNd64	lld	SCNdLEAST8	hhd	SCNdLEAST16	hd
SCNdLEAST32	ld	SCNdLEAST64	lld	SCNdFAST8	hhd
SCNdFAST16	hd	SCNdFAST32	ld	SCNdFAST64	lld
SCNdMAX	lld	SCNdPTR	ld	SCNi8	hhi
SCNi16	hi	SCNi32	li	SCNi64	lli
SCNiLEAST8	hhi	SCNiLEAST16	hi	SCNiLEAST32	li

Table 19.3 Fscanf Greatest-Width Format Specifiers (*continued*)

Macro Substitution	Specifier	Macro Substitution	Specifier	Macro Substitution	Specifier
SCNiLEAST64	lli	SCNiFAST8	hhi	SCNiFAST16	hi
SCNiFAST32	li	SCNiFAST64	lli	SCNiMAX	lli
SCNiPTR	li	SCNo8	hho	SCNo16	ho
SCNo32	lo	SCNo64	llo	SCNoLEAST8	hho
SCNoLEAST16	ho	SCNoLEAST32	lo	SCNoLEAST64	llo
SCNoFAST8	hho	SCNoFAST16	ho	SCNoFAST32	lo
SCNoFAST64	llo	SCNoMAX	llo	SCNoPTR	lo
SCNu8	hhu	SCNu16	hu	SCNu32	lu
SCNu64	llu	SCNuLEAST8	hhu	SCNuLEAST16	hu
SCNuLEAST32	lu	SCNuLEAST64	llu	SCNuFAST8	hhu
SCNuFAST16	hu	SCNuFAST32	lu	SCNuFAST64	llu
SCNuMAX	llu	SCNuPTR	lu	SCNx8	hhx
SCNx16	hx	SCNx32	lx	SCNx64	llx
SCNxLEAST8	hhx	SCNxLEAST16	hx	SCNxLEAST32	lx
SCNxLEAST64	llx	SCNxFAST8	hhx	SCNxFAST16	hx
SCNxFAST32	lx	SCNxFAST64	llx	SCNxMAX	llx
SCNxPTR	lx				

Greatest-Width Integer Functions

The header `inttype.h` includes several functions for greatest-width integer manipulation and conversions.

imaxabs

Computes the absolute value of a greatest-width integer.

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
```

Table 19.4 imaxabs

j	intmax_t	The value being computed

Remarks

The behavior is undefined if the result can not be represented.

The imaxabs function returns the absolute value.

This function may not be implemented on all platforms.

See Also

"abs" on page 406

"labs" on page 424

imaxdiv

Compute the greatest-width integer quotient and remainder.

```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Table 19.5 imaxdiv

numer	intmax_t	The numerator
denom	intmax_t	The denominator

Remarks

The result is undefined behavior when either part of the result cannot be represented.

The imaxdiv function returns a structure of type "imaxdiv_t" on page 146 storing both the quotient and the remainder values.

This function may not be implemented on all platforms.

See Also

"div" on page 419

"ldiv" on page 425

strtoimax

Character array conversion to a greatest-width integral value.

```
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr,
    char ** restrict endptr, int base);
```

Table 19.6 strtoimax

nptr	const char *	A character array to convert
endptr	char **	A pointer to a position in `nptr` that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `strtoimax()` function converts a character array, pointed to by `nptr`, that is expected to represent an integer expressed with the radix `base` to an integer value of type `UINTMAX_MIN, or UINTMAX_MAX`. A plus or minus sign (+ or -) prefixing the number string is optional.

The `base` argument in `strtoimax()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then a `strtol family function`

converts the character array based on its format. Character arrays beginning with'0' are assumed to be octal, number strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr.` This position marks the first character that is not convertible to a long int value.

This function skips leading white space.

The converted value is return upon success. If a failure occurs zero is returned. If the value is outside of a representable range, the appropriate `INTMAX_MAX,` or `INTMAX_MIN` value is returned and `ERANGE` is stored in `errno.`

This function may not be implemented on all platforms.

See Also

"strtoumax" on page 152

"strtol" on page 438

"strtoll" on page 442

strtoumax

Character array conversion to a greatest-width integer value.

```
#include <inttypes.h>
uintmax_t strtoumax(const char * restrict nptr,
    char ** restrict endptr, int base);
```

Table 19.7 strtoumax

nptr	const char *	A character array to convert
endptr	char **	A pointer to a position in `nptr` that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `strtoumax()` function converts a character array, pointed to by `nptr`, to an integer value of type `UINTMAX_MIN,or UINTMAX_MAX`, in `base`. A plus or minus sign prefix is ignored.

The `base` argument in `strtoumax()` specifies the base used for conversion. It must have a value between 2 and 36, or 0.The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If `base` is 0, then a `strtoul family` function converts the character array based on its format. Character arrays beginning with `'0'` are assumed to be octal, number strings beginning with `'0x'` or `'0X'` are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to the functions' respective types.

This function skips leading white space.

The converted value is return upon success. If a failure occurs zero is returned. If the value is outside of a representable range, the appropriate `INTMAX_MIN, or UINTMAX_MAX` value is returned and `ERANGE is` stored in `errno.`

This function may not be implemented on all platforms.

See Also

"strtoimax" on page 151

"strtoul" on page 443

"strtoull" on page 444

wcstoimax

Wide character array conversion to a greatest-width integral value.

```
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
```

Table 19.8 wcstoimax

nptr	const wchar_t *	A wide character array to convert
endptr	wchar_t **	A pointer to a position in `nptr` that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `wcstoimax()` function converts a wide character array, pointed to by `nptr,` expected to represent an integer expressed in radix base to an integer value of type `INTMAX_MIN, or INTMAX_MAX`. A plus or minus sign (+ or -) prefixing the number string is optional.

The base argument in `wcstoimax()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then a `wcstoll family function` converts the wide character array based on its format. Numerical strings beginning with'0' are assumed to be octal, numerical strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other numerical strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr.` This position marks the first wide character that is not convertible to a long int value.

This function skips leading white space.

The converted value is return upon success. If a failure occurs zero is returned. If the value is outside of a representable range, the appropriate `INTMAX_MAX,` or `INTMAX_MIN` value is returned and `ERANGE is` stored in `errno.`

This function may not be implemented on all platforms.

See Also

"wcstoumax" on page 155

wcstoumax

Wide character array conversion to a greatest-width integer value.

```
#include <inttypes.h>
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t **
     restrict endptr, int base);
```

Table 19.9 wcstoumax

nptr	const wchar_t *	A wide character array to convert
endptr	wchar_t **	A pointer to a position in `nptr` that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `wcstoumax()` function converts a wide character array, pointed to by `nptr`, to an integer value of type `UINTMAX_MIN,or UINTMAX_MAX,` in base. A plus or minus sign prefix is ignored.

The `base` argument in `wcstoumax()` specifies the base used for conversion. It must have a value between 2 and 36, or 0.The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If `base` is 0, then and a `wcstoul` family function converts the wide character array based on its format. Numerical strings beginning with'0' are assumed to be octal, numerical strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other numerical strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to the functions' respective types.

This function skips leading white space.

The converted value is return upon success. If a failure occurs zero is returned. If the value is outside of a representable range, the appropriate `UINTMAX_MIN,or UINTMAX_MAX` value is returned and `ERANGE is` stored in `errno.`

This function may not be implemented on all platforms.

See Also

"wcstoimax" on page 153

20

io.h

The header io.h defines several Windows console functions.

Overview of io.h

This header file defines the facilities as follows:

- “ findclose” on page 158 closes a directory search.
- “ findfirst” on page 158 opens a directory search.
- “ findnext” on page 159 searches a directory.
- “ setmode” on page 160 sets the translation for unformatted input and output.

NOTE If you’re porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

Table 1.1 for information on `POSIX` naming conventions.

_finddata_t

The structure `_finddata_t` is defined to store directory information. This structure is used in the directory searching functions.

Listing 20.1 Example of _finddata_t Usage

```
struct _finddata_t {
 unsigned        attrib;
__std(time_t)    time_create;     /* -1 for FAT file systems */
__std(time_t)    time_access;     /* -1 for FAT file systems */
__std(time_t)    time_write;
_fsize_t         size;
char             name[260];
}
```

_findclose

Closes the handle opened by `_findfirst`.

```
#include <io.h>
int _findclose(long fhandle);
```

Table 20.1 _findclose

fhandle	long	The size in bytes of the allocation

Remarks

The fhandle is returned by `_findfirst`.

Zero is returned on success and a negative one on failure.

Windows compatible header yet may also be implemented in other headers.

See Also

" findfirst" on page 158

" findnext" on page 159

_findfirst

Searches a directory folder.

```
#include <io.h>
long _findfirst(const char *pathname,
   struct _finddata_t * fdata);
```

Table 20.2 _findfirst

pathname	const char *	The pathname of the file to be found
fdata	_finddata_t *	A pointer to a directory information structure

Remarks

Wildcards are allowed in the directory search.

A file handle is returned.

Windows compatible header yet may also be implemented in other headers.

See Also

"_finddata_t" on page 157

"_findnext" on page 159

"_findclose" on page 158

_findnext

Continues a directory search began with `_findfirst`.

```
#include <io.h>
int _findnext(long fhandle, struct _finddata_t * fdata);
```

Table 20.3 _findnext

fhandle	long	The handle returned from calling _findfirst
fdata	_finddata_t *	A pointer to a directory information structure

Remarks

Wildcards are allowed in the directory search.

Zero is returned if the file is found, if no file is found then a negative one is returned.

Windows compatible header yet may also be implemented in other headers.

See Also

"_finddata_t" on page 157

"_findfirst" on page 158

"_findclose" on page 158

_setmode

The _setmode function sets the translation mode of the file given by handle

```
#include <io.h>
int _setmode(int handle, int mode);
```

Table 20.4 _setmode

int	handle *	handle
int	mode *	The translation mode

Remarks

The mode must be one of two manifest constants, _O_TEXT or _O_BINARY.

The _O_TEXT mode sets text (a translated) mode. Carriage return-linefeed (CR-LF) combinations are translated into a single linefeed character on input. Linefeed characters are translated into CR-LF combinations on output.

The _O_BINARY mode sets binary (an untranslated) mode, in which linefeed translations are suppressed.

While _setmode is typically used to modify the default translation mode of stdin and stdout, you can use it on any file. If you apply _setmode to the file handle for a stream, call _setmode before performing any input or output operations on the stream.

The previous translation mode is returned or negative one on failure.

Windows compatible header yet may also be implemented in other headers.

21

iso646.h

The header `iso646.h` defines keyword alternates for the C operator symbols.

Overview of iso646.h

This header file consists of equivalent “words” for standard C operators as shown in Table 21.1.

Table 21.1 Operator Keyword Equivalents

Operator	Keyword Equivalent
&&	and
&=	and_eq
&	bitand
|	bitor
~	compl
!=	not_eq
||	or
|=	or_eq
^	xor
^=	xor_eq

22

limits.h

The `limits.h` header file macros describe the maximum and minimum integral type limits.

Overview of limits.h

This header file consists of macros listed in "Integral type limits" on page 163.

Integral type limits

The `limits.h` header file macros describe the maximum and minimum values of integral types. Table 22.1 describes the macros.

Table 22.1 Integral Type Limits

Macro	Description
CHAR_BIT	Number of bits of smallest object that is not a bit field.
CHAR_MAX	Maximum value for an object of type `char`.
CHAR_MIN	Minimum value for an object of type `char`.
SCHAR_MAX	Maximum value for an object of type `signed char`.
SCHAR_MIN	Minimum value for an object of type `signed char`.
UCHAR_MAX	Maximum value for an object of type `unsigned char`.
SHRT_MAX	Maximum value for an object of type `short int`.

Table 22.1 Integral Type Limits (*continued*)

Macro	Description
SHRT_MIN	Minimum value for an object of type `short int`.
USHRT_MAX	Maximum value for an object of type `unsigned short int`.
INT_MAX	Maximum value for an object of type `int`.
INT_MIN	Minimum value for an object of type `int`.
LONG_MAX	Maximum value for an object of type `long int`.
LONG_MIN	Minimum value for an object of type `long int`.
ULONG_MAX	Maximum value for an object of type `unsigned long int`
MB_LEN_MAX	Maximum number of bytes in a multibyte character
LLONG_MIN	minimum value for an object of type long long int
LLONG_MAX	Maximum value for an object of type long long int
ULLONG_MAX	Maximum value for an object of type unsigned long long int

23

locale.h

The `locale.h` header file provides facilities for handling different character sets and numeric and monetary formats.

Overview of locale.h

This header file defines the facilities as follows:

- "localeconv" gets the locale.
- "setlocale" sets the locale.

Locale Specification

The ANSI C Standard specifies that certain aspects of the C compiler are adaptable to different geographic locales. The `locale.h` header file provides facilities for handling different character sets and numeric and monetary formats. CodeWarrior C supports the `"C"` locale by default and a vendor implementation.

The `lconv` structure, defined in `locale.h`, specifies numeric and monetary formatting characteristics for converting numeric values to character strings. A call to `localeconv()` will return a pointer to an `lconv` structure containing the settings for the `"C"` locale Listing 23.1. An `lconv` member is assigned "CHAR_MAX" on page 163 value if it is not applicable to the current locale.

Listing 23.1 Example of lconv Structure and Contents Returned by localeconv() Usage

```
struct lconv {
   char   * decimal_point;
   char   * thousands_sep;
   char   * grouping;
   char   * int_curr_symbol;
   char   * currency_symbol;
   char   * mon_decimal_point;
   char   * mon_thousands_sep;
   char   * mon_grouping;
   char   * positive_sign;
   char   * negative_sign;
   char   int_frac_digits;
   char   frac_digits;
```

```
    char    p_cs_precedes;
    char    p_sep_by_space;
    char    n_cs_precedes;
    char    n_sep_by_space;
    char    p_sign_posn;
    char    n_sign_posn;
    char    *int_curr_symbol;
    char    int_p_cs_precedes;
    char    int_n_cs_precedes;
    char    int_p_sep_by_space;
    char    int_n_sep_by_space;
    char    int_p_sign_posn;
    char    int_n_sign_posn;
};
```

localeconv

Return the `lconv` settings for the current locale.

```
#include <locale.h>

struct lconv *localeconv(void);
```

Remarks

`localeconv()` returns a pointer to an `lconv` structure for the `"C"` locale. Refer to Figure 1.

This function may not be implemented on all platforms.

setlocale

Query or set locale information for the C compiler.

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

Table 23.1 setlocale

category	int	The part of the C compiler to query or set.
locale	char *	A pointer to the locale

Remarks

The `category` argument specifies the part of the C compiler to query or set.

The argument can have one of six values defined as macros in `locale.h`: `LC_ALL` for all aspects, `LC_COLLATE` for the collating function `strcoll()`, `LC_CTYPE` for `ctype.h` functions and the multibyte conversion functions in `stdlib.h`, `LC_MONETARY` for monetary formatting, `LC_NUMERIC` for numeric formatting, and `LC_TIME` for time and date formatting.

If the `locale` argument is a null pointer, a query is made. The `setlocale()` function returns a pointer to a character string indicating which locale the specified compiler part is set to. The CodeWarrior C compiler supports the `"C"` and `""` locale.

Attempting to set a part of the CodeWarrior C compiler's locale will have no effect.

This function may not be implemented on all platforms.

See Also

"strcoll" on page 462

24

malloc.h

The header `malloc.h` defines one function, `alloca`, which lets you allocate memory quickly on from the stack.

Overview of malloc.h

This header file consists of alloca, which allocates memory from the stack.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4 for information on `POSIX` naming conventions.

---

alloca

Allocates memory quickly on the stack.

```
#include <malloc.h>
void *alloca(size_t nbytes);
```

Table 24.1 alloca

nbytes	size_t	The size in bytes of the allocation

Remarks

This function returns a pointer to a block of memory that is `nbytes` long. The block is on the function's stack. This function works quickly since it decrements the current stack pointer. When your function exits, it automatically releases the storage.

If you use `alloca()` to allocate a lot of storage, be sure to increase the Stack Size for your project in the Project preferences panel.

If it is successful, `alloca()` returns a pointer to a block of memory. If it encounters an error, `alloca()` returns `NULL`.

This function may not be implemented on all platforms.

See Also

"calloc" on page 417

"free" on page 422

"malloc" on page 427

"realloc" on page 434

Non Standard <malloc.h> Functions

Various non standard functions are included in the header `malloc.h` for legacy source code and compatibility with operating system frameworks and application programming interfaces

For the function `heapmin` see "heapmin" on page 87 for a full description.

25

math.h

The `math.h` header file provides floating point mathematical and conversion functions.

Overview of math.h

This header file defines the facilities as follows:

Classification Macros

- "fpclassify" on page 175 classifies floating point numbers.
- "isfinite" on page 176 tests if a value is a finite number.
- "isnan" on page 177 tests if a value is a computable number.
- "isnormal" on page 177 tests for normal numbers.
- "signbit" on page 178 tests for a negative number.

Functions

- "acos" on page 178 determines the arccosine.
- "asin" on page 179 determines the arcsine.
- "atan" on page 180 determines the arctangent.
- "atan2" on page 181 determines the arctangent of two variables.
- "ceil" on page 183 determines the smallest int not less than x.
- "cos" on page 184 determines the cosine.
- "cosh" on page 186 determines the hyperbolic cosine.
- "exp" on page 187 computes the exponential.
- "fabs" on page 188 determines the absolute value.
- "floor" on page 190 determines the largest integer not greater than x.
- "fmod" on page 191 determines the remainder of a division.
- "frexp" on page 193 extracts a value of the mantissa and exponent.
- "ldexp" on page 197 computes a value from a mantissa and exponent.
- "log" on page 198 determines the natural logarithm.
- "log10" on page 200 determines the logarithm to base 10.
- "modf" on page 201 separates integer and fractional parts.

- "pow" on page 202 raises to a power.
- "sin" on page 204 determines the sine.
- "sinh" on page 205 determines the hyperbolic sine.
- "sqrt" on page 207 determines the square root.
- "tan" on page 208 determines the tangent.
- "tanh" on page 209 determines the hyperbolic tangent.

C9X Implementations

- "acosh" on page 211 computes the (non-negative) arc hyperbolic cosine.
- "asinh" on page 212 computes the arc hyperbolic sine.
- "atanh" on page 213 computes the arc hyperbolic tangent.
- "cbrt" on page 214 computes the real cube root.
- "copysign" on page 215 gives a value with the magnitude of x and the sign of y.
- "erf" on page 216 computes the error function.
- "erfc" on page 217 complementary error function.
- "exp2" on page 217 computes the base-2 exponential.
- "expm1" on page 218 computes the exponential minus 1.
- "fdim" on page 219 computes the positive difference of its arguments.
- "fma" on page 220 a ternary operation to multiply and add.
- "fmax" on page 221 computes the maximum numeric value of its argument.
- "fmin" on page 222 computes the minimum numeric value of its arguments.
- "gamma" on page 223 computes the gamma function.
- "hypot" on page 224 computes the square root of the sum of the squares of the arguments.
- "isgreater" on page 194 compares two numbers for x greater than y.
- "isgreaterless" on page 195 compares numbers for x not equal to y.
- "isless" on page 195 compares two numbers for x less than y.
- "islessequal" on page 196 compares two numbers for x is less than or equal to y.
- "isunordered" on page 196 compares two numbers for lack of order.
- "ilogb" on page 225 determines the exponent.
- "lgamma" on page 226 computes the log of the absolute value.
- "log1p" on page 227 computes the natural- log of x plus 1.
- "log2" on page 228 computes the base-2 logarithm.
- "logb" on page 229 extracts the exponent of a double value.

- "nan" on page 230 stores floating point information.
- "nan" on page 230 Tests for NaN.
- "nearbyint" on page 230 rounds off the argument to an integral value.
- "nextafter" on page 231 determines the next representable value in the type of the function.
- "remainder" on page 232 computes the remainder x REM y required by IEC 559.
- "remquo" on page 233 computes the same remainder as the remainder function.
- "rint" on page 234 rounds off the argument to an integral value.
- "rinttol" on page 235 rounds its argument to the nearest long integral value.
- "round" on page 236 rounds its argument to an integral value in floating-point format.
- "roundtol" on page 237 rounds its argument to the nearest integral value.
- "scalb" on page 237 computes x * FLT_RADIX^n.
- "trunc" on page 238 rounds its argument to an integral value in floating-point format nearest to but no larger than the argument.

Floating Point Mathematics

The `HUGE_VAL` macro, defined in `math.h`, is returned as an error value by the `strtod()` function. See "strtold" on page 441 for information on `strtod()`.

Un-optimized x86 `math.h` functions may use the "errno" on page 75 global variable to indicate an error condition. In particular, many functions set `errno` to `EDOM` when an argument is beyond a legal domain. See Table 13.1.

NaN Not a Number

`NaN` stands for 'Not a Number' meaning that it has no relationship with any other number. A `NaN` is neither greater, less, or equal to a number. Whereas infinity is comparable to a number that is, it is greater than all numbers and negative infinity is less than all numbers.

There are two types of `NaN` the `signalling NaN` and `quiet NaN`. The difference between a `signalling NaN` and a `quiet NaN` is that both have a full exponent and both have at least one non-zero significant bit, but the `signalling NaN` has its 2 most significant bits as 1 whereas a `quiet NaN` has only the second most significant bit as 1.

Quiet NaN

`A quiet NaN` is the result of an indeterminate calculation such as zero divided by zero, infinity minus infinity. The IEEE floating-point standard guarantees that a quiet `NaN` is detectable by requiring that the invalid exception be raised whenever an `NaN` appears as an operand to any basic arithmetic(+,/,-,*) or non-arithmetic operation (load/store). The Main Standard Library for C follows the IEEE specification.

Signaling NaN

A `signalling NaN` does not occur as a result of arithmetic. A signalling `NaN` occurs when you load a bad memory value into a floating-point register that happens to have the same bit pattern as a `signalling NaN`. IEEE 754 requires that in such a situation the invalid exception be raised and the `signalling NaN` be converted to a `quiet NaN` so the lifetime of a signalling `NaN` may be brief.

Floating point error testing.

The math library used for PowerPC Mac OS and Windows (when optimized) is not fully compliant with the 1990 ANSI C standard. One way it deviates is that none of the math functions set errno.

The setting of errno is considered an obsolete mechanism because it is inefficient as well as un-informative. Further more various math facilities may set errno haphazardly for 68k Mac OS.

The MSL math libraries provide better means of error detection. Using fpclassify (which is fully portable) provides a better error reporting mechanism. Listing 25.1 shows an example code used for error detection that allows you to recover in your algorithm based on the value returned from fpclassify.

Inlined Intrinsics Option

For the Win32 x86 compilers CodeWarrior has an optimization option, “inline intrinsics”. If this option is on the math functions do not set the global variable errno. The debug version of the ANSI C libraries built by CodeWarrior has “inline intrinsics” option off and errno is set. The optimized release version of the library has “inline intrinsics” option on, and errno is not set.

Floating Point Classification Macros

Several facilities are available for floating point error classification.

Enumerated Constants

The Main Standard Library for C includes the following constant types for Floating point evaluation.

`FP_NAN` represents a quiet `NaN`

`FP_INFINITE` represents a positive or negative infinity

`FP_ZERO` represents a positive or negative zero

`FP_NORMAL` represents all normal numbers

`FP_SUBNORMAL` represents denormal numbers

Remarks

This function may not be implemented on all platforms.

See Also

"NaN Not a Number" on page 173

fpclassify

Classifies floating point numbers.

```
#include <math.h>
int __fpclassify(long double x);
int __fpclassifyd(double x);
int __fpclassifyf(float x);
```

Table 25.1 fpclassify

x	float, double or long double	number evaluated

Remarks

An integral value `FP_NAN, FP_INFINITE, FP_ZERO, FP_NORMAL and FP_SUBNORMAL.`

This function may not be implemented on all platforms.

See Also

"isfinite" on page 176
"isnan" on page 177
"isnormal" on page 177
"signbit" on page 178
"NaN Not a Number" on page 173

Listing 25.1 Example of Error Detection Usage

```
switch(fpclassify(pow(x,y))
{
case FP_NAN: // we know y is not an int and <0
case FP_INFINITY: // we know y is an int <0
case FP_NORMAL: // given x=0 we know y=0
case FP_ZERO:// given x<0 we know y >0
}
```

isfinite

The facility isfinite tests if a value is a finite number.

```
#include <math.h>
int isfinite(double x);
```

Table 25.2 isfinite

x	float, double or long double	number evaluated

Remarks

The facility returns true if the value tested is finite otherwise it returns false.

This function may not be implemented on all platforms.

See Also

"fpclassify" on page 175

isnan

The facility `isnan` test if a value is a computable number.

```
#include <math.h>
int isnan(double x);
```

Table 25.3 isnan

x	float, double or long double	number evaluated

Remarks

This facility is true if the argument is not a number.

This function may not be implemented on all platforms.

See Also

"fpclassify" on page 175

"NaN Not a Number" on page 173

isnormal

A test of a normal number.

```
#include <math.h>
int isnormal(double x);
```

Table 25.4 isnormal

x	float, double or long double	number evaluated

Remarks

This facility is true if the argument is a normal number.

This function may not be implemented on all platforms.

See Also

"fpclassify" on page 175

signbit

A test for a number that includes a signed bit

```
#include <math.h>
int __signbit(long double x);
int __signbitd(double x);
int __signbit(float x);
```

Table 25.5 signbit

x	float, double or long double	number evaluated

Remarks

This facility is true if the sign of the argument value is negative.

This function may not be implemented on all platforms.

See Also

"fpclassify" on page 175

Floating Point Math Facilities

Several facilities are available for floating point manipulations.

acos

This function computes the arc values of cosine, sine, and tangent.

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

Table 25.6 acos

x	float, double or long double	value to be computed

Remarks

The function `acos()` may set `errno` to `EDOM` if the argument is not in the range of -1 to +1. See "Floating point error testing." on page 174, for information on newer error testing procedures. For example usage, see Listing 25.2.

The function `acos()` returns the arccosine of the argument `x` in radians. If the argument to `acos()` is not in the range of -1 to +1, the global variable `errno` may be set to `EDOM` and returns 0. See "Floating point error testing." on page 174, for information on newer error testing procedures.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"cos" on page 184

"errno" on page 75

acosf

Implements the acos() function for float type values. See "acos" on page 178.

acosl

Implements the acos() function for long double type values. See "acos" on page 178.

asin

Arcsine function.

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

Table 25.7 asin

x	float, double or long double	value to be computed

Remarks

This function computes the arc values of sine. For example usage, see Listing 25.2.

The function `asin()` may set `errno` to `EDOM` if the argument is not in the range of -1 to +1. See "Floating point error testing." on page 174, for information on newer error testing procedures.

The function `asin()` returns the arcsine of `x` in radians. If the argument to `asin()` is not in the range of -1 to +1, the global variable `errno` may be set to `EDOM` and returns 0. See "Floating point error testing." on page 174, for information on newer error testing procedures.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"sin" on page 204

"errno" on page 75

asinf

Implements the asin() function for float type values. See "asin" on page 179.

asinl

Implements the asin() function for long double type values. See "asin" on page 179.

atan

Arctangent function. This function computes the value of the arc tangent of the argument.

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

Table 25.8 atan

x	float, double or long double	value to be computed

Remarks

The function `atan()` returns the arc tangent of the argument `x` in the range `[-¼/2, +¼/2]` radians. For example usage, see Listing 25.2.

This function may not be implemented on all platforms.

See Also

"tan" on page 208

"errno" on page 75

atanf

Implements the atan() function for float type values. See "atan" on page 180.

atanl

Implements the atan() function for long double type values. See "atan" on page 180.

atan2

Arctangent function. This function computes the value of the tangent of x/y using the sines of both arguments.

```
#include <math.h>
double atan2(double y, double x);
float atan2f(float, float);
long double atan2l(long double, long double);
```

Table 25.9 atan2

y	double, float or long double	Value one
x	double, float or long double	Value two

Remarks

A domain error occurs if both x and y are zero. For example usage, see Listing 25.2.

The function `atan2()` returns the arc tangent of `y/x` in the range `[-¼, ++` radians.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"tan" on page 208

"errno" on page 75

Listing 25.2 Example of acos(), asin(), atan(), atan2() Usage

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

int main(void)
{
   double x = 0.5, y = -1.0;

   printf("arccos (%f) = %f\n", x, acos(x));
   printf("arcsin (%f) = %f\n", x, asin(x));
   printf("arctan (%f) = %f\n", x, atan(x));
   printf("arctan (%f / %f) = %f\n", y, x, atan2(y, x));

   return 0;
}
```

```
Output:
arccos (0.500000) = 1.047198
arcsin (0.500000) = 0.523599
arctan (0.500000) = 0.463648
arctan (-1.000000 / 0.500000) = -1.107149
```

atan2f

Implements the atan2() function for float type values. See "atan2" on page 181.

atan2l

Implements the atan2() function for long double type values. See "atan2" on page 181.

ceil

Compute the smallest floating point number not less than *x*.

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

Table 25.10 ceil

x	float, double or long double	value to be computed

Remarks

`ceil()` returns the smallest integer not less than `x`.

This function may not be implemented on all platforms.

See Also

"floor" on page 190

"fmod" on page 191

"round" on page 236

Listing 25.3 Example of ceil() Usage

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

```
int main(void)
{
   double x = 100.001, y = 9.99;

   printf("The ceiling of %f is %f.\n", x, ceil(x));
   printf("The ceiling of %f is %f.\n", y, ceil(y));

   return 0;
}
```

```
Output:
The ceiling of 100.001000 is 101.000000.
The ceiling of 9.990000 is 10.000000.
```

ceilf

Implements the ceil() function for float type values. See "ceil" on page 183.

ceill

Implements the ceil() function for long double type values. See "ceil" on page 183.

cos

Compute cosine.

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

Table 25.11 cos

x	float, double or long double	value to be computed

Remarks

`cos()` returns the cosine of `x`. `x` is measured in radians.

This function may not be implemented on all platforms.

See Also

"sin" on page 204

"tan" on page 208

Listing 25.4 Example of cos() Usage

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

int main(void)
{
   double x = 0.0;
   printf("The cosine of %f is %f.\n", x, cos(x));

   return 0;
}
```

```
Output:
The cosine of 0.000000 is 1.000000.
```

cosf

Implements the cos() function for float type values. See "cos" on page 184.

cosl

Implements the cos() function for long double type values. See "cos" on page 184.

cosh

Compute the hyperbolic cosine.

```
double cosh(double x);
float coshf(float);
long double coshl(long double);
```

Table 25.12 cosh

x	float, double or long double	value to be computed

Remarks

`cosh()` returns the hyperbolic cosine of `x`.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"sinh" on page 205

"tanh" on page 209

Listing 25.5 Example of cosh() Usage

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

int main(void)
{
   double x = 0.0;

   printf("Hyperbolic cosine of %f is %f.\n",x,cosh(x));

   return 0;
}
```

```
Output:
Hyperbolic cosine of 0.000000 is 1.000000.
```

coshf

Implements the cosh() function for float type values. See "cosh" on page 186.

coshl

Implements the cosh() function for long double type values. See "cosh" on page 186.

exp

Computes the exponent of the function's argument

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

Table 25.13 exp

x	float, double or long double	value to be computed

Remarks

A range error may occur for larger numbers.

`exp()` returns e^x, where e is the natural logarithm base value.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"log" on page 198

"expm1" on page 218

"exp2" on page 217

"pow" on page 202

Listing 25.6 Example of exp() Usage

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

int main(void)
{
   double x = 4.0;
   printf("The natural logarithm base e raised to the\n");
   printf("power of %f is %f.\n", x, exp(x));

   return 0;
}
```

```
Output:
The natural logarithm base e raised to the
power of 4.000000 is 54.598150.
```

expf

Implements the exp() function for float type values. See "exp" on page 187.

expl

Implements the exp() function for long double type values. See "exp" on page 187.

fabs

Compute the floating point absolute value.

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

Table 25.14 fabs

x	float, double or long double	value to be computed

Remarks

`fabs()` returns the absolute value of `x`.

This function may not be implemented on all platforms.

See Also

"floor" on page 190

"ceil" on page 183

"fmod" on page 191

Listing 25.7 Example of fabs() Usage

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

int main(void)
{
   double s = -5.0, t = 5.0;
   printf("Absolute value of %f is %f.\n", s, fabs(s));
   printf("Absolute value of %f is %f.\n", t, fabs(t));

   return 0;
}
```

```
Output:
Absolute value of -5.000000 is 5.000000.
Absolute value of 5.000000 is 5.000000.
```

fabsf

Implements the fabs() function for float type values. See "fabs" on page 188.

fabsl

Implements the fabs() function for long double type values. See "fabs" on page 188.

floor

Compute the largest floating point not greater than *x*.

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

Table 25.15 floor

x	float, double or long double	value to be computed

Remarks

The function `floor()` returns the largest integer not greater than `x`.

This function may not be implemented on all platforms.

See Also

"ceil" on page 183

"fmod" on page 191

"fabs" on page 188

Listing 25.8 Example of floor() Usage

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

int main(void)
{
   double x = 12.03, y = 10.999;

   printf("Floor value of %f is %f.\n", x, floor(x));
   printf("Floor value of %f is %f.\n", y, floor(y));
```

```
   return 0;
}
```

```
Output:
Floor value of 12.030000 is 12.000000.
Floor value of 10.999000 is 10.000000.
```

floorf

Implements the floor() function for float type values. See "floor" on page 190.

floorl

Implements the floor() function for long double type values. See "floor" on page 190.

fmod

Return the floating point remainder of *x* / *y*.

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

Table 25.16 fmod

x	double, float or long double	The value to compute
y	double, float or long double	The divider

Remarks

`fmod()` returns, when possible, the value *f* such that `x` = *i* `y` + *f* for some integer *i*, and |*f*| < |`y`|. The sign of *f* matches the sign of `x`.

This function may not be implemented on all platforms.

See Also

"floor" on page 190

"ceil" on page 183

"fmod" on page 191

"fabs" on page 188

Listing 25.9 Example of fmod() Usage

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

int main(void)
{
   double x = -54.4, y = 10.0;
   printf("Remainder of %f / %f = %f.\n",   x, y, fmod(x, y));

   return 0;
}
```

```
Output:
Remainder of -54.400000 / 10.000000 = -4.400000.
```

fmodf

Implements the fmod() function for float type values. See "fmod" on page 191.

fmodl

Implements the fmod() function for long double type values. See "fmod" on page 191.

frexp

Extract the mantissa and exponent. The `frexp()` function extracts the mantissa and exponent of `value` based on the formula $x*2^n$, where the mantissa is 0.5 ð |x| < 1.0 and n is an integer exponent.

```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float, int *);
long double frexpl(long double, int *);
```

Table 25.17 frexp

x	double, float or long double	The value to compute
exp	int	Exponent

Remarks

`frexp()` returns the `double` mantissa of `value`. It stores the integer exponent value at the address referenced by `exp`.

This function may not be implemented on all platforms.

See Also

"ldexp" on page 197

"fmod" on page 191

Listing 25.10 Example of frexp() Usage

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

int main(void)
{
   double m, value = 12.0;
   int e;

   m = frexp(value, &e);

   printf("%f = %f * 2 to the power of %d.\n",value, m, e);

   return 0;
```

```
}
```

```
Output:
12.000000 = 0.750000 * 2 to the power of 4.
```

frexpf

Implements the frexp() function for float type values. See "frexp" on page 193.

frexpl

Implements the frexp() function for long double type values. See "frexp" on page 193.

isgreater

The facility determine the greater of two doubles. Unlike `x>y isgreater` does not raise an invalid exception when x and y are unordered.

```
#include <math.h>
```

```
int isgreater(x, y);
```

Table 25.18 isgreater

x	float, double or long double	number compared
y	float, double or long double	number compared

Remarks

This facility is true if x is greater than y.

This function may not be implemented on all platforms.

isgreaterless

The facility determines if two numbers are unequal. Unlike `x>y || x<y` `isgreaterless` does not raise an invalid exception when x and y are unordered.

```
#include <math.h>
int isgreaterless(x, y)
```

Table 25.19 isgreaterless

x	float, double or long double	number compared
y	float, double or long double	number compared

Remarks

This facility returns true if x is greater than or less than y.

This function may not be implemented on all platforms.

isless

The facility determines the lesser of two numbers.

```
#include <math.h>
int isless(x, y);
```

Table 25.20 isless

x	float, double or long double	number compared
y	float, double or long double	number compared

Remarks

This facility is true if x is less than y.

This function may not be implemented on all platforms.

islessequal

The facility test for less than or equal to comparison. Unlike `x<y || x==y` `islessequal` does not raise an invalid exception when x and y are unordered.

```
#include <math.h>
int islessequal(x, y);
```

Table 25.21 islessequal

x	float, double or long double	number compared
y	float, double or long double	number compared

Remarks

This facility is true if x is less than or equal to y.

This function may not be implemented on all platforms.

isunordered

The facility compares the order of the arguments.

```
#include <math.h>
int isunordered(x, y);
```

Table 25.22 isunordered

x	float, double or long double	number compared
y	float, double or long double	number compared

Remarks

This facility is true if the arguments are unordered false otherwise.

This function may not be implemented on all platforms.

ldexp

Compute a value from a mantissa and exponent. The `ldexp()` function computes `x` $*2^{exp}$. This function can be used to construct a `double` value from the values returned by the `frexp()` function.

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

Table 25.23 ldexp

x	double, float or long double	The value to compute
exp	int	Exponent

Remarks

The Function `ldexp()` returns `x * 2exp`.

This function may not be implemented on all platforms.

See Also

"frexp" on page 193

"modf" on page 201

Listing 25.11 Example of ldexp() Usage

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

int main(void)
{
   double value, x = 0.75;
   int e = 4;

   value = ldexp(x, e);

   printf("%f * 2 to the power of %d is %f.\n",x, e, value);

   return 0;
```

```
}
```

```
Output:
0.750000 * 2 to the power of 4 is 12.000000.
```

ldexpf

Implements the ldexp() function for float type values. See "ldexp" on page 197.

ldexpl

Implements the ldexp() function for long double type values. See "ldexp" on page 197.

log

Compute the natural logarithms.

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

Table 25.24 log

x	float, double or long double	value to be computed

Remarks

`log()` returns $\log_e x$. If `x` < 0 the `log()` may assign `EDOM` to `errno`

See "Floating point error testing." on page 174, for information on newer error testing procedures.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"exp" on page 187

"errno" on page 75

Listing 25.12 Example of log(), log10() Usage

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

int main(void)
{
   double x = 100.0;

   printf("The natural logarithm of %f is %f\n",x, log(x));
   printf("The base 10 logarithm of %f is %f\n",x, log10(x));

   return 0;
}
```

```
Output:
The natural logarithm of 100.000000 is 4.605170
The base 10 logarithm of 100.000000 is 2.000000
```

logf

Implements the log() function for float type values. See "log" on page 198.

logl

Implements the log() function for long double type values. See "log" on page 198.

log10

Compute the base 10 logarithms.

```
#include <math.h>
double log10(double x);
float log10f(float);
long double log10l(long double);
```

Table 25.25 log10

x	float, double or long double	value to be computed

Remarks

`log10()` returns $\log_{10}x$. If `x` < 0 `log10()` may assign `EDOM` to `errno`. See "Floating point error testing." on page 174, for information on newer error testing procedures. For example usage, see Listing 25.12.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"exp" on page 187

"errno" on page 75

log10f

Implements the log10() function for float type values. See "log10" on page 200.

log10l

Implements the log10() function for long double type values. See "log10" on page 200.

modf

Separates integer and fractional parts.

```
#include <math.h>
double modf(double value, double *iptr);
float fmodf(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Table 25.26 modf

value	double, float, or long double	The value to separate
iptr	double, float, or long double	integer part

Remarks

The `modf()` function separates `value` into its integer and fractional parts. In other words, `modf()` separates `value` such that `value` = $f + i$ where 0 ð f < 1, and i is the largest integer that is not greater than `value`.

`modf()` returns the signed fractional part of `value`, and stores the integer part in the integer pointed to by `iptr`.

This function may not be implemented on all platforms.

See Also

"frexp" on page 193

"ldexp" on page 197

Listing 25.13 Example of modf() Usage

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

int main(void)
{
   double i, f, value = 27.04;

   f = modf(value, &i);
   printf("The fractional part of %f is %f.\n", value, f);
```

```
   printf("The integer part of %f is %f.\n", value, i);

   return 0;
}
```

```
Output:
The fractional part of 27.040000 is 0.040000.
The integer part of 27.040000 is 27.000000.
```

modff

Implements the modf() function for float type values. See "modf" on page 201.

modfl

Implements the modf() function for long double type values. See "modf" on page 201.

pow

Calculate x^y.

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

Table 25.27 pow

x	float, double or long double	value to be computed

Remarks

The `pow()` function may assign `EDOM` to `errno` if `x` is 0.0 and `y` is less than or equal to zero or if `x` is less than zero and `y` is not an integer. For example of error detection usage, see Listing 25.1.

See "Floating point error testing." on page 174, for information on newer error testing procedures.

The function `pow()` returns x^y.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"sqrt" on page 207

Listing 25.14 Example of pow() Usage

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

int main(void)
{
   double x;

   printf("Powers of 2:\n");
   for (x = 1.0; x <= 10.0; x += 1.0)
   printf("2 to the %4.0f is %4.0f.\n", x, pow(2, x));

   return 0;
}
```

```
Output:
Powers of 2:
2 to the    1 is    2.
2 to the    2 is    4.
2 to the    3 is    8.
2 to the    4 is   16.
2 to the    5 is   32.
2 to the    6 is   64.
2 to the    7 is  128.
2 to the    8 is  256.
2 to the    9 is  512.
2 to the   10 is 1024.
```

powf

Implements the pow() function for float type values. See "pow" on page 202.

powl

Implements the pow() function for long double type values. See "pow" on page 202.

sin

Compute sine.

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

Table 25.28 sin

x	float, double or long double	value to be computed

Remarks

The argument for the sin() function should be in radians. One radian is equal to 360/2¼ degrees.

The function `sin()` returns the sine of `x`. `x` is measured in radians.

This function may not be implemented on all platforms.

See Also

"cos" on page 184

"tan" on page 208

Listing 25.15 Example of sin() Usage

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

#define DtoR 2*pi/360

int main(void)
{
   double x = 57.0;
   double xRad = x*DtoR;
```

```
   printf("The sine of %.2f degrees is %.4f.\n",x, sin(xRad));

   return 0;
}
```

```
Output:
The sine of 57.00 degrees is 0.8387.
```

sinf

Implements the sin() function for float type values. See "sin" on page 204.

sinl

Implements the sin() function for long double type values. See "sin" on page 204.

sinh

Compute the hyperbolic sine.

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

Table 25.29 sinh

x	float, double or long double	value to be computed

Remarks

A range error can occur if the absolute value of the argument is to large.

`sinh()` returns the hyperbolic sine of `x`.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"cosh" on page 186

"tanh" on page 209

Listing 25.16 Example of sinh() Usage

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

int main(void)
{
   double x = 0.5;
   printf("Hyperbolic sine of %f is %f.\n", x, sinh(x));

   return 0;
}
```

```
Output:
Hyperbolic sine of 0.500000 is 0.521095.
```

sinhf

Implements the sinh() function for float type values. See "sinh" on page 205.

sinhl

Implements the sinh() function for long double type values. See "sinh" on page 205.

sqrt

Calculate the square root.

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

Table 25.30 sqrt

x	float, double or long double	value to compute

Remarks

A domain error occurs if the argument is a negative value.

`sqrt()` returns the square root of `x`.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"pow" on page 202

Listing 25.17 Example of sqrt() Usage

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

int main(void)
{
   double x = 64.0;

   printf("The square root of %f is %f.\n", x, sqrt(x));

   return 0;
}
```

```
Output:
The square root of 64.000000 is 8.000000.
```

sqrtf

Implements the sqrt() function for float type values. See "sqrt" on page 207.

sqrtl

Implements the sqrt() function for long double type values. See "sqrt" on page 207.

tan

Computes tangent of the argument.

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

Table 25.31 tan

x	float, double or long double	value to compute

Remarks

A range error may occur if the argument is close to an odd multiple of pi divided by 2

`tan()` returns the tangent of `x`. `x` is measured in radians.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

"cos" on page 184

"sin" on page 204

Listing 25.18 Example of tan() Usage

```
#include <math.h>
```

```
#include <stdio.h>

int main(void)
{
   double x = 0.5;

   printf("The tangent of %f is %f.\n", x, tan(x));

   return 0;
}
```

```
Output:
The tangent of 0.500000 is 0.546302.
```

tanf

Implements the tan() function for float type values. See "tan" on page 208.

tanl

Implements the tan() function for long double type values. See "tan" on page 208.

tanh

Compute the hyperbolic tangent.

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

Table 25.32 tanh

x	float, double or long double	value to compute

Remarks

`tanh()` returns the hyperbolic tangent of `x`.

This function may not be implemented on all platforms.

See Also

"cosh" on page 186

"sinh" on page 205

Listing 25.19 Example of tanh() Usage

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

int main(void)
{
   double x = 0.5;

   printf("The hyperbolic tangent of %f is %f.\n",x, tanh(x));

   return 0;

}
```

```
Output:
The hyperbolic tangent of 0.500000 is 0.462117.
```

tanhf

Implements the tanh() function for float type values. See "tanh" on page 209.

tanhl

Implements the tanh() function for long double type values. See "tanh" on page 209.

HUGE_VAL

The largest floating point value with the same sign possible for a function's return.

```
#include <math.h>
```

```
Varies by CPU
```

Remarks

If the result of a function is too large to be represented as a value by the return type, the function should return HUGE_VAL. It is the largest floating point value with the same sign as the expected return type.

This function may not be implemented on all platforms.

C99 Implementations

Although not formally accepted by the ANSI/ISO committee these proposed math functions are already implemented on some platforms.

acosh

`Acosh` computes the (non-negative) arc or inverse hyperbolic `cosine` of `x` in the range `[01, +INF]`.

```
#include <math.h>
```

```
double acosh(double x);
```

Table 25.33 acosh

x	double	The value to compute

Remarks

A domain error occurs for argument `x` is less than 1 and a range error occurs if `x` is too large.

The (non-negative) arc hyperbolic cosine of x.

This function may not be implemented on all platforms.

See Also

"acos" on page 178

Listing 25.20 Example of acosh() Usage

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

int main(void)
{
   double a = 3.14;
   printf("The arc hyperbolic cosine of %f is %f.\n\n",
               a,acosh(a));
   return 0;
}
```

```
Output:
The arc hyperbolic cosine of 3.140000 is 1.810991.
```

asinh

Asinh computes the arc or inverse hyperbolic sine.

```
#include <math.h>
double asinh(double x);
```

Table 25.34 asinh

x	double	The value to compute

Remarks

A range error occurs if the magnitude of x is too large.

The arc hyperbolic sine of the argument x.

This function may not be implemented on all platforms.

See Also

"asin" on page 179

Listing 25.21 Example of asinh() Usage

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

int main(void)
{
   double b = 3.14;
   printf("The arc hyperbolic sine of %f is %f.\n\n",
            b,asinh(b));
   return 0;
}
```

```
Output:
The arc hyperbolic sine of 3.140000 is 1.861813
```

atanh

The function atanh computes the arc hyperbolic tangent.

```
#include <math.h>

double atanh(double x);
```

Table 25.35 atanh

x	double	The value to compute

Remarks

The arc hyperbolic tangent of `x`.

A domain error occurs for arguments not in the range [-1,+1]

This function may not be implemented on all platforms.

See Also

"atan" on page 180

Listing 25.22 Example of atanh() Usage

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

```
int main(void)
{
   double c = 0.5;
   printf("The arc hyperbolic tan of %f is %f.\n\n",
            c,atanh(c));
   return 0;
}
```

```
Output:
The arc hyperbolic tan of 0.500000 is 0.549306.
```

cbrt

The cbrt functions compute the real cube root

```
#include <math.h>
double cbrt(double x);
float cbrtf(float fx);
long double cbrtl(long double lx);
```

Table 25.36 cbrt

x	double	The value being cubed
fx	float	The value being cubed
lx	long double	The value being cubed

Remarks

The cbrt functions returns the real cube root of the argument.

This function may not be implemented on all platforms.

See Also

"sqrt" on page 207

copysign

The function copysign produces a value with the magnitude of x and the sign of y. The copysign function regards the sign of zero as positive. It produces a NaN with the sign of y if x is NaN.

```
#include <math.h>

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

Table 25.37 copysign

x	double	Magnitude
y	double	The sign argument

Remarks

A value with the magnitude of x and the sign of y.

This function may not be implemented on all platforms.

Listing 25.23 Example of copysign() Usage

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

int main(void)
{
   double e = +10.0;
   double f = -3.0;
   printf("Copysign(%f, %f) = %f.\n\n",
            e,f,copysign(e,f));
   return 0;
}
```

```
Output:
Copysign(10.000000, -3.000000) = -10.000000.
```

erf

The erf function is used in probability.

```
#include <math.h>
double erf(double x);
```

Table 25.38 erf

x	double	The value to be computed

Remarks

The function erf( ) is defined as:

```
erf (x) = ( 2/sqrt(pi) *
( integral from 0 to x of exp(-t^2)dt ))
```

The error function of `x` is returned.

This function may not be implemented on all platforms.

Listing 25.24 Example of erf() Usage

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

int main(void)
{
   double g = +10.0;
   printf("The error function of (%f) = %f.\n\n",
            g,erf(g));
   return 0;
}
```

```
Output:
The error function of (10.000000) = 1.000000
```

erfc

The erfc( ) function computes the complement of the error function.

```
#include <math.h>
double erfc(double x);
```

Table 25.39 erfc

x	double	The value to be computed

The erfc( ) function computes the complement of the error function of `x`:

```
erfc(x) = 1 - ( erf (x) = 2/sqrt(pi) *
( integral from 0 to x of exp(-t^2) dt ))
```

The complementary error function of `x` is returned.

This function may not be implemented on all platforms.

Listing 25.25 Example of erfc() Usage

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

int main(void)
{
   double h = +10.0;
   printf("The inverse error function of (%f) = %f.\n\n",
            h,erfc(h));
   return 0;
```

```
Output:
The inverse error functions of (10.000000) = 0.000000}
```

exp2

The function exp2 computes the base-2 exponential.

```
#include <math.h>
double exp2(double x);
```

Table 25.40 exp2

x	double	The value to compute

Remarks

The function exp2 computes the base-2 exponential. In other words `exp2( b )` solves for the x in `( b = 2x ).`

A range error occurs if the magnitude of x is too large

The function returns the base-2 exponential of x: 2^x

This function may not be implemented on all platforms.

See Also

"pow" on page 202

Listing 25.26 Example of exp2() Usage

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

int main(void)
{
   double i = 12;
   printf("2^%f = %f.\n\n",i,i, exp2(i));
   return 0;
}
```

```
Output:
2^(12.000000) = 4096.000000.
```

expm1

The function `expm1` computes the base-e exponential minus 1.

```
#include <math.h>
double expm1(double x);
```

Table 25.41 expm1

x	double	The value to compute

Remarks

The function `expm1` computes the base-e exponential minus 1. Written as:

```
expm1( x ) = ( ex ) - 1.0
```

A range error occurs if x is too large. For small magnitude x, expm1(x) is expected to be more accurate than exp(x) -1

The base-e exponential of x, minus 1: (e^x) -1 is returned.

This function may not be implemented on all platforms.

Listing 25.27 Example of expm1() Usage

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

int main(void)
{
   double j = 12;
   printf("(e - 1)^%f = %f.\n\n",j,expm1(j));
   return 0;
}
```

```
Output:
(e - 1)^12.000000 = 162753.791419.
```

fdim

The function fdim computes the positive difference of its arguments

```
#include <math.h>

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

Table 25.42 fdim

x	double	Value one
y	double	Value two

Remarks

This function returns the value of x - y if x is greater than y else zero. If x is less than or equal to y a range error may occur

This function may not be implemented on all platforms.

Listing 25.28 Example of fdim() Usage

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

int main(void)
{
   double k = 12;
   double l = 4;
   printf("|(%f - %f)| = %f.\n\n",k,l,fdim(k,l));
   return 0;
}
```

```
Output:
| ( 12.000000 - 4.0000000 ) |  =  8.0000000.
```

fma

The fma functions return the sum of the third argument plus the product of the first two arguments rounded as one ternary operation.

```
#include <math.h>

double fma(double x, double y, double z);

float fmaf(float fx, float fy, float fz);

long double fmal(long double lx, long double ly,long double
     lz);
```

Table 25.43 fma

fx	float	An argument to be multiplied
fy	float	The second argument being multiplied
fz	float	The argument to be added
x	double	An argument to be multiplied
y	double	The second argument being multiplied
z	double	The argument to be added
lx	long double	An argument to be multiplied
ly	long double	The second argument being multiplied
lz	long double	The argument to be added

Remarks

The fma functions compute `(x * y) + z,` rounded as one ternary operation:

The fma functions compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of `FLT_ROUNDS`.

The fma functions returns the result of `(x*y) + z`.

This function may not be implemented on all platforms.

See Also

"round" on page 236

fmax

The function fmax computes the maximum numeric value of its argument

```
#include <math.h>
double fmax(double x, double y);
```

Table 25.44 fmax

x	double	First argument
y	double	Second argument

Remarks

The maximum value of x or y is returned.

This function may not be implemented on all platforms.

See Also

"fmin" on page 222

Listing 25.29 Example of fmax() Usage

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

int main(void)
{
   double m = 4;
   double n = 6;
   printf("fmax(%f, %f)=%f.\n\n",m,n,fmax(m,n));
   return 0;
}
```

```
Output:
fmax(4.000000, 6.000000) = 6.000000.
```

fmin

The function fmin computes the minimum numeric value of its arguments.

```
#include <math.h>
double fmin(double x, double y);
```

Table 25.45 fmin

x	double	First argument
y	double	Second argument

Remarks

Fmin returns the minimum numeric value of its arguments

This function may not be implemented on all platforms.

See Also

"fmax" on page 221

Listing 25.30 Example of fmin() Usage

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

int main(void)
{
   double o = 4;
   double p = 6;
   printf("fmin(%f, %f) = %f.\n\n", o,p,fmin(o,p));
   return 0;
}
```

```
Output:
fmin(4.000000, 6.000000) = 4.000000.
```

gamma

The `gamma( )` function computes `loge G(x)`.

```
#include <math.h>
double gamma (double x);
```

Table 25.46 gamma

x	double	The value to be computed

Remarks

The `gamma( )` function computes `loge G(x)`, where `G(x)` is defined as the integral of `( e(-t) * t(x-1) )dt` from `0` to infinity. The sign of `G(x)` is returned in the external integer signgam.

The argument *x* need not be a non-positive integer, `( G(x)` is defined over the real numbers, except the non-positive integers).

An application wishing to check for error situations should set `errno` to `0` before calling `lgamma()`. If `errno` is non-zero on return.

- If the return value is `NaN`, an error has occurred.
- A domain error occurs if `x` is equal to zero or if `x` is a negative integer.
- A range error may occur.

The gamma function of `x` is returned.

This function may not be implemented on all platforms.

See Also

"lgamma" on page 226

hypot

The function hypot computes the square root of the sum of the squares of the arguments.

```
#include <math.h>
double hypot(double x, double y);
```

Table 25.47 hypot

x	double	The first value to be squared
y	double	The second value to be squared

Remarks

Hypot computes the square root of the sum of the squares of x and y without undue overflow or underflow.

A range error may occur.

The square root of the sum of the squares of `x` and `y` is returned.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

Listing 25.31 Example of hypot() Usage

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

int main(void)
{
   double r = 4;
   double s = 4;
   printf("(%f^2 + %f^2)^(.5) = %f\n\n",r,s,hypot(r,s));
   return 0;
}
```

```
Output:
(4.000000^2 + 4.000000^2)^(.5) = 5.656854
```

ilogb

The ilogb functions determine the exponent of the argument as a signed int value.

```
#include <math.h>
int ilogb(double x);
int ilogbf(float y);
int ilogbl(long double z);
```

Table 25.48 flogb

x	double	The whose exponent is determined
y	float	The whose exponent is determined
z	long double	The whose exponent is determined

Remarks

The ilogb functions extract the exponent of x as a signed int value. If x is zero they compute the value FP_ILOGB0; if x is infinite they compute the value

INT_MAX; if x is a NaN they compute the value FP_ILOGBNAN; otherwise, they are equivalent to calling the corresponding logb function and casting the returned value to type int.

A rangeerror may occur if x is 0.

The ilogb functions return the exponent of x as a signed int value.

This function may not be implemented on all platforms.

This function may not be implemented on all platforms.

See Also

"exp" on page 187

lgamma

The `lgamma( )` function computes the same thing as the `gamma()`.

```
#include <math.h>
double lgamma(double x);
```

Table 25.49 lgamma

x	double	The value to be computed

Remarks

The `lgamma( )` function computes the same thing as the `gamma()` with the addition of absolute value signs `loge | G(x) |`, where `G(x)` is defined as the integral of `( e(-t) * t(x-1) )dt` from `0` to infinity.

The sign of `G(x)` is returned in the external integer signgam. The argument x need not be a non-positive integer, `( G(x)` is defined over the real numbers, except the non-positive integers).

An application wishing to check for error situations should set `errno` to `0` before calling `lgamma()`. If `errno` is non-zero on return, or the return value is `NaN`, an error has occurred.

`lgamma()` may create a range error occurs if x is too large

The log of the absolute value of gamma of `x`.

This function may not be implemented on all platforms.

See Also

"gamma" on page 223

log1p

The function log1p computes the base-e logarithm.

```
#include <math.h>
double log1p(double x);
```

Table 25.50 log1p

x	double	The value being computed

Remarks

The function log1p computes the base-e logarithm. Which can be denoted as

```
log1p = loge(1.0 + x )
```

The value of x must be greater than `-1.0`.

For small magnitude x, log1p(x) is expected to be more accurate than log(x+1)

- A domain error occurs if x is less than negative one.
- A range error may occur if x is equal to one.

The base-e logarithm of 1 plus x is returned.

This function may not be implemented on all platforms.

See Also

"log" on page 198

Listing 25.32 Example of log1p() Usage

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

int main(void)
{
   double u = 5;
   printf("log1p computes log1p(%f) = %f\n\n",u,log1p(u));
   return 0;
```

```
}
```

```
Output:
log1p computes log1p(5.000000) = 1.791759
```

log2

The function `log2` computes the base-2 logarithm.

```
#include <math.h>
double log2(double x);
```

Table 25.51 log2

x	double	The value being computed

Remarks

The function `log2` computes the base-2 logarithm which can be denoted as:

`log2(x) = log2(x)`

The value of `x` must be positive.

- A domain error may occur if x is less than zero.
- A range error may occur if x is equal to zero.

The base-2 logarithm of x is returned.

This function may not be implemented on all platforms.

See Also

"log" on page 198

Listing 25.33 Example of log2() Usage

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

int main(void)
{
   double v = 5;
   printf("log2(%f) = %f\n\n",v,log2(v));
   return 0;
```

```
}
```

```
Output:
log2(5.000000) = 2.321928
```

logb

The `logb( )` function computes the exponent of x.

```
#include <math.h>
double logb(double x);
```

Table 25.52 logb

x	double	The value being computed

Remarks

The `logb( )` function computes the exponent of `x`, which is the integral part of `logr | x |`, as a signed floating point value, for non-zero `x`, where `r` is the radix of the machine's floating-point arithmetic. If x is subnormal it is treated as though it were normalized.

A range error may occur if x is equal to zero.

The exponent of x as a signed integral value in the format of the `x` argument.

This function may not be implemented on all platforms.

See Also

"Inlined Intrinsics Option" on page 174

Listing 25.34 Example of logb() Usage

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

int main(void)
{
   double w = 5;
   printf("logb(%f) = %f\n\n",w,logb(w));
   return 0;
```

```
}
```

```
Output:
logb(5.000000) = 2.000000
```

nan

The function nan tests for NaN.

```
#include <math.h>
double nan(const char *tagp);
```

Table 25.53 nan

tagp	const char *	A character string

Remarks

See "Quiet NaN" on page 174, fore more information.

A quiet NAN is returned, if available.

This function may not be implemented on all platforms.

See Also

"isnan" on page 177

"NaN Not a Number" on page 173

nearbyint

The function `nearbyint` rounds off the argument to an integral value.

```
#include <math.h>
double nearbyint(double x);
```

Table 25.54 nearbyint

x	double	The value to be computed

Remarks

Nearbyint, computes like rint but doesn't raise an inexact exception.

The argument is returned as an integral value in floating point format.

This function may not be implemented on all platforms.

Listing 25.35 Example of nearbyint() Usage

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

int main(void)
{
   double x = 5.7;
   printf("nearbyint(%f) = %f\n\n",x,nearbyint(x));
   return 0;
}
```

```
Output:
nearbyint(5.700000) = 6.000000
```

nextafter

The `nextafter` function determines the next representable value in the type of the function.

```
#include <math.h>

float nextafter(float x, float y);

double nextafter(double x, double y);

long double nextafter(long double x, long double y);
```

Table 25.55 nextafter

x	float double long double	current representable value
y	float double long double	direction

Remarks

The `nextafter` function determines the next representable value in the type of the function, after `x` in the direction of `y`, where `x` and `y` are first converted to the type of the function. Thus, if `y` is less than `x, nextafter( )` returns the largest representable floating-point number less than `x`.

The next representable value after `x` is returned.

This function may not be implemented on all platforms.

Listing 25.36 Example of nextafter() Usage

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

int main(void)
{
   double y = 7;
   double z = 10;
   printf("nextafter(%f,%f) = %f\n\n",y,z,nextafter(y,z));
   return 0;
}
```

```
Output:
Nextafter(7.000000,10.000000) = 7.000000
```

remainder

Remainder computes the remainder `x REM y` required by `IEC 559.`

```
#include <math.h>
double remainder(double x, double y);
```

Table 25.56 remainder

x	double	The first value
y	double	The second value

Remarks

The `remainder( )` function returns the floating point remainder `r = x - ny` when `y` is non-zero. The value `n` is the integral value nearest the exact value `x/y`. When `| n - x/y | = _ ,` the value `n` is chosen to be even.

The behavior of `remainder()` is independent of the rounding mode.

The remainder x REM y is returned.

This function may not be implemented on all platforms.

See Also

"remquo" on page 233

Listing 25.37 Example of remainder() Usage

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

int main(void)
{
   double var1 = 2;
   double var2 = 4;
   printf("remainder(%f,%f) = %f\n\n",var1,var2,remainder(var1,var2));
   return 0;
}
```

```
Output:
remainder(2.000000,4.000000) = 2.000000
```

remquo

The function remquo computes the same remainder as the remainder function.

```
#include <math.h>
double remquo(double x, double y, int *quo);
```

Table 25.57 remquo

x	double	First value

Table 25.57 remquo (*continued*)

y	double	Second value
quo	int*	Pointer to an object quotient

Remarks

The argument quo points to an object whose sign is the sign as x/y and whose magnitude is congruent mod 2^n to the magnitude of the integral quotient of x/y, where n >= 3.

The value of x may be so large in magnitude relative to y that an exact representation of the quotient is not practical.

The remainder of x and y is returned.

This function may not be implemented on all platforms.

See Also

"remainder" on page 232

rint

The function rint rounds off the argument to an integral value. Rounds its argument to an integral value in floating-point format using the current rounding direction.

```
#include <math.h>
double rint(double x);
```

Table 25.58 rint

x	double	The value to be computed

Remarks

The argument in integral value in floating point format is returned.

This function may not be implemented on all platforms.

See Also

"rinttol" on page 235

Listing 25.38 Example of rint() Usage

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

int main(void)
{
   double varONE = 2.5;
   printf("rint(%f) = %f\n\n",varONE,rint(varONE));
   return 0;
}
```

```
Output:
rint(2.500000) = 2.000000
```

rinttol

Rintrol rounds its argument to the nearest integral value using the current rounding direction.

```
#include <math.h>
```

```
long int rinttol(double x);
```

Table 25.59 rinttol

x	double	Value being rounded

Remarks

If the rounded range is outside the range of long, result is unspecified

The argument in integral value in floating point format is returned.

This function may not be implemented on all platforms.

See Also

"rint" on page 234

round

Round rounds its argument to an integral value in floating-point format.

```
#include <math.h>
double round(double x);
```

Table 25.60 round

x	double	The value to be rounded

Remarks

Rounding halfway cases away from zero, regardless of the current rounding direction.

The argument rounded to an integral value in floating point format nearest to is returned but is never larger in magnitude than the argument.

This function may not be implemented on all platforms.

See Also

"roundtol" on page 237

Listing 25.39 Example of round() Usage

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

int main(void)
{
   double varALPHA = 2.4;
   printf("round(%f) = %f\n\n",varALPHA,round(varALPHA));
   return 0;
}
```

```
Output:
round(2.400000) = 2.000000
```

roundtol

The function roundtol rounds its argument to the nearest integral value. Rounding halfway cases away from zero, regardless of the current rounding direction.

```
#include <math.h>
long int roundtol(double round);
```

Table 25.61 rooundtol

round	double	The value being rounded

Remarks

If the rounded range is outside the range of long, result is unspecified

The argument rounded to an integral value in long int format is returned.

This function may not be implemented on all platforms.

See Also

"round" on page 236

scalb

The function scalb computes x * FLT_RADIX^n efficiently, not normally by computing FLT_RADIX^n explicitly.

```
#include <math.h>
double scalb(double x, long int n);
```

Table 25.62 scalb

x	double	The original value
n	long int	Power value

Remarks

A range error may occur

The function scalb returns `x * FLT_RADIX^n`.

This function may not be implemented on all platforms.

trunc

Trunc rounds its argument to an integral value in floating-point format nearest to but no larger in magnitude than the argument.

```
#include <math.h>

double trunc(double x);
```

Table 25.63 trunc

x	double	The value to be truncated.

Remarks

For 68k processors returns an integral value.

The trunc function returns an argument to an integral value in floating-point format.

This function may not be implemented on all platforms.

Listing 25.40 Example of trunc() Usage

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

int main(void)
{
   double varPHI = 2.4108;
   printf("trunc(%f) = %f\n\n",varPHI,trunc(varPHI));
   return 0;
}
```

```
Output:
trunc(2.410800) = 2.000000
```

26

Process.h

The header Process.h defines the threadex functions _beginthreadex and _endthreadex.

Overview of Process.h

This header file defines the facilities as follows:

- “ beginthread” on page 239 begins a thread.
- “ beginthreadex” on page 240 begins a with local data.
- “ endthread” on page 241 ends thread for `_beginthread`.
- “ endthreadex” on page 242 ends thread for `_beginthreadex`.

_beginthread

This function starts a thread for a multi-threaded application.

```
#include <process.h>
long _beginthread(void (__cdecl *inCodeAddress)(void *)
     thread,
   unsigned int inStackSize, void *inParameter);
```

Table 26.1 _beginthread

thread	void *	The address of the thread function
inStackSize	int	Set by the linkers /STACK switch, 1MB is the default
inParameter	void *	The same as the lpvThreadParameter used in CreateThread() that is used to pass an initialization routine.

Remarks

The thread runs concurrently with the rest of the code.

The thread handle is returned or zero upon failure.

A Windows only function—this function may not be implemented on all versions.

See Also

"_endthread" on page 241

"_beginthreadex" on page 240

_beginthreadex

Begins a thread.

```
#include <process.h>
HANDLE __cdecl _beginthreadex(
    LPSECURITY_ATTRIBUTES inSecurity,
    DWORD inStacksize,
    LPTHREAD_START_ROUTINE inCodeAddress,
    LPVOID inParameter,
    DWORD inCreationFlags,
    LPDWORD inThreadID);
```

Table 26.2 _beginthreadex

inSecurity	LPSECURITY_ATTRIBUTES	Security Attributes, NULL is the default attributes.
inStacksize	`DWORD` *	Set by the linkers /STACK switch, 1MB is the default
inCodeAddress	LPTHREAD_START_ROUTINE	The address of the function containing the code where the new thread should start.
inParameter	LPVOID	The same as the lpvThreadParameter used in CreateThread() that is used to pass an initialization routine.

Table 26.2 _beginthreadex (*continued*)

inCreationFlags	DWORD	If zero begins thread immediately, if CREATE_SUSPENDED it waits before executing.
inThreadID	LPDWORD	An variable to store the ID assigned to a new thread.

The function _beginthreadex is similar to the Windows call CreateThread except this functions properly creates the local data used by MSL.

A `HANDLE` variable is returned if successful.

A Windows only function—this function may not be implemented on all versions.

See Also

" endthreadex" on page 242

_endthread

This function ends a thread called by `_beginthread`.

```
#include <process.h>
void _endthread(void);
```

Remarks

This facility has no parameters.

There is no return value for this function.

Windows only compatible function.

See Also

" beginthread" on page 239

" endthreadex" on page 242

_endthreadex

Exits the thread.

```
#include <process.h>
VOID __cdecl _endthreadex(DWORD inReturnCode);
```

Table 26.3 _endthreadex

inReturnCode	DWORD	The exit code is passed through this argument.

Remarks

The function_endthreadex is similar to the Windows call ExitThread except this functions properly destroys the thread local data used by MSL.

There is no return.

A Windows only function—this function may not be implemented on all versions.

See Also

“ beginthreadex” on page 240

27

setjmp.h

The `setjmp.h` header file provides a means of saving and restoring a processor state. The facilities that do this are "longjmp" and "setjmp":

Overview of setjmp.h

The `setjmp.h` header file provides a means of saving and restoring a processor state. The `setjmp.h` functions are typically used for programming error and low-level interrupt handlers.

- The function "setjmp" on page 245 saves the current calling environment—the current processor state—in its `jmp_buf` argument. The `jmp_buf` type, an array, holds the processor program counter, stack pointer, and relevant data and address registers.
- The function "longjmp" on page 244 restores the processor to its state at the time of the last `setjmp()` call. In other words, `longjmp()` returns program execution to the last `setjmp()` call if the `setjmp()` and `longjmp()` pair use the same `jmp_buf` variable as arguments.

Non-local Jumps and Exception Handling

Because the `jmp_buf` variable can be global, the `setjmp` and `longjmp` calls do not have to be in the same function body.

A `jmp_buf` variable must be initialized with a call to `setjmp()` before being used with `longjmp()`. Calling `longjmp()` with an un-initialized `jmp_buf` variable may crash the program. Variables assigned to registers through compiler optimization may be corrupted during execution between `setjmp()` and `longjmp()` calls. This situation can be avoided by declaring affected variables as `volatile`.

longjmp

Restore the processor state saved by `setjmp()`.

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

Table 27.1 longjmp

env	jmp_buf	The current processor state
val	int	A value returned by setjmp()

Remarks

The `longjmp()` function restores the calling environment (i.e. returns program execution) to the state saved by the last called `setjmp()` to use the `env` variable. Program execution continues from the `setjmp()` function. The `val` argument is the value returned by `setjmp()` when the processor state is restored.

The longjmp function is redefined when AltiVec support is enabled. The programmer must ensure that both the “to” compilation unit and “from” compilation unit have AltiVec enabled. Failure to do so will create an undesired result.

After longjmp is completed, program execution continues as if the corresponding invocation of the setjmp macro had just returned the value specified by val. The longjmp function cannot cause the setjmp macro to return the value 0; if val is 0, the setjmp macro returns the value 1.

The `env` variable must be initialized by a previously executed `setjmp()` before being used by `longjmp()` to avoid undesired results in program execution.

This function may not be implemented on all platforms.

See Also

“setjmp” on page 245

“signal” on page 251

“abort” on page 405

setjmp

Save the processor state for `longjmp()`.

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

Table 27.2 setjmp

env	jmp_buf	The current processor state

Remarks

The `setjmp()` function saves the calling environment—data and address registers, the stack pointer, and the program counter—in the `env` argument. The argument must be initialized by `setjmp()` before being passed as an argument to `longjmp()`. For example usage, see Listing 27.1.

The setjmp function is redefined when AltiVec support is enabled. The programmer must ensure that both the "from" compilation unit and "to" compilation unit have AltiVec enabled. Failure to do so will create an undesired result.

When it is first called, `setjmp()` saves the processor state and returns 0. When `longjmp()` is called program execution jumps to the `setjmp()` that saved the processor state in `env`. When activated through a call to `longjmp()`, `setjmp()` returns `longjmp()`'s `val` argument.

This function may not be implemented on all platforms.

See Also

"longjmp" on page 244

"signal" on page 251

"abort" on page 405

Listing 27.1 Example of setjmp() Usage

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

// Let main() and doerr() both have
// access to global env
```

```
volatile jmp_buf env;

void doerr(void);
int main(void)
{

   int i, j, k;

   printf("Enter 3 integers that total less than 100.\n");
   printf("A zero sum will quit.\n\n");

   // If the total of entered numbers is not less than 100,
   // program execution is restarted from this point.

   if (setjmp(env) != 0)
      printf("Try again, please.\n");

   do {
      scanf("%d %d %d", &i, &j, &k);
      if ( (i + j + k) == 0)
         exit(0);       // quit program
      printf("%d + %d + %d = %d\n\n", i, j, k, i+j+k);
      if ( (i + j + k) >= 100)
         doerr();     // error!
   } while (1);       // loop forever

   return 0;
}

void doerr(void)       // this is the error handler
{
   printf("The total is >= 100!\n");
   longjmp(env, 1);
}
```

```
Output:

Enter 3 integers that total less than 100.
A zero sum will quit.

10 20 30
10 + 20 + 30 = 60

-4 5 1000
-4 + 5 + 1000 = 1001

The total is >= 100!
```

```
Try again, please.
0 0 0
```

28

signal.h

The `signal.h` header file lists the software interrupt specifications.

Overview of signal.h

Signals are software interrupts. There are signals for aborting a program, floating point exceptions, illegal instruction traps, user-signaled interrupts, segment violation, and program termination.

- Table 28.1 lists the macros in the `signal.h` file.
- "signal" on page 251 specifies how a signal is handled: a signal can be ignored, handled in a default manner, or be handled by a programmer-supplied signal handling function.
- "raise" on page 253 calls the signal handling function.
- "Signal Function Handling Arguments" on page 251 describes the pre-defined signal handling macros that expand to functions.

Signal handling

Signals are invoked, or raised, using the `raise()` function. When a signal is raised its associated function is executed.

With the MSL C implementation of `signal.h`, a signal can only be invoked through the function "raise" on page 253, and, in the case of the `SIGABRT` signal, through the function "abort" on page 405. When a signal is raised, its signal handling function is executed as a normal function call.

The default signal handler for all signals except `SIGTERM` is `SIG_DFL`. The `SIG_DFL` function aborts a program with the `abort()` function, while the `SIGTERM` signal terminates a program normally with the `exit()` function.

The ANSI C Standard Library specifies that the SIG prefix used by the signal.h macros is reserved for future use. The programmer should avoid using the prefix to prevent conflicts with future specifications of the Standard Library.

The type typedef `char sig_atomic_t` in `signal.h` can be accessed as an incorruptible, atomic entity during an asynchronous interrupt.

The number of signals is defined by `__signal_max` given a value in this header.

CAUTION Using unprotected re-entrant functions such as printf(), getchar(), malloc(), etc. functions from within a signal handler is not recommended in any system that can throw signals in hardware. Signals are in effect interrupts, and can happen anywhere, including when you're already within a function. Even functions that protect themselves from re-entry in a multi-threaded case can fail if you re-enter them from a signal handler.

Table 28.1 signal.h Macros

Macro	Details
SIGABRT	Abort signal. This macro is defined as a positive integer value. This signal is called by the `abort()` function.
SIGBREAK	Terminates calling program.
SIGFPE	Floating point exception signal. This macro is defined as a positive integer value.
SIGILL	Illegal instruction signal. This macro is defined as a positive integer value.
SIGINT	Interactive user interrupt signal. This macro is defined as a positive integer value.
SIGSEGV	Segment violation signal. This macro is defined as a positive integer value.
SIGTERM	Terminal signal. This macro is defined as a positive integer value. When raised this signal terminates the calling program by calling the exit() function.

The `signal()` function specifies how a signal is handled: a signal can be ignored, handled in a default manner, or be handled by a programmer-supplied signal handling function. Table 28.2 lists the pre-defined signal handling macros.

Table 28.2 Signal Function Handling Arguments

Macro	Description
SIG_IGN	This macro expands to a pointer to a function that returns void. It is used as a function argument in `signal()` to designate that a signal be ignored.
SIG_DFL	This macro expands to a pointer to a function that returns void. This signal handler quits the program without flushing and closing open streams.
SIG_ERR	A macro defined like `SIG_IGN` and `SIG_DFL` as a function pointer. This value is returned when signal() cannot honor a request passed to it.

signal

Set signal handling.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Table 28.3 signal

sig	int	A number associated with the signal handling function
func	void *	A pointer to a signal handling function

Remarks

The `signal()` function returns a pointer to a signal handling routine that takes an `int` value argument. For example usage, see Listing 28.1.

The sig argument is the signal number associated with the signal handling function. The signals defined in `signal.h` are listed in Table 28.1.

The func argument is the signal handling function. This function is either programmer-supplied or one of the pre-defined signal handlers described in "Signal Function Handling Arguments" on page 251.

When it is raised, a signal handler's execution is preceded by the invocation of `signal(sig, SIG_DFL)`. This call to `signal()` effectively disables the user's handler. It can be reinstalled by placing a call within the user handler to `signal()` with the user's handler as its function argument.

`signal()` returns a pointer to the signal handling function set by the last call to `signal()` for signal sig. If the request cannot be honored, `signal()` returns `SIG_ERR`.

This function may not be implemented on all platforms.

See Also

"raise" on page 253

"abort" on page 405

"atexit" on page 408

"exit" on page 420

Listing 28.1 Example of signal() Usage

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

void userhandler(int);

void userhandler(int sig)
{
   char c;

   printf("userhandler!\nPress return.\n");

   /* wait for the return key to be pressed */
   c = getchar();
}
int main(void)
{
   void (*handlerptr)(int);
   int i;

   handlerptr = signal(SIGINT, userhandler);
   if (handlerptr  == SIG_ERR)
      printf("Can't assign signal handler.\n");
```

```
   for (i = 0; i < 10; i++) {
   printf("%d\n", i);
      if (i == 5) raise(SIGINT);
   }

   return 0;
}
```

```
Output:

0
1
2
3
4
5
userhandler!
Press return.

6
7
8
9
```

raise

Raise a signal.

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

Table 28.4 raise

sig	int	A signal handling function

Remarks

The `raise()` function calls the signal handling function associated with signal sig.

`raise()` returns a zero if the signal is successful; it returns a nonzero value if it is unsuccessful.

This function may not be implemented on all platforms.

See Also

"longjmp" on page 244
"signal" on page 251
"abort" on page 405
"atexit" on page 408
"exit" on page 420

29

SIOUX.h

The SIOUX (Simple Input and Output User eXchange) libraries handle Graphical User Interface issues. Such items as menus, windows, and events are handled so your program doesn't need to for C and C++ programs.

Overview of SIOUX

The following section describes the Macintosh versions of the console emulation interface known as SIOUX. The facilities and structure members for the Standard Input Output User eXchange console interface are "Using SIOUX" and "SIOUX for Macintosh".

- "Using SIOUX" on page 255 is a general description of SIOUX properties.
- "SIOUX for Macintosh" on page 256 explains the (Simple Input and Output User eXchange) library for the Macintosh Operating Systems.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4 for information on `POSIX` naming conventions.

Using SIOUX

Sometimes you need to port a program that was originally written for a command line interface such as DOS or UNIX. Or you need to write a new program quickly and don't have the time to write a complete Graphical User Interface that handles windows, menus, and events.

To help you, CodeWarrior provides you with the SIOUX libraries, which handles all the Graphical User Interface items such as menus, windows, and titles so your program doesn't need to. It creates a window that's much like a dumb terminal or TTY but with scrolling. You can write to it and read from it with the standard C functions and C++ operators, such as printf(), scanf(), getchar(), putchar() and the C++ inserter and extractor operators << and >>. The SIOUX and WinSIOUX

libraries also creates a File menu that lets you save and print the contents of the window. The Macintosh hosted SIOUX includes an Edit menu that lets you cut, copy, and paste the contents in the window. For information on Macintosh redirecting to or from file the stdin, stdout, cout, and cin input output or commandline arguments.

Macintosh only—this function may not be implemented on all Mac OS versions.

See Also

"Overview of console.h" on page 41.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

SIOUX for Macintosh

SIOUX for Macintosh contains the following segments.

- "Creating a Project with SIOUX" on page 257 shows a running SIOUX program.
- "Customizing SIOUX" on page 258 shows how to customize your SIOUX window.
  - "The SIOUXSettings Structure" on page 259 list structure members that may be set for altering SIOUX's appearance
- "Using SIOUX windows in your own application" on page 264 contains information for using Mac OS facilities with in your SIOUX project.
  - "path2fss" on page 265 a function similar to PBMakeFSSpec.
  - "SIOUXHandleOneEvent" on page 265 allows you to use an even in SIOUX
  - "SIOUXSetTitle" on page 267 allows you to specify a custom title for SIOUX's window

NOTE A **WASTE**© by **Marco Piovanelli** based SIOUX console is available as a pre-release version. This will allow screen output of over 32k characters. All normal SIOUX functions should work but normal pre-release precautions should be taken. Please read all release notes.

The window is a re-sizable, scrolling text window, where your program reads and writes text. It saves up to 32K of your program's text.

With the commands from the Edit menu, you can cut and copy text from the SIOUX window and paste text from other applications into the SIOUX window. With the commands in the File menu, you can print or save the contents of the SIOUX window.

To stop your program at any time, press Command-Period or Control-C. The SIOUX application keeps running so you can edit or save the window's contents. If you want to exit when your program is done or avoid the dialog asking whether to save the window, see "Changing what happens on quit" on page 263

To quit out of the SIOUX application at any time, choose Quit from the File menu. If you haven't saved the contents of the window, the application displays a dialog asking you whether you want to save the contents of the window now. If you want to remove the status line, see "Showing the status line" on page 264.

Creating a Project with SIOUX

To use the SIOUX library, create a project from a project stationery pads that creates an Console style project.

In this chapter, standard input and standard output refer to `stdin`, `stdout`, `cin`, and `cout`. Standard error reporting such as `stderr`, `clog`, and `cerr` is not redirected to a file using `ccommand()`.

If you want only to write to or read from standard input and output, you don't need to call any special functions or include any special header files. When your program refers to standard input or output, the SIOUX library kicks in automatically and creates a SIOUX window for it.

NOTE Remember that functions like `printf()` and `scanf()` use standard input and output even though these symbols do not appear in their parameter lists.

If you want to customize the SIOUX environment, you must #include `SIOUX.h` and modify `SIOUXSettings` before you use standard input or output. As soon as you use one of them, SIOUX creates a window and you cannot modify it. For more information, see "Customizing SIOUX" on page 258.

If you want to use a SIOUX window in a program that has its own event loop, you must modify `SIOUXSettings` and call the function `SIOUXHandleOneEvent()`. For more information, see "Using SIOUX windows in your own application" on page 264.

If you want to add SIOUX to a project you already created, the project must contain certain libraries.

A PPC project must either contain at least these libraries:

- MSL_All_.PPC.Lib
- InterfaceLib
- MathLib

Or at least:

- MSL C.PPC.Lib

- MSL C++.PPC.Lib (for C++)
- MSL_SIOUX_PPC.Lib
- MSL_Runtime_PPC.Lib
- InterfaceLib
- MathLib

A Carbon project must either contain at least these libraries:

- MSL_All_Carbon.Lib
- CarbonLib

Or at least:

- MSL_C_Carbon.Lib
- MSL_C++_Carbon.Lib (for C++)
- MSL_SIOUX_Carbon.Lib
- CarbonLib

A Mach-O project must contain at least these libraries:

- MSL_All_Mach-O.lib
- MSL_SIOUX_Mach-O.lib (to be implemented)

Or at least:

- MSL_C_Mach-O.lib
- MSL_C++_Mach-O.lib (for C++)
- MSL_SIOUX_Mach-O.lib (to be implemented)
- MSL_Runtime_Mach-O.lib

Customizing SIOUX

This following sections describe how you can customize the SIOUX environment by modifying the structure `SIOUXSettings`. SIOUX examines the data fields of `SIOUXSettings` to determine how to create the SIOUX window and environment.

NOTE To customize SIOUX, you must modify `SIOUXSettings` before you call any function that uses standard input or output. If you modify `SIOUXSettings` afterwards, SIOUX does not change its window.

The first three sections, "Changing the font and tabs" on page 261, "Changing the size and location" on page 262, and "Showing the status line" on page 264, describe how to customize the SIOUX window. The next section, "Changing what happens on quit" on page 263, describe how to modify how SIOUX acts when you quit it. The last section,

"Using SIOUX windows in your own application" on page 264, describes how you can use a SIOUX window in your own Macintosh program.

Table 29.1 summarizes what's in the `SIOUXSettings` structure.

Table 29.1 The SIOUXSettings Structure

This field...		Specifies...
char	initializeTB	Whether to initialize the Macintosh toolbox.
char	standalone	Whether to use your own event loop or SIOUX's.
char	setupmenus	Whether to create File and Edit menus for the application.
char	autocloseonquit	Whether to quit the application automatically when your program is done.
char	asktosaveonclose	Query the user whether to save the SIOUX output as a file, when the program is done.
char	showstatusline	Whether to draw the status line in the SIOUX window.
short	tabspaces	If greater than zero, substitute a tab with that number of spaces. If zero, print the tabs.
short	column	The number of characters per line that the SIOUX window will contain.
short	rows	The number of lines of text that the SIOUX window will contain.
short	toppixel	The location of the top of the SIOUX window.
short	leftpixel	The location of the left of the SIOUX window.

Table 29.1 The SIOUXSettings Structure (*continued*)

This field…		Specifies…
short	fontid	The font in the SIOUX window.
short	fontsize	The size of the font in the SIOUX window.
short	stubmode	SIOUX acts like a stubs library
char	usefloatingwindows	(Carbon) use non floating front window
short	fontface	The style of the font in the SIOUX window.
int	sleep	The default value for the `sleep` setting is zero. Zero gets the most speed out of `SIOUX` by telling the system to not give time to other processes during a WaitNextEvent call. A more appropriate setting (that is more friendly to other processes) is to set the `sleep` value to `GetCaretTime()`.

Listing 29.1 contains a small program that customizes a SIOUX window.

Listing 29.1 Example of Customizing a SIOUX Window

```
#include <stdio.h>
#include <sioux.h>
#include <MacTypes.h>
#include <Fonts.h>

int main(void)
{
   short familyID;

   /* Don't exit the program after it runs or ask whether
      to save the window when the program exit */
   SIOUXSettings.autocloseonquit = false;
```

```
    SIOUXSettings.asktosaveonclose = false;

    /* Don't show the status line */
    SIOUXSettings.showstatusline = false;

    /* Make the window large enough to fit 1 line
       of text that contains 12 characters. */
    SIOUXSettings.columns = 12;
    SIOUXSettings.rows = 1;

    /* Place the window's top left corner at (5,40).  */
    SIOUXSettings.toppixel = 40;
    SIOUXSettings.leftpixel = 5;

    /* Set the font to be 48-point, bold, italic Times.   */
    SIOUXSettings.fontsize = 48;
    SIOUXSettings.fontface = bold + italic;
    GetFNum("\ptimes", &familyID);
    SIOUXSettings.fontid = familyID;

    printf("Hello World!");

    return 0;
}
```

Changing the font and tabs

This section describes how to change how SIOUX handles tabs with the field `tabspaces` and how to change the font with the fields `fontid`, `fontsize`, and `fontface`.

NOTE The status line in the SIOUX window writes its messages with the font specified in the fields `fontid`, `fontsize`, and `fontface`. If that font is too large, the status line may be unreadable. You can remove the status line by setting the field `showstatusline` to `false`, as described in "Showing the status line" on page 264.

To change the font in the SIOUX window, set fontid to one of these values defined in the header file Fonts.h:

- `courier where the ID is kFontIDCourier`
- `geneva where the ID is kFontIDGeneva`
- `helvetica where the ID is kFontIDHelvetica`
- `monaco where the ID is kFontIDMonaco`

- `newYork where the ID is kFontIDNewYork`
- `symbol where the ID is kFontIDSymbol`
- `times where the ID is kFontIDTimes`

By default, fontid is monaco.

To change the character style for the font, set fontface to one of these values:

- `normal`
- `bold`
- `italic`
- `underline`
- `outline`
- `shadow`
- `condense`
- `extend`

To combine styles, add them together. For example, to write text that's bold and italic, set `fontface` to `bold + italic`. By default, `fontface` is `normal`.

To change the size of the font, set `fontsize` to the size. By default, `fontsize` is `9`.

The field `tabspaces` controls how SIOUX handles tabs. If `tabspaces` is any number greater than `0`, SIOUX prints that number of spaces required to get to the next tab position instead of a tab. If `tabspaces` is `0`, it prints a tab. In the SIOUX window, a tab looks like a single space, so if you are printing a table, you should set `tabspaces` to an appropriate number, such as `4` or `8`. By default, `tabspaces` is `4`.

The sample below sets the font to 12-point, bold, italic New York and substitutes 4 spaces for every tab:

```
SIOUXSettings.fontsize = 12;
SIOUXSettings.fontface = bold + italic;
SIOUXSettings.fontid = kFontIDNewYork;
SIOUXSettings.tabspaces = 4;
```

Changing the size and location

SIOUX lets you change the size and location of the SIOUX window.

To change the size of the window, set `rows` to the number of lines of text in the window and set `columns` to the number of characters in each line. SIOUX checks the font you specified in `fontid`, `fontsize`, and `fontface` and creates a window that will be large enough to contain the number of lines and characters you specified. If the window is

too large to fit on your monitor, SIOUX creates a window only as large as the monitor can contain.

For example, the code below creates a window that contains 10 lines with 40 characters per line:

SIOUXSettings.rows = 10;

SIOUXSettings.columns = 40;

By default, the SIOUX window contains 24 rows with 80 characters per row.

To change the position of the SIOUX window, set `toppixel` and `leftpixel` to the point where you want the top left corner of the SIOUX window to be. By setting `toppixel` to `38` and `leftpixel` to 0, you can place the window as far left as possible and just under the menu bar. Notice that if `toppixel` is less than `38`, the SIOUX window is under the menu bar. If `toppixel` and `leftpixel` are both 0, SIOUX doesn't place the window at that point but instead centers it on the monitor.

For example, the code below places the window just under the menu bar and near the left edge of the monitor:

```
SIOUXSettings.toppixel = 40;
SIOUXSettings.leftpixel = 5;
```

Changing what happens on quit

The fields `autocloseonquit` and `asktosaveonclose` let you control what SIOUX does when your program is over and SIOUX closes its window.

The field `autocloseonquit` determines what SIOUX does when your program has finished running. If `autocloseonquit` is `true`, SIOUX automatically exits. If `autocloseonquit` is `false`, SIOUX continues to run, and you must choose Quit from the File menu to exit. By default, `autocloseonquit` is `false`.

NOTE You can save the contents of the SIOUX window at any time by choosing Save from the File menu.

The field `asktosaveonclose` determines what SIOUX does when it exits. If `asktosaveonclose` is `true`, SIOUX displays a dialog asking whether you want to save the contents of the SIOUX window. If `asktosaveonclose` is `false`, SIOUX exits without displaying the dialog. By default, `asktosaveonclose` is `true`.

For example, the code below quits the SIOUX application as soon as your program is done and doesn't ask you to save the output:

```
SIOUXSettings.autocloseonquit = true;
SIOUXSettings.asktosaveonclose = false;
```

Showing the status line

The field `showstatusline` lets you control whether the SIOUX window displays a status line, which contains such information as whether the program is running, handling output, or waiting for input. If `showstatusline` is `true`, the status line is displayed. If `showstatusline` is `false`, the status line is not displayed. By default, `showstatusline` is `false`.

Using SIOUX windows in your own application

This section explains how you can limit how much SIOUX controls your program. But first, you need to understand how SIOUX works with your program. You can consider the SIOUX environment to be an application that calls your `main()` function as just another function. Before SIOUX calls `main()`, it performs some initialization to set up the Macintosh Toolbox and its menu. After `main()` completes, SIOUX cleans up what it created. Even while `main()` is running, SIOUX sneaks in whenever it performs input or output, acting on any menu you've chosen or command key you've pressed.

However, SIOUX lets you choose how much work it does for you. You can choose to handle your own events, set up your own menus, and initialize the Macintosh Toolbox yourself.

When you want to write an application that handles its own events and uses SIOUX windows for easy input and output, set the field `standalone` to `false` before you use standard input or output. SIOUX doesn't use its event loop and sets the field `autocloseonquite` to `true` for you, so the application exits as soon as your program is done. In your event loop, you need to call the function `SIOUXHandleOneEvent()`, described on "Using SIOUX windows in your own application" on page 264.

When you don't want to use SIOUX's menus, set the field `setupmenus` to `false`. If `standalone` is also `false`, you won't be able to create menus, and your program will have none. If `standalone` is `true`, you can create and handle your own menus.

When you want to initialize the Macintosh Toolbox yourself, set the field `initializeTB` to `false`. The field `standalone` does not affect `initializeTB`.

For example, these lines set up SIOUX for an application that handles its own events, creates its own menus, and initializes the Toolbox:

```
SIOUXSettings.standalone = false;
SIOUXSettings.setupmenus = false;
SIOUXSettings.initializeTB = false;
```

path2fss

This function is similar to PBMakeFSSpec.

```
#include <path2fss.h>
OSErr __path2fss
   (const char * pathName, FSSpecPtr spec)
```

Table 29.2 path2fss

pathname	const char *	The path name
spec	FSSpecPtr	A file specification pointer

Remarks

This function is similar to PBMakeFSSpec with four major differences:

- Takes only a path name as input (as a C string) no parameter block.
- Only makes FSSpecs for files, not directories.
- Works on *any* HFS Mac (Mac 512KE, Mac Plus or later) under any system version that supports HFS.
- Deals correctly with MFS disks (correctly traps file names longer than 63 chars and returns bdNamErr).

Like PBMakeFSSpec, this function returns fnfErr if the specified file does not exist but the FSSpec is still valid for the purposes of creating a new file.

Errors are returned for invalid path names or path names that specify directories rather than files.

Macintosh only—this function may not be implemented on all Mac OS versions.

SIOUXHandleOneEvent

Handles an event for a SIOUX window.

```
#include <SIOUX.h>
Boolean SIOUXHandleOneEvent(EventRecord *event);
```

Table 29.3 SIOUXHandleOneEvent

event	EventRecord*	A pointer to a toolbox event

Remarks

Before you handle an event, call `SIOUXHandleOneEvent()` so SIOUX can update its windows when necessary. The argument `event` should be an event that `WaitNextEvent()` or `GetNextEvent()` returned. The function returns `true` if it handled the event and `false` if it didn't. If `event` is a `NULL` pointer, the function polls the event queue until it receives an event.

If it handles the event, `SIOUXHandleOneEvent()` returns `true`. Otherwise, `SIOUXHandleOneEvent()` returns `false`.

Macintosh only—this function may not be implemented on all Mac OS versions.

Listing 29.2 Example of SIOUXHandleOneEvent() Usage

```
void MyEventLoop(void)
{
   EventRecord event;
   RgnHandle cursorRgn;
   Boolean gotEvent, SIOUXDidEvent;

   cursorRgn = NewRgn();

   do {
      gotEvent = WaitNextEvent(everyEvent, &event,
                  MyGetSleep(), cursorRgn);

      /* Before handling the event on your own,
       * call SIOUXHandleOneEvent() to see whether
       * the event is for SIOUX.
       */
      if (gotEvent)
         SIOUXDidEvent = SIOUXHandleOneEvent(&event);

    if (!SIOUXDidEvent)
      DoEvent(&event);

   } while (!gDone)
}
```

SIOUXSetTitle

To set the title of the SIOUX output window.

```
include <SIOUX.h>
extern void SIOUXSetTitle(unsigned char title[256])
```

Table 29.4 SIOUXSetTitle

title	unsigned char []	A pascal string

Remarks

You must call the SIOUXSetTitle() function after an output to the SIOUX window. The function SIOUXSetTitle() does not return an error if the title is not set. A write to console is not performed until a new line is written, the stream is flushed or the end of the program occurs.

NOTE The argument for SIOUXSetTitle() is a pascal string, not a C style string.

There is no return value from SIOUXSetTitle()

Macintosh only—this function may not be implemented on all Mac OS versions.

Listing 29.3 Example of SIOUXSetTitle() Usage

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

int main(void)
{
   printf("Hello World\n");
   SIOUXSetTitle("\pMy Title");

   return 0;
}
```

30

stat.h

The `stat.h` header file contains several functions that are useful for porting a program from UNIX.

Overview of stat.h

This header file defines the facilities as follows:

- "Stat Structure and Definitions" on page 269 explains the stat struct and types.
- "chmod" on page 272 gets or sets a files attributes.
- "fstat" on page 273 gets information about an open file.
- "mkdir" on page 275 makes a directory for folder.
- "stat" on page 276 gets statistics of a file.

stat.h and UNIX Compatibility

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4 for information on `POSIX` naming conventions.

Stat Structure and Definitions

The header stat.h includes the `stat` (and `_stat`, for Windows) structure, listed in Table 30.2. It also has several type definitions and file mode definitions. The necessary types are listed in Table 30.1. The file modes are listed in Table 30.3. File mode macros are listed in Table 30.4 and Table 30.5.

Table 30.1 Defined Types

Type	Used to Store
dev_t	Device type
gid_t	The file size
ino_t	File information
mode_t	File Attributes
nlink_t	The number of links
off_t	The file size in bytes
uid_t	The user's ID

Table 30.2 The stat or _stat Structure

Type	Variable	Purpose
mode_t	st_mode	File mode, see "File Modes" on page 271
ino_t	st_ino	File serial number
dev_t	st_dev	ID of device containing this file
dev_t	std_rdev (Windows)	ID of device containing this file
nlink_t	st_nlink	Number of links
uid_t	st_uid	User ID of the file's owner
gid_t	st_gid	Group ID of the file's group
off_t	st_size	File size in bytes
__std(time_t)	st_atime	Time of last access
__std(time_t)	st_mtime	Time of last data modification
__std(time_t)	st_ctime	Time of last file status change

Table 30.2 The stat or _stat Structure (*continued*)

Type	Variable	Purpose
long	st_blksize	Optimal blocksize
long	st_blocks	Blocks allocated for file

File Modes

File mode information.

Table 30.3 File Modes

File Mode	Purpose
S_IFMT	File type
S_IFIFO	FIFO queue
S_IFCHR	Character special
S_IFDIR	Directory
S_IFBLK	Blocking stream (non Windows)
S_IFREG	Regular
S_IFLNK	Symbolic link (non Windows)
S_IFSOCK	Socket (non Windows)

Table 30.4 File Mode Macros Non-Windows

File Mode	Purpose
S_IRGRP	Read permission file group class
S_IROTH	Read permission file other class
S_IRUSR	Read permission file owner class
S_IRWXG	Permissions for file group class
S_IRWXO	Permissions for file other class
S_IRWXU	Permissions for file owner class

Table 30.4 File Mode Macros Non-Windows (*continued*)

File Mode	Purpose
S_ISGID	Set group ID on execution
S_ISUID	Set user ID on execution
S_IWGRP	Write permission file group class
S_IWOTH	Write permission file other class
S_IWUSR	Write permission file owner class
S_IXGRP	Exec permission file group class
S_IXOTH	Exec permission file other class
S_IXUSR	Exec permission file owner class

Table 30.5 File Mode Macros Windows Only

File Mode	Purpose
S_IEXEC	Execute/search permission, owner (Windows)
S_IREAD	Read permission, owner (Windows Only)
S_IWRITE	Write permission, owner (Windows Only)

chmod

Gets or sets file attributes.

```
#include <stat.h>
int chmod(const char *, mode_t);
int _chmod(const char *, mode_t);
```

Table 30.6 chmod

path	const char *	The file to change modes on
mod	mode_t	A mask of the new file attributes

Remarks

The file attributes as a mask is returned or a negative one on failure.

This function may not be implemented on all platforms.

See Also

"fstat" on page 273

fstat

Gets information about an open file.

```
#include <stat.h>
int fstat(int fildes, struct stat *buf);
int _fstat(int fildes, struct stat *buf);
```

Table 30.7 fstat

fildes	int	A file descriptor
buf	struct stat *	The stat structure address

Remarks

This function gets information on the file associated with `fildes` and puts the information in the structure that `buf` points to. The structure contains the fields listed in "Stat Structure and Definitions" on page 269.

If it is successful, `fstat()` returns zero. If it encounters an error, `fstat()` returns `-1` and sets `errno`.

This function may not be implemented on all platforms.

See Also

"stat" on page 276

Listing 30.1 Example of fstat() Usage

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

int main(void)
{
   struct stat info;
   int fd;

   fd = open("mytest", O_WRONLY | O_CREAT | O_TRUNC);
   write(fd, "Hello world!\n", 13);

   fstat(fd, &info);
      /* Get information on the open file.                */

   printf("File mode:          0x%lX\n", info.st_mode);
   printf("File ID:            0x%lX\n", info.st_ino);
   printf("Volume ref. no.:    0x%lX\n", info.st_dev);
   printf("Number of links:    %hd\n", info.st_nlink);
   printf("User ID:            %lu\n", info.st_uid);
   printf("Group ID:           %lu\n", info.st_gid);
   printf("File size:          %ld\n", info.st_size);
   printf("Access time:        %s", ctime(&info.st_atime));
   printf("Modification time: %s", ctime(&info.st_mtime));
   printf("Creation time:      %s", ctime(&info.st_ctime));

   close(fd);

   return 0;
}
```

```
This program may print the following:
File mode:          0x800
File ID:            0x5ACA
Volume ref. no.:    0xFFFFFFFF
Number of links:    1
User ID:            200
Device type:        0
File size:          13
```

mkdir

Makes a folder.

```
#include <stat.h>
int mkdir(const char *path, int mode);
int _mkdir(const char *path);
```

Table 30.8 mkdir

path	const char *	The path name
mode	int	The open mode (Not applicable for Windows)

Remarks

This function creates the new folder specified in `path`. It ignores the argument `mode`.

If it is successful, `mkdir()` returns zero. If it encounters an error, `mkdir()` returns `-1` and sets `errno`.

This function may not be implemented on all platforms.

See Also

"unlink" on page 535

"rmdir" on page 529

Listing 30.2 Example of mkdir() on Macintosh OS

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

int main(void)
{
   if( mkdir(":Asok", 0) == 0)
         printf("Folder Asok is created");

   return 0;
}
Windows
#include <stdio.h>
```

```
#include <stat.h>

int main(void)
{
   if( mkdir(".\\Asok") == 0)
         printf("Folder Asok is created");

   return 0;
}
```

```
Output
Creates a folder named Asok as a sub-folder of the current folder
```

stat

Gets information about a file.

```
#include <stat.h>
int stat(const char *path, struct stat *buf);
int _stat(const char *path, struct stat *buf);
```

Table 30.9 stat

path	const char *	The path name
buf	struct stat *	A pointer to the stat struct

Remarks

This function gets information on the file specified in `path` and puts the information in the structure that `buf` points to. The structure contains the fields listed in "Stat Structure and Definitions" on page 269.

If it is successful, `stat()` returns zero.

This function may not be implemented on all platforms.

See Also

"fstat" on page 273

"uname" on page 549

Listing 30.3 Example of stat() Usage

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

int main(void)
{
   struct stat info;

   stat("Akbar:System Folder:System", &info);
      /* Get information on the System file.              */

   printf("File mode:          0x%lX\n", info.st_mode);
   printf("File ID:            0x%lX\n", info.st_ino);
   printf("Volume ref. no.:    0x%lX\n", info.st_dev);
   printf("Number of links:    %hd\n", info.st_nlink);
   printf("User ID:            %lu\n", info.st_uid);
   printf("Group ID:           %lu\n", info.st_gid);
   printf("File size:          %ld\n", info.st_size);
   printf("Access time:        %s", ctime(&info.st_atime));
   printf("Modification time: %s", ctime(&info.st_mtime));
   printf("Creation time:      %s", ctime(&info.st_ctime));

   return 0;
}
```

```
This program may print the following:
File mode:          0x800
File ID:            0x4574
Volume ref. no.:    0x0
Number of links:    1
User ID:            200
Group ID:           100
File size:          30480
Access time:        Mon Apr 17 19:46:37 1995
Modification time: Mon Apr 17 19:46:37 1995
Creation time:      Fri Oct  7 12:00:00 1994
```

umask

Sets a UNIX style file creation mask.

```
#include <stat.h> /* Macintosh */
mode_t umask(mode_t cmask)
mode_t _umask(mode_t cmask)
```

Table 30.10 umask

cmask	mode_t	permission bitmask

Remarks

The function `umask` is used for calls to `open()`, `creat()` and `mkdir()` to turn off permission bits in the mode argument.

If `_MSL_POSIX true` and you are compiling for Macintosh or Windows `umask` returns `mode_t` and takes a `mode_t` otherwise it takes an `int` type.

NOTE The permission bits are not used on either the Mac nor Windows. The function is provided merely to allow compilation and compatibility.

The previous mask. Zero is returned for Mac and Windows operating systems. if _MSL_POSIX is on and you are on mac and win umask returns mode_t.

This function may not be implemented on all platforms.

See Also

"creat, _wcreate" on page 118

"open, _wopen" on page 121

"mkdir" on page 275

31

stdarg.h

The `stdarg.h` header file allows the creation of functions that accept a variable number of arguments.

Overview of stdarg.h

This header file defines the facilities as follows:

- "va_arg" on page 280 returns an argument value.
- "va_copy" on page 280 copies and initializes a variable argument list.
- "va_end" on page 281 cleans the stack to allow a proper function return.
- "va_start" on page 282 initializes the variable-length argument list.

Variable Arguments for Functions

The `stdarg.h` header file allows the creation of functions that accept a variable number of arguments.

A variable-length argument function is defined with an ellipsis (`...`) as its last argument. For example:

```
int funnyfunc(int a, char c, ...);
```

The function is written using the `va_list` type, the `va_start()`, `va_arg()` and the `va_end()` macros.

The function has a `va_list` variable declared within it to hold the list of function arguments. The macro °"va_start" on page 282 initializes the `va_list` variable and is called before gaining access to the arguments. The macro "va_arg" on page 280 returns each of the arguments in `va_list`. When all the arguments have been processed through `va_arg()`, the macro "va_end" on page 281 is called to allow a normal return from the function.

va_arg

Macro to return an argument value.

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

Table 31.1 va_arg

ap	va_list	A variable list
type	type	The type of the argument value to be obtained

Remarks

The `va_arg()` macro returns the next argument on the function's argument list. The argument returned has the type defined by *type*. The `ap` argument must first be initialized by the `va_start()` macro.

The `va_arg()` macro returns the next argument on the function's argument list of `type`.

This function may not be implemented on all platforms.

See Also

"va_end" on page 281

"va_start" on page 282

For example of va() usage, see "Example of va_start() Usage" on page 282.

va_copy

Copies and initializes a variable argument list.

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src)
```

Table 31.2 va_copy

dest	va_list	The va_list being initialized
src	va_list	The source va_list being copied

Remarks

The `va_copy()` macro makes a copy of the variable list `src` in a state as if the `va_start` macro had been applied to it followed by the same sequence of `va_arg` macros as had been applied to `src` to bring it into its present state

There is no return for this facility.

This function may not be implemented on all platforms.

See Also

"Variable Arguments for Functions" on page 279

va_end

Prepare a normal function return.

```
#include <stdarg.h>
void va_end(va_list ap);
```

Table 31.3 va_end

ap	va_list	A variable list

Remarks

The `va_end()` function cleans the stack to allow a proper function return. The function is called after the function's arguments are accessed with the `va_arg()` macro.

This function may not be implemented on all platforms.

See Also

"va_arg" on page 280

"va_start" on page 282

va_start

Initialize the variable-length argument list.

```
#include <stdarg.h>
void va_start(va_list ap, ParmN Parm);
```

Table 31.4 va_start

ap	va_list	A variable list
Parm	ParmN	The last named parameter

Remarks

The `va_start()` macro initializes and assigns the argument list to `ap`. The `ParmN` parameter is the last named parameter before the ellipsis (`...`) in the function prototype. For example usage, see Listing 31.1.

This function may not be implemented on all platforms.

See Also

"va_arg" on page 280

"va_end" on page 281

Listing 31.1 Example of va_start() Usage

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

void multisum(int *dest, ...);

int main(void)
{
   int all;

   all = 0;
   multisum(&all, 13, 1, 18, 3, 0);
   printf("%d\n", all);

   return 0;
}
```

```
void multisum(int *dest, ...)
{
   va_list ap;
   int n, sum = 0;

   va_start(ap, dest);

   while ((n = va_arg(ap, int)) != 0)
      sum += n;     /* add next argument to dest */
   *dest = sum;
   va_end(ap);     /* clean things up before leaving */
}
```

```
 Output:
3
```

32

stdbool.h

The `stdbool.h` header file defines types used for boolean integral values.

Overview of stdbool.h

The `stdbool.h` header file consists of definitions in Table 32.1 only if the compiler support for c99 has been turned on using the C99 pragma.

```
#pragma c99 on | off | reset
```

Table 32.1 Defines in stdbool.h

bool	_Bool
true	1
false	0
__bool_true_false_are_defined	1

Remarks

There are no other defines in this header.

This function may not be implemented on all platforms.

33

stddef.h

The `stddef.h` header file defines commonly used macros and types that are used throughout the ANSI C Standard Library.

Overview of stddef.h

The commonly used definitions macros and types are defined in `stddef.h`

- "NULL" on page 287 defines NULL.
- "offsetof" on page 287 is the offset of a structure's member.
- "ptrdiff_t" on page 288 is used for pointer differences.
- "size_t" on page 288 is the return from a size of operation.
- "wchar_t" on page 288 is a wide character type.

NULL

The `NULL` macro is the null pointer constant used in the Standard Library.

Remarks

This definition may not be implemented on all platforms.

offsetof

The `offsetof(structure, member)` macro expands to an integral expression of type `size_t`. The value returned is the offset in bytes of a member, from the base of its `structure`.

NOTE If the member is a bit field the result is undefined.

Remarks

This macro may not be implemented on all platforms.

ptrdiff_t

The `ptrdiff_t` type is the signed integral type used for holding the result of subtracting one pointer's value from another.

Remarks

This type may not be implemented on all platforms.

size_t

The `size_t` type is an unsigned integral type returned by the `sizeof()` operator.

Remarks

This type may not be implemented on all platforms.

wchar_t

The `wchar_t` type is an integral type capable of holding all character representations of the wide character set.

Remarks

This type may not be implemented on all platforms.

34

stdint.h

The header `stdint.h` defines types used for standard integral values.

Overview of stdint.h

The `stdint.h` header file consists of integer types listed as follows:

- "Integer Types" on page 289
- "Limits of Specified-width Integer Types" on page 291
- "Macros for Integer Constants" on page 295
- "Macros for Greatest-width Integer Constants" on page 295

NOTE The types in this header may not be implemented on all platforms.

Integer Types

The header `stdint.h` contains several integer types.

- Table 34.1, "Exact Width Integer Type"
- Table 34.2, "Minimum Width Integer Type"
- Table 34.3, "Fastest Minimum-Width Integer Types"
- Table 34.4, "Integer Types Capable of Holding Object Pointers"
- Table 34.5, "Greatest Width Integer Types"
- Table 34.6, "Mac OS X Specific Integer Types"

Table 34.1 Exact Width Integer Type

Type	Equivalent	Type	Equivalent
int8_t	signed char	int16_t	short int
int32_t	long int	uint8_t	unsigned char

Table 34.1 Exact Width Integer Type (*continued*)

Type	Equivalent	Type	Equivalent
uint16_t	unsigned short int	uint32_t	unsigned long int
int64_t	long long	uint64_t	unsigned long long

Table 34.2 Minimum Width Integer Type

Type	Equivalent	Type	Equivalent
int_least8_t	signed char	int_least16_t	short int
int_least32_t	long int	uint_least8_t	unsigned char
uint_least16_t	unsigned short int	uint_least32_t	unsigned long int
int_least64_t	long long	uint_least64_t	unsigned long long

Table 34.3 Fastest Minimum-Width Integer Types

Type	Equivalent	Type	Equivalent
int_fast8_t	signed char	int_fast16_t	short int
int_fast32_t	long int	uint_fast8_t	unsigned char
uint_fast16_t	unsigned short int	uint_fast32_t	unsigned long int
int_fast64_t	long long	uint_fast64_t	unsigned long long

Table 34.4 Integer Types Capable of Holding Object Pointers

Type	Equivalent	Type	Equivalent
intptr_t	int32_t	uintptr_t	uint32_t

Table 34.5 Greatest Width Integer Types

Type	Equivalent	Type	Equivalent
intmax_t	int64_t	uintmax_t	int32_t

Table 34.6 Mac OS X Specific Integer Types

Type	Equivalent	Type	Equivalent
u_int8_t	unsigned char	u_int16_t	unsigned short
u_int32_t	unsigned int	u_int64_t	unsigned long long
register_t	__std(int32_t)		

Limits of Specified-width Integer Types

The limits of exact-width integer types are defined in the following tables.

- Table 34.7, "Minimum Values of Exact Width Signed Integer Types"
- Table 34.8, "Maximum Values of Exact Width Signed Integer Types"
- Table 34.9, "Maximum Values of Exact Width Unsigned Integer Types"
- Table 34.10, "Minimum Values of Minimum Width Signed Integer Types"
- Table 34.11, "Maximum Values of Minimum Width Signed Integer Types"
- Table 34.12, "Maximum Values of Minimum Width Unsigned Integer Types"
- Table 34.13, "Minimum Values of Fastest Minimum Width Signed Integer Types"
- Table 34.14, "Maximum Values of Fastest Minimum Width Signed Integer Types"
- Table 34.15, "Maximum Values of Fastest Minimum Width Unsigned Integer Types"
- Table 34.16, "Minimum Value of Pointer Holding Signed Integer Types"
- Table 34.17, "Minimum Value of Greatest Width Signed Integer Type"
- Table 34.18, "Maximum Value of Greatest Width Signed Integer Type"
- Table 34.19, "Limits of Other Integer Types"

Table 34.7 Minimum Values of Exact Width Signed Integer Types

Type	Equivalent
INT8_MIN	SCHAR_MIN
INT16_MIN	SHRT_MIN
INT32_MIN	LONG_MIN
INT64_MIN	LLONG_MIN

Table 34.8 Maximum Values of Exact Width Signed Integer Types

Type	Equivalent
INT8_MAX	SCHAR_MAX
INT16_MAX	SHRT_MAX
INT32_MAX	LONG_MAX
INT64_MAX	LLONG_MAX

Table 34.9 Maximum Values of Exact Width Unsigned Integer Types

Type	Equivalent
UINT8_MAX	UCHAR_MAX
UINT16_MAX	USHRT_MAX
UINT32_MAX	ULONG_MAX
UINT64_MAX	ULLONG_MAX

Table 34.10 Minimum Values of Minimum Width Signed Integer Types

Type	Equivalent
INT_LEAST8_MIN	SCHAR_MIN
INT_LEAST16_MIN	SHRT_MIN
INT_LEAST32_MIN	LONG_MIN
INT_LEAST64_MIN	LLONG_MIN

Table 34.11 Maximum Values of Minimum Width Signed Integer Types

Type	Equivalent
INT_LEAST8_MAX	SCHAR_MAX
INT_LEAST16_MAX	SHRT_MAX

Table 34.11 Maximum Values of Minimum Width Signed Integer Types (*continued*)

Type	Equivalent
INT_LEAST32_MAX	LONG_MAX
INT_LEAST64_MAX	LLONG_MAX

Table 34.12 Maximum Values of Minimum Width Unsigned Integer Types

Type	Equivalent
UINT_LEAST8_MAX	UCHAR_MAX
UINT_LEAST16_MAX	USHRT_MAX
UINT_LEAST32_MAX	ULONG_MAX
UINT_LEAST64_MAX	ULLONG_MAX

Table 34.13 Minimum Values of Fastest Minimum Width Signed Integer Types

Type	Equivalent
INT_FAST8_MIN	SCHAR_MIN
INT_FAST16_MIN	SHRT_MIN
INT_FAST32_MIN	LONG_MIN
INT_FAST64_MIN	LLONG_MIN

Table 34.14 Maximum Values of Fastest Minimum Width Signed Integer Types

Type	Equivalent
INT_FAST8_MAX	SCHAR_MAX
INT_FAST16_MAX	SHRT_MAX
INT_FAST32_MAX	LONG_MAX
INT_FAST64_MAX	LLONG_MAX

Table 34.15 Maximum Values of Fastest Minimum Width Unsigned Integer Types

Type	Equivalent
UINT_FAST8_MAX	UCHAR_MAX
UINT_FAST16_MAX	USHRT_MAX
UINT_FAST32_MAX	ULONG_MAX
UINT_FAST64_MAX	ULLONG_MAX

Table 34.16 Minimum Value of Pointer Holding Signed Integer Types

Type	Equivalent
INTPTR_MIN	LONG_MIN
INTPTR_MAX	LONG_MAX
UINTPTR_MAX	ULONG_MAX

Table 34.17 Minimum Value of Greatest Width Signed Integer Type

Type	Equivalent
INTMAX_MIN	LLONG_MIN

Table 34.18 Maximum Value of Greatest Width Signed Integer Type

Type	Equivalent
UINTMAX_MAX	ULLONG_MAX

Table 34.19 Limits of Other Integer Types

Type	Equivalent
PTRDIFF_MIN	LONG_MIN
PTRDIFF_MAX	LONG_MAX

Table 34.19 Limits of Other Integer Types (*continued*)

Type	Equivalent
SIG_ATOMIC_MIN	INT_MIN
SIG_ATOMIC_MAX	INT_MAX

Macros for Integer Constants

The macros expand to integer constants suitable for initializing objects that have integer types.

```
INT8_C(value)    value
INT16_C(value)    value
INT32_C(value)    value ## L
INT64_C(value)    value ## LL
UINT8_C(value)     value ## U
UINT16_C(value)    value ## U
UINT32_C(value)    value ## UL
UINT64_C(value)    value ## ULL
```

Macros for Greatest-width Integer Constants

The `INTMAX_C` macro expands to an integer constant with the value of its argument and the type `intmax_t`.

```
INTMAX_C(value)    value ## LL
```

The `UINTMAX_C` macro expands to an integer constant with the value of its argument and the type `uintmax_t`.

```
UINTMAX_C(value)    value ## ULL
```

35

stdio.h

The `stdio.h` header file provides functions for input/output control.

Overview of stdio.h

The `stdio.h` header file provides functions for input/output control. There are functions for creating, deleting, and renaming files, functions to allow random access, as well as to write and read text and binary data.

This header file defines the facilities as follows:

- "clearerr" on page 302 clears an error from a stream.
- "fclose" on page 304 closes a file.
- "fdopen" on page 306 converts a file descriptor to a stream.
- "feof" on page 307 detects the end of a file.
- "ferror" on page 309 checks a file error status.
- "fflush" on page 310 flushes a stream.
- "fgetc" on page 312 gets a character from a file.
- "fgetpos" on page 314 gets a file position from large files.
- "fgets" on page 316 gets a string from a file.
- "_fileno" on page 317 gets the file number (Windows version only).
- "fopen" on page 317 opens a file for manipulation.
- "fprintf" on page 320 prints formatted output to a file.
- "fputc" on page 328 writes a character to a file.
- "fputs" on page 330 writes a string to a file.
- "fread" on page 331 reads a file.
- "freopen" on page 333 reopens a file.
- "fscanf" on page 335 scans a file.
- "fseek" on page 341 moves to a file position.
- "fsetpos" on page 343 sets a file position for large files.
- "ftell" on page 344 tells a file offset.

- "fwide" on page 345 determines a character orientation.
- "fwrite" on page 347 writes to a file.
- "getc" on page 348 gets a character from a stream.
- "getchar" on page 349 gets a character from stdin.
- "gets" on page 351 gets a string from stdin.
- "perror" on page 352 writes an error to stderr.
- "printf" on page 353 writes a formatted output to stdout.
- "putc" on page 362 writes a character to a stream.
- "putchar" on page 363 writes a character to stdout.
- "puts" on page 365 writes a string to stdout.
- "remove" on page 366 removes a file.
- "rename" on page 367 renames a file.
- "rewind" on page 368 resets the file indicator to the beginning.
- "scanf" on page 370 scans stdin for input.
- "setbuf" on page 375 sets the buffer size for a stream.
- "setvbuf" on page 377 sets the stream buffer size and scheme.
- "snprintf" on page 379 writes a number of characters to a buffer.
- "sprintf" on page 380 writes to a character buffer.
- "sscanf" on page 381 scans a string.
- "tmpfile" on page 382 creates a temporary file.
- "tmpnam" on page 384 creates a temporary name.
- "ungetc" on page 385 places a character back in a stream.
- "vfprintf" on page 387 writes variable arguments to file.
- "vfscanf" on page 389 a variable argument scanf.
- "vprintf" on page 391 writes variable arguments to stdout.
- "vsnprintf" on page 393 writes variable arguments to a char array buffer with a number limit.
- "vsprintf" on page 395 writes variable arguments to a char array buffer.
- "vsscanf" on page 397 reads formatted text from a character strin (variable scanf).
- "_wfopen" on page 399 opens a wide character file.
- "_wfreopen" on page 399 reopens a wide character file.
- "_wremove" on page 400 removes a wide character file.
- "_wrename" on page 401 renames wide character file.

- “ wtmpnam” on page 401 creates a temporary wide character file name.

Standard input/output

Streams

A stream is a logical abstraction that isolates input and output operations from the physical characteristics of terminals and structured storage devices. They provide a mapping between a program’s data and the data as actually stored on the external devices. Two forms of mapping are supported, for `text streams` and for `binary streams`. See “Text Streams and Binary Streams” on page 300, for more information.

Streams also provide buffering, which is an abstraction of a file designed to reduce hardware I/O requests. Without buffering, data on an I/O device must be accessed one item at a time. This inefficient I/O processing slows program execution considerably. The `stdio.h` functions use buffers in primary storage to intercept and collect data as it is written to or read from a file. When a buffer is full its contents are actually written to or read from the file, thereby reducing the number of I/O accesses. A buffer's contents can be sent to the file prematurely by using the `fflush()` function.

The `stdio.h` header offers three buffering schemes: unbuffered, block buffered, and line buffered. The `setvbuf()` function is used to change the buffering scheme of any output stream.

When an output stream is unbuffered, data sent to it are immediately read from or written to the file.

When an output stream is block buffered, data are accumulated in a buffer in primary storage. When full, the buffer's contents are sent to the destination file, the buffer is cleared, and the process is repeated until the stream is closed. Output streams are block buffered by default if the output refers to a file.

A line buffered output stream operates similarly to a block buffered output stream. Data are collected in the buffer, but are sent to the file when the line is completed with a newline character (`'\n'`).

A stream is declared using a pointer to a `FILE`. There are three `FILE` pointers that are automatically opened for a program: `FILE *stdin`, `FILE *stdout`, and `FILE *stderr`. The `FILE` pointers `stdin` and `stdout` are the standard input and output files, respectively, for interactive console I/O. The `stderr` file pointer is the standard error output file, where error messages are written to. The `stderr` stream is written to the console. The `stdin` and `stdout` streams are line buffered while the `stderr` stream is unbuffered.

Text Streams and Binary Streams

In a binary stream, there is no transformation of the characters during input or output and what is recorded on the physical device is identical to the program's internal data representation.

A text stream consists of sequence of characters organized into lines, each line terminated by a new-line character. To conform to the host system's convention for representing text on physical devices, characters may have to be added altered or deleted during input and output. Thus, there may not be a one-to-one correspondence between the characters in a stream and those in the external representation. These changes occur automatically as part of the mapping associated with text streams. Of course, the input mapping is the inverse of the output mapping and data that are output and then input through text streams will compare equal with the original data.

In MSL, the text stream mapping affects only the linefeed (LF) character, '\n' and the carriage return (CR) character, '\r'. The semantics of these two control characters are:

- \n Moves the current print location to the start of the next line.
- \r Moves the current print location to the start of the current line.

where "current print location "is defined as "that location on a display device where the next character output by the fputc function would appear".

The ASCII character set defines the value of LF as 0x0a and CR as 0x0d and these are the values that these characters have when they are part of a program's data. On physical devices in the Macintosh operating system, newline characters are represented by 0x0d and CR as 0x0a; in other words, the values are interchanged. To meet this requirement, the MSL C library for the Mac, interchanges these values while writing a file and again while reading so that a text stream will be unchanged by writing to a file and then reading back. MPW chose 0x0a for the newline character in its text file, so, when the MPW switch is on, this interchange of values does not take place. However, if you use this option, you must use the MSL C and C++ libraries that were compiled with this option on.

These versions of the libraries are marked with an N (on 68k) or NL (on PPC), for example ANSI (N/2i) C.68k.Lib or ANSI (NL) C.PPC.Lib. See the notes on the mpwc_newline pragma in the CodeWarrior C Compilers Reference.

On Windows, the situation is different. There, lines are terminated with the character pair CR/LF. As a consequence, in the Windows implementation of MSL, when a text stream is written to a file, a single newline character is converted to the character pair CR/LF and the reverse transformation is made during reading.

The library routines that read a file have no means of determining the mode in which text files were written and thus some assumptions have to be made. On the Mac, it is assumed that the Mac convention is used. Under MPW, it is assumed that the MPW convention is to be used and on Windows, the DOS convention.

File position indicator

The file position indicator is another concept introduced by the `stdio.h` header. Each opened stream has a file position indicator acting as a cursor within a file. The file position indicator marks the character position of the next read or write operation. A read or write operation advances the file position indicator. Other functions are available to adjust the indicator without reading or writing, thus providing random access to a file.

Note that console streams, `stdin`, `stdout`, and `stderr` in particular, do not have file position indicators.

End-of-file and errors

Many functions that read from a stream return the `EOF` value, defined in `stdio.h`. The `EOF` value is a nonzero value indicating that the end-of-file has been reached during the last read or write.

Some `stdio.h` functions also use the `errno` global variable. Refer to the `errno.h` header section. The use of `errno` is described in the relevant function descriptions below.

Wide Character and Byte Character Stream Orientation

There are two types of stream orientation for input and output, a wide character (wchar_t) oriented and a byte (char) oriented. A stream is without orientation after that stream has been associated with a file, until an operation occurs.

Once any operation is performed on that stream, that stream becomes oriented by that operation to be either byte oriented or wide character oriented and remains that way until the file has been closed and reopened.

After a stream orientation is established, any call to a function of the other orientation is not applied. That is, a byte-oriented input/output function does not have an effect on a wide-oriented stream.

Unicode

Unicode encoded characters are represented and manipulated in MSL as wide characters of type wchar_t and can be manipulated with the wide character functions defined in the C Standard.

Table 35.1 Byte Oriented Functions

fgetc	fgets	fprintf	fputc	fputs
fread	fscanf	fwrite	getc	getchar

Table 35.1 Byte Oriented Functions (*continued*)

gets	printf	putc	putchar	puts
scanf	ungetc	vfprintf	vfscanf	vprintf

Table 35.2 The Wide Character Oriented Functions in wchar.h

fgetwc	fgetws	fwprintf	fputwc	fputws	fwscanf
getwc	getwchar	putwc	putwchar	swprintf	swscanf
towctrans	vfwscanf	vswscanf	vwscanf	vfwprintf	vswprintf
vwprintf	wasctime	watof	wcscat	wcschr	wcscmp
wcscoll	wcscspn	wcscpy	wcslen	wcsncat	wcsncmp
wcsncpy	wcspbrk	wcsspn	wcsrchr	wcsstr	wcstod
wcstok	wcsftime	wcsxfrm	wctime	wctrans	wmemchr
wmemcmp	wmemcpy	wmemmove	wmemset	wprintf	wscanf

Stream Orientation and Standard Input/Output

The three predefined associated streams, stdin, stdout, and stderr are without orientation at program startup. If any of the standard input/output streams is closed it is not possible to reopen and reconnect that stream to the console. However, it is possible to reopen and connect the stream to a named file.

The C and C++ input/output facilities share the same stdin, stdout and stderr streams.

clearerr

Clear a stream's end-of-file and error status.

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Table 35.3 clearerr

stream	FILE *	A pointer to a FILE stream

Remarks

The `clearerr()` function resets the end-of-file status and error status for `stream`. The end-of-file status and error status are also reset when a stream is opened.

This function may not be implemented on all platforms.

See Also

"feof" on page 307

"ferror" on page 309

"fopen" on page 317

"fseek" on page 341

"rewind" on page 368

Listing 35.1 Example of clearerr() Usage

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

int main(void)
{
   FILE *f;

   static char name[] = "myfoo";
   char buf[80];

   // create a file for output
   if ( (f = fopen(name, "w")) == NULL) {
      printf("Can't open %s.\n", name);
      exit(1);
   }
   // output text to the file
   fprintf(f, "chair table chest\n");
   fprintf(f, "desk raccoon\n");

   // close the file
   fclose(f);

   // open the same file again for input
```

```
   if ( (f = fopen(name, "r")) == NULL) {
      printf("Can't open %s.\n", name);
      exit(1);
   }


   // read all the text until end-of-file
   for (; feof(f) == 0; fgets(buf, 80, f))
      fputs(buf, stdout);

   printf("feof() for file %s is %d.\n", name, feof(f));
   printf("Clearing end-of-file status. . .\n");
   clearerr(f);
   printf("feof() for file %s is %d.\n", name, feof(f));

   // close the file
   fclose(f);

   return 0;
}
```

```
Output
chair table chest
desk raccoon
feof() for file myfoo is 256.
Clearing end-of-file status. . .
feof() for file myfoo is 0.
```

fclose

Close an open file.

```
#include <stdio.h>
int fclose(FILE *stream);
```

Table 35.4 fclose

stream	FILE *	A pointer to a FILE stream

Remarks

The `fclose()` function closes a file created by `fopen()`, `freopen()`, or `tmpfile()`. The function flushes any buffered data to its file and closes the

stream. After calling `fclose()`, `stream` is no longer valid and cannot be used with file functions unless it is reassigned using `fopen()`, `freopen()`, or `tmpfile()`.

All of a program's open streams are flushed and closed when a program terminates normally.

`fclose()` closes then deletes a file created by `tmpfile()`.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fclose()` returns a zero if it is successful and returns an `EOF` if it fails to close a file.

This function may not be implemented on all platforms.

See Also

"fopen" on page 317

"freopen" on page 333

"tmpfile" on page 382

"abort" on page 405

"exit" on page 420

Listing 35.2 Example of fclose() Usage

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   FILE *f;
   static char name[] = "myfoo";

   // create a new file for output
   if ( (f = fopen(name, "w")) == NULL) {
      printf("Can't open %s.\n", name);
      exit(1);
   }
   // output text to the file
   fprintf(f, "pizza sushi falafel\n");
   fprintf(f, "escargot sprocket\n");

   // close the file
   if (fclose(f) == -1) {
      printf("Can't close %s.\n", name);
      exit(1);
   }
```

```
   return 0;
}
```

```
Output to file myfoo:
pizza sushi falafel
escargot sprocket
```

fdopen

Converts a file descriptor to a stream.

```
#include <stdio.h>
FILE *fdopen(int fildes, char *mode);
FILE *_fdopen(int fildes, char *mode);
```

Table 35.5 fdopen

fildes	int	A file descriptor, which is integer file number that can be obtained from the function fileno().
mode	char *	The file opening mode

Remarks

This function creates a stream for the file descriptor `fildes`. You can use the stream with such standard I/O functions as `fprintf()` and `getchar()`. In MSL C/C++, it ignores the value of the `mode` argument.

If it is successful, `fdopen()` returns a stream. If it encounters an error, `fdopen()` returns `NULL`.

This function may not be implemented on all platforms.

See Also

"fileno" on page 83

"open, _wopen" on page 121

Listing 35.3 Example of fdopen() Usage

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

int main(void)
{
   int fd;
   FILE *str;

   fd = open("mytest", O_WRONLY | O_CREAT);

   /* Write to the file descriptor */
   write(fd, "Hello world!\n", 13);
   /* Convert the file descriptor to a  stream */

   str = fdopen(fd,"w");

   /* Write to the stream */
   fprintf(str, "My name is %s.\n", getlogin());

   /* Close the stream. */
   fclose(str);
   /* Close the file descriptor */
   close(fd);

   return 0;
}
```

feof

Check the end-of-file status of a stream.

```
#include <stdio.h>
int feof(FILE *stream);
```

Table 35.6 feof

stream	FILE *	A pointer to a FILE stream

Remarks

The `feof()` function checks the end-of-file status of the last read operation on `stream`. The function does not reset the end-of-file status.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`feof()` returns a nonzero value if the stream is at the end-of-file and returns zero if the stream is not at the end-of-file.

This function may not be implemented on all platforms.

See Also

"clearerr" on page 302

"ferror" on page 309

Listing 35.4 Example of feof() Usage

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

int main(void)
{
   FILE *f;
   static char filename[80], buf[80] = "";

   // get a filename from the user
   printf("Enter a filename to read.\n");
   gets(filename);

   // open the file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   // read text lines from the file until
   // feof() indicates the end-of-file
   for (; feof(f) == 0 ; fgets(buf, 80, f) )
      printf(buf);

   // close the file
   fclose(f);

   return 0;
}
```

```
Output:
Enter a filename to read.
itwerks
```

```
The quick brown fox
jumped over the moon.
```

ferror

Check the error status of a stream.

```
#include <stdio.h>
int ferror(FILE *stream);
```

Table 35.7 ferror

stream	FILE *	A pointer to a FILE stream

Remarks

The `ferror()` function returns the error status of the last read or write operation on `stream`. The function does not reset its error status.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`ferror()` returns a nonzero value if `stream`'s error status is on, and returns zero if `stream`'s error status is off.

This function may not be implemented on all platforms.

See Also

"clearerr" on page 302

"feof" on page 307

Listing 35.5 Example of ferror() Usage

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

int main(void)
{
   FILE *f;
   char filename[80], buf[80];
   int ln = 0;

   // get a filename from the user
```

```
   printf("Enter a filename to read.\n");
   gets(filename);

   // open the file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   // read the file one line at a time until end-of-file
   do {
      fgets(buf, 80, f);
      printf("Status for line %d: %d.\n", ln++, ferror(f));
   } while (feof(f) == 0);

   // close the file
   fclose(f);

   return 0;
}
```

```
Output:
Enter a filename to read.
itwerks
Status for line 0: 0.
Status for line 1: 0.
Status for line 2: 0.
```

fflush

Empty a stream's buffer to its host environment.

```
#include <stdio.h>
int fflush(FILE *stream);
```

Table 35.8 fflush

stream	FILE *	A pointer to a FILE stream

Remarks

The `fflush()` function empties `stream`'s buffer to the file associated with `stream`. If the stream points to an output stream or an update stream in which the

most recent operation was not input, the fflush function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise the behavior is undefined.

The fflush() function should not be used after an input operation.

Using `fflush()` for input streams especially the standard input stream (stdin) is undefined and is not supported and will not flush the input buffer.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

The function `fflush()` returns `EOF` if a write error occurs, otherwise it returns zero.

This function may not be implemented on all platforms.

See Also

"setvbuf" on page 377

Listing 35.6 Example of fflush() Usage

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

int main(void)
{
   FILE *f;
   int count;

   // create a new file for output
   if (( f = fopen("foofoo", "w")) == NULL) {
      printf("Can't open file.\n");
      exit(1);
   }
   for (count = 0; count < 100; count++) {
      fprintf(f, "%5d", count);
      if( (count % 10) == 9 )
      {
         fprintf(f, "\n");
         fflush(f);       /* flush buffer every 10 numbers */
      }
   }
   fclose(f);

   return 0;
}
```

```
Output to file foofoo:
    0    1    2    3    4    5    6    7    8    9
   10   11   12   13   14   15   16   17   18   19
   20   21   22   23   24   25   26   27   28   29
   30   31   32   33   34   35   36   37   38   39
   40   41   42   43   44   45   46   47   48   49
   50   51   52   53   54   55   56   57   58   59
   60   61   62   63   64   65   66   67   68   69
   70   71   72   73   74   75   76   77   78   79
   80   81   82   83   84   85   86   87   88   89
   90   91   92   93   94   95   96   97   98   99
```

fgetc

Read the next character from a stream.

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Table 35.9 fgetc

stream	FILE *	A pointer to a FILE stream

Remarks

The `fgetc()` function reads the next character from `stream` and advances its file position indicator.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fgetc()` returns the character as an unsigned char converted to an `int`. If the end-of-file has been reached or a read error is detected, `fgetc()` returns `EOF`. The difference between a read error and end-of-file can be determined by the use of `feof()`..

If the file is opened in update mode (+) a file cannot be read from and then written to without repositioning the file using one of the file positioning functions (fseek(), fsetpos(), or rewind()) unless the last read or write reached the end-of-file.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"getc" on page 348

"getchar" on page 349

Listing 35.7 Example of fgetc() Usage

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

int main(void)
{
   FILE *f;
   char filename[80], c;

   // get a filename from the user
   printf("Enter a filename to read.\n");
   gets(filename);

   // open the file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   // read the file one character at a time until
   // end-of-file is reached
   while ( (c = fgetc(f)) != EOF)
      putchar(c);          // print the character

   // close the file
   fclose(f);

   return 0;
}
```

```
Output:
Enter a filename to read.
foofoo
    0    1    2    3    4    5    6    7    8    9
   10   11   12   13   14   15   16   17   18   19
   20   21   22   23   24   25   26   27   28   29
   30   31   32   33   34   35   36   37   38   39
   40   41   42   43   44   45   46   47   48   49
   50   51   52   53   54   55   56   57   58   59
```

```
60   61   62   63   64   65   66   67   68   69
70   71   72   73   74   75   76   77   78   79
80   81   82   83   84   85   86   87   88   89
90   91   92   93   94   95   96   97   98   99
```

fgetpos

Get a stream's current file position indicator value.

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Table 35.10 fgetpos

stream	FILE *	A pointer to a FILE stream
pos	fpos_t *	A pointer to a file position type

Remarks

The `fgetpos()` function is used in conjunction with the `fsetpos()` function to allow random access to a file. The `fgetpos()` function gives unreliable results when used with streams associated with a console (`stdin`, `stderr`, `stdout`).

While the `fseek()` and `ftell()` functions use `long` integers to read and set the file position indicator, `fgetpos()` and `fsetpos()` use `fpos_t` values to operate on larger files. The `fpos_t` type, defined in `stdio.h`, can hold file position indicator values that do not fit in a `long int`.

The `fgetpos()` function stores the current value of the file position indicator for `stream` in the `fpos_t` variable `pos` points to.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fgetpos()` returns zero when successful and returns a nonzero value when it fails.

This function may not be implemented on all platforms.

See Also

"fseek" on page 341

"fsetpos" on page 343

"ftell" on page 344

Listing 35.8 Example of fgetpos() Usage

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

int main(void)
{
   FILE *f;
   fpos_t pos;
   char filename[80], buf[80];

   // get a filename from the user
   printf("Enter a filename to read.\n");
   gets(filename);

   // open the file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }
   printf("Reading each line twice.\n");

   // get the initial file position indicator value
   // (which is at the beginning of the file)
   fgetpos(f, &pos);

   // read each line until end-of-file is reached
   while (fgets(buf, 80, f) != NULL) {
      printf("Once: %s", buf);

      // move to the beginning of the line to read it again
      fsetpos(f, &pos);
      fgets(buf, 80, f);
      printf("Twice: %s", buf);

      // get the file position of the next line
      fgetpos(f, &pos);
   }

   // close the file
   fclose(f);

   return 0;
```

```
}
```

```
Output:
Enter a filename to read.
myfoo
Reading each line twice.
Once: chair table chest
Twice: chair table chest
Once: desk raccoon
Twice: desk raccoon
```

fgets

Read a character array from a stream.

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

```
char *fgets(char *s, int n, FILE *stream);
```

Table 35.11 fgets

s	char *	The destination string
n	int	The maximum number of chars read
stream	FILE *	A pointer to a FILE stream

Remarks

The `fgets()` function reads characters sequentially from `stream` beginning at the current file position, and assembles them into `s` as a character array. The function stops reading characters when `n-1` characters have been read. The `fgets()` function finishes reading prematurely if it reaches a newline (`'\n'`) character or the end-of-file. For example usage, see Listing 35.16.

If the file is opened in update mode (+) a file cannot be read from and then written to without repositioning the file using one of the file positioning functions (fseek(), fsetpos(), or rewind()) unless the last read or write reached the end-of-file.

Unlike the `gets()` function, `fgets()` appends the newline character (`'\n'`) to `s`. It also null terminates the characters written into the character array.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fgets()` returns a pointer to `s` if it is successful. If it reaches the end-of-file before reading any characters, `s` is untouched and `fgets()` returns a null pointer (`NULL`). If an error occurs `fgets()` returns a null pointer and the contents of `s` may be corrupted.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"gets" on page 351

_fileno

This function is described in extras.h as "fileno" on page 83 in this header it is Windows only.

fopen

Open a file as a stream.

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Table 35.12 fopen

filename	const char *	The filename of the file to open
mode	const char *	The file opening mode

Remarks

The `fopen()` function opens a file specified by `filename`, and associates a stream with it. The `fopen()` function returns a pointer to a `FILE`. This pointer is used to refer to the file when performing I/O operations.

The mode argument specifies how the file is to be used. Table 35.13 shows the values for mode.

UPDATE MODE

A file opened with an update mode (“+”) is buffered. The file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the fflush() function or one of the file positioning operations (fseek(), fsetpos(), or rewind()). Similarly, a file cannot be read from and then written to without repositioning the file using one of the file positioning functions unless the last read or write reached the end-of-file.

All file modes, except the append modes (`"a"`, `"a+"`, `"ab"`, `"ab+"`) set the file position indicator to the beginning of the file. The append modes set the file position indicator to the end-of-file.

NOTE Write modes, even if in Write and Read (w+, wb+) delete any current data in a file when the file is opened.

Table 35.13 Open Modes for fopen()

Mode	Description
“r”	Open an existing text file for reading only.
“w”	Create a new text file for writing, or open and truncate an existing file
“a”	Open an existing text file, or create a new one if it does not exist, for appending. Writing occurs at the end-of-file position.
“r+”	Update mode. Open an existing text file for reading and writing. See Remarks.
“w+”	Update mode. Create a new text file for writing, or open and truncate an existing file, for writing and reading. See Remarks.
“a+”	Update mode. Open an existing text file or create a new one for reading and writing. Writing occurs at the end-of-file position. See Remarks.
“rb”	Open an existing binary file for reading only.
“wb”	Create a new binary file or open and truncate an existing file, for writing

Table 35.13 Open Modes for fopen() (*continued*)

Mode	Description
“ab”	Open an existing binary file, or create a new one if it does not exist, and append. Writing occurs at the end-of-file.
“r+b” or “rb+”	Update mode. Open an existing binary file for reading and writing. See Remarks.
“w+b” or “wb+”	Update mode. Create a new binary file or open and truncate an existing file, for writing and reading. See Remarks.
“a+b” or “ab+”	Update mode. Open an existing binary file or create a new one for reading and writing. Writing occurs at the end-of-file position. See Remarks.

`fopen()` returns a pointer to a `FILE` if it successfully opens the specified file for the specified operation. `fopen()` returns a null pointer (`NULL`) when it is not successful.

This function may not be implemented on all platforms.

See Also

“fclose” on page 304

“_wfopen” on page 399

Listing 35.9 Example of fopen() Usage

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

int main(void)
{
   FILE *f;
   int count;

   // create a new file for output
   if (( f = fopen("foofoo", "w")) == NULL) {
      printf("Can't create file.\n");
      exit(1);
   }

   // output numbers 0 to 9
```

```
   for (count = 0; count < 10; count++)
      fprintf(f, "%5d", count);

   // close the file
   fclose(f);

   // open the file to append
   if (( f = fopen("foofoo", "a")) == NULL) {
      printf("Can't append to file.\n");
      exit(1);
   }

   // output numbers 10 to 19
   for (; count <20; count++)
      fprintf(f, "%5d\n", count);

   // close file
   fclose(f);

   return 0;
}
```

```
Output to file foofoo:
0    1    2    3    4    5    6    7    8    9   10
   11
   12
   13
   14
   15
   16
   17
   18
   19
```

fprintf

Send formatted text to a stream.

```
#include <stdio.h>
int fprintf(FILE *stream,
      const char *format, ...);
```

Table 35.14 fprintf

stream	FILE *	A pointer to a FILE stream
format	const char *	The format string

Remarks

The `fprintf()` function writes formatted text to `stream` and advances the file position indicator. Its operation is the same as `printf()` with the addition of the `stream` argument. Refer to the description of `printf()`.

If the file is opened in update mode (+) the file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the fflush() function or one of the file positioning operations (fseek(), fsetpos(), or rewind()).

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Output Control String and Conversion Specifiers

The `format` character array contains normal text and conversion specifications. Conversion specifications must have matching arguments in the same order in which they occur in `format`.

The various elements of the format string is specified in the ANSI standards to be in this order from left to right.

- A percent sign
- Optional flags -,+,0,# or space
- Optional minimum field width specification
- Optional precision specification
- Optional size specification
- Conversion specifier c,d,e,E,f, Fg,G,i,n,o,p,s,u,x,X or %

A conversion specification describes the format its associated argument is to be converted to. A specification starts with a percent sign (`%`), optional flag characters, an optional minimum width, an optional precision width, and the necessary, terminating conversion type. Doubling the percent sign (`%%`) results in the output of a single `%`.

An optional flag character modifies the formatting of the output; it can be left or right justified, and numerical values can be padded with zeroes or output in alternate forms. More than one optional flag character can be used in a conversion specification. Table 35.16 describes the flag characters.

The optional minimum width is a decimal digit string. If the converted value has more characters that the minimum width, it is expanded as required. If the converted value has fewer characters than the minimum width, it is, by default, right justified (padded on the left). If the - flag character is used, the converted value is left justified (padded on the right).

The maximum `minimum field width` allowed in MSL Standard Libraries is 509 characters.

The optional precision width is a period character (`.`) followed by decimal digit string. For floating point values, the precision width specifies the number of digits to print after the decimal point. For integer values, the precision width functions identically to, and cancels, the minimum width specification. When used with a character array, the precision width indicates the maximum width of the output.

A minimum width and a precision width can also be specified with an asterisk (`*`) instead of a decimal digit string. An asterisk indicates that there is a matching argument, preceding the conversion argument, specifying the minimum width or precision width.

The terminating character, the conversion type, specifies the conversion applied to the conversion specification's matching argument. Table 35.17 describes the conversion type characters.

Table 35.15 and Table 35.18 also include other modifiers for formatted output functions.

MSL AltiVec Extensions for Fprintf

The AltiVec extensions to the standard printf family of functions is supported in Main Standard Libraries.

Separator arguments after % and before any specifier may be any character or may be the @ symbol. The @ symbol is a non-Motorola extension that will use a specified string as a specifier.

In the specific case of a 'c' specifier any char may be used as a separator. For all other specifiers '-', '+', '#', ' ' may not be used.

Listing 35.23 demonstrates their use.

Table 35.15 Length Modifiers for Formatted Output Functions

Modifier	Description
h	The h flag followed by `d, i, o, u, x, or X` conversion specifier indicates that the corresponding argument is a `short int` or `unsigned short int`.
l	The lower case L followed by `d, i, o, u, x, or X` conversion specifier indicates the argument is a `long int` or `unsigned long int`. The lower case L followed by a `c` conversion specifier, indicates that the argument is of type `wint_t`. The lower case L followed by an `s` conversion specifier, indicates that the argument is of type `wchar_t`.
ll	The double l followed by `d, i, o, u, x, or X` conversion specifier indicates the argument is a `long long or unsigned long long`.
L	The upper case L followed by `e, E, f, g, or G conversion specifier indicates a long double`.
v	**AltiVec**: `A vector bool char, vector signed char or vector unsigned char` when followed by `c, d, i, o, u, x or X` A vector float, when followed by `f`.
vh hv	**AltiVec**: A `vector short, vector unsigned short, vector bool short or vector pixel` when followed by `c, d, i, o, u, x or X`
vl lv	**AltiVec**: A `vector int, vector unsigned int or vector bool int` when followed by `c, d, i, o, u, x or X`

Table 35.16 Flag Specifiers For formatted Output Functions

Modifier	Description
-	The conversion will be left justified.
+	The conversion, if numeric, will be prefixed with a sign (+ or -). By default, only negative numeric values are prefixed with a minus sign (-).
space	If the first character of the conversion is not a sign character, it is prefixed with a space. Because the plus sign flag character (+) always prefixes a numeric value with a sign, the space flag has no effect when combined with the plus flag.
#	For c, d, i, and u conversion types, the # flag has no effect. For s conversion types, a pointer to a Pascal string, is output as a character string. For o conversion types, the # flag prefixes the conversion with a 0. For x conversion types with this flag, the conversion is prefixed with a 0x. For e, E, f, g, and G conversions, the # flag forces a decimal point in the output. For g and G conversions with this flag, trailing zeroes after the decimal point are not removed.
0	This flag pads zeroes on the left of the conversion. It applies to d, i, o, u, x, X, e, E, f, g, and G conversion types. The leading zeroes follow sign and base indication characters, replacing what would normally be space characters. The minus sign flag character overrides the 0 flag character. The 0 flag is ignored when used with a precision width for d, i , o, u, x, and X conversion types.
@	**AltiVec**: This flag indicates a pointer to a string specified by an argument. This string will be used as a separator for vector elements.

Table 35.17 Conversion Specifiers for Formatted Output Functions

Modifier	Description
d	The corresponding argument is converted to a signed decimal.
i	The corresponding argument is converted to a signed decimal.
o	The argument is converted to an unsigned octal.
u	The argument is converted to an unsigned decimal.
x, X	The argument is converted to an unsigned hexadecimal. The x conversion type uses lowercase letters (abcdef) while X uses uppercase letters (ABCDEF).
n	This conversion type stores the number of items output by printf() so far. Its corresponding argument must be a pointer to an int.
f, F	The corresponding floating point argument (float, or double) is printed in decimal notation. The default precision is 6 (6 digits after the decimal point). If the precision width is explicitly 0, the decimal point is not printed. For the `f` conversion specifier, a double argument representing infinity produces `[-]inf;` a double argument representing a `NaN` (Not a number) produces `[-]nan.` For the *F* conversion specifier, `[-]INF` or `[-]NAN` are produced instead.

Table 35.17 Conversion Specifiers for Formatted Output Functions (*continued*)

Modifier	Description
e, E	The floating point argument (float or double) is output in scientific notation: `[-]b.aaae±Eee`. There is one digit (*b*) before the decimal point. Unless indicated by an optional precision width, the default is 6 digits after the decimal point (`aaa`). If the precision width is 0, no decimal point is output. The exponent (*ee*) is at least 2 digits long. The e conversion type uses lowercase e as the exponent prefix. The E conversion type uses uppercase E as the exponent prefix.
g, G	The g conversion type uses the f or e conversion types and the G conversion type uses the f or E conversion types. Conversion type e (or E) is used only if the converted exponent is less than -4 or greater than the precision width. The precision width indicates the number of significant digits. No decimal point is output if there are no digits following it.
c	The corresponding argument is output as a character.
s	The corresponding argument, a pointer to a character array, is output as a character string. Character string output is completed when a null character is reached. The null character is not output.
p	The corresponding argument is taken to be a pointer. The argument is output using the X conversion type format.

Table 35.18 CodeWarrior Extensions for Formatted Output Functions

Modifier	Description
#s	The corresponding argument, a pointer to a Pascal string, is output as a character string. A Pascal character string is a length byte followed by the number characters specified in the length byte. **Note**: This conversion type is an extension to the ANSI C library but applied in the same manner as for other format variations.

`fprintf()` returns the number of arguments written or a negative number if an error occurs.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"printf" on page 353

"sprintf" on page 380

"vfprintf" on page 387

"vprintf" on page 391

"vsprintf" on page 395

Listing 35.10 Example of fpritnf() Usage

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

int main(void)
{
   FILE *f;
   static char filename[] = "myfoo";
   int a = 56;
   char c = 'M';
   double x = 483.582;

   // create a new file for output
   if (( f = fopen(filename, "w")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
```

```
    }

    // output formatted text to the file
    fprintf(f, "%10s %4.4f %-10d\n%10c", filename, x, a, c);

    // close the file
    fclose(f);

    return 0;
}
```

```
Output to file foo:
myfoo 483.5820 56
         M
```

fputc

Write a character to a stream.

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

Table 35.19 fputc

c	int	The character to write to a file
stream	FILE *	A pointer to a FILE stream

Remarks

The `fputc()` function writes the character `c` to `stream` and advances `stream`'s file position indicator. Although the `c` argument is an `unsigned int`, it is converted to a `char` before being written to `stream`. `fputc()` is written as a function, not as a macro.

If the file is opened in update mode (+) the file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the fflush() function or one of the file positioning operations (fseek(), fsetpos(), or rewind()).

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fputc()` returns the character written if it is successful, and returns `EOF` if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"putc" on page 362

"putchar" on page 363

Listing 35.11 Example of fputc() Usage

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

int main(void)
{
   FILE *f;
   int letter;

   // create a new file for output
   if (( f = fopen("foofoo", "w")) == NULL) {
      printf("Can't create file.\n");
      exit(1);
   }

   // output the alphabet to the file one letter
   // at a time
   for (letter = 'A'; letter <= 'Z'; letter++)
      fputc(letter, f);
   fclose(f);

   return 0;
}
```

```
Output to file foofoo:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

fputs

Write a character array to a stream.

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Table 35.20 fputs

s	const char *	The string to write to a file
stream	FILE *	A pointer to a FILE stream

Remarks

The `fputs()` function writes the array pointed to by `s` to `stream` and advances the file position indicator. The function writes all characters in `s` up to, but not including, the terminating null character. Unlike `puts()`, `fputs()` does not terminate the output of `s` with a newline (`'\n'`).

If the file is opened in update mode (+) the file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the fflush() function or one of the file positioning operations (fseek(), fsetpos(), or rewind()).

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fputs()` returns a zero if successful, and returns a nonzero value when it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"puts" on page 365

Listing 35.12 Example of fputs() Usage

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

int main(void)
{
   FILE *f;
```

```
   // create a new file for output
   if (( f = fopen("foofoo", "w")) == NULL) {
      printf("Can't create file.\n");
      exit(1);
   }

   // output character strings to the file
   fputs("undo\n", f);
   fputs("copy\n", f);
   fputs("cut\n", f);
   fputs("rickshaw\n", f);

   // close the file
   fclose(f);

   return 0;
}
```

```
Output to file foofoo:
undo
copy
cut
rickshaw
```

fread

Read binary data from a stream.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size,
      size_t nmemb, FILE *stream);
```

Table 35.21 fread

ptr	void *	A pointer to the read destination
size	size_t	The size of the array elements pointed to

Table 35.21 fread (*continued*)

nmemb	size_t	Number of elements to be read
stream	FILE *	A pointer to a FILE stream

Remarks

The `fread()` function reads a block of binary or text data and updates the file position indicator. The data read from `stream` are stored in the array pointed to by `ptr`. The `size` and `nmemb` arguments describe the size of each item and the number of items to read, respectively.

The `fread()` function reads `nmemb` items unless it reaches the end-of-file or a read error occurs.

If the file is opened in update mode (+) a file cannot be read from and then written to without repositioning the file using one of the file positioning functions (fseek(), fsetpos(), or rewind()) unless the last read or write reached the end-of-file.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fread()` returns the number of items read successfully.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fgets" on page 316

"fwrite" on page 347

Listing 35.13 Example of fread() Usage

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


// define the item size in bytes
#define BUFSIZE  40

int main(void)
{
   FILE *f;
   static char s[BUFSIZE] = "The quick brown fox";
   char target[BUFSIZE];
```

```
   // create a new file for output and input
   if (( f = fopen("foo", "w+")) == NULL) {
      printf("Can't create file.\n");
      exit(1);
   }

   // output to the stream using fwrite()
   fwrite(s, sizeof(char), BUFSIZE, f);

   // move to the beginning of the file
   rewind(f);

   // now read from the stream using fread()
   fread(target, sizeof(char), BUFSIZE, f);

   // output the results to the console
   puts(s);
   puts(target);

   // close the file
   fclose(f);

   return 0;
}
```

```
Output:
The quick brown fox
The quick brown fox
```

freopen

Re-direct a stream to another file.

```
#include <stdio.h>

FILE *freopen(const char *filename,

const char *mode, FILE *stream);
```

Table 35.22 freopen

filename	const char *	The name of the file to re-open
mode	const char *	The file opening mode
stream	FILE *	A pointer to a FILE stream

Remarks

The `freopen()` function changes the file that `stream` is associated with to another file. The function first closes the file the stream is associated with, and opens the new file, `filename`, with the specified `mode`, using the same stream.

`fopen()` returns the value of `stream`, if it is successful. If `fopen()` fails it returns a null pointer (`NULL`).

This function may not be implemented on all platforms.

See Also

"fopen" on page 317

Listing 35.14 Example of freopen() Usage

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

int main(void)
{
   FILE *f;

   // re-direct output from the console to a new file
   if (( f = freopen("newstdout", "w+", stdout)) == NULL) {
      printf("Can't create new stdout file.\n");
      exit(1);
   }
   printf("If all goes well, this text should be in\n");
   printf("a text file, not on the screen via stdout.\n");
   fclose(f);

   return 0;
}
```

fscanf

Read formatted text from a stream.

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Table 35.23 fscanf

stream	FILE *	A pointer to a FILE stream
format	const char *	A format string

Remarks

The `fscanf()` function reads programmer-defined, formatted text from `stream`. The function operates identically to the `scanf()` function with the addition of the `stream` argument indicating the stream to read from. Refer to the `scanf()` function description.

If the file is opened in update mode (+) a file cannot be read from and then written to without repositioning the file using one of the file positioning functions (fseek(), fsetpos(), or rewind()) unless the last read or write reached the end-of-file.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Input Control String and Conversion Specifiers

The `format` argument is a character array containing normal text, white space (space, tab, newline), and conversion specifications. The normal text specifies literal characters that must be matched in the input stream. A white space character indicates that white space characters are skipped until a non-white space character is reached. The conversion specifications indicate what characters in the input stream are to be converted and stored.

The conversion specifications must have matching arguments in the order they appear in `format`. Because `scanf()` stores data in memory, the arguments matching the conversion specification arguments must be pointers to objects of the relevant types.

A conversion specification consists of the percent sign (`%`) prefix, followed by an optional maximum width or assignment suppression, and ending with a conversion type. A percent sign can be skipped by doubling it in format; `%%` signifies a single `%` in the input stream.

An optional width is a decimal number specifying the maximum width of an input field. `scanf()` will not read more characters for a conversion than is specified by the width.

An optional assignment suppression character (`*`) can be used to skip an item by reading it but not assigning it. A conversion specification with assignment suppression must not have a corresponding argument.

The last character, the conversion type, specifies the kind of conversion requested. Table 35.24 and Table 35.24 describe the length specifier and conversion type characters.

MSL AltiVec Extensions for Scanf

The AltiVec extensions to the standard scanf family of functions is supported in Main Standard Libraries.

Separator arguments after % and before any specifier may be any character or may be the @ symbol. The @ symbol is a non-Motorola extension that will use a specified string as a specifier.

In the specific case of a 'c' specifier any char may be used as a separator. For all other specifiers '-', '+', '#', ' ' may not be used.

Listing 35.31 demonstrates their use.

Table 35.24 Length Specifiers for Formatted Input

Modifier	Description
hh	The `hh` flag indicates that the following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument that is of type `char` or `unsigned char`.
h	The `h` flag indicates that the following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument that is of type `short int` or `unsigned short int`.
l	When used with integer conversion specifier, the `l` flag indicates `long int` or an `unsigned long int` type. When used with floating point conversion specifier, the `l` flag indicates a `double`. When used with a `c` or `s` conversion specifier, the `l` flag indicates that the corresponding argument with type pointer to `wchar_t`.

Table 35.24 Length Specifiers for Formatted Input (*continued*)

Modifier	Description
ll	When used with integer conversion specifier, the ll flag indicates that the corresponding argument is of type long long or an unsigned long long.
L	The L flag indicates that the corresponding float conversion specifier corresponds to an argument of type long double.
v	**AltiVec**: `A vector bool char, vector signed char or vector unsigned char` when followed by `c, d, i, o, u, x or X` A vector float, when followed by `f`.
vh hv	**AltiVec**: `vector short, vector unsigned short, vector bool short or vector pixel` when followed by `c, d, i, o, u, x or X`
vl lv	**AltiVec**: `vector long, vector unsigned long or vector bool` when followed by `c, d, i, o, u, x or X`

Table 35.25 Conversion Specifiers for Formatted Input

Modifier	Description
d	A decimal integer is read.
i	A decimal, octal, or hexadecimal integer is read. The integer can be prefixed with a plus or minus sign (+, -), 0 for octal numbers, 0x or 0X for hexadecimal numbers.
o	An octal integer is read.
u	An unsigned decimal integer is read.
x, X	A hexadecimal integer is read.

Table 35.25 Conversion Specifiers for Formatted Input (*continued*)

Modifier	Description
e, E, f, g, G	A floating point number is read. The number can be in plain decimal format (e.g. 3456.483) or in scientific notation `([-]b.aaae[-]dd)`.
s	A character string is read. The input character string is considered terminated when a white space character is reached or the maximum width has been reached. The null character is appended to the end of the array.
c	A character is read. White space characters are not skipped, but read using this conversion specifier.
p	A pointer address is read. The input format should be the same as that output by the p conversion type in printf().
n	This conversion type does not read from the input stream but stores the number of characters read so far in its corresponding argument.
[scanset]	Input stream characters are read and filtered determined by the `scanset`. See "Scanset" for a full description.

Scanset

The conversion specifier %[allows you to specify a `scanset`, which is a sequence of characters that will be read and stored in the string pointed to by the scanset's corresponding argument. The characters between the `[` and the terminating `]` define the `scanset`. A `null` character is appended to the end of the character sequence.

Input stream characters are read until a character is found that is not in the `scanset`. If the first character of scanset is a circumflex (^) then input stream characters are read until a character from the `scanset` is read. A `null` character is appended to the end of the character array.

Thus, the conversion specifier `%[abcdef]` specifies that the `scanset` is `abcdef` and any of the characters `'a'` through `'f'` are to be accepted and stored. As soon as

any character outside this set is encountered, reading and storing will cease. Thus, for example, assuming we have the declaration:

```
char str[20];
```

the execution of

```
sscanf("acdfxbe", "%[abcdef]", str);
```

will store `acdf` in `str;` the `'x'` and following characters will not be stored because the `'x'` is not in the scanset.

If the first character of the scanset is the circumflex, ^, then the following characters will define a set of characters such that encountering any one of them will cause reading and storing to stop; any character outside a scanset defined in this way will be accepted, we will call this an exclusionary scanset. Thus execution of

```
sscanf("stuvawxyz", "%[^abcdef]", str);
```

will store `stuv` in `str.` If you want ^ to be part of the scanset, you cannot list it as the first character otherwise it will be interpreted as introducing the members of an `exclusionary scanset.` Thus `%[^abc]` defines the `exclusionary scanset abc` whereas `%[a^bc]` defines the `scanset abc^. %[^a^bc]` defines the `exclusionary scanset abc^,` as does `%[^^abc].`

If you want `]` to be in the `scanset,` it must be the first character of the `scanset,` immediately following the `%[` or, to be in an `exclusionary scanset,` immediately after the ^, for example, `%[]abc] or %[^]abc].` In any other position, the `]` will be interpreted as terminating the `scanset.`

To include the `-` character in the `scanset,` it must be either listed first (possibly after an initial ^ or last, thus for example, `%[-abc], %[abc-], %[^-abc], or%[^abc-].` The C Standard explicitly states:

- If a - character is in the scanlist and is not the first, nor the second where the first character is a ^, nor the last character, the behavior is implementation defined.

MSL interprets such a use of - in a scanlist as defining a range of characters; thus, the specification `%[a-z]` as being the equivalent of `%[abcdefghijklmnopqrstuvwxyz].` You should bear in mind that this is MSL's interpretation and such usage may be interpreted differently in other C library implementations. Note also that it is assumed that the numeric value of the character before the `-` is less than that of the one after. If this relationship does not hold undefined and probably unwanted effects may be experienced.

`fscanf()` returns the number of items read or, if an input error occurs before any conversions, the value `EOF.` If there is an error in reading data that is inconsistent with the format string, `fscanf()` sets `errno` to a nonzero value. `fscanf()` returns `EOF` if it reaches the end-of-file.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"errno" on page 75

"scanf" on page 370

Listing 35.15 Example of fscanf() Usage

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

int main(void)
{
   FILE *f;
   int i;
   double x;
   char c;

   // create a new file for output and input
   if (( f = fopen("foobar", "w+")) == NULL) {
      printf("Can't create new file.\n");
      exit(1);
   }

   // output formatted text to the file
   fprintf(f, "%d\n%f\n%c\n", 45, 983.3923, 'M');

   // go to the beginning of the file
   rewind(f);

   // read from the stream using fscanf()
   fscanf(f, "%d %lf %c", &i, &x, &c);

   // close the file
   fclose(f);

   printf("The integer read is %d.\n", i);
   printf("The floating point value is %f.\n", x);
   printf("The character is %c.\n", c);

   return 0;

}
```

```
Output:
The integer read is 45.
```

```
The floating point value is 983.392300.
The character is M.
```

fseek

Move the file position indicator.

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Table 35.26 fseek

stream	FILE *	A pointer to a FILE stream
offset	long	The offset to move in bytes
whence	int	The starting position of the offset

Remarks

The `fseek()` function moves the file position indicator to allow random access to a file. For example usage, see Listing 35.16.

The function moves the file position indicator either absolutely or relatively. The `whence` argument can be one of three values defined in `stdio.h`: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`.

The `SEEK_SET` value causes the file position indicator to be set `offset` bytes from the beginning of the file. In this case `offset` must be equal or greater than zero.

The `SEEK_CUR` value causes the file position indicator to be set `offset` bytes from its current position. The `offset` argument can be a negative or positive value.

The `SEEK_END` value causes the file position indicator to be set `offset` bytes from the end of the file. The `offset` argument must be equal or less than zero.

The `fseek()` function undoes the last `ungetc()` call and clears the end-of-file status of `stream`.

NOTE The function `fseek` has limited use when used with `MS DOS` text files opened in `text mode` because of the carriage return / line feed translations. For more information review "Text Streams and Binary Streams" on page 300. The `fseek` operations may be incorrect near the end of the file due to eof translations.

The only `fseek` operations guaranteed to work in `MS DOS` text files opened in `text mode` are:

- Using the offset returned from ftell() and seeking from the beginning of the file.
- Seeking with an offset of zero from `SEEK_SET`, `SEEK_CUR` and `SEEK_END`.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fseek()` returns zero if it is successful and returns a nonzero value if it fails.

This function may not be implemented on all platforms.

See Also

"fgetpos" on page 314

"fsetpos" on page 343

"ftell" on page 344

Listing 35.16 Example of fseek() Usage

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

int main(void)
{
   FILE *f;
   long int pos1, pos2, newpos;
   char filename[80], buf[80];

   // get a filename from the user
   printf("Enter a filename to read.\n");
   gets(filename);

   // open a file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   printf("Reading last half of first line.\n");

   // get the file position indicator before and after
   // reading the first line
   pos1 = ftell(f);
   fgets(buf, 80, f);
   pos2 = ftell(f);
   printf("Whole line: %s\n", buf);
```

```
    // calculate the middle of the line
    newpos = (pos2 - pos1) / 2;

    fseek(f, newpos, SEEK_SET);
    fgets(buf, 80, f);
    printf("Last half: %s\n", buf);

    // close the file
    fclose(f);

    return 0;
}
```

```
Output:
Enter a filename to read.
itwerks
Reading last half of first line.
Whole line: The quick brown fox

Last half: brown fox
```

fsetpos

Set the file position indicator.

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
```

Table 35.27 fsetpos

stream	FILE *	A pointer to a FILE stream
pos	fpos_t	A pointer to a file positioning type

Remarks

The `fsetpos()` function sets the file position indicator for `stream` using the value pointed to by `pos`. The function is used in conjunction with `fgetpos()` when dealing with files having sizes greater than what can be represented by the `long int` argument used by `fseek()`.

`fsetpos()` undoes the previous call to `ungetc()` and clears the end-of-file status.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fsetpos()` returns zero if it is successful and returns a nonzero value if it fails.

This function may not be implemented on all platforms.

See Also

"fgetpos" on page 314

"fseek" on page 341

"ftell" on page 344

For example usage of "fgetpos", see Listing 35.8.

ftell

Return the current file position indicator value.

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Table 35.28 ftell

stream	FILE *	A pointer to a FILE stream

Remarks

The `ftell()` function returns the current value of stream's file position indicator. It is used in conjunction with `fseek()` to provide random access to a file.

The function will not work correctly when it is given a stream associated to a console file, such as `stdin`, `stdout`, or `stderr`, where a file indicator position is not applicable. Also, `ftell()` cannot handle files with sizes larger than what can be represented with a `long int`. In such a case, use the `fgetpos()` and `fsetpos()` functions.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`ftell()`, when successful, returns the current file position indicator value. If it fails, `ftell()` returns `-1L` and sets the global variable `errno` to a nonzero value.

This function may not be implemented on all platforms.

See Also

"errno" on page 75, "fgetpos" on page 314

fwide

Determine the orientation of a stream.

```
#include <stdio.h>
int fwide(FILE *stream, int orientation);
```

Table 35.29 fwide

stream	FILE *	A pointer to the stream being tested
orientation	int	The desired orientation

Remarks

The fwide function determines the orientation of the stream pointed to by stream. If the value of orientation is greater than zero and stream is without orientation, stream is made to be wide oriented. If the value of orientation is less than zero and stream is without orientation, stream is made to be byte oriented. Otherwise, the value of orientation is zero and the function does not alter the orientation of the stream. In all cases, if stream already has an orientation, it will not be changed.

The `fwide` function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

This function may not be implemented on all platforms.

Listing 35.17 Example of fwide() Usage

```
#include <stdio.h>

int main()
{
   FILE * fp;
   char filename[FILENAME_MAX];
   int orientation;
   char * cptr;
```

```
    cptr = tmpnam(filename);
    fp = fopen(filename, "w");
    orientation = fwide(fp, 0);

    // A newly opened file has no orientation
    printf("Initial orientation = %i\n", orientation);
    fprintf(fp, "abcdefghijklmnopqrstuvwxyz\n");

    // A byte oriented output operation will set the orientation
    // to byte oriented
    orientation = fwide(fp, 0);

    printf("Orientation after fprintf = %i\n", orientation);
    fclose(fp);
    fp = fopen(filename, "r");
    orientation = fwide(fp, 0);
    printf("Orientation after reopening = %i\n", orientation);
    orientation = fwide(fp, -1);

    // fwide with a non-zero orientation argument will set an
    // unoriented file's orientation
    printf("Orientation after fwide  = %i\n", orientation);
    orientation = fwide(fp, 1);

    // but will not change the file's orientation if it
    // already has an orientation
    printf("Orientation after second fwide  = %i\n",orientation);
    fclose(fp);
    remove(filename);

    return 0;
}
```

```
Output:
Initial orientation = 0
Orientation after fprintf = -1
Orientation after reopening = 0
Orientation after fwide  = -1
Orientation after second fwide  = -1
```

fwrite

Write binary data to a stream.

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
      size_t nmemb, FILE *stream);
```

Table 35.30 fwrite

ptr	void *	A pointer to the item being written
size	size_t	The size of the item being written
nmemb	size_t	The number of items being written
stream	FILE *	A pointer to a FILE stream

Remarks

The `fwrite()` function writes `nmemb` items of `size` bytes each to `stream`. The items are contained in the array pointed to by `ptr`. After writing the array to `stream`, `fwrite()` advances the file position indicator accordingly.

If the file is opened in update mode (+) the file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the fflush() function or one of the file positioning operations (fseek(), fsetpos(), or rewind()).

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`fwrite()` returns the number of items successfully written to `stream`.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fread" on page 331

For example of fread() usage, see Listing 35.13.

getc

Read the next character from a stream.

```
#include <stdio.h>
int getc(FILE *stream);
```

Table 35.31 getc

stream	FILE *	A pointer to a FILE stream

Remarks

The `getc()` function reads the next character from `stream`, advances the file position indicator, and returns the character as an `int` value. Unlike the `fgetc()` function, `getc()` is implemented as a macro.

If the file is opened in update mode (+) it cannot be read from and then written to without being repositioned using one of the file positioning functions (fseek(), fsetpos(), or rewind()) unless the last read or write reached the end-of-file.

`getc()` returns the next character from the stream or returns `EOF` if the end-of-file has been reached or a read error has occurred.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fgetc" on page 312

"fputc" on page 328

"getchar" on page 349

"putchar" on page 363

Listing 35.18 Example of getc() Usage

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

int main(void)
{
   FILE *f;
   char filename[80], c;
```

```
   // get a filename from the user
   printf("Enter a filename to read.\n");
   scanf("%s", filename);

   // open a file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   // read one character at a time until end-of-file
   while ( (c = getc(f)) != EOF)
      putchar(c);

   // close the file
   fclose(f);

   return 0;
}
```

```
Output

Enter a filename to read.
foofoo
    0    1    2    3    4    5    6    7    8    9
   10   11   12   13   14   15   16   17   18   19
   20   21   22   23   24   25   26   27   28   29
   30   31   32   33   34   35   36   37   38   39
   40   41   42   43   44   45   46   47   48   49
   50   51   52   53   54   55   56   57   58   59
   60   61   62   63   64   65   66   67   68   69
   70   71   72   73   74   75   76   77   78   79
   80   81   82   83   84   85   86   87   88   89
   90   91   92   93   94   95   96   97   98   99
```

getchar

Get the next character from `stdin`.

```
#include <stdio.h>

int getchar(void);
```

Remarks

The `getchar()` function reads a character from the `stdin` stream.

The function `getchar()` is implemented as `getc(stdin)` and as such `getchar's` return may be delayed or optimized out of program order if `stdin` is buffered. For most implementations, `stdin` is line buffered.

`getchar()` returns the value of the next character from `stdin` as an `int` if it is successful. `getchar()` returns `EOF` if it reaches an end-of-file or an error occurs.

This function may not be implemented on all platforms.

See also:

"Wide Character and Byte Character Stream Orientation" on page 301

"fgetc" on page 312

"getc" on page 348

"putchar" on page 363

Listing 35.19 Example of getchar() Usage

```
#include <stdio.h>

int main(void)
{
   int c;

   printf("Enter characters to echo, * to quit.\n");

   // characters entered from the console are echoed
   // to it until a * character is read
   while ( (c = getchar()) != '*')
      putchar(c);

   printf("\nDone!\n");

   return 0;
}
```

```
Output:
Enter characters to echo, * to quit.
I'm experiencing deja-vu *
I'm experiencing deja-vu
Done!
```

gets

Read a character array from stdin.

```
#include <stdio.h>
char *gets(char *s);
```

Table 35.32 gets

s	char s	The string being written in to

Remarks

The gets() function reads characters from stdin and stores them sequentially in the character array pointed to by s. Characters are read until either a newline or an end-of-file is reached.

Unlike fgets(), the programmer cannot specify a limit on the number of characters to read. Also, gets() reads and ignores the newline character ('\n') so that it can advance the file position indicator to the next line. The newline character is not stored s. Like fgets(), gets() terminates the character string with a null character.

If an end-of-file is reached before any characters are read, gets() returns a null pointer (NULL) without affecting the character array at s. If a read error occurs, the contents of s may be corrupted.

gets() returns s if it is successful and returns a null pointer if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fgets" on page 316

Listing 35.20 Example of gets() Usage

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

int main(void)
{
   char buf[100];
```

```
   printf("Enter text lines to echo.\n");
   printf("Enter an empty line to quit.\n");

   // read character strings from the console
   // until an empty line is read
   while (strlen(gets(buf)) > 0)
      puts(buf);   // puts() appends a newline to its output

   printf("Done!\n");

   return 0;
}
```

```
Output:
Enter text lines to echo.
Enter an empty line to quit.
I'm experiencing deja-vu
I'm experiencing deja-vu
Now go to work
Now go to work

Done!
```

perror

Output an error message to `stderr`.

```
#include <stdio.h>
void perror(const char *s);
```

Table 35.33 perror

s	const char *	Prints an errno and message

Remarks

If s is not NULL or a pointer to a null string the `perror()` function outputs to `stderr` the character array pointed to by `s` followed by a colon and a space `': '`. Then, the error message that would be returned by `strerror()` for the current value of the global variable `errno`.

This function may not be implemented on all platforms.

See Also

"abort" on page 405

"errno" on page 75

Listing 35.21 Example of perror() Usage

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

int main()
{
   perror("No error reported as");
   errno = EDOM;
   perror("Domain error reported as");
   errno = ERANGE;
   perror("Range error reported as");

   return 0;
}
```

```
Output
No error reported as: No Error
Domain error reported as: Domain Error
Range error reported as: Range Error
```

printf

Output formatted text.

```
#include <stdio.h>
int printf(const char *format,  ...);
```

Table 35.34 printf

format	const char *	A format string

Remarks

The `printf()` function outputs formatted text. The function takes one or more arguments, the first being `format`, a character array pointer. The optional arguments following format are items (integers, characters, floating point values,

etc.) that are to be converted to character strings and inserted into the output of `format` at specified points.

The `printf()` function sends its output to `stdout`.

Printf Control String and Conversion Specifiers

The `format` character array contains normal text and conversion specifications. Conversion specifications must have matching arguments in the same order in which they occur in `format`.

The various elements of the format string is specified in the ANSI standards to be in this order from left to right.

- A percent sign
- Optional flags -,+,0,# or space
- Optional minimum field width specification
- Optional precision specification
- Optional size specification
- Conversion specifier c,d,e,E,f,F,g,G,i,n,o,p,s,u,x,X or %

A conversion specification describes the format its associated argument is to be converted to. A specification starts with a percent sign (`%`), optional flag characters, an optional minimum width, an optional precision width, and the necessary, terminating conversion type. Doubling the percent sign (`%%`) results in the output of a single `%`.

An optional flag character modifies the formatting of the output; it can be left or right justified, and numerical values can be padded with zeroes or output in alternate forms. More than one optional flag character can be used in a conversion specification. Table 35.36 describes the flag characters.

The optional minimum width is a decimal digit string. If the converted value has more characters that the minimum width, it is expanded as required. If the converted value has fewer characters than the minimum width, it is, by default, right justified (padded on the left). If the `-` flag character is used, the converted value is left justified (padded on the right).

The maximum `minimum field width` allowed in MSL Standard Libraries is 509 characters.

The optional precision width is a period character (`.`) followed by decimal digit string. For floating point values, the precision width specifies the number of digits to print after the decimal point. For integer values, the precision width functions identically to, and cancels, the minimum width specification. When used with a character array, the precision width indicates the maximum width of the output.

A minimum width and a precision width can also be specified with an asterisk (*) instead of a decimal digit string. An asterisk indicates that there is a matching argument, preceding the conversion argument, specifying the minimum width or precision width.

The terminating character, the conversion type, specifies the conversion applied to the conversion specification's matching argument. Table 35.37 describes the conversion type characters.

Table 35.35 and Table 35.38 also include other modifiers for formatted output functions.

MSL AltiVec Extensions for Printf

The AltiVec extensions to the standard printf family of functions is supported in Main Standard Libraries.

Separator arguments after % and before any specifier may be any character or may be the @ symbol. The @ symbol is a non-Motorola extension that will use a specified string as a specifier.

In the specific case of a 'c' specifier any char may be used as a separator. For all other specifiers '-', '+', '#', ' ' may not be used.

Listing 35.23 demonstrates their use.

Table 35.35 Length Modifiers for Formatted Output Functions

Modifier	Description
h	The h flag followed by d, i, o, u, x, or X conversion specifier indicates that the corresponding argument is a short int or unsigned short int.
l	The lower case L followed by d, i, o, u, x, or X conversion specifier indicates the argument is a long int or unsigned long int. The lower case L followed by a `c` conversion specifier indicates that the argument is of type `wint_t`. The lower case L followed by an `s` conversion specifier indicates that the argument is of type `wchar_t`.
ll	The double l followed by d, i, o, u, x, or X conversion specifier indicates the argument is a long long or unsigned long long

Table 35.35 Length Modifiers for Formatted Output Functions (*continued*)

Modifier	Description
L	The upper case L followed by e, E, f, g, or G conversion specifier indicates a long double.
v	**AltiVec**: `A vector bool char, vector signed char or vector unsigned char` when followed by `c, d, i, o, u, x or X` A vector float, when followed by `f`.
vh hv	**AltiVec**: A `vector short, vector unsigned short, vector bool short or vector pixel` when followed by `c, d, i, o, u, x or X`
vl lv	**AltiVec**: A `vector int, vector unsigned int or vector bool int` when followed by `c, d, i, o, u, x or X`

Table 35.36 Flag Specifiers For formatted Output Functions

Modifier	Description
-	The conversion will be left justified.
+	The conversion, if numeric, will be prefixed with a sign (+ or -). By default, only negative numeric values are prefixed with a minus sign (-).
space	If the first character of the conversion is not a sign character, it is prefixed with a space. Because the plus sign flag character (+) always prefixes a numeric value with a sign, the space flag has no effect when combined with the plus flag.

Table 35.36 Flag Specifiers For formatted Output Functions (*continued*)

Modifier	Description
#	For c, d, i, and u conversion types, the # flag has no effect. For s conversion types, a pointer to a Pascal string, is output as a character string. For o conversion types, the # flag prefixes the conversion with a 0. For x conversion types with this flag, the conversion is prefixed with a 0x. For e, E, f, g, and G conversions, the # flag forces a decimal point in the output. For g and G conversions with this flag, trailing zeroes after the decimal point are not removed.
0	This flag pads zeroes on the left of the conversion. It applies to d, i, o, u, x, X, e, E, f, g, and G conversion types. The leading zeroes follow sign and base indication characters, replacing what would normally be space characters. The minus sign flag character overrides the 0 flag character. The 0 flag is ignored when used with a precision width for d, i , o, u, x, and X conversion types.
@	**AltiVec**: This flag indicates a pointer to a string specified by an argument. This string will be used as a separator for vector elements.

Table 35.37 Conversion Specifiers for Formatted Output Functions

Modifier	Description
d	The corresponding argument is converted to a signed decimal.
i	The corresponding argument is converted to a signed decimal.
o	The argument is converted to an unsigned octal.
u	The argument is converted to an unsigned decimal.

Table 35.37 Conversion Specifiers for Formatted Output Functions (*continued*)

Modifier	Description
x, X	The argument is converted to an unsigned hexadecimal. The x conversion type uses lowercase letters (abcdef) while X uses uppercase letters (ABCDEF).
n	This conversion type stores the number of items output by printf() so far. Its corresponding argument must be a pointer to an int.
f, F	The corresponding floating point argument (float, or double) is printed in decimal notation. The default precision is 6 (6 digits after the decimal point). If the precision width is explicitly 0, the decimal point is not printed. For the `f` conversion specifier, a double argument representing infinity produces `[-]inf;` a double argument representing a `NaN` (Not a number) produces `[-]nan.` For the *F* conversion specifier, `[-]INF` or `[-]NAN` are produced instead.
e, E	The floating point argument (float or double) is output in scientific notation: `[-]b.aaae±Eee.` There is one digit (*b*) before the decimal point. Unless indicated by an optional precision width, the default is 6 digits after the decimal point (`aaa`). If the precision width is 0, no decimal point is output. The exponent (*ee*) is at least 2 digits long. The e conversion type uses lowercase e as the exponent prefix. The E conversion type uses uppercase E as the exponent prefix.
g, G	The g conversion type uses the f or e conversion types and the G conversion type uses the f or E conversion types. Conversion type e (or E) is used only if the converted exponent is less than -4 or greater than the precision width. The precision width indicates the number of significant digits. No decimal point is output if there are no digits following it.

Table 35.37 Conversion Specifiers for Formatted Output Functions (*continued*)

Modifier	Description
c	The corresponding argument is output as a character.
s	The corresponding argument, a pointer to a character array, is output as a character string. Character string output is completed when a null character is reached. The null character is not output.
p	The corresponding argument is taken to be a pointer. The argument is output using the X conversion type format.

Table 35.38 CodeWarrior Extensions for Formatted Output Functions

Modifier	Description
#s	The corresponding argument, a pointer to a Pascal string, is output as a character string. A Pascal character string is a length byte followed by the number characters specified in the length byte. **Note**: This conversion type is an extension to the ANSI C library but applied in the same manner as for other format variations.

`printf()`, like `fprintf()`, `sprintf()`, `vfprintf()`, and `vprintf()`, returns the number of arguments that were successfully output. `printf()` returns a negative value if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fprintf" on page 320

"sprintf" on page 380

"vfprintf" on page 387

"vprintf" on page 391

"vsprintf" on page 395

Listing 35.22 Example of printf() Usage

```
#include <stdio.h>

int main(void)
{
   int i = 25;
   char c = 'M';
   short int d = 'm';
   static char s[] = "CodeWarrior!";
   static char pas[] = "\pCodeWarrior  again!";
   float f = 49.95;
   double x = 1038.11005;
   int count;
   printf("%s printf() demonstration:\n%n", s, &count);
   printf("The last line contained %d characters\n",count);
   printf("Pascal string output: %#20s\n", pas);
   printf("%-4d %x %06x %-5o\n", i, i, i, i);
   printf("%*d\n", 5, i);
   printf("%4c %4u %4.10d\n", c, c, c);
   printf("%4c %4hu %3.10hd\n", d, d, d);
   printf("$%5.2f\n", f);
   printf("%5.2f\n%6.3f\n%7.4f\n", x, x, x);
   printf("%*.*f\n", 8, 5, x);

   return 0;
}
```

```
The output is:
CodeWarrior! printf() demonstration:
The last line contained 36 characters
Pascal string output:    CodeWarrior again!
25   19 000019 31
   25
   M   77 0000000077
   m  109 0000000109
$49.95
1038.11
1038.110
1038.1101
1038.11005
```

Listing 35.23 Example of AltiVec printf Extensions

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

```
int main(void)
{
   vector signed char s =
      (vector signed char)(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
   vector unsigned short us16 =
      (vector unsigned short)('a','b','c','d','e','f','g','h');
   vector signed int sv32 =
      (vector signed int)(100, 2000, 30000, 4);
   vector signed int vs32 =
      (vector signed int)(0, -1, 2, 3);
   vector float flt32 =
      (vector float)(1.1, 2.22, 3.3, 4.444);

   printf("s = %vd\n", s);

   printf("s = %,vd\n", s);

   printf("vector=%@vd\n", "\nvector=", s);

      // c specifier so no space is added.
   printf("us16 = %vhc\n", us16);

   printf("sv32 = %,5lvd\n", sv32);

   printf("vs32 = 0x%@.8lvX\n", ", 0x", vs32);

   printf("flt32 = %,5.2vf\n", flt32);


   return 0;
}
```

```
The Result is:
s = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
s = 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
vector=1
vector=2
vector=3
vector=4
vector=5
vector=6
vector=7
vector=8
vector=9
vector=10
vector=11
vector=12
```

```
vector=13
vector=14
vector=15
vector=16
us16 = abcdefgh
sv32 =   100, 2000,30000,    4
vs32 = 0x00000000, 0xFFFFFFFF, 0x00000002, 0x00000003
flt32 =  1.10, 2.22, 3.30, 4.44
```

putc

Write a character to a stream.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Table 35.39 putc

c	int	The character to write to a file
stream	FILE *	A pointer to a FILE stream

Remarks

The `putc()` function outputs `c` to `stream` and advances `stream`'s file position indicator.

The `putc()` works identically to the `fputc()` function, except that it is written as a macro.

If the file is opened in update mode (+) the file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the fflush() function or one of the file positioning operations (fseek(), fsetpos(), or rewind()).

`putc()` returns the character written when successful and return `EOF` when it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fputc" on page 328

"putchar" on page 363

Listing 35.24 Example of putc() Usage

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

int main(void)
{
   FILE *f;
   static char filename[] = "checkputc";
   static char test[] = "flying fish and quail eggs";
   int i;

   // create a new file for output
   if (( f = fopen(filename, "w")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   // output the test character array
   // one character at a time using putc()
   for (i = 0; test[i] > 0; i++)
      putc(test[i], f);

   // close the file
   fclose(f);

   return 0;
}
```

```
Output to file checkputc
flying fish and quail eggs
```

putchar

Write a character to `stdout`.

```
#include <stdio.h>
int putchar(int c);
```

Table 35.40 putchar

c	int	The character to write to stdout

Remarks

The `putchar()` function writes character `c` to `stdout`.

`putchar()` returns `c` if it is successful and returns `EOF` if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fputc" on page 328

"putc" on page 362

Listing 35.25 Example of putchar() Usage

```
#include <stdio.h>

int main(void)
{
   static char test[] = "running jumping walking tree\n";
   int i;

   // output the test character one character
   // at a time until the null character is found.
   for (i = 0; test[i] != '\0'; i++)
      putchar(test[i]);

   return 0;
}
```

```
Output:
running jumping walking tree
```

puts

Write a character string to `stdout`.

```
#include <stdio.h>
int puts(const char *s);
```

Table 35.41 puts

s	const char *	The string written to stdout

Remarks

The `puts()` function writes a character string array to `stdout`, stopping at, but not including the terminating null character. The function also appends a newline (`'\n'`) to the output.

`puts()` returns zero if successful and returns a nonzero value if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fputs" on page 330

Listing 35.26 Example of puts() Usage

```
#include <stdio.h>

int main(void)
{
   static char s[] = "car bus metro werks";
   int i;

   // output the string 10 times
   for (i = 0; i < 10; i++)
      puts(s);

   return 0;
}
```

```
Output:
car bus metro werks
```

```
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
car bus metro werks
```

remove

Delete a file.

```
#include <stdio.h>
int remove(const char *filename);
```

Table 35.42 remove

filename	const char *	The name of the file to be deleted

Remarks

The `remove()` function deletes the named file specified by `filename`.

`remove()` returns 0 if the file deletion is successful, and returns a nonzero value if it fails.

This function may not be implemented on all platforms.

See Also

"_wremove" on page 400

"fopen" on page 317

"rename" on page 367

Listing 35.27 Example of remove() Usage

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

int main(void)
{
```

```
   char filename[40];

   // get a filename from the user
   printf("Enter the name of the file to delete.\n");
   gets(filename);

   // delete the file
   if (remove(filename) != 0) {
      printf("Can't remove %s.\n", filename);
      exit(1);
   }

   return 0;
}
```

rename

Change the name of a file.

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Table 35.43 rename

old	const char *	The old file name
new	const char *	The new file name

Remarks

The `rename()` function changes the name of a file, specified by `old` to the name specified by `new`.

`rename()` returns a nonzero if it fails and returns zero if successful

This function may not be implemented on all platforms.

See Also

“ wrename” on page 401

“freopen” on page 333

“remove” on page 366

Listing 35.28 Example of rename() Usage

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

int main(void)
{
   char oldname[50];   // current filename
   char newname[50];   // new filename

   // get the current filename from the user
   printf("Please enter the current filename.\n");
   gets(oldname);

   // get the new filename from the user
   printf("Please enter the new filename.\n");
   gets(newname);

   // rename oldname to newname
   if (rename(oldname, newname) != 0) {
      printf("Can't rename %s to %s.\n", oldname,
        newname);
      exit(1);
   }

   return 0;
}
```

```
Output:
Please enter the current filename.
boots.txt
Please enter the new filename.
sandals.txt
```

rewind

Reset the file position indicator to the beginning of the file.

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

```
void rewind(FILE *stream);
```

Table 35.44 rewind

stream	FILE *	A pointer to a FILE stream

Remarks

The `rewind()` function sets the file indicator position of `stream` such that the next write or read operation will be from the beginning of the file. It also undoes any previous call to `ungetc()` and clears `stream`'s end-of-file and error status.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

This function may not be implemented on all platforms.

See Also

"fseek" on page 341

"fsetpos" on page 343

Listing 35.29 Example of rewind() Usage

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

int main(void)
{
   FILE *f;
   char filename[80], buf[80];

   // get a filename from the user
   printf("Enter a filename to read.\n");
   gets(filename);

   // open a file for input
   if (( f = fopen(filename, "r")) == NULL) {
      printf("Can't open %s.\n", filename);
      exit(1);
   }

   printf("Reading first line twice.\n");

   // move the file position indicator to the beginning
   // of the file
   rewind(f);
   // read the first line
   fgets(buf, 80, f);
```

```
    printf("Once: %s\n", buf);

    // move the file position indicator to the
    //beginning of the file
    rewind(f);

    // read the first line again
    fgets(buf, 80, f);
    printf("Twice: %s\n", buf);

    // close the file
    fclose(f);

    return 0;
}
```

```
Output:
Enter a filename to read.
itwerks
Reading first line twice.
Once: flying fish and quail eggs
Twice: flying fish and quail eggs
```

scanf

Read formatted text.

```
#include <stdio.h>

int scanf(const char *format, ...);
```

Table 35.45 scanf

format	const char *	The format string

Remarks

The `scanf()` function reads text and converts the text read to programmer specified types.

Scanf Control String and Conversion Specifiers

The `format` argument is a character array containing normal text, white space (space, tab, newline), and conversion specifications. The normal text specifies literal characters that must be matched in the input stream. A white space character indicates that white space characters are skipped until a non-white space character is reached. The conversion specifications indicate what characters in the input stream are to be converted and stored.

The conversion specifications must have matching arguments in the order they appear in `format`. Because `scanf()` stores data in memory, the matching conversion specification arguments must be pointers to objects of the relevant types.

A conversion specification consists of the percent sign (`%`) prefix, followed by an optional maximum width or assignment suppression, and ending with a conversion type. A percent sign can be skipped by doubling it in format; `%%` signifies a single `%` in the input stream.

An optional width is a decimal number specifying the maximum width of an input field. `scanf()` will not read more characters for a conversion than is specified by the width.

An optional assignment suppression character (`*`) can be used to skip an item by reading it but not assigning it. A conversion specification with assignment suppression must not have a corresponding argument.

The last character, the conversion type, specifies the kind of conversion requested. Table 35.46 and Table 35.47 describe the length specifier and conversion type characters.

MSL AltiVec Extensions for Scanf

The AltiVec extensions to the standard scanf family of functions is supported in Main Standard Libraries.

Separator arguments after % and before any specifier may be any character or may be the @ symbol. The @ symbol is a non-Motorola extension that will use a specified string as a specifier.

In the specific case of a'c' specifier any char may be used as a separator. For all other specifiers '-', '+', '#', ' ' may not be used.

Listing 35.31 demonstrates their use.

Table 35.46 Length Specifiers for Formatted Input

Modifier	Description
hh	The `hh` flag indicates that the following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument that is of type `char` or `unsigned char`.
h	The `h` flag indicates that the following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument that is of type `short int` or `unsigned short int`.
l	When used with integer conversion specifier, the `l` flag indicates `long int` or an `unsigned long int` type. When used with floating point conversion specifier, the `l` flag indicates a `double`. When used with a `c` or `s` conversion specifier, the `l` flag indicates that the corresponding argument with type pointer to `wchar_t`.
ll	When used with integer conversion specifier, the ll flag indicates that the corresponding argument is of type long long or an unsigned long long.
L	The L flag indicates that the corresponding float conversion specifier corresponds to an argument of type long double.
v	**AltiVec**: `A vector bool char, vector signed char or vector unsigned char` when followed by `c, d, i, o, u, x or X` A vector float, when followed by `f`.
vh hv	**AltiVec**: `vector short, vector unsigned short, vector bool short or vector pixel` when followed by `c, d, i, o, u, x or X`
vl lv	**AltiVec**: `vector long, vector unsigned long or vector bool` when followed by `c, d, i, o, u, x or X`

Table 35.47 Conversion Specifiers for Formatted Input

Modifier	Description
d	A decimal integer is read.
i	A decimal, octal, or hexadecimal integer is read. The integer can be prefixed with a plus or minus sign (+, -), 0 for octal numbers, 0x or 0X for hexadecimal numbers.
o	An octal integer is read.
u	An unsigned decimal integer is read.
x, X	A hexadecimal integer is read.
e, E, f, g, G	A floating point number is read. The number can be in plain decimal format (e.g. 3456.483) or in scientific notation `([-]b.aaae[-]dd)`.
s	A character string is read. The input character string is considered terminated when a white space character is reached or the maximum width has been reached. The null character is appended to the end of the array.
c	A character is read. White space characters are not skipped, but read using this conversion specifier.
p	A pointer address is read. The input format should be the same as that output by the p conversion type in printf().
n	This conversion type does not read from the input stream but stores the number of characters read so far in its corresponding argument.
[scanset]	Input stream characters are read and filtered determined by the `scanset`. See "Scanset" for a full description.

`scanf()` returns the number of items successfully read and returns `EOF` if a conversion type does not match its argument or and end-of-file is reached.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fscanf" on page 335

"sscanf" on page 381

Listing 35.30 Example of scanf() Usage

```
#include <stdio.h>

int main(void)
{
   int i;
   unsigned int j;
   char c;
   char s[40];
   double x;

   printf("Enter an integer surrounded by ! marks\n");
   scanf("!%d!", &i);
   printf("Enter three integers\n");
   printf("in hexadecimal, octal, or decimal.\n");
   // note that 3 integers are read, but only the last two
   // are assigned to i and j
   scanf("%*i %i %ui", &i, &j);

   printf("Enter a character and a character string.\n");
   scanf("%c %10s", &c, s);

   printf("Enter a floating point value.\n");
   scanf("%lf", &x);

   return 0;
}
```

```
Output:
Enter an integer surrounded by ! marks
!94!
Enter three integers
in hexadecimal, octal, or decimal.
1A 6 24
Enter a character and a character string.
Enter a floating point value.
A
```

```
Sounds like 'works'!
3.4
```

Listing 35.31 Example of AltiVec Scanf Extensions

```
#include <stdio.h>

int main(void)
{
   vector signed char v8, vs8;
   vector unsigned short v16;
   vector signed long v32;
   vector float vf32;

   sscanf("1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16", "%vd", &v8);
   sscanf("1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16", "%,vd", &vs8);
   sscanf("abcdefgh", "%vhc", &v16);
   sscanf("1, 4, 300, 400", "%,3lvd", &v32);
   sscanf("1.10, 2.22, 3.333, 4.4444", "%,5vf", &vf32);

   return 0;
}
```

```
The Result is:
v8  = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16;
vs8 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16;
v16 = 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'
v32 =  1, 4, 300, 400
vf32 = 1.1000, 2.2200, 3.3330, 4.4444
```

setbuf

Change the buffer size of a stream.

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Table 35.48 setbuf

stream	FILE *	A pointer to a FILE stream
buf	char *	A buffer for input or output

Remarks

The `setbuf()` function allows the programmer to set the buffer size for `stream`. It should be called after `stream` is opened, but before it is read from or written to.

The function makes the array pointed to by `buf` the buffer used by `stream`. The `buf` argument can either be a null pointer or point to an array of size `BUFSIZ` defined in `stdio.h`.

If `buf` is a null pointer, the stream becomes unbuffered.

This function may not be implemented on all platforms.

See Also

"setvbuf" on page 377

"malloc" on page 427

Listing 35.32 Example of setbuf() Usage

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

int main(void)
{
   FILE *f;
   char name[80];

   // get a filename from the user
   printf("Enter the name of the file to write to.\n");
   gets(name);

   // create a new file for output
   if ( (f = fopen(name, "w")) == NULL) {
      printf("Can't open file %s.\n", name);
      exit(1);
   }

   setbuf(f, NULL);      // turn off buffering

   // this text is sent directly to the file without
   // buffering
   fprintf(f, "Buffering is now off\n");
   fprintf(f, "for this file.\n");

   // close the file
   fclose(f);
```

```
    return 0;
}
```

```
Output:
Enter the name of the file to write to.
bufftest
```

setvbuf

Change the buffering scheme for a stream.

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
      size_t size);
```

Table 35.49 setvbuf

stream	FILE *	A pointer to a FILE stream
buf	char *	A buffer for input and output
mode	int	A buffering mode
size	size_t	The size of the buffer

Remarks

The `setvbuf()` allows the manipulation of the buffering scheme as well as the size of the buffer used by stream. The function should be called after the stream is opened but before it is written to or read from.

The `buf` argument is a pointer to a character array. The `size` argument indicates the size of the character array pointed to by `buf`. The most efficient buffer size is a multiple of `BUFSIZ`, defined in `stdio.h`.

If `buf` is a null pointer, then the operating system creates its own buffer of `size` bytes.

The `mode` argument specifies the buffering scheme to be used with `stream`. `mode` can have one of three values defined in `stdio.h`: `_IOFBF`, `_IOLBF`, and `_IONBF`.

- `_IOFBF` specifies that `stream` be buffered.
- `_IOLBF` specifies that `stream` be line buffered.

- `_IONBF` specifies that `stream` be unbuffered

`setvbuf()` returns zero if it is successful and returns a nonzero value if it fails.

This function may not be implemented on all platforms.

See Also

"setbuf" on page 375

"malloc" on page 427

Listing 35.33 Example of setvbuf() Usage

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

int main(void)
{
   FILE *f;
   char name[80];

   // get a filename from the user
   printf("Enter the name of the file to write to.\n");
   gets(name);

   // create a new file for output
   if ( (f = fopen(name, "w")) == NULL) {
      printf("Can't open file %s.\n", name);
      exit(1);
   }

   setvbuf(f, NULL, _IOLBF, 0);   // line buffering
   fprintf(f, "This file is now\n");
   fprintf(f, "line buffered.\n");

   // close the file
   fclose(f);

   return 0;
}
```

```
Output:
Enter the name of the file to write to.
buffy
```

snprintf

Format a character string array.

```
#include <stdio.h>
int snprintf(char * s, size_t n, const char * format, ...);
```

Table 35.50 snprintf

s	char *	A string to write to
n	size_t	Max number of chars to be written to s
format	const char *	The format string

Remarks

The `snprintf()` function works identically to `fprintf()` except that the output is written into the array s instead of to a stream. If `n` is zero nothing is written; otherwise, any characters beyond the `n-1st` are discarded rather than being written to the array and a `null` character is appended at the end.

For specifications concerning the output control string and conversion specifiers, see "Output Control String and Conversion Specifiers" on page 321.

`Snprintf()` returns the number of characters that would have been assigned to `s`, had `n` been sufficiently large, not including the `null` character or a negative value if an encoding error occurred. Thus, the null-terminated output will have been completely written if and only if the returned value is nonnegative and less than `n`.

This function may not be implemented on all platforms.

Listing 35.34 Example of snprintf() Usage

```
#include <stdio.h>

int main()
{
   int i = 1;
   static char s[] = "Programmer";
   char dest[50];
   int retval;

   retval = snprintf(dest, 5, "%s is number %d!", s, i);
```

```
    printf("n too small, dest = |%s|, retval = %i\n", dest, retval);
    retval = snprintf(dest, retval, "%s is number %d!", s, i);
    printf("n right size, dest = |%s|, retval = %i\n", dest, retval);

    return 0;
}
```

```
Output:
n too small, dest = |Prog|, retval = 23
n right size, dest = |Programmer is number 1|, retval = 23
```

sprintf

Format a character string array.

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

```
int sprintf(char *s, const char *format, ...);
```

Table 35.51 sprintf

s	char *	A string to write to
format	const char *	The format string

Remarks

The `sprintf()` function works identically to `printf()` with the addition of the `s` parameter. Output is stored in the character array pointed to by `s` instead of being sent to `stdout`. The function terminates the output character string with a null character.

,For specifications concerning the output control string and conversion specifiers, see "Output Control String and Conversion Specifiers" on page 321.

`sprintf()` returns the number of characters assigned to `s`, not including the null character.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fprintf" on page 320

"printf" on page 353

Listing 35.35 Example of sprintf() Usage

```
#include <stdio.h>

int main(void)
{
   int i = 1;
   static char s[] = "CodeWarrior";
   char dest[50];

   sprintf(dest, "%s is number %d!", s, i);
   puts(dest);

   return 0;
}
```

```
Output:
CodeWarrior is number 1!
```

sscanf

Read formatted text into a character string.

```
#include <stdio.h>
int sscanf(char *s, const char *format, ...);
```

Table 35.52 sscanf

s	char *	The string to be scanned
format	const char *	The format string

Remarks

The `sscanf()` operates identically to `scanf()` but reads its input from the character array pointed to by `s` instead of `stdin`. The character array pointed to `s` must be null terminated.

For specifications concerning the input control string and conversion specifications, see "Input Control String and Conversion Specifiers" on page 335. Also see "Scanset" on page 338, for a full description of the use of `scansets`.

`scanf()` returns the number of items successfully read and converted and returns `EOF` if it reaches the end of the string or a conversion specification does not match its argument.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fscanf" on page 335

"scanf" on page 370

Listing 35.36 Example of sscanf() Usage

```
#include <stdio.h>

int main(void)
{
   static char in[] = "figs cat pear 394 road 16!";
   char s1[20], s2[20], s3[20];
   int i;

   // get the words figs, cat, road,
   // and the integer 16
   // from in and store them in s1, s2, s3, and i,
   // respectively
   sscanf(in, "%s %s pear 394 %s %d!", s1, s2, s3, &i);
   printf("%s %s %s %d\n", s1, s2, s3, i);

   return 0;
}
```

```
Output:
figs cat road 16
```

tmpfile

Open a temporary file.

```
#include <stdio.h>
FILE *tmpfile(void);
```

Remarks

The `tmpfile()` function creates and opens a binary file that is automatically removed when it is closed or when the program terminates.

`tmpfile()` returns a pointer to the `FILE` variable of the temporary file if it is successful. If it fails, `tmpfile()` returns a null pointer (`NULL`).

This function may not be implemented on all platforms.

See Also

"fopen" on page 317

"tmpnam" on page 384

Listing 35.37 Example of tmpfile() Usage

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

int main(void)
{
   FILE *f;

   // create a new temporary file for output
   if ( (f = tmpfile()) == NULL) {
      printf("Can't open temporary file.\n");
      exit(1);
   }

   // output text to the temporary file
   fprintf(f, "watch clock timer glue\n");

   // close AND DELETE the temporary file
   // using fclose()
   fclose(f);

   return 0;
}
```

tmpnam

Creates a unique temporary filename.

```
#include <stdio.h>
char *tmpnam(char *s);
```

Table 35.53 tmpnam

s	char *	A temporary file name

Remarks

The `tmpnam()` functions creates a valid filename character string that will not conflict with any existing filename. A program can call the function up to `TMP_MAX` times before exhausting the unique filenames `tmpnam()` generates. The `TMP_MAX` macro is defined in `stdio.h`.

The `s` argument can either be a null pointer or pointer to a character array. The character array must be at least `L_tmpnam` characters long. The new temporary filename is placed in this array. The `L_tmpnam` macro is defined in `stdio.h`.

If `s` is `NULL`, `tmpnam()` returns with a pointer to an internal static object that can be modified by the calling program.

Unlike `tmpfile()`, a file created using a filename generated by the `tmpnam()` function is not automatically removed when it is closed.

`tmpnam()` returns a pointer to a character array containing a unique, non-conflicting filename. If `s` is a null pointer (`NULL`), the pointer refers to an internal static object. If `s` points to a character array, `tmpnam()` returns the same pointer.

This function may not be implemented on all platforms.

See Also

"fopen" on page 317

"tmpfile" on page 382

Listing 35.38 Example of tmpnam() Usage

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

int main(void)
{
```

```
    FILE *f;
    char *tempname;
    int c;

    // get a unique filename
    tempname = tmpnam("tempwerks");

    // create a new file for output
    if ( (f = fopen(tempname, "w")) == NULL) {
       printf("Can't open temporary file %s.\n", tempname);
       exit(1);
    }

    // output text to the file
    fprintf(f, "shoe shirt tie trousers\n");
    fprintf(f, "province\n");

    // close the file
    fclose(f);

    // delete the file
    remove(tempname);

    return 0;
}
```

ungetc

Place a character back into a stream.

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Table 35.54 ungetc

c	int	The character to return to a file
stream	FILE *	A pointer to a FILE stream

Remarks

The `ungetc()` function places character `c` back into `stream`'s buffer. The next read operation will read the character placed by `ungetc()`. Only one character can be pushed back into a buffer until a read operation is performed.

The function's effect is ignored when an `fseek()`, `fsetpos()`, or `rewind()` operation is performed.

`ungetc()` returns `c` if it is successful and returns `EOF` if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fseek" on page 341

"fsetpos" on page 343

"rewind" on page 368

Listing 35.39 Example of ungetc() Usage

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

int main(void)
{
   FILE *f;
   int c;

   // create a new file for output and input
   if ( (f = fopen("myfoo", "w+")) == NULL) {
      printf("Can't open myfoo.\n");
      exit(1);
   }

   // output text to the file
   fprintf(f, "The quick brown fox\n");
   fprintf(f, "jumped over the moon.\n");

   // move the file position indicator
   // to the beginning of the file
   rewind(f);

   printf("Reading each character twice.\n");

   // read a character
   while ( (c = fgetc(f)) != EOF) {
```

```
      putchar(c);
   // put the character back into the stream
      ungetc(c, f);
      c = fgetc(f);// read the same character again
      putchar(c);
   }

   fclose(f);

   return 0;
}
```

```
Output
Reading each character twice.
TThhee  qquuiicckk  bbrroowwnn  ffooxx
jjuummppeedd  oovveerr  tthhee  mmoooonn..
```

vfprintf

Write formatted output to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format,va_list arg);
```

Table 35.55 vfprintf

stream	FILE *	A pointer to a FILE stream
format	const char *	The format string
arg	va_list	The variable argument list

Remarks

The `vfprintf()` function works identically to the `fprintf()` function. Instead of the variable list of arguments that can be passed to `fprintf()`, `vfprintf()` accepts its arguments in the array `arg` of type `va_list` which must have been initialized by the `va_start()` macro from the `stdarg.h` header file. The `vfprintf()` does not invoke the `va_end` macro.

NOTE On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

For specifications concerning the output control string and conversion specifiers, see "Output Control String and Conversion Specifiers" on page 321.

`vfprintf()` returns the number of characters written or `EOF` if it failed.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fprintf" on page 320

"printf" on page 353

"Overview of stdarg.h" on page 279

Listing 35.40 Example of vfprintf() Usage

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

int fpr(FILE *, char *, ...);

int main(void)
{
   FILE *f;
   static char name[] = "foo";
   int a = 56, result;
   double x = 483.582;

   // create a new file for output
   if (( f = fopen(name, "w")) == NULL) {
      printf("Can't open %s.\n", name);
      exit(1);
   }

   // format and output a variable number of arguments
   // to the file
   result = fpr(f, "%10s %4.4f %-10d\n", name, x, a);

   // close the file
   fclose(f);

   return 0;
```

```
}

// fpr() formats and outputs a variable
// number of arguments to a stream using
// the vfprintf() function
int fpr(FILE *stream, char *format, ...)
{
   va_list args;
   int retval;

   va_start(args, format);      // prepare the arguments
   retval = vfprintf(stream, format, args);
   // output them
   va_end(args);               // clean the stack
   return retval;
}
```

```
Output to file foo:
foo 483.5820 56
```

vfscanf

Read formatted text from a stream.

```
 #include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE *stream, const char *format, va_list arg);
```

Table 35.56 vfscanf

stream	FILE *	A pointer to a FILE stream
format	const char *	The format string
arg	va_list	The variable argument list

Remarks

The vfscantf() function works identically to the fscanf() function. Instead of the variable list of arguments that can be passed to fscanf(), vfscanf() accepts its arguments in the array arg of type va_list, which must have been initialized by the

va_start() macro from the stdarg.h header file. The vfscanf() does not invoke the va_end macro.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

For specifications concerning the output control string and conversion specifiers, see Table 35.35, Table 35.36, Table 35.37, and Table 35.38.

vfscanf() returns the number of items assigned, which can be fewer than provided for in the case of an early matching failure. If an input failure occurs before any conversion, vfscanf() returns EOF.

This function may not be implemented on all platforms.

See Also

"scanf" on page 370

"fscanf" on page 335

Listing 35.41 Example of vfscanf() Usage

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

int fsc(FILE *, char *, ...);

int main(void)
{
   FILE *f;
   int i;
   double x;
   char c;
   int numassigned;
   // create a new file for output and input
   if (( f = fopen("foobar", "w+")) == NULL) {
   printf("Can't create new file.\n");
   exit(1);
   }
   // output formatted text to the file
   fprintf(f, "%d\n%f\n%c\n", 45, 983.3923, 'M');
   // go to the beginning of the file
   rewind(f);
   // read from the stream using fscanf()
   numassigned = fsc(f, "%d %lf %c", &i, &x, &c);
   // close the file
   fclose(f);
   printf("The number of asssignments is %d.\n", numassigned);
```

```
   printf("The integer read is %d.\n", i);
   printf("The floating point value is %f.\n", x);
   printf("The character is %c.\n", c);
   return 0;
}

// fsc() scans an input stream and inputs
// a variable number of arguments using
// the vfscanf() function
int fsc(FILE *stream, char *format, ...)
{
   va_list args;
   int retval;

   va_start(args, format);       // prepare the arguments
   retval = vfscanf(stream, format, args);
   va_end(args);                 // clean the stack
   return retval;
}
```

```
Output:
The number of asssignments is 3.
The integer read is 45.
The floating point value is 983.392300.
The character is M.
```

vprintf

Write formatted output to `stdout`.

```
#include <stdio.h>

int vprintf(const char *format, va_list arg);
```

Table 35.57 vprintf

format	const char *	The format string
arg	va_list	A variable argument list

Remarks

The `vprintf()` function works identically to the `printf()` function. Instead of the variable list of arguments that can be passed to `printf()`, `vprintf()`

accepts its arguments in the array of type `va_list` processed by the `va_start()` macro from the `stdarg.h` header file.

For specifications concerning the output control string and conversion specifiers, see "Output Control String and Conversion Specifiers" on page 321.

`vprintf()` returns the number of characters written or a negative value if it failed.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"fprintf" on page 320

"printf" on page 353

"Overview of stdarg.h" on page 279

Listing 35.42 Example of vprintf() Usage

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

int pr(char *, ...);

int main(void)
{
   int a = 56;
   double f = 483.582;
   static char s[] = "Valerie";

   // output a variable number of arguments to stdout
   pr("%15s %4.4f %-10d*\n", s, f, a);

   return 0;

}

// pr() formats and outputs a variable number of arguments
// to stdout using the vprintf() function
int pr(char *format, ...)
{
   va_list args;
   int retval;
   va_start(args, format); // prepare the arguments
   retval = vprintf(format, args);
   va_end(args);            // clean the stack
   return retval;
```

```
}
```

```
Output:
Valerie 483.5820 56        *
```

vsnprintf

Format a character string array.

```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char * s, size_t n,
    const char * format, va_list arg);
```

Table 35.58 vsnprintf

s	char *	A string to write to
n	size_t	Max number of chars to be written to s
format	const char *	The format string
arg	va_list	A variable argument list

Remarks

The `vsnprintf()` function works identically to `snprintf()`, except that the variable list of arguments that can be passed to `snprintf()` is replaced by an array `arg` of type `va_list`, which must have been initialized by the `va_start()` macro from the `stdarg.h` header file. The `vsnprintf()` does not invoke the `va_end` macro. If `n` is zero nothing is written; otherwise, any characters beyond the `n-1st` are discarded rather than being written to the array and a null character is appended at the end.

For specifications concerning the output control string and conversion specifiers, see "Output Control String and Conversion Specifiers" on page 321.

`Vsnprintf()` returns the number of characters that would have been assigned to `s`, had n been sufficiently large, not including the null character or a negative value if an encoding error occurred. Thus, the null-terminated output will have

been completely written if and only if the returned value is nonnegative and less than `n`.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"printf" on page 353

"sprintf" on page 380

"Overview of stdarg.h" on page 279

Listing 35.43 Example of vsnprintf() Usage

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

int sp(char *, size_t, char *, ...);

int main()
{
   int i = 1;
   static char s[] = "Isabelle";
   char dest[50];
   int retval;

   retval = sp(dest, 5, "%s is number %d!", s, i);
   printf("n too small, dest = |%s|, retval = %i\n", dest, retval);
   retval = sp(dest, retval, "%s is number %d!", s, i);
   printf("n right size, dest = |%s|, retval = %i\n", dest, retval);

   return 0;
}

// sp() formats and outputs a variable number of arguments
// to a character string using the vsnprintf() function
int sp(char * s, size_t n, char *format,...)
{
   va_list args;
   int retval;

   va_start(args, format);      // prepare the arguments
   retval = vsnprintf(s, n, format, args);
   va_end(args);               // clean the stack
   return retval;
}
```

```
Output:
n too small, dest = |Isab|, retval = 21
n right size, dest = |Isabelle is number 1|, retval = 21
```

vsprintf

Write formatted output to a string.

```
#include <stdio.h>
int vsprintf(char *s,
      const char *format, va_list arg);
```

Table 35.59 vsprintf

s	char *	A string to write to
format	const char *	The format string
arg	va_list	A variable argument list

Remarks

The `vsprintf()` function works identically to the `sprintf()` function. Instead of the variable list of arguments that can be passed to `sprintf()`, `vsprintf()` accepts its arguments in the array of type `va_list` processed by the `va_start()` macro from the `stdarg.h` header file.

For specifications concerning the output control string and conversion specifiers, see "Output Control String and Conversion Specifiers" on page 321.

`vsprintf()` returns the number of characters written to `s` not counting the terminating null character. Otherwise, `EOF` on failure.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 301

"printf" on page 353

"sprintf" on page 380

"Overview of stdarg.h" on page 279

Listing 35.44 Example of vsprintf() Usage

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

int spr(char *, char *, ...);

int main(void)
{
   int a = 56;
   double x = 1.003;
   static char name[] = "Charlie";
   char s[50];

   // format and send a variable number of arguments
   // to character array s
   spr(s, "%10s\n %f\n %-10d\n", name, x, a);
   puts(s);

   return 0;
}

// spr() formats and sends a variable number of
// arguments to a character array using the sprintf()
// function
int spr(char *s, char *format, ...)
{
   va_list args;
   int retval;

   va_start(args, format); // prepare the arguments
   retval = vsprintf(s, format, args);
   va_end(args);             // clean the stack
   return retval;
}
```

```
Output:
Charlie
 1.003000
 56
```

vsscanf

Read formatted text from a character string.

```
 #include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * s,
   const char * format, va_list arg);
```

Table 35.60 vsscanf

s	char *	The character string to be scanned
format	char *	The format string
arg	va_list	The variable argument list

Remarks

The `vsscantf()` function works identically to the `sscanf()` function. Instead of the variable list of arguments that can be passed to `sscanf()`, `vsscanf()` accepts its arguments in the array `arg` of type `va_list`, which must have been initialized by the `va_start()` macro from the `stdarg.h` header file. The `vfscanf()` does not invoke the `va_end` macro.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

For specifications concerning the output control string and conversion specifiers, see Table 35.35, Table 35.36, Table 35.37, Table 35.38.

vfscanf() returns the number of items assigned, which can be fewer than provided for in the case of an early matching failure. If an input failure occurs before any conversion, vfscanf() returns EOF.

This function may not be implemented on all platforms.

See Also

"scanf" on page 370

"fscanf" on page 335

Listing 35.45 Example of Vsscanf() Usage

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

int ssc(char *, char *, ...);

int main(void)
{
   static char in[] = "figs cat pear 394 road 16!";
   char s1[20], s2[20], s3[20];
   int i;

   // get the words figs, cat, road,
   // and the integer 16
   // from in and store them in s1, s2, s3, and i,
   // respectively
   ssc(in, "%s %s pear 394 %s %d!", s1, s2, s3, &i);
   printf("%s %s %s %d\n", s1, s2, s3, i);

   return 0;
}


// ssc() scans a character string and inputs
// a variable number of arguments using
// the vsscanf() function
int ssc(char * s, char *format, ...)
{
   va_list args;
   int retval;

   va_start(args, format);      // prepare the arguments
   retval = vsscanf(s, format, args);
   va_end(args);               // clean the stack
   return retval;
}
```

```
Output:
figs cat road 16
```

_wfopen

Open a file as a stream with a wide character file name.

```
#include <stdio.h>
FILE *_wfopen(const wchar_t *wfilename, const wchar_t
     *wmode);
```

Table 35.61 _wfopen

wfilename	const wchar_t *	The wide character filename of the file to open
wmode	const wchar_t *	The wide character file opening mode

Remarks

The `_wfopen()` function is a wide character implementation of "fopen" on page 317

This function may not be implemented on all platforms.

See Also

"freopen" on page 333

"fopen" on page 317

_wfreopen

Re-direct a stream to another file as a wide character version.

```
#include <stdio.h>
FILE *_wfreopen(const wchar_t *wfilename,
const wchar_t *wmode, FILE *stream);
```

Table 35.62 _wfreopen

wfilename	const wchar_t *	The wide character name of the file to re-open
wmode	const wchar_t *	The wide character file opening mode
stream	FILE *	A pointer to a FILE stream

Remarks

The `_wfreopen()` function is a wide character implementation of "freopen" on page 333

This function may not be implemented on all platforms.

See Also

"freopen" on page 333

"_wfopen" on page 399

_wremove

Delete a file.

```
#include <stdio.h>
int _wremove(const wchar_t *wfilename);
```

Table 35.63 _wremove

wfilename	const wchar_t *	The name of the wide character file to be deleted

Remarks

The `_fremove()` function is a wide character variation of "remove" on page 366

This function may not be implemented on all platforms.

See Also

"remove" on page 366

_wrename

Change the name of a file.

```
#include <stdio.h>
int _wrename(const char *wold, const wchar_t *wnew);
```

Table 35.64 _wrename

wold	const `wchar_t` *	The old wide character file name
wnew	const wchar_t *	The new wide character file name

Remarks

The `_wrename()` function implements a wide character variation of "rename" on page 367.

This function may not be implemented on all platforms.

See Also

"rename" on page 367

" wtmpnam"

_wtmpnam

Create a unique temporary filename wide character variant.

```
#include <stdio.h>
wchar_t *_wtmpnam(wchar_t *ws);
```

Table 35.65 _wtmpnam

ws	`wchar_t` *	A temporary wide character file name

Remarks

The `_wtmpnam()` functions creates a valid filename wide character string that will not conflict with any existing filename. It is implemented for a wide character array in the same manner as "tmpnam" on page 384

This function may not be implemented on all platforms.

See Also

"fopen" on page 317

"tmpfile" on page 382

36

stdlib.h

The `stdlib.h` header file provides groups of closely related functions for string conversion, pseudo-random number generation, memory management, environment communication, searching and sorting, multibyte character conversion, and integer arithmetic.

Overview of stdlib.h

The `stdlib.h` header file provides groups of closely related functions as follows:

String Conversion Functions

The string conversion functions are as follows:

Pseudo-random Number Generation Functions

The pseudo-random number generation functions are

- “rand” on page 432
- “rand_r” on page 433
- “srand” on page 435

Memory Management Functions

The memory management functions are

- “calloc” on page 417
- “free” on page 422
- “malloc” on page 427
- “realloc” on page 434
- “vec_calloc” on page 446
- “vec_free” on page 447
- “vec_malloc” on page 448
- “vec_realloc” on page 448

Environment Communication Functions

The environment communication functions are

- “abort” on page 405
- “atexit” on page 408
- “exit” on page 420
- “_Exit” on page 422
- “getenv” on page 423
- “putenv” on page 430
- “system” on page 446

Searching And Sorting Functions

The searching and sorting functions are

- “bsearch” on page 413
- “qsort” on page 431

Multibyte Conversion Functions

The multibyte conversion functions convert `UTF-8` multibyte characters to `wchar_t` type characters (defined in `stddef.h`). The functions are

- "mblen" on page 428
- "mbstowcs" on page 428
- "mbtowc" on page 429
- "wcstombs" on page 449
- "wctomb" on page 450

Integer Arithmetic Functions

The integer arithmetic functions are

- "abs" on page 406
- "div" on page 419
- "labs" on page 424
- "llabs" on page 425
- "ldiv" on page 425

Many of the `stdlib.h` functions use the `size_t` type as well as the `NULL and MB_CUR_MAX` macros, which are defined in `stdlib.h`. The macro `MB_CUR_MAX` defines the maximum number of bytes in a single multibyte character.

abort

Abnormal program termination.

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

Remarks

The `abort()` function raises the `SIGABRT` signal and quits the program to return to the operating system.

The `abort()` function will not terminate the program if a programmer-installed signal handler uses `longjmp()` instead of returning normally.

This function may not be implemented on all platforms.

See Also

"assert" on page 27
"longjmp" on page 244
"raise" on page 253
"signal" on page 251
"atexit" on page 408
"exit" on page 420

Listing 36.1 Example of abort() Usage

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

int main(void)
{
   char c;

   printf("Aborting the program.\n");
   printf("Press return.\n");

   // wait for the return key to be pressed
   c = getchar();

   // abort the program
   abort();

   return 0;
}
```

```
Output:
Aborting the program.
Press return.
```

abs

Compute the absolute value of an integer.

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

Table 36.1 abs

i	int	The value being computed

`abs()` returns the absolute value of its argument. Note that the two's complement representation of the smallest negative number has no matching absolute integer representation.

This function may not be implemented on all platforms.

See Also

"fabs" on page 188

"labs" on page 424

Listing 36.2 Example of abs() Usage

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

int main(void)
{
   int i = -20;
   long int j = -48323;
   long long k = -9223372036854773307;


   printf("Absolute value of %d is %d.\n", i, abs(i));
   printf("Absolute value of %ld is %ld.\n", j, labs(j));
   printf("Absolute value of %lld is %lld.\n", k, llabs(k));

   return 0;
}
```

```
Output:
Absolute value of -20 is 20.
Absolute value of -48323 is 48323.
Absolute value of -9223372036854773307 is 9223372036854773307.
```

atexit

Install a function to be executed at a program's exit.

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

Table 36.2 atexit

func	void *	The function to execute at exit

Remarks

The `atexit()` function adds the function pointed to by `func` to a list. When `exit()` is called, each function on the list is called in the reverse order in which they were installed with `atexit()`. After all the functions on the list have been called, `exit()` terminates the program.

The `stdio.h` library, for example, installs its own exit function using `atexit()`. This function flushes all buffers and closes all open streams.

`atexit()` returns a zero when it succeeds in installing a new exit function and returns a nonzero value when it fails.

This function may not be implemented on all platforms.

See Also

"exit" on page 420

Listing 36.3 Example of atexit() Usage

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

// Prototypes
void first(void);
void second(void);
void third(void);

int main(void)
{
   atexit(first);
   atexit(second);
   atexit(third);
```

```
   printf("exiting program\n\n");
   return 0;
}

void first(void)
{
   int c;

   printf("First exit function.\n");
   printf("Press return.\n");
// wait for the return key to be pressed
   c = getchar();
}

void second(void)
{
   int c;

   printf("Second exit function.\n");
   printf("Press return.\n");
   c = getchar();
}

void third(void)
{
   int c;

   printf("Third exit function.\n");
   printf("Press return.\n");
   c = getchar();
}
```

```
Output:
Third exit function.
Press return.

Second exit function.
Press return.

First exit function.
Press return.
```

atof

Convert a character string to a numeric value of type double.

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

Table 36.3 atof

nptr	const char *	The character being converted

Remarks

The `atof()` function converts the character array pointed to by `nptr` to a floating point value of type `double`. Except for its behavior on error, this function is the equivalent of the call `strtod(nptr, NULL);`

This function sets the global variable `errno` to `ERANGE` if the converted value cannot be expressed as a floating point value of type double.

`atof()` returns a floating point value of type `double`.

This function may not be implemented on all platforms.

See Also

"atoi" on page 411

"atol" on page 412

"errno" on page 75

"strtod" on page 435

"scanf" on page 370

Listing 36.4 Example of atof(), atoi(), atol() Usage

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

int main(void)
{
   int i;
   long int j;
   float f;
   static char si[] = "-493", sli[] = "63870";
```

```
   static char sf[] = "1823.4034";

   f = atof(sf);
   i = atoi(si);
   j = atol(sli);

   printf("%f %d %ld\n", f, i, j);

   return 0;
}
```

```
Output:
1823.403400 -493 63870
```

atoi

Convert a character string to a value of type int.

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

Table 36.4 atoi

nptr	const char *	The string to be converted

Remarks

The `atoi()` function converts the character array pointed to by `nptr` to an integer value. Except for its behavior on error, this function is the equivalent of the call `(int)strtol(nptr, (char **)NULL, 10);`

This function sets the global variable `errno` to `ERANGE` if the converted value cannot be expressed as a value of type int.

`atoi()` returns an integer value of type `int`.

This function may not be implemented on all platforms.

See Also

"atof" on page 410

"atol" on page 412

"errno" on page 75

"strtol" on page 438

"scanf" on page 370

atol

Convert a character string to a value of type long.

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

Table 36.5 atol

nptr	const char *	The string to be converted

Remarks

The `atol()` function converts the character array pointed to by `nptr` to an integer of type `long int`. Except for its behavior on error, this function is the equivalent of the call `strtol(nptr, (char **)NULL, 10);`

This function sets the global variable `errno` to `ERANGE` if the converted value cannot be expressed as a value of type long int.

`atol()` returns an integer value of type `long int`.

This function may not be implemented on all platforms.

See Also

"atof" on page 410

"atoi" on page 411

"errno" on page 75

"strtol" on page 438

"scanf" on page 370

atoll

Convert a character string to a value of type long long.

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

Table 36.6 atoll

nptr	const char *	The string to be converted

Remarks

The `atoll()` function converts the character array pointed to by `nptr` to an integer of type `long long`. Except for its behavior on error, this function is the equivalent of the call `strtoll(nptr, (char **)NULL, 10);`

This function sets the global variable `errno` to `ERANGE` if the converted value cannot be expressed as a value of type long int.

`atoll()` returns an integer value of type `long long`.

This function may not be implemented on all platforms.

See Also

"atof" on page 410

"atoi" on page 411

bsearch

This function uses the binary search algorithm to make an efficient search of a sorted array for an item.

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

Table 36.7 bsearch

key	const void *	Search criteria see remarks
base	const void *	The array to be searched see remarks
num	size_t	Number of elements see remarks

Table 36.7 bsearch (*continued*)

size	size_t	Size of an array element see remarks
compare	const void *	A pointer to a function used for comparison see remarks

Remarks

The `key` argument points to the item you want to search for.

The `base` argument points to the first byte of the array to be searched. This array must already be sorted in ascending order. This order is based on the comparison requirements of the function pointed to by the `compare` argument.

The `num` argument specifies the number of array elements to search.

The `size` argument specifies the size of an array element.

The compare argument is a pointer to a programmer-supplied function that is used to compare two elements of the array. That compare function takes two array element pointers as arguments. The first argument is the key that was passed to `bsearch`() as the first argument to `bsearch()`. The second argument is a pointer to an element of the array passed as the second argument to `bsearch()`.

For explanation, we will call the arguments search_key and array_element. The compare function compares the search_key to the array element. If the search_key and the array_element are equal, the function will return zero. If the search_key is less than the array_element, the function will return a negative value. If the search_key is greater than the array_element, the function will return a positive value.

`bsearch()` returns a pointer to the element in the array matching the item pointed to by `key`. If no match was found, `bsearch()` returns a null pointer (`NULL`).

This function may not be implemented on all platforms.

See Also

"qsort" on page 431

Listing 36.5 Example of bsearch Usage

```
// A simple telephone directory manager
// This program accepts a list of names and
// telephone numbers, sorts the list, then
// searches for specified names.
```

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

// Maximum number of records in the directory.
#define MAXDIR 40

typedef struct
{
   char lname[15];            // keyfield--see comp() function
   char fname[15];
   char phone[15];
} DIRENTRY;                  // telephone directory record

int comp(const DIRENTRY *, const DIRENTRY *);
DIRENTRY *look(char *);
DIRENTRY directory[MAXDIR];             // the directory itself
int reccount;                  // the number of records entered

int main(void)
{
   DIRENTRY *ptr;
   int lastlen;
   char lookstr[15];

   printf("Telephone directory program.\n");
   printf("Enter blank last name when done.\n");

   reccount = 0;
   ptr = directory;
   do {
      printf("\nLast name: ");
      gets(ptr->lname);
      printf("First name: ");
      gets(ptr->fname);
      printf("Phone number: ");
      gets(ptr->phone);
      if ( (lastlen = strlen(ptr->lname)) > 0) {
         reccount++;
         ptr++;
      }
   } while ( (lastlen > 0) && (reccount < MAXDIR) );

   printf("Thank you.  Now sorting. . .\n");

   // sort the array using qsort()
   qsort(directory, reccount,
```

```
            sizeof(directory[0]),(void *)comp);

   printf("Enter last name to search for,\n");
   printf("blank to quit.\n");
   printf("\nLast name: ");
   gets(lookstr);

   while ( (lastlen = strlen(lookstr)) > 0) {
      ptr = look(lookstr);
      if (ptr != NULL)
         printf("%s, %s: %s\n",
            ptr->lname,
            ptr->fname,
            ptr->phone);
      else   printf("Can't find %s.\n", lookstr);
      printf("\nLast name: ");
      gets(lookstr);
   }

   printf("Done.\n");

   return 0;
}

int comp(const DIRENTRY *rec1, const DIRENTRY *rec2)
{
   return (strcmp((char *)rec1->lname,
                  (char *)rec2->lname));
}

// search through the array using bsearch()
DIRENTRY *look(char k[])
{
   return (DIRENTRY *) bsearch(k, directory, reccount,
sizeof(directory[0]), (void *)comp);
}
```

```
Output
Telephone directory program.
Enter blank last name when done.

Last name: Mation
First name: Infor
Phone number: 555-1212

Last name: Bell
First name: Alexander
```

```
Phone number: 555-1111

Last name: Johnson
First name: Betty
Phone number: 555-1010

Last name:
First name:
Phone number:
Thank you.  Now sorting. . .
Enter last name to search for,
blank to quit.

Last name: Mation
Infor, Mation: 555-1212

Last name: Johnson
Johnson, Betty: 555-1010

Last name:
Done.
```

calloc

Allocate space for a group of objects.

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t elemsize);
```

Table 36.8 calloc

nmemb	size_t	Number of elements
elemsize	size_t	The size of the elements

Remarks

The `calloc()` function allocates contiguous space for `nmemb` elements of size `elemsize`. The space is initialized with all bits zero.

`calloc()` returns a pointer to the first byte of the memory area allocated. `calloc()` returns a null pointer (`NULL`) if no space could be allocated.

This function may not be implemented on all platforms.

See Also

"vec_calloc" on page 446

"malloc" on page 427

"realloc" on page 434

Listing 36.6 Example of calloc() Usage

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

int main(void)
{
   static char s[] = "CodeWarrior  compilers";
   char *sptr1, *sptr2, *sptr3;

   // allocate the memory three different ways
   // one: allocate a thirty byte block of
   //   uninitialized memory
   sptr1 = (char *) malloc(30);
   strcpy(sptr1, s);
   printf("Address of sptr1: %p\n", sptr1);

   // two: allocate twenty bytes of unitialized memory
   sptr2 = (char *) malloc(20);
   printf("sptr2 before reallocation: %p\n", sptr2);
   strcpy(sptr2, s);
   // now re-allocate ten extra bytes (for a total of
   // thirty bytes)
   //
   // note that the memory block pointed to by sptr2 is
   // still contiguous after the call to realloc()
   sptr2 = (char *) realloc(sptr2, 30);
   printf("sptr2 after reallocation: %p\n", sptr2);

   // three: allocate thirty bytes of initialized memory
   sptr3 = (char *) calloc(strlen(s), sizeof(char));
   strcpy(sptr3, s);
   printf("Address of sptr3: %p\n", sptr3);

   puts(sptr1);
   puts(sptr2);
   puts(sptr3);

   // release the allocated memory to the heap
   free(sptr1);
```

```
    free(sptr2);
    free(sptr3);

    return 0;
}
```

```
Output:
Address of sptr1: 5e5432
sptr2 before reallocation: 5e5452
sptr2 after reallocation: 5e5468
Address of sptr3: 5e5488
CodeWarrior  compilers
CodeWarrior  compilers
CodeWarrior  compilers
```

div

Compute the integer quotient and remainder.

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

Table 36.9 div

numer	int	The numerator
denom	int	The denominator

Remarks

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

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

`div()` divides `denom` into `numer` and returns the quotient and remainder as a `div_t` type.

This function may not be implemented on all platforms.

See Also

"fmod" on page 191

"ldiv" on page 425

"div_t" on page 73

Listing 36.7 Example of div() Usage

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

int main(void)
{
   div_t result;
   ldiv_t lresult;

   int d = 10, n = 103;
   long int ld = 1000L, ln = 1000005L;

   result = div(n, d);
   lresult = ldiv(ln, ld);

   printf("%d / %d has a quotient of %d\n",
            n, d, result.quot);
   printf("and a remainder of %d\n", result.rem);
   printf("%ld / %ld has a quotient of %ld\n",
            ln, ld, lresult.quot);
   printf("and a remainder of %ld\n", lresult.rem);

   return 0;
}
```

```
Output:
103 / 10 has a quotient of 10
and a remainder of 3
1000005 / 1000 has a quotient of 1000
and a remainder of 5
```

exit

Terminate a program normally.

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

```
void exit(int status);
```

Table 36.10 exit

status	int	The exit error value

Remarks

The `exit()` function calls every function installed with `atexit()` in the reverse order of their installation, flushes the buffers and closes all open streams, then calls the Toolbox system call `ExitToShell()`.

`exit()` does not return any value to the operating system. The `status` argument is kept to conform to the ANSI C Standard Library specification.

This function may not be implemented on all platforms.

See Also

"abort" on page 405

"atexit" on page 408

Listing 36.8 Example of exit() Usage

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

int main(void)
{
   FILE *f;
   int count;

   // create a new file for output exit on failure
   if (( f = fopen("foofoo", "w")) == NULL) {
      printf("Can't create file.\n");
      exit(1);
   }

   // output numbers 0 to 9
   for (count = 0; count < 10; count++)
      fprintf(f, "%5d", count);

   // close the file
   fclose(f);

   return 0;
}
```

_Exit

The _Exit function causes normal program termination

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

Table 36.11 _Exit

status	int	exit status

Remarks

This function can not return but a status value is passed back to the calling host in the same manner as in the function "exit" on page 420.

The effects on open buffers is implementation defined.

This function may not be implemented on all platforms.

See Also

"abort" on page 405

"atexit" on page 408

For example usage of exit(). see Listing 36.8.

free

Release previously allocated memory to heap.

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

Table 36.12 free

ptr	void *	A pointer to the allocated memory

Remarks

The `free()` function releases a previously allocated memory block, pointed to by `ptr`, to the heap. The `ptr` argument should hold an address returned by the

memory allocation functions `calloc()`, `malloc()`, or `realloc()`. Once the memory block pointed to by `ptr` has been released, it is no longer valid. The `ptr` variable should not be used to reference memory again until it is assigned a value from the memory allocation functions. For example usage, see Listing 36.6.

This function may not be implemented on all platforms.

See Also

"vec_free" on page 447

"calloc" on page 417

"malloc" on page 427

"realloc" on page 434

getenv

Environment list access.

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

Table 36.13 getenv

name	char *	A buffer for the environment list

Remarks

For Macintosh systems, the `getenv()` is an empty function that always returns a null pointer (`NULL`). It is included in the CodeWarrior `stdlib.h` header file to conform to the ANSI C Standard Library specification.

`getenv()` returns `NULL` for the Mac. For Windows, `getenv()` returns zero on failure or the environmental variable.

This function may not be implemented on all platforms.

See Also

"system" on page 446

Listing 36.9 Example of getenv() Usage

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

```
#include <stdlib.h>

int main(void)
{
   char *value;
   char *var = "path";

   if( (value = getenv(var)) == NULL)
   { printf("%s is not a environmental variable", var);}
   else
   { printf("%s = %s \n", var, value);}

   return 0;
}
```

```
Result:
path = c:\program files\Freescale\codewarrior;c:\WINNT\system32
```

labs

Compute long integer absolute value.

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

Table 36.14 labs

j	long int	The variable to be computed

Remarks

`labs()` returns the absolute value of its argument as a value of type `long int`. For example usage, see Listing 36.2.

This function may not be implemented on all platforms.

See Also

"fabs" on page 188

"abs" on page 406

ldiv

Compute the long integer quotient and remainder.

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

Table 36.15 ldiv

numer	long int	The numerator
denom	long int	The denominator

Remarks

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

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

`ldiv()` divides `denom` into `numer` and returns the quotient and remainder as an `ldiv_t` type. For example usage, see Listing 36.7.

This function may not be implemented on all platforms.

See Also

"fmod" on page 191

"div" on page 419

"ldiv_t" on page 74

"lldiv_t" on page 74

llabs

Compute long long integer absolute value.

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

Table 36.16 llabs

j	long long	The variable to be computed

Remarks

`llabs()` returns the absolute value of its argument as a value of type `long long`. For example usage, see Listing 36.2.

This function may not be implemented on all platforms.

See Also

"fabs" on page 188

"abs" on page 406

lldiv

Compute the long long integer quotient and remainder.

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

Table 36.17 ldiv

numer	long long	The numerator
denom	long long	The denominator

Remarks

The `lldiv_t` type is defined in `<div_t.h>` as

```
typedef struct {
   long long quot, rem;
} lldiv_t;
```

`lldiv()` divides `denom` into `numer` and returns the quotient and remainder as an `lldiv_t` type.

This function may not be implemented on all platforms.

See Also

"ldiv" on page 425
"fmod" on page 191
"lldiv_t" on page 74

malloc

Allocate a block of heap memory.

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

Table 36.18 malloc

size	size_t	The size in bytes of the allocation

Remarks

The `malloc()` function allocates a block of contiguous heap memory `size` bytes large. If the argument for `malloc()` is zero, the behavior is unspecified. Dependent upon the system, either a null pointer is returned, or the behavior is as if the size was not zero, except that the returned pointer can not be used to access an object. For example usage, see Listing 36.6.

`malloc()` returns a pointer to the first byte of the allocated block if it is successful and return a null pointer if it fails.

This function may not be implemented on all platforms.

See Also

"vec_malloc" on page 448
"calloc" on page 417
"free" on page 422
"realloc" on page 434

mblen

Compute the length of an encoded multibyte character, encoded as defined by the LC_CTYPE category of the current locale.

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Table 36.19 mblen

s	const char *	The multibyte array to measure
n	size_t	The maximum size

Remarks

The mblen() function returns the length of the multibyte character pointed to by s. It examines a maximum of n characters.

If s is a null pointer, the mblen function returns a nonzero or zero value signifying whether multibyte encoding do or do not have state-dependent encoding. If s is not a null pointer, the mblen function either returns 0 (if s points to the null character), or returns the number of bytes that are contained in the multibyte.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165

"mbtowc" on page 429

mbstowcs

Convert a multibyte character array encoded as defined by the LC_CTYPE category of the current locale to a wchar_t array.

```
#include <stddlib.h>
size_t mbstowcs(wchar_t *pwcs,
      const char *s, size_t n);
```

Table 36.20 mbstowcs

pwcs	wchar_t *	The wide character destination
s	const char *s	The multibyte string to convert
n	size_t	The maximum wide characters to convert

Remarks

The MSL C implementation of `mbstowcs()` converts a sequence of multibyte characters encoded as defined by the LC_CTYPE category of the current locale from the character array pointed to by `s` and stores not more than `n` of the corresponding Unicode characters into the wide character array pointed to by `pwcs`. No multibyte characters that follow a `null` character (which is converted into a `null` wide character) will be examined or converted.

If an invalidly encoded character is encountered, `mbstowcs()` returns the value `(size_t)(-1)`. Otherwise `mbstowcs` returns the number of elements of the array pointed to by `pwcs` modified, not including any terminating null wide character.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165

"wcstombs" on page 449

mbtowc

Translate a multibyte character, encoded as defined by the `LC_CTYPE` category of the current locale, to a `wchar_t` type.

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Table 36.21 mbtowc

pwc	wchar_t *	The wide character destination
s	const char *s	The string to convert
n	size_t	The maximum wide characters to convert

Remarks

If `s` is not a `null pointer`, the `mbtowc()` function examines at most `n` bytes starting with the byte pointed to by `s` to determine whether the next multibyte character is a complete and valid encoding of a Unicode character encoded as defined by the LC_CTYPE category of the current locale. If so, and `pwc` is not a `null pointer`, it converts the multibyte character, pointed to by `s`, to a character of type `wchar_t`, pointed to by `pwc`.

`mbtowc()` returns `-1` if n is zero and `s` is not a null pointer or if *s* points to an incomplete or invalid multibyte encoding.

`mbtowc()` returns 0 if `s` is a null pointer or s points to a null character (`'\0'`).

`mbtowc()` returns the number of bytes of `s` required to form a complete and valid multibyte encoding of the Unicode character.

In no case will the value returned be greater than n or the value of the macro `MB_CUR_MAX`.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165

"mblen" on page 428

"wctomb" on page 450

_putenv

This functions lets you enter an argument into the environment list.

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

Table 36.22 _putenv

name	const char *	The item to add to the environment list

The function `_putenv()` returns `NULL` on success or minus one on failure to enter the environmental variable.

This function may not be implemented on all platforms.

See Also

"getenv" on page 423

"system" on page 446

qsort

Sort an array.

```
#include <stdlib.h>
void qsort(void *base,size_t nmemb,size_t size,
    int (*compare) (const void *, const void *))
```

Table 36.23 qsort

base	void *	A pointer to the array to be sorted
nmemb	size_t	The number of elements
size	size_t	The size of the elements
compare	void *	A pointer to a comparison function

Remarks

The `qsort()` function sorts an array using the quicksort algorithm. It sorts the array without displacing it; the array occupies the same memory it had before the call to `qsort()`. For example usage, see Listing 36.5.

The `base` argument is a pointer to the base of the array to be sorted.

The `nmemb` argument specifies the number of array elements to sort.

The `size` argument specifies the size of an array element.

The `compare` argument is a pointer to a programmer-supplied compare function. The function takes two pointers to different array elements and compares them based on the key. If the two elements are equal, `compare` must return a zero. The `compare` function must return a negative number if the first element is less than the second. Likewise, the function must return a positive number if the first argument is greater than the second.

This function may not be implemented on all platforms.

See Also

"bsearch" on page 413

rand

Generate a pseudo-random integer value.

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

Remarks

A sequence of calls to the `rand()` function generates and returns a sequence of pseudo-random integer values from 0 to `RAND_MAX`. The `RAND_MAX` macro is defined in `stdlib.h`.

By seeding the random number generator using `srand()`, different random number sequences can be generated with `rand()`.

`rand()` returns a pseudo-random integer value between 0 and `RAND_MAX`.

This function may not be implemented on all platforms.

See Also

"srand" on page 435

Listing 36.10 Example of rand() Usage

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

int main(void)
{
   int i;
   unsigned int seed;
```

```
   for (seed = 1; seed <= 5; seed++) {
      srand(seed);
      printf("First five random numbers for seed %d:\n", seed);
      for (i = 0; i < 5; i++)
         printf("%10d", rand());
      printf("\n\n");      // terminate the line
   }

   return 0;
}
```

```
Output:
First five random numbers for seed 1:
     16838     5758     10113     17515     31051

First five random numbers for seed 2:
     908     22817     10239     12914     25837

First five random numbers for seed 3:
     17747     7107     10365     8312     20622

First five random numbers for seed 4:
     1817     24166     10491     3711     15407

First five random numbers for seed 5:
     18655     8457     10616     31877     10193
```

rand_r

Reentrant function to generate a pseudo-random integer value.

```
#include <stdlib.h>
int rand_r(unsigned int *context);
```

Table 36.24 rand_r

context	unsigned int *	The context seed value

Remarks

The `rand_r()` function provides the same service as "rand" on page 432, yet it also combines the functionality of `srand()` as well. The result of `rand_r()`

is equivalent to calling `srand()` with a context seed value, then calling `rand()`. The difference is that for `rand_r()`, the caller provides the storage for the context seed value.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"rand" on page 432

"srand" on page 435

realloc

Change the size of an allocated block of heap memory.

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

Table 36.25 realloc

ptr	void *	A pointer to an allocated block of memory
size	size_t	The size of memory to reallocate

Remarks

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The `size` argument can have a value smaller or larger than the current size of the block `ptr` points to. The `ptr` argument should be a value assigned by the memory allocation functions `calloc()` and `malloc()`. For example usage, see Listing 36.6.

If `size` is 0, the memory block pointed to by `ptr` is released. If `ptr` is a null pointer, `realloc()` allocates `size` bytes.

The old contents of the memory block are preserved in the new block if the new block is larger than the old. If the new block is smaller, the extra bytes are cut from the end of the old block.

`realloc()` returns a pointer to the new block if it is successful and `size` is greater than 0. `realloc()` returns a null pointer if it fails or `size` is 0.

This function may not be implemented on all platforms.

See Also

"vec_realloc" on page 448

"calloc" on page 417

"free" on page 422

"malloc" on page 427

srand

Sets the seed for the pseudo-random number generator.

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

Table 36.26 srand

seed	unsigned int	A seeding value

Remarks

The `srand()` function sets the seed for the pseudo-random number generator to `start`. Further calls of srand() with the same seed value produces the same sequence of random numbers. For example usage, see Listing 36.10.

This function may not be implemented on all platforms.

This function may not be implemented on all platforms.

See Also

"rand" on page 432

strtod

Character array conversion to floating point value of type double.

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

Table 36.27 strtod

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptr that is follows the converted part.

Remarks

The strtod() function converts a character array, pointed to by nptr, to a floating point value of type double. The character array can be in either decimal or hexadecimal floating point constant notation (e.g. `103.578`, `1.03578e+02`, or `0x1.9efef9p+6`).

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to a value of type double.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`strtod()` returns a floating point value of type `double`. If `nptr` cannot be converted to an expressible double value, `strtod()` returns `HUGE_VAL`, defined in `math.h`, and sets `errno` to `ERANGE`.

This function may not be implemented on all platforms.

See Also

"strtol" on page 438

"strtoul" on page 443

"errno" on page 75

"Integral type limits" on page 163

"Overview of math.h" on page 171

"scanf" on page 370

Listing 36.11 Example of strtod() and strtold() Usage

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

int main(void)
```

```
{
   double f;
   long double lf;
   static char sf1[] = "103.578 777";
   static char sf2[] = "1.03578e+02 777";
   static char sf3[] = "0x1.9efef9p+6 777";
   char *endptr;

   f = strtod(sf1, &endptr);
   printf("Value = %f remainder of string = |%s|\n", f, endptr);

   f = strtod(sf2, &endptr);
   printf("Value = %f remainder of string = |%s|\n", f, endptr);

   f = strtod(sf3, &endptr);
   printf("Value = %f remainder of string = |%s|\n", f, endptr);

   lf = strtold(sf1, &endptr);
   printf("Value = %lf remainder of string = |%s|\n", lf, endptr);

   lf = strtold(sf2, &endptr);
   printf("Value = %lf remainder of string = |%s|\n", lf, endptr);

   lf = strtold(sf3, &endptr);
   printf("Value = %lf remainder of string = |%s|\n", lf, endptr);

   return 0;
}
```

```
Output:
Value = 103.578000 remainder of string = | 777|
Value = 103.578000 remainder of string = | 777|
Value = 103.748997 remainder of string = | 777|
Value = 103.578000 remainder of string = | 777|
Value = 103.578000 remainder of string = | 777|
Value = 103.748997 remainder of string = | 777|
```

strtof

Character array conversion to floating point value of type float.

```
#include <stdlib.h>

float strtof(const char *nptr, char **endptr);
```

Table 36.28 strtof

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptr that is follows the converted part.

Remarks

The strtof() function converts a character array, pointed to by nptr, to a floating point value of type float. The character array can be in either decimal or hexadecimal floating point constant notation (e.g. `103.578`, `1.03578e+02`, or `0x1.9efef9p+6`).

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to a value of type `float`.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`strtof()` returns a floating point value of type `float`. If `nptr` cannot be converted to an expressible float value, `strtof()` returns `HUGE_VAL`, defined in `math.h`, and sets `errno` to `ERANGE`.

This function may not be implemented on all platforms.

See Also

"strtol" on page 438

"strtod" on page 435

strtol

Character array conversion to an integral value of type long int.

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

Table 36.29 strtol

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `strtol()` function converts a character array, pointed to by `nptr`, expected to represent an integer expressed in radix base, to an integer value of type `long int`. If the sequence pointed to by nptr is a minus sign, the value resulting from the conversion is negated in the return value.

The `base` argument in `strtol()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If `base` is 0, then `strtol()` converts the character array based on its format. Character arrays beginning with `'0'` are assumed to be octal, number strings beginning with `'0x'` or `'0X'` are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to a long int value.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`strtol()` returns an integer value of type `long int`. If the converted value is less than `LONG_MIN`, `strtol()` returns `LONG_MIN` and sets `errno` to `ERANGE`. If the converted value is greater than `LONG_MAX`, `strtol()` returns `LONG_MAX` and sets `errno` to `ERANGE`. The `LONG_MIN` and `LONG_MAX` macros are defined in `limits.h`.

This function may not be implemented on all platforms.

See Also

"strtod" on page 435

"strtoul" on page 443

"errno" on page 75
"Integral type limits" on page 163
"Overview of math.h" on page 171
"scanf" on page 370

Listing 36.12 Example of strtol(), strtoul(), strtoll(), and strtoull() Usage

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   long int i;
   unsigned long int j;
   long long int lli;
   unsigned long long ull;
   static char si[] = "4733 777";
   static char sb[] = "0x10*****";
   static char sc[] = "66E00M???";
   static char sd[] = "Q0N50M abcd";

   char *endptr;
   i = strtol(si, &endptr, 10);
   printf("%ld remainder of string = |%s|\n", i, endptr);
   i = strtol(si, &endptr, 8);
   printf("%ld remainder of string = |%s|\n", i, endptr);
   j = strtoul(sb, &endptr, 0);
   printf("%lu remainder of string = |%s|\n", j, endptr);
   j = strtoul(sb, &endptr, 16);
   printf("%lu remainder of string = |%s|\n", j, endptr);
   lli = strtoll(sc, &endptr, 36);
   printf("%lld remainder of string = |%s|\n", lli, endptr);
   ull = strtoull(sd, &endptr, 27);
   printf("%llu remainder of string = |%s|\n", ull, endptr);
   return 0;
}
```

```
Output:
4733 remainder of string = | 777|
2523 remainder of string = | 777|
16 remainder of string = |*****|
16 remainder of string = |*****|
373527958 remainder of string = |???|
373527958 remainder of string = | abcd|
```

strtold

Character array conversion to floating point value of type long double.

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

Table 36.30 strtold

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptr that follows the converted part

Remarks

The strtold() function converts a character array, pointed to by nptr, to a floating point value of type long double. The character array can be in either decimal or hexadecimal floating point constant notation (e.g. `103.578`, `1.03578e+02`, or `0x1.9efef9p+6`). For example usage, see Listing 36.11.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to a value of type double.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`strtold()` returns a floating point value of type `long double`. If `nptr` cannot be converted to an expressible double value, `strtold()` returns `HUGE_VAL`, defined in `math.h`, and sets `errno` to `ERANGE`.

This function may not be implemented on all platforms.

See Also

"strtol" on page 438

"errno" on page 75

"Overview of math.h" on page 171

strtoll

Character array conversion to integer value of type long long int.

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

Table 36.31 strtoul

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `strtoll()` function converts a character array, pointed to by `nptr`, expected to represent an integer expressed in radix base to an integer value of type long long int. If the sequence pointed to by nptr is a minus sign, the value resulting from the conversion is negated in the return value. For example usage, see Listing 36.12.

The base argument in `strtoll()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then `strtoll()` converts the character array based on its format. Character arrays beginning with'0' are assumed to be octal, number strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to a long int value.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`strtoll()` returns an unsigned integer value of type `long long int`. If the converted value is less than `LLONG_MIN`, `strtoll()` returns `LLONG_MIN` and sets `errno` to `ERANGE`. If the converted value is greater than `LLONG_MAX`, `strtoll()` returns `LLONG_MAX` and sets `errno` to `ERANGE`. The `LLONG_MIN` and `LLONG_MAX` macros are defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"strtod" on page 435

"strtol" on page 438

"errno" on page 75

"Integral type limits" on page 163

"Overview of math.h" on page 171

strtoul

Character array conversion to integer value of type unsigned long int.

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

Table 36.32 strtoul

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `strtoul()` function converts a character array, pointed to by `nptr`, to an integer value of type `unsigned long int`, in base. If the sequence pointed to by nptr is a minus sign, the value resulting from the conversion is negated in the return value. For example usage, see Listing 36.12.

The `base` argument in `strtoul()` specifies the base used for conversion. It must have a value between 2 and 36, or 0.The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If `base` is 0, then `strtol()` and `strtoul()` convert the character array based on its format. Character arrays beginning with `'0'` are assumed to be octal, number strings beginning with `'0x'` or `'0X'` are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to the functions' respective types.

In other than the `"C" locale`, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

The function `strtoul()` returns an `unsigned integer` value of type `unsigned long int` (which may have a negative sign if the original string was negative.) If the converted value is greater than `ULONG_MAX`, `strtoul()` returns `ULONG_MAX` and sets `errno` to `ERANGE`. The `ULONG_MAX` macro is defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"strtod" on page 435

"strtol" on page 438

"errno" on page 75

"Integral type limits" on page 163

"Overview of math.h" on page 171

strtoull

Character array conversion to integer value of type unsigned long long int.

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

Table 36.33 strtoul

nptr	const char *	A Null terminated array to convert
endptr	char **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `strtoull()` function converts a character array, pointed to by `nptr`, expected to represent an integer expressed in radix base to an integer value of type `unsigned long long int`. If the sequence pointed to by nptr is a minus sign, the value resulting from the conversion is negated in the return value. For example usage, see Listing 36.12.

The base argument in `strtoull()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then `strtoull()` converts the character array based on its format. Character arrays beginning with'0' are assumed to be octal, number strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a `null` pointer, it is assigned a pointer to a position within the character array pointed to by `nptr`. This position marks the first character that is not convertible to a long int value.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

The function `strtoull()` returns an `unsigned integer` value of type `unsigned long long int` (which may have a negative sign if the original string was negative.) If the converted value is greater than `ULLONG_MAX`, `strtoull()` returns `ULLONG_MAX` and sets `errno` to `ERANGE`. The `ULLONG_MAX` macro is defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"strtoul" on page 443

"strtoll" on page 442

system

Environment list assignment.

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

NOTE The `system()` function is an empty function on MacOS t

Table 36.34 system

string	const char *	A OS system command

`system()` returns zero if successful or minus one on failure.

This function may not be implemented on all platforms.

See Also

"getenv" on page 423

vec_calloc

Clears and allocates memory on a 16 byte alignment.

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

Table 36.35 vec_calloc

nmemb	size_t	Number of elements
size	size_t	The size of the elements

Remarks

The `vec_calloc()` function allocates contiguous space for `nmemb` elements of size. The space is initialized with zeroes.

`vec_calloc()` returns a pointer to the first byte of the memory area allocated. `vec_calloc()` returns a null pointer (`NULL`) if no space could be allocated.

This function may not be implemented on all platforms.

See Also

"calloc" on page 417

"vec_free" on page 447

"vec_malloc" on page 448

"vec_realloc" on page 448

vec_free

Frees memory allocated by `vec_malloc`, `vec_calloc` and `vec_realloc`

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

Table 36.36 vec_free

ptr	void *	A pointer to the allocated memory

Remarks

The `vec_free()` function releases a previously allocated memory block, pointed to by `ptr`, to the heap. The `ptr` argument should hold an address returned by the memory allocation functions `vec_calloc()`, `vec_malloc()`, or `vec_realloc()`. Once the memory block `ptr` points to has been released, it is no longer valid. The `ptr` variable should not be used to reference memory again until it is assigned a value from the memory allocation functions.

There is no return.

This function may not be implemented on all platforms.

See Also

"free" on page 422

"vec_calloc" on page 446

"vec_malloc" on page 448

"vec_realloc" on page 448

vec_malloc

Allocates memory on a 16 byte alignment.

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

Table 36.37 vec_malloc

size	size_t	The size in bytes of the allocation

Remarks

The vec_malloc() function allocates a block of contiguous heap memory size bytes large.

vec_malloc() returns a pointer to the first byte of the allocated block if it is successful and return a null pointer if it fails.

This function may not be implemented on all platforms.

See Also

"malloc" on page 427

"vec_free" on page 447

"vec_calloc" on page 446

"vec_realloc" on page 448

vec_realloc

Reallocates memory on a 16 byte alignment.

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

Table 36.38 vec_realloc

ptr	void *	A pointer to an allocated block of memory
size	size_t	The size of memory to reallocate

`vec_realloc()` returns a pointer to the new block if it is successful and `size` is greater than 0. `realloc()` returns a null pointer if it fails or `size` is 0.

This function may not be implemented on all platforms.

See Also

"realloc" on page 434

"vec_free" on page 447

"vec_calloc" on page 446

"vec_malloc" on page 448

wcstombs

Translate a `wchar_t` type character array to a multibyte character array encoded as defined by the `LC_CTYPE` category of the current locale.

```
#include <stdlib.h>
size_t wcstombs(char *s, const
      wchar_t *pwcs, size_t n);
```

Table 36.39 wcstombs

s	char *	A multibyte string buffer
pwcs	const wchar_t *	A pointer to a wide character string to be converted
n	size_t	The maximum length to convert

Remarks

The MSL C implementation of the `wcstombs()` function converts a character array containing `wchar_t` type Unicode characters to a character array containing multibyte characters encoded as defined by the LC_CTYPE category of the current locale. The `wchar_t` type is defined in `stddef.h`. Each wide character is converted as if by a call to the `wctomb()` function. No more than `n` bytes will be modified in the array pointed to by `s`.

The function terminates prematurely if a `null` character is reached.

`wcstombs()` returns the number of bytes modified in the character array pointed to by `s`, not including a terminating null character, if any.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165, "mbstowcs" on page 428

wctomb

Translate a `wchar_t` type to a multibyte character encoded as defined by the LC_CTYPE category of the current locale.

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Table 36.40 wctomb

s	char *	A multibyte string buffer
wchar	wchar_t	A wide character to convert

Remarks

The MSL C implementation of the `wctomb()` function converts a `wchar_t` type Unicode character to a multibyte character encoded as defined by the LC_CTYPE category of the current locale. If s is not a null pointer, the encoded multibyte character is stored in the array whose first element is pointed to by `s`. At most `MB_CUR_MAX` characters are stored. If `wchar` is a null wide character, a null byte is stored.

`wctomb()` returns `1` if s is not null and returns `0`, otherwise it returns the number of bytes that are contained in the multibyte character stored in the array whose first element is pointed to by `s`.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165

"mbtowc" on page 429

Non Standard <stdlib.h> Functions

Various non standard functions are included in the header `stdlib.h` for legacy source code and compatibility with operating system frameworks and application programming interfaces.

- For the function `gcvt`, see "gcvt" on page 85 for a full description.
- For the function itoa, see "itoa" on page 87 for a full description.
- For the function `itow`, see "itow" on page 88 for a full description.
- For the function `ltoa`, see "ltoa" on page 88 for a full description.
- For the function `makepath`, see "makepath" on page 90 for a full description.
- For the function `splitpath`, see "splitpath" on page 92 for a full description.
- For the function `ultoa`, see "ultoa" on page 105 for a full description.
- For the function `wtoi`, see "wtoi" on page 115 for a full description.

37

string.h

The `string.h` header file provides functions for comparing, copying, concatenating, and searching character arrays and arrays of larger items.

Overview of string.h

The `string.h` header file provides multiple functions with and without length limits, with and without case sensitivity for comparing, copying, concatenating, and searching character arrays and generic arrays of larger items in memory.

The function naming convention used in `string.h` determines the type of data structure(s) a function manipulates.

A function with an `str` prefix operates on character arrays terminated with a null character (`'\0'`). The `str` functions are as follows:

- “strcat” on page 459 concatenates strings.
- “strchr” on page 460 searches by character.
- “strcmp” on page 461 compares strings.
- “strcpy” on page 464 copies strings.
- “strcoll” on page 462 compares string lexicographically.
- “strcspn” on page 465 finds a substring in a string.
- “strerror” on page 466 retrieves an error message from an errno variable.
- “strerror_r” on page 467 translates an error number into an error message (re-entrant version of strerror).
- “strlen” on page 468 returns string’s length.
- “strpbrk” on page 473 looks for an occurrence of a character from one string in another.
- “strrchr” on page 474 searches a string for a character.
- “strspn” on page 475 searches for a character not in one string in another.
- “strstr” on page 476 searches a string for a string.
- “strtok” on page 477 retrieves the next token or substring.
- “strxfrm” on page 479 transforms a string to a locale.

A function with an `strn` prefix operates on character arrays of a length specified as a function argument. The `strn` functions are:

- "strncat" on page 469 concatenates strings with length specified.
- "strncmp" on page 470 compares strings with length specified.
- "strncpy" on page 472 copies a specified number of characters.

A function with a `mem` prefix operates on arrays of items or contiguous blocks of memory. The size of the array or block of memory is specified as a function argument. The `mem` functions are:

- "memchr" on page 454 searches a memory block for a character.
- "memcmp" on page 456 compares a memory block.
- "memcpy" on page 457 copies a memory block.
- "memmove" on page 458 moves a memory block.
- "memset" on page 458 sets a value for a memory block.

The nonstandard functions with a 'stri' prefix operate on strings ignoring case.

memchr

Search for an occurrence of a character.

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

Table 37.1 memchr

s	const void *	The memory to search
c	int	The char to search for
n	size_t	The maximum length to search

Remarks

The `memchr()` function looks for the first occurrence of `c` in the first `n` characters of the memory area pointed to by `s`.

`memchr()` returns a pointer to the found character, or a null pointer (`NULL`) if `c` cannot be found.

This function may not be implemented on all platforms.

See Also

"strchr" on page 460

"strrchr" on page 474

Listing 37.1 Example of memchr() Usage

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

#define ARRAYSIZE 100

int main(void)
{
   // s1 must by same length as s2 for this example!
   static char s1[ARRAYSIZE] = "laugh* giggle 231!";
   static char s2[ARRAYSIZE] = "grunt sigh# snort!";
   char dest[ARRAYSIZE];
   char *strptr;
   int len1, len2, lendest;

   // Clear destination string using memset()
   memset( (char *)dest, '\0', ARRAYSIZE);

   // String lengths are needed by the mem functions
   // Add 1 to include the terminating '\0' character
   len1 = strlen(s1) + 1;
   len2 = strlen(s2) + 1;
   lendest = strlen(dest) + 1;

   printf(" s1=%s\n s2=%s\n dest=%s\n\n", s1, s2, dest);

   if (memcmp( (char *)s1, (char *)s2, len1) > 0)
      memcpy( (char *)dest, (char *)s1, len1);
   else
      memcpy( (char *)dest, (char *)s2, len2);

   printf(" s1=%s\n s2=%s\n dest=%s\n\n", s1, s2, dest);

   // copy s1 onto itself using memchr() and memmove()
   strptr = (char *)memchr( (char *)s1, '*', len1);
   memmove( (char *)strptr, (char *)s1, len1);

   printf(" s1=%s\n s2=%s\n dest=%s\n\n", s1, s2, dest);

   return 0;
}
```

```
Output:
s1=laugh* giggle 231!
 s2=grunt sigh# snort!
 dest=

 s1=laugh* giggle 231!
 s2=grunt sigh# snort!
 dest=laugh* giggle 231!

 s1=laughlaugh* giggle 231!
 s2=grunt sigh# snort!
 dest=laugh* giggle 231!
```

memcmp

Compare two blocks of memory.

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

Table 37.2 memcmp

s1	const void *	The memory to compare
s2	const void *	The comparison memory
n	size_t	The maximum length to compare

Remarks

The `memcmp()` function compares the first `n` characters of `s1` to `s2` one character at a time. For example usage, see Listing 37.1.

`memcmp()` returns a zero if all `n` characters pointed to by `s1` and `s2` are equal.

`memcmp()` returns a negative value if the first non-matching character pointed to by `s1` is less than the character pointed to by `s2`.

`memcmp()` returns a positive value if the first non-matching character pointed to by `s1` is greater than the character pointed to by `s2`.

This function may not be implemented on all platforms.

See Also

"strcmp" on page 461

"strncmp" on page 470

memcpy

Copy a contiguous memory block.

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

Table 37.3 memcpy

dest	void *	The destination memory
source	const void *	The source to copy
n	size_t	The maximum length to copy

Remarks

The `memcpy()` function copies the first `n` characters from the item pointed to by `source` to the item pointed to by `dest`. The behavior of `memcpy()` is undefined if the areas pointed to by `dest` and `source` overlap. The `memmove()` function reliably copies overlapping memory blocks. For example usage, see Listing 37.1.

`memcpy()` returns the value of `dest`.

This function may not be implemented on all platforms.

See Also

"memmove" on page 458

"strcpy" on page 464

"strncpy" on page 472

memmove

Copy an overlapping contiguous memory block.

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

Table 37.4 memmove

dest	void *	The Memory destination
source	const void *	The source to be moved
n	size_t	The maximum length to move

Remarks

The `memmove()` function copies the first `n` characters of the item pointed to by `source` to the item pointed to by `dest`. For example usage, see Listing 37.1.

Unlike `memcpy()`, the `memmove()` function safely copies overlapping memory blocks.

`memmove()` returns the value of `dest`.

This function may not be implemented on all platforms.

See Also

"memcpy" on page 457

"memset" on page 458

"strcpy" on page 464

"strncpy" on page 472

memset

Set the contents of a block of memory to the value of a single character.

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

Table 37.5 memset

dest	void *	The destination memory
c	int	The char to set
n	size_t	The maximum length to set

Remarks

The `memset()` function assigns `c` to the first `n` characters of the item pointed to by `dest`. For example usage, see Listing 37.1.

`memset()` returns the value of `dest`.

This function may not be implemented on all platforms.

strcat

Concatenate two character arrays.

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

Table 37.6 strcat

dest	char *	The destination string
source	const char *	The source to append

Remarks

The `strcat()` function appends a copy of the character array pointed to by `source` to the end of the character array pointed to by `dest`. The `dest` and `source` arguments must both point to null terminated character arrays. `strcat()` null terminates the resulting character array.

`strcat()` returns the value of dest.

This function may not be implemented on all platforms.

See Also

"strcpy" on page 464

Listing 37.2 Example of strcat() Usage

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

int main(void)
{
   char s1[100] = "The quick brown fox ";
   static char s2[] = "jumped over the lazy dog.";

      strcat(s1, s2);
   puts(s1);

   return 0;
}
```

```
Output:
The quick brown fox jumped over the lazy dog.
```

strchr

Search for an occurrence of a character.

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

```
char *strchr(const char *s, int c);
```

Table 37.7 strchr

s	const char *	The string to search
c	int	The char to search for

Remarks

The `strchr()` function searches for the first occurrence of the character `c` in the character array pointed to by `s`. The `s` argument must point to a null terminated character array.

`strchr()` returns a pointer to the successfully located character. If it fails, `strchr()` returns a null pointer (`NULL`).

This function may not be implemented on all platforms.

See Also

"memchr" on page 454

"strrchr" on page 474

Listing 37.3 Example of strchr() Usage

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

int main(void)
{
   static char s[] = "tree * tomato eggplant garlic";
   char *strptr;

   strptr = strchr(s, '*');
   puts(strptr);

   return 0;
}
```

```
Output:
* tomato eggplant garlic
```

strcmp

Compare two character arrays.

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

Table 37.8 strcmp

s1	const char *	The string to compare
s2	const char *	The comparison string

Remarks

The `strcmp()` function compares the character array pointed to by `s1` to the character array pointed to by `s2`. Both `s1` and `s2` must point to null terminated character arrays.

`strcmp()` returns a zero if `s1` and `s2` are equal, a negative value if `s1` is less than `s2`, and a positive value if `s1` is greater than `s2`.

This function may not be implemented on all platforms.

See Also

"memcmp" on page 456

"strcoll" on page 462

"strncmp" on page 470

Listing 37.4 Example of strcmp() Usage

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

int main (void)
{
   static char s1[] = "butter", s2[] = "olive oil";
   char dest[20];

   if (strcmp(s1, s2) < 0)
      strcpy(dest, s2);
   else
      strcpy(dest, s1);

   printf(" s1=%s\n s2=%s\n dest=%s\n", s1, s2, dest);

   return 0;
}
```

```
Output:
 s1=butter
 s2=olive oil
 dest=olive oil
```

strcoll

Compare two character arrays according to locale.

```
#include <string.h>

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

Table 37.9 strcoll

s1	const char *	The string to compare
s2	const char *	The comparison string

Remarks

The `strcoll()` function compares two character arrays based on the `LC_COLLATE` component of the current locale.

The MSL C implementation of `strcoll()` compares two character arrays using `strcmp()`. It is included in the string library to conform to the ANSI C Standard Library specification.

`strcoll()` returns zero if `s1` is equal to `s2`, a negative value if `s1` is less than `s2`, and a positive value if `s1` is greater than `s2`.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165

"memcmp" on page 456

"strcmp" on page 461

"strncmp" on page 470,

Listing 37.5 Example of strcoll() Usage

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

int main(void)
{
   static char s1[] = "aardvark", s2[] = "xylophone";
   int result;

   result = strcoll(s1, s2);

   if (result < 1)
      printf("%s is less than %s\n", s1, s2);
   else
      printf("%s is equal or greater than %s\n", s1, s2);

   return 0;
}
```

```
Output:
aardvark is less than xylophone
```

strcpy

Copy one character array to another.

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

Table 37.10 strcpy

dest	char *	The destination string
source	const char *	The string being copied

Remarks

The `strcpy()` function copies the character array pointed to by `source` to the character array pointed to `dest`. The `source` argument must point to a null terminated character array. The resulting character array at `dest` is null terminated as well.

If the arrays pointed to by `dest` and `source` overlap, the operation of `strcpy()` is undefined.

`strcpy()` returns the value of dest.

This function may not be implemented on all platforms.

See Also

"memcpy" on page 457

"memmove" on page 458

"strncpy" on page 472

Listing 37.6 Example of strcpy() Usage

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

int main(void)
{
```

```
    char d[30] = "";
    static char s[] = "CodeWarrior";

    printf(" s=%s\n d=%s\n", s, d);
    strcpy(d, s);
    printf(" s=%s\n d=%s\n", s, d);

    return 0;
}
```

```
Output:
 s=CodeWarrior
 d=
 s=CodeWarrior
 d=CodeWarrior
```

strcspn

Find the first character in one string that is in another.

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

Table 37.11 strcspn

s1	const char *	The string to count
s2	const char *	The list string of character to search for

Remarks

The `strcspn()` function finds the first character in the `null` terminated character string `s1` that is also in the null terminated string `s2`. For this purpose, the null terminators are considered part of the strings. The function starts examining characters at the beginning of `s1` and continues searching until a character in `s1` matches a character in `s2`.

`strcspn()` returns the index of the first character in `s1` that matches a character in `s2`.

This function may not be implemented on all platforms.

See Also

"strpbrk" on page 473

"strspn" on page 475

Listing 37.7 Example of strcspn() Usage

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

int main(void)
{
   static char s1[] = "chocolate *cinnamon* 2 ginger";
   static char s2[] = "1234*";
   int i;

   printf(" s1 = %s\n s2 = %s\n", s1, s2);
   i = strcspn(s1, s2);
   printf("Index returned by strcspn %d\n", i);
   printf("Indexed character  = %c\n", s1[i]);

   return 0;
}
```

```
Output:
s1 = chocolate *cinnamon* 2 ginger
s2 = 1234*
Index returned by strcspn 10
Indexed character  = *
```

strerror

Translate an error number into an error message.

```
#include <string.h>

char *strerror(int errnum);
```

Table 37.12 strerror

errnum	int	The error number to be translated.

Remarks

The `strerror()` function returns a pointer to a null terminated character array that contains an error message. The array pointed to may not be modified by the program, but may be overwritten by a subsequent call to the strerror function

`strerror()` returns a pointer to a null terminated character array containing an error message that corresponds to `errnum`. Although normally the integer value in `errnum` will come from the global variable `errno`, `strerror()` will provide a message translation for any value of type int.

This function may not be implemented on all platforms.

Listing 37.8 Example of strerror() Usage

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

int main(void)
{
   puts(strerror(8));
   puts(strerror(ESIGPARM));

   return 0;
}
```

```
Output:
unknown error (8)
Signal Error
```

strerror_r

Translate an error number into an error message.

```
#include <string.h>
int strerror_r(int errnum, char *str, size_t bufflen);
```

Table 37.13 strerror_r

errnum	int	The error number to be translated.

Table 37.13 strerror_r (*continued*)

str	char *	A pointer to the storage error string
bufflen	size_t	The size of the storage buffer

Remarks

The `strerror_r()` function provides the same service as "strerror" on page 466 but is reentrant. The difference is that `strerror()` would return a pointer to the error string, and that pointer was internal to the library implementation. For `strerror_r()`, the caller provides the storage `str` and the size of the storage `bufflen`.

On a successful call to `strerror_r()`, the function result is zero. If any error occurs, the function result is an error code.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"strerror" on page 466

strlen

Compute the length of a character array.

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

Table 37.14 strlen

s1	const char *	The string to evaluate

Remark

The `strlen()` function computes the number of characters in a null terminated character array pointed to by `s`. The null character (`'\0'`) is not added to the character count.

`strlen()` returns the number of characters in a character array not including the terminating null character.

This function may not be implemented on all platforms.

Listing 37.9 Example of strlen() Usage

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

int main(void)
{
   static char s[] = "antidisestablishmentarianism";

   printf("The length of %s is %ld.\n", s, strlen(s));

   return 0;
}
```

```
Output:
The length of antidisestablishmentarianism is 28.
```

strncat

Append a specified number of characters to a character array.

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

Table 37.15 strncat

dest	char *	The destination string
source	const char *	The source to append
n	size_t	The maximum length to append

Remarks

The `strncat()` function appends a maximum of `n` characters from the character array pointed to by `source` to the character array pointed to by `dest`. The `dest` argument must point to a null terminated character array. The `source` argument does not necessarily have to point to a null terminated character array.

If a null character is reached in `source` before `n` characters have been appended, `strncat()` stops.

When done, `strncat()` terminates `dest` with a null character (`'\0'`).

`strncat()` returns the value of `dest`.

This function may not be implemented on all platforms.

See Also

"strcat" on page 459

Listing 37.10 Example of strncat() Usage

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

int main(void)
{
   static char s1[100] = "abcdefghijklmnopqrstuv";
   static char s2[] = "wxyz0123456789";

   strncat(s1, s2, 4);
   puts(s1);

   return 0;
}
```

```
Output:
abcdefghijklmnopqrstuvwxyz
```

strncmp

Compare a specified number of characters.

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

Table 37.16 strncmp

s1	const char *	The string to compare

Table 37.16 strncmp (*continued*)

s2	const char *	The comparison string
n	size_t	The maximum number of characters to compare

Remarks

The `strncmp()` function compares `n` characters of the character array pointed to by `s1` to `n` characters of the character array pointed to by `s2`. Neither `s1` nor `s2` needs to be null terminated character arrays.

The function stops prematurely if it reaches a null character before `n` characters have been compared.

`strncmp()` returns a zero if the first `n` characters of `s1` and `s2` are equal, a negative value if `s1` is less than `s2`, and a positive value if `s1` is greater than `s2`.

This function may not be implemented on all platforms.

See Also

"memcmp" on page 456

"strcmp" on page 461

Listing 37.11 Example of strncmp() Usage

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

int main(void)
{
   static char s1[] = "12345anchor", s2[] = "12345zebra";

   if (strncmp(s1, s2, 5) == 0)
      printf("%s is equal to %s\n", s1, s2);
   else
      printf("%s is not equal to %s\n", s1, s2);

   return 0;
}
```

```
Output:
12345anchor is equal to 12345zebra
```

strncpy

Copy a specified number of characters.

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

Table 37.17 strncpy

dest	char *	The destination string
source	const char *	The source to copy
n	size_t	The maximum length to copy

Remarks

The `strncpy()` function copies a maximum of `n` characters from the character array pointed to by `source` to the character array pointed to by `dest`. Neither `dest` nor `source` need necessarily point to null terminated character arrays. Also, `dest` and `source` must not overlap.

If a null character (`'\0'`) is reached in `source` before `n` characters have been copied, `strncpy()` continues padding `dest` with null characters until `n` characters have been added to `dest`.

The function does not terminate `dest` with a null character if `n` characters are copied from `source` before reaching a null character.

`strncpy()` returns the value of `dest`.

This function may not be implemented on all platforms.

See Also

"memcpy" on page 457

"memmove" on page 458

"strcpy" on page 464

Listing 37.12 Example of strncpy Usage

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

int main(void)
```

```
{
   char d[50];
   static char s[] = "123456789ABCDEFG";

   strncpy(d, s, 9);
   puts(d);

   return 0;
}
```

```
Output:
123456789
```

strpbrk

Look for the first occurrence of any one of an array of characters in another.

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

Table 37.18 strpbrk

s1	const char *	The string being searched
s2	const char *	A list of characters to search for

Remarks

The `strpbrk()` function searches the character array pointed to by `s1` for the first occurrence of a character in the character array pointed to by `s2`.

Both `s1` and `s2` must point to null terminated character arrays.

`strpbrk()` returns a pointer to the first character in `s1` that matches any character in `s2`, and returns a null pointer (`NULL`) if no match was found.

This function may not be implemented on all platforms.

See Also

"strcspn" on page 465

Listing 37.13 Example of strpbrk Usage

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

int main(void)
{
   static char s1[] = "orange banana pineapple *plum";

   static char s2[] = "*%#$";
   puts(strpbrk(s1, s2));

   return 0;
}
```

```
Output:
*plum
```

strrchr

Searches a string for the last occurrence of a character.

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

Table 37.19 strrchr

s	const char *	The string to search
c	int	A character to search for

Remarks

The `strrchr()` function searches for the last occurrence of `c` in the character array pointed to by `s`. The `s` argument must point to a null terminated character array.

`strrchr()` returns a pointer to the character found or returns a null pointer (`NULL`) if it fails.

This function may not be implemented on all platforms.

See Also

"memchr" on page 454

"strchr" on page 460

Listing 37.14 Example of strrchr() Usage

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

int main(void)
{
   static char s[] = "Marvin Melany CodeWarrior";
   puts(strrchr(s, 'M'));

   return 0;
}
```

```
Output:
CodeWarrior
```

strspn

Find the first character in one string that is not in another.

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

Table 37.20 strspn

s1	const char *	The string to search
s2	const char *	A list of characters to look for

Remarks

The `strspn()` function finds the first character in the null terminated character string `s1` that is not in the null terminated string `s2`. The function starts examining characters at the beginning of `s1` and continues searching until a character in `s1` does not match any character in `s2`.

Both `s1` and `s2` must point to null terminated character arrays.

`strspn()` returns the index of the first character in `s1` that does not match a character in `s2`.

This function may not be implemented on all platforms.

See Also

"strpbrk" on page 473

"strcspn" on page 465

Listing 37.15 Example of strspn() Usage

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

int main(void)
{
   static char s1[] = "create *build* construct";
   static char s2[] = "create *";

   printf(" s1 = %s\n s2 = %s\n", s1, s2);
   printf(" %d\n", strspn(s1, s2));

   return 0;
}
```

```
Output:
s1 = create *build* construct
 s2 = create *
 8
```

strstr

Search for a character array within another.

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

Table 37.21 strstr

s1	const char *	The string to search
s2	const char *	The string to search for

Remarks

The `strstr()` function searches the character array pointed to by `s1` for the first occurrence of the character array pointed to by `s2`.

Both `s1` and `s2` must point to null terminated (`'\0'`) character arrays.

`strstr()` returns a pointer to the first occurrence of `s2` in `s1` and returns a null pointer (`NULL`) if `s2` cannot be found.

This function may not be implemented on all platforms.

See Also

"memchr" on page 454

"strchr" on page 460

Listing 37.16 Example of strstr() Usage

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

int main(void)
{
   static char s1[] = "tomato carrot onion";
   static char s2[] = "on";
   puts(strstr(s1, s2));

   return 0;
}
```

```
Output:
onion
```

strtok

Extract tokens within a character array.

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

```
char *strtok(char *str, const char *sep);
```

Table 37.22 strtok

str	char *	The string to be separate
sep	const char *	The separator string

Remarks

The `strtok()` function divides a null terminated character array pointed to by `str` into separate "tokens". The `sep` argument points to a null terminated character array containing one or more separator characters. The tokens in `str` are extracted by successive calls to `strtok()`.

`Strtok()` works by a sequence of calls to the `strtok()` function. The first call is made with the string to be divided into tokens, as the first argument. Subsequent calls use `NULL` as the first argument and returns pointers to successive tokens of the separated string.

The first call to `strtok()` causes it to search for the first character in `str` that does not occur in `sep`. If no character other than those in the sep string can be found, strtok() returns a null pointer (NULL). If no characters from the sep string are found it returns a pointer to the original string. Otherwise the function returns a pointer to the beginning of this first token.

Subsequent calls to `strtok()` are made with a `NULL` str argument causing it to return pointers to successive tokens in the original `str` character array. If no further tokens exist, `strtok()` returns a `null pointer`.

Both `str` and `sep` must be null terminated character arrays.

The sep argument can be different for each call to `strtok()`. `strtok()` modifies the character array pointed to by str.

When first called strtok() returns a pointer to the first token in `str`. If nothing but separator characters are found strtok returns a null pointer.

Subsequent calls to `strtok()` with a NULL `str` argument causes `strtok()` to return a pointer to the next token or return a null pointer (`NULL`) when no more tokens exist.

This function may not be implemented on all platforms.

Listing 37.17 Example of strtok() Usage

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

int main(void)
{
```

```
   static char s[50] = "(ape+bear)*(cat+dog)";
   char *token;
   char *separator = "()+*";

   /* first call to strtok() */
   token = strtok(s, separator);

   while(token != NULL)
   {
      puts(token);
      token = strtok(NULL, separator);
   }

   return 0;
}
```

```
Output:
ape
bear
cat
dog
```

strxfrm

Transform a locale-specific character array.

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

Table 37.23 strxfrm

dest	char *	The destination string
source	const char *	The source to be transformed
n	size_t	The maximum length to transform

Remarks

The `strxfrm()` function copies characters from the character array pointed to by `source` to the character array pointed to by `dest`, transforming each character as

specified by the LC_COLLATE component of the current locale. The strxfrm function transforms the string pointed to by source and places the resulting string into the array pointed to by dest. The transformation is such that if the strcmp function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the strcoll function applied to the same two original strings.

The MSL C implementation of `strxfrm()` copies a maximum of `n` characters from the character array pointed to by `source` to the character array pointed to by `dest` using the `strncpy()` function. It is included in the string library to conform to the ANSI C Standard Library specification.

`strxfrm()` returns the length of `dest` after it has received `source`.

This function may not be implemented on all platforms.

See Also

"Locale Specification" on page 165

"strcpy" on page 464

Listing 37.18 Example of strxfrm() Usage

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

int main(void)
{
   char d[50];
   static char s[] = "123456789ABCDEFG";
   size_t result;

   result = strxfrm(d, s, 30);

   printf("%d characters copied: %s\n", result, d);

   return 0;
}
```

```
Output:
16 characters copied: 123456789ABCDEFG
```

Non Standard <string.h> Functions

Various non standard functions are included in the header `string.h` for legacy source code and compatibility with operating system frameworks and application programming interfaces.

- For the function `strdup`, see "strdup" on page 95 for a full description.
- For the function `stricmp`, see "stricmp" on page 96 for a full description.
- For the function `strlwr`, see "strlwr" on page 97 for a full description.
- For the function `strnicmp`, see "strnicmp" on page 100 for a full description.
- For the function `strnset`, see "strnset" on page 101 for a full description.
- For the function `strrev`, see "strrev" on page 102 for a full description.
- For the function `strset`, see "strset" on page 102 for a full description.
- For the function `strupr`, see "strupr" on page 104 for a full description.
- For the function `wcsdup`, see "wcsdup" on page 107, for a full description.
- For the function `wcsicmp`, see "wcsicmp" on page 108, for a full description.
- For the function `wcslwr`, see "wcslwr" on page 109, for a full description.
- For the function `wcsincmp`, see "wcsnicmp" on page 111, for a full description.
- For the function `wcsnset`, see "wcsnset" on page 112, for a full description.
- For the function `wcsrevv` "wcsrev" on page 113, for a full description.
- For the function `wcsset`, see "wcsset" on page 113, for a full description.
- For the function `wcspnp`, see "wcsspnp" on page 114, for a full description.
- For the function `wcsupr`, see "wcsupr" on page 114, for a full description.
- For the function `wtoi`, see "wtoi" on page 115, for a full description.

38

tgmath.h

The header `tgmath.h` includes the header `math.h` and defines type-generic macros for those math functions.

Overview of tgmath.h

The `tgmath.h` header file consists of type-generic macros for most math functions listed in Table 38.1.

NOTE Main Standard Library for C does not include complex types and complex type-generic equivalents

The macros in this header may not be implemented on all platforms.

Table 38.1 Type-Generic Macro for Math Functions

Function	Type-Generic Macro	Function	Type-Generic Macro
acos	acos	acosh	acosh
asin	asin	asinh	asinh
atan	atan	atan2	atan2
atanh	atanh	cbrt	cbrt
ceil	ceil	copysign	copysign
cos	cos	cosh	cosh
erf	erf	erfc	erfc
exp	exp	exp2	exp2
expm1	expm1	fabs	fabs
fdim	fdim	floor	floor

Table 38.1 Type-Generic Macro for Math Functions (*continued*)

Function	Type-Generic Macro	Function	Type-Generic Macro
acos	acos	acosh	acosh
fma	fma	fmax	fmax
fmin	fmin	fmod	fmod
frexp	frexp	hypot	hypot
ilogb	ilogb	ldexp	ldexp
lgamma	lgamma	llrint	llrint
llround	llround	log	log
log10	log10	log1p	log1p
log2	log2	logb	logb
lrint	lrint	lround	lround
nearbyint	nearbyint	nextafter	nextafter
nexttoward	nexttoward	pow	pow
remainder	remainder	remquo	remquo
rint	rint	round	round
scalbln	scalbln	scalbn	scalbn
sin	sin	sinh	sinh
sqrt	sqrt	tan	tan
tanh	tanh	tgamma	tgamma
trunc	trunc		

39

time.h

The `time.h` header file provides access to the computer system clock, date and time conversion functions, and time-formatting functions.

Overview of time.h

This header file defines the facilities as follows:

- "struct tm" on page 487 is a structure for storing time data.
- "tzname" on page 488 contains an array that stores the time zone abbreviations.
- "asctime" on page 488 converts a tm structure type to a char array.
- "asctime_r" on page 489 is a reentrant version of asctime.
- "clock" on page 490 determines the relative time since the system was started.
- "ctime" on page 492 converts a time_t type to a char array.
- "ctime_r" on page 493 is a reentrant version of ctime.
- "difftime" on page 494 determines the difference between two times.
- "gmtime" on page 495 determines Greenwich Mean Time.
- "gmtime_r" on page 496 is a reentrant version of gmtime.
- "localtime" on page 497 determines the local time.
- "localtime_r" on page 497 is a reentrant version of localtime.
- "mktime" on page 498 convert a tm structure to time_t type.
- "strftime" on page 499 formats time as a C string.
- "time" on page 505 determines a number of seconds from a set time.
- "tzset" on page 506 internalizes the time zone to that of the application.

Date and time

The `time.h` header file provides access to the computer system clock, date and time conversion functions, and formatting functions.

Three data types are defined in `time.h`: `clock_t`, `time_t`, and `tm`.

Type clock_t

The `clock_t` type is a numeric, system dependent type returned by the `clock()` function.

Type time_t

The `time_t` type is a system dependent type used to represent a calendar date and time.

Remarks

The type `time_t's` range and precision are defined in the C standard as implementation defined. The MSL C implementation uses an unsigned long for time_t and it represents the number of UTC seconds since 1900 January 1. If his value exceeds the size of the maximum value for unsigned long (ULONG_MAX = 4,294,967,295) the result is undefined. A value of greater than 136 for tm_year will exceed this limit. Similarly, since time_t is unsigned, negative values for tm_year are also out of range.

The ANSI/ISO C Standard does not specify a start date, therefore an arbitrarily chosen Jan. 1, 1970 is used for the MSL C Library. These routines are not meant to be intermixed with any specific API time functions. However some conversion constants are available in the OS specific headers (e.g. time.mac.h).

struct tm

The `struct tm` type contains a field for each part of a calendar date and time.

```
#include <time.h>
    struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

Remarks

See Table 39.1 for the `tm` structure members.

The `tm_isdst` flag is positive if Daylight Savings Time is in effect, zero if it is not, and negative if such information is not available.

This structure may not be implemented on all platforms.

Table 39.1 tm Structure Members

Field	Description	Range, min - max
int tm_sec	Seconds after the minute	0 - 59
int tm_min	Minutes after the hour	0 - 59
int tm_hour	Hours after midnight	0 - 23
int tm_mday	Day of the month	1 - 31
int tm_mon	Months after January	0 - 11
int tm_year	Up to 136 years after 1900	1900 - 2036
int tm_wday	Days after Sunday	0 - 6

Table 39.1 tm Structure Members (*continued*)

Field	Description	Range, min - max
int tm_yday	Days after January 1	0 - 365
int tm_isdst	Daylight Savings Time flag	

tzname

The _tzname_ array contains the names (abbreviations) of the time zones for local standard time and DST.

This function may not be implemented on all platforms.

```
#include <time.h>
extern char *tzname[2];
```

This function may not be implemented on all platforms.

See Also

"tzset" on page 506

asctime

Convert a `tm` structure to a character array.

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Table 39.2 asctime

timeptr	const struct tm *	A pointer to a tm structure that holds the time information

Remarks

The `asctime()` function converts a `tm` structure, pointed to by `timeptr`, to a character array. The `asctime()` and `ctime()` functions use the same calendar

time format. This format, expressed as a `strftime()` format string is `"%a %b %e %H:%M: %S %Y"`. For example: Tue Apr 4 15:17:23 2000.

`asctime()` returns a null terminated character array pointer containing the converted `tm structure`.

This function may not be implemented on all platforms.

See Also

"ctime" on page 492

"strftime" on page 499

Listing 39.1 Example of asctime() Usage

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

int main(void)
{
   time_t systime;
   struct tm *currtime;

   systime = time(NULL);
   currtime = localtime(&systime);

   puts(asctime(currtime));

   return 0;
}
```

```
Output:
Tue Nov 30 12:56:05 1993
```

asctime_r

Reentrant function to convert a `tm` structure to a character array.

```
#include <time.h>
char * asctime_r(const struct tm * tm, char * s);
```

Table 39.3 asctime_r

tm	const struct tm *	A pointer to a tm structure that holds the time information
s	char *	Storage for the time string

Remarks

The `asctime_r()` function provides a reentrant version of "asctime" on page 488. The difference is that `asctime()` will return a pointer to the time string, and that pointer is internal to the library implementation. For `asctime_r()`, the caller provides the storage for string `s` and the size of the storage must be at least 26 characters long.

The `asctime_r()` function always returns the value of `s`.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"asctime" on page 488

"ctime" on page 492

"strftime" on page 499

clock

A program relative invocation of the system time.

```
#include <time.h>
clock_t clock(void);
```

Remarks

This function is used to obtain values of type `clock_t` which may be used to calculate elapsed times during the execution of a program. To compute the elapsed time in seconds, divide the `clock_t` value by `CLOCKS_PER_SEC`, a macro defined in time.h.

The programmer should be aware that `clock_t`, defined in `time.h`, has a finite size that varies depending upon the target system.

The `clock()` function returns a `clock_t` type value representing the approximation of time since the system was started. There is no error value returned if an error occurs.

This function may not be implemented on all platforms.

Listing 39.2 Example of clock() Usage

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

int main()
{
   clock_t start, end;
   double  secs = 0;
   int stop = 0;

   fprintf( stderr, "Press enter to start");
   getchar();
   start = clock();

   while(stop != 'x')
   {
      fprintf( stderr, "Press enter to terminate");
      getchar();
      end = clock();
      secs =  (double)(end - start) /CLOCKS_PER_SEC;
      fprintf( stderr, "Elapsed seconds = %f \n", secs);
      fprintf( stderr,"Press enter to start again ");
      fprintf(stderr, "or enter x to terminate:  ");
      stop = getchar();
      start = clock();
   }

   printf("\n** Program has terminated ** \n");
   return 0;
}
```

```
Output:
Press enter to start <enter>
Press enter to terminate <enter>
Elapsed seconds = 1.200000
Press enter to start again or enter x to terminate: <enter>
Press enter to terminate <enter>
Elapsed seconds = 1.033333
Press enter to start again or enter x to terminate: <x enter>
```

```
** Program has terminated **
```

ctime

Convert a `time_t` type to a character array.

```
#include <time.h>
char *ctime(const time_t *timer);
```

Table 39.4 ctime

timer	const time_t *	The address of the time_t variable

Remarks

The `ctime()` function converts a `time_t` type to a character array with the same format as generated by `asctime()`.

`ctime()` returns a null terminated character array pointer containing the converted `time_t` value.

This function may not be implemented on all platforms.

See Also

"asctime" on page 488

"strftime" on page 499

Listing 39.3 Example of ctime() Usage

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

int main(void)
{
   time_t systime;

   systime = time(NULL);
   puts(ctime(&systime));

   return 0;
}
```

```
Output:
Wed Jul 20 13:32:17 1994
```

ctime_r

Convert a `time_t` type to a character array but reentrant safe.

```
#include <time.h>
char * ctime_r(const time_t * timer, char * s);
```

Table 39.5 ctime_r

timer	const time_t *	The address of the time_t variable
s	char *	The storage string

Remarks

The `ctime_r()` function provides the same service as "ctime" on page 492. The difference is that `ctime()` would return a pointer to the time string, and that pointer was internal to the library implementation. For `ctime_r()`, the caller provides the storage for string `s` and the size of the storage must be at least 26 characters long.

The `ctime_r()` function always returns the value of `s`.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"asctime" on page 488

"ctime" on page 492

"strftime" on page 499

difftime

Compute the difference between two `time_t` types.

```
#include <time.h>
double difftime(time_t t1, time_t t2);
```

Table 39.6 difftime

t1	time_t	A time_t variable to compare
t2	time_t	A time_t variable to compare

`difftime()` returns the difference of `t1` minus `t2` expressed in seconds.

This function may not be implemented on all platforms.

Listing 39.4 Example of difftime Usage

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

int main(void)
{
   time_t t1, t2;
   struct tm *currtime;
   double midnight;

   time(&t1);
   currtime = localtime(&t1);

   currtime->tm_sec = 0;
   currtime->tm_min = 0;
   currtime->tm_hour = 0;
   currtime->tm_mday++;

   t2 = mktime(currtime);

   midnight = difftime(t1, t2);
   printf("There are %f seconds until midnight.\n",
                                        midnight);

   return 0;
```

```
}
```

```
Output:
There are 27892.000000 seconds until midnight.
```

gmtime

Convert a `time_t` value to Coordinated Universal Time (UTC), which is the new name for Greenwich Mean Time.

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

```
struct tm *gmtime(const time_t *timer);
```

Table 39.7 gmtime

timer	const time_t *	The address of the time_t variable

Remarks

The `gmtime` function converts the calendar time pointed to by `timer` into a broken-down time, expressed as UTC.

The `gmtime()` function returns a pointer to that object.

This function may not be implemented on all platforms.

See Also

"gmtime_r" on page 496

"localtime" on page 497

Listing 39.5 Example of gmtime Usage

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

int main(void)
{
   time_t systime;
   struct tm *utc;

   systime = time(NULL);
```

```
    utc = gmtime(&systime);

    printf("Universal Coordinated Time:\n");
    puts(asctime(utc));

    return 0;
}
```

```
Output:
Universal Coordinated Time:
Thu Feb 24 18:06:10 1994
```

gmtime_r

Convert a `time_t` value to Coordinated Universal Time (UTC), which is the new name for Greenwich Mean Time but is reentrant safe.

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

```
struct tm * gmtime_r(const time_t * timer, struct tm * tm);
```

Table 39.8 gmtime_r

timer	const time_t *	The address of the time_t variable
tm	struct tm *	A storage location for the converted time

Remarks

The `gmtime_r()` function provides the same service as "gmtime" on page 495. The difference is that `gmtime()` would return a pointer to the converted time, and that pointer was internal to the library implementation. For `gmtime_r()`, the caller provides the storage for the `tm struct`.

The `gmtime_r()` function always returns the value of `tm`.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"gmtime" on page 495

localtime

Convert a `time_t` type to a `struct tm` type.

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Table 39.9 localtime

timer	const time_t *	The address of the time_t variable

Remarks

The `localtime()` function converts a `time_t` type, pointed to by `timer`, and returns it as a pointer to an internal `struct tm` type. The `struct tm` pointer is `static`; it is overwritten each time `localtime()` is called. For example usage, see Listing 39.4.

`localtime()` converts `timer` and returns a pointer to a `struct tm`.

This function may not be implemented on all platforms.

See Also

"mktime" on page 498

localtime_r

Convert a `time_t` type to a `struct tm` type but is reentrant.

```
#include <time.h>
struct tm * localtime_r(const time_t * timer, struct tm * tm);
```

Table 39.10 localtime_r

timer	const time_t *	The address of the time_t variable
tn	struct tm *	The storage location

Remarks

The `localtime_r()` function provides the same service as "localtime" on page 497 but is reentrant. The difference is that `localtime()` would return a pointer to the converted time, and that pointer was internal to the library implementation. For `localtime_r()`, the caller provides the storage for the `tm struct`.

The `localtime_r()` function always returns the value of `tm`.

This function may require extra library support.

This function may not be implemented on all platforms.

See Also

"localtime" on page 497

"mktime" on page 498

mktime

Convert a `struct tm` item to a `time_t` type.

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Table 39.11 mktime

timeptr	struct tm *	The address of the tm structure

Remarks

The `mktime()` function converts a `struct tm` type and returns it as a `time_t` type. For example usage, see Listing 39.4.

The function also adjusts the fields in `timeptr` if necessary. The `tm_sec`, `tm_min`, `tm_hour`, and `tm_day` are processed such that if they are greater than their maximum, the appropriate carry-overs are computed. For example, if `timeptr->tm_min` is 65, `timeptr->tm_hour` will be incremented by 1 and `timeptr->min` will be set to 5.

The function also computes the correct values for `timeptr->tm_wday` and `timeptr->tm_yday`.

`mktime()` returns the converted `tm` structure as a `time_t` type.

This function may not be implemented on all platforms.

See Also

"localtime" on page 497

strftime

Format a `tm` structure.

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
      const char *format,
      const struct tm *timeptr);
```

Table 39.12 strftime

s	char *	A string to hold the formatted time
maxsize	size_t	Max length of formatted string
format	const char *	The format string
timeptr	const struct tm*	The address of the time structure

Remarks

The `strftime()` function converts a `tm` structure to a character array using a programmer supplied format.

The `s` argument is a pointer to the array to hold the formatted time.

The `maxsize` argument specifies the maximum length of the formatted character array.

The `timeptr` argument points to a `tm` structure containing the calendar time to convert and format.

The format argument points to a character array containing normal text and format specifications similar to a `printf()` function format string. Format specifiers are prefixed with a percent sign (`%`). Doubling the percent sign (`%%`) will output a single `%`.

If any of the specified values are outside the normal range, the characters stored are unspecified.

In the “C” locale, the E and O modifiers are ignored. Also, some of the formats are dependent on the LC_TIME component of the current locale.

See Table 39.13 for a list of format specifiers.

Table 39.13 strftime() Conversion Characters

Character	Description
a	Locale’s abbreviated weekday name.
A	Locale’s full weekday name.
b	Locale’s abbreviated month name.
B	Locale’s full month name.
c	The locale's appropriate date and time representation equivalent to the format string of “%A %B %d %T %Y”.
C	The year divided by 100 and truncated to an integer, as a decimal number [00 - 99]
d	Day of the month as a decimal number [01 - 31].
D	The month date year, equivalent to '%m/%d/%y"
e	The day of the month as a decimal number; a single digit is preceded by a space.
F	The year, month and day separated by hyphens, the equivalent to "%Y-%m-%d"
g	The last 2 digits of the week-based year as a decimal number. For example: 03 99
G	The week-based year as a decimal number
h	The month name, equivalent to "%b"
H	The hour (24-hour clock) as a decimal number from 00 to 23.
I	The hour (12-hour clock) as a decimal number from 01 to 12

Table 39.13 strftime() Conversion Characters (*continued*)

Character	Description
j	The day of the year as a decimal number from 001 to 366
m	The month as a decimal number from 01 to 12.
M	The minute as a decimal number from 00 to 59.
n	A newline character
p	`Locale's equivalent of "am"` or `"pm"`.
r	The locale's 12-hour clock time, equivalent of “%I:%M:%S %p”
R	The hour, minute, equivalent to ''%H:%M
S	The second as a decimal number from 00 to 59.
t	A horizontal-tab character
T	The hour minute second, equivalent to ''%H:%M:%S'
u	The weekday as a decimal number 1 to 7, where Monday is 1.
U	The week number of the year as a decimal number from 00 to 53. Sunday is considered the first day of the week.
w	The weekday as a decimal number from 0 to 6. Sunday is (0) zero.
W	The week of the year as a decimal number from 00 to 51. Monday is the first day of the week.
x	The date representation of the current locale, equivalent to “%A %B %d %Y”
X	The time representation of the current locale, equivalent to “%T”

Table 39.13 strftime() Conversion Characters (*continued*)

Character	Description
y	The last two digits of the year as a decimal number.
Y	The year as a four digit decimal number.
z	The time zone offset from UTC. for example, -0430 is 4 hours 30 minutes behind UTC. Or nothing if the time zone is unknown.
Z	The locale's time zone name or abbreviation, or by no characters if no time zone is unknown.
%	The percent sign is displayed.

The `strftime()` function returns the total number of characters in the argument `'s'` if the total number of characters including the null character in the string argument `'s'` is less than the value of `'maxlen'` argument. If it is greater, `strftime()` returns 0.

This function may not be implemented on all platforms.

Listing 39.6 Example of strftime() Usage

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

int main(void)
{
 time_t lclTime;
 struct tm *now;
 char ts[256];  /* time string */

 lclTime = time(NULL);
 now = localtime(&lclTime);

 strftime(ts, 256,
   "Today's abr.name is %a", now);
 puts(ts);

 strftime(ts, 256,
   "Today's full name is %A", now);
```

```
 puts(ts);

 strftime(ts, 256,
   "Today's aabr.month name is %b", now);
 puts(ts);

 strftime(ts, 256,
   "Today's full month name is %B", now);
 puts(ts);

 strftime(ts, 256,
   "Today's date and time is %c",now);
 puts(ts);
 strftime(ts, 256,
"The day of the month is %d", now);
 puts(ts);

 strftime(ts, 256,
"The 24-hour clock hour is %H",now);
 puts(ts);

 strftime(ts, 256,
"The 12-hour clock hour is %H", now);
 puts(ts);

 strftime(ts, 256,
"Today's day number is %j", now);
 puts(ts);

 strftime(ts, 256,
"Today's month number is %m", now);
 puts(ts);

 strftime(ts, 256,
"The minute is %M", now);
 puts(ts);

 strftime(ts, 256,
"The AM/PM is %p", now);
 puts(ts);

 strftime(ts, 256,
"The second is %S", now);
 puts(ts);

 strftime(ts, 256,
"The week number of the year,\
starting on a Sunday is %U", now);
```

```
 puts(ts);

 strftime(ts, 256,
"The number of the week is %w", now);
 puts(ts);

 strftime(ts, 256, "The week number of the year,\
starting on a Monday is %W", now);
 puts(ts);

 strftime(ts, 256, "The date is %x", now);
 puts(ts);

 strftime(ts, 256, "The time is %X", now);
 puts(ts);

 strftime(ts, 256,
   "The last two digits of the year are %y", now);
 puts(ts);

 strftime(ts, 256, "The year is %Y", now);
 puts(ts);

 strftime(ts, 256, "%Z", now);
 if (strlen(ts) == 0)
  printf("The time zone cannot be determined\n");
 else
  printf("The time zone is %s\n", ts);

   return 0;
}
```

```
Results
Today's abr.name is Wed
Today's full name is Wednesday
Today's aabr.month name is Apr
Today's full month name is April
Today's date and time is Wednesday April 05 10:50:44 2000
The day of the month is 05
The 24-hour clock hour is 10
The 12-hour clock hour is 10
Today's day number is 096
Today's month number is 04
The minute is 50
The AM/PM is am
The second is 44
The week number of the year,starting on a Sunday is 14
```

```
The number of the week is 3
The week number of the year,starting on a Monday is 14
The date is Wednesday April 05 2000
The time is 10:50:44
The last two digits of the year are 00
The year is 2000
The time zone cannot be determined
```

time

Return the current system calendar time.

```
#include <time.h>
time_t time(time_t *timer);
```

Table 39.14 time

timer	time_t *	The address of the time_t variable

Remarks

The `time()` function returns the computer system's calendar time. If `timer` is not a null pointer, the calendar time is also assigned to the item it points to.

`time()` returns the system current calendar time.

This function may not be implemented on all platforms.

Listing 39.7 Example of time() Usage

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

int main(void)
{
   time_t systime;
   systime = time(NULL);

   puts(ctime(&systime));

   return 0;
}
```

```
Output:
Tue Nov 30 13:06:47 1993
```

tzset

The function tzset() reads the value of the “TZ” environment variable and internalizes it into the time zone functionality of the program.

```
#include <time.h>
void tzset(void);
```

Remarks

The function tzset() reads the value of the “TZ” environment variable and internalizes it into the time zone functionality of the program.

This function may not be implemented on all platforms.

See Also

“tzname” on page 488

Non Standard <time.h> Functions

Various non standard functions are included in the header `time.h` for legacy source code and compatibility with operating system frameworks and application programming interfaces.

- For the function `strdate` see “strdate” on page 94, for a full description
- “asctime_r” on page 489 is a reentrant version of asctime
- “ctime_r” on page 493 is a reentrant version of ctime
- “gmtime_r” on page 496 is a reentrant version of gmtime
- “localtime_r” on page 497 is a reentrant version of localtime

40

timeb.h

The `timeb.h` header file provides access to the computer system clock, date and time conversion functions, and time formatting functions. Currently, this header is implemented for Windows only.

Overview of timeb.h

This header file defines the facilities as follows:

- "struct timeb" on page 507 lists the elements of the timeb structure.
- "ftime" on page 508 stores the current time in a buffer that the programmer can allocate.

This struct is Windows x86 only at this time.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4 for information on `POSIX` naming conventions.

struct timeb

The timeb struct is used to store the time of a milliseconds timer, timezone, and timezone flag.

Table 40.1 timeb Structure Members

Field	Description
time_t time	A type used to represent calendar date and time See "Type time_t" on page 486
unsigned short millitm	Current time in milliseconds

Table 40.1 timeb Structure Members (*continued*)

Field	Description
short timezone	The difference, in minutes, between local time and Greenwich Mean time
short dstflag	True if daylight savings time is in effect

Remarks

The dstflag flag is true if the daylight savings time is in effect

This structure may not be implemented on all platforms.

ftime

The function ftime and _ftime gets the current time and stores it in a struct that is allocated by the programmer.

```
#include <timeb.h>
void ftime(struct timeb * timebptr);
void _ftime(struct timeb * timebptr);
```

Table 40.2 ftime

timebptr	struct timeb *	A pointer to a timeb structure that holds the time information

Remarks

The elements of the struct timeb are listed in Table 40.1.

This function may not be implemented on all platforms.

See Also

"ctime" on page 492.

Listing 40.1 Example of ftime Usage

```
#include <sys\timeb.h>
#include <stdio.h>
```

```
int main()
{
   struct timeb tbuf;
   ftime( &tbuf);

   printf("Time is %s", ctime(&tbuf.time));

   return 0;
}
```

41

unistd.h

The header file `unistd.h` contains several functions that are useful for porting a program from UNIX.

Overview of unistd.h

The header file `unistd.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and other operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions explain the differences.

This header file defines the facilities as follows:

- "access" on page 512 determines the files accessibility.
- "chdir" on page 513 changes the directory.
- "close" on page 515 closes a file opened with open.
- "cuserid" on page 518 retrieves the current user's ID.
- "cwait" on page 519 instructs to wait for a process to end.
- "dup" on page 520 duplicates a file handle.
- "dup2" on page 520 duplicates a file handle unto an existing handle.
- "exec functions" on page 521 executes programs from within a program.
- "getcwd" on page 523 gets the current working directory.
- "getlogin" on page 524 returns a login name.
- "getpid" on page 525 returns the process ID.
- "isatty" on page 526 determines if a file ID is attached to a terminal.
- "lseek" on page 527 seeks a position on a file stream.
- "read" on page 528 reads when opened with open.
- "rmdir" on page 529 removes a directory or folder.
- "sleep" on page 532 pauses a program.
- "spawn functions" on page 533 spawns a child process.
- "ttyname" on page 534 determines a terminal ID.

- "unlink" on page 535 deletes a file.
- "write" on page 536 writes to a binary file stream.

unistd.h and UNIX Compatibility

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

Table 1.1 for information on `POSIX` naming conventions.

access

Determines the files accessibility.

```
#include <unistd.h>
int access(const char *fname, int mode);
```

Table 41.1 access

fname	const char *	The file to check
mode	int	The file mode to test for

Remarks

If the access is allowed then zero is returned. A negative number is returned if the access is not allowed.

This function may not be implemented on all platforms.

Table 41.2 lists the file modes that may be tested.

Table 41.2 Access Test Modes

Macro	Description
F_OK	Test for existence of file
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission

See Also

"creat, _wcreate" on page 118

"open, _wopen" on page 121

"close" on page 515

chdir

Change the current directory.

```
#include <unistd.h>
int chdir(const char *path);
int _chdir(const char *path);
```

Table 41.3 chdir

path	char *	The new pathname

Remarks

The function `chdir()` is used to change from one directory to a different directory or folder. For example usage, see Listing 41.1.

`chdir()` returns zero, if successful. If unsuccessful `chdir()` returns negative one and sets `errno`.

This function may not be implemented on all platforms.

See Also

"errno" on page 75

Listing 41.1 Example of chdir() Usage

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

#define SIZE FILENAME_MAX
#define READ_OR_WRITE   0x0 /* fake a UNIX mode */

int main(void)
{
char folder[SIZE];
char curFolder[SIZE];
char newFolder[SIZE];
int folderExisted = 0;

   /* Get the name of the current folder or directory */
   getcwd( folder, SIZE );
   printf("The current Folder is: %s", folder);

   /*  create a new sub folder */
   /* note mode parameter ignored on Mac */
   sprintf(newFolder,"%s%s", folder, "Sub" );
   if( mkdir(newFolder, READ_OR_WRITE ) == -1 )
   {
      printf("\nFailed to Create folder");
      folderExisted = 1;
   }

   /* change to new folder */
   if( chdir( newFolder) )
   {
      puts("\nCannot change to new folder");
      exit(EXIT_FAILURE);
   }

   /* show the new folder or folder  */
   getcwd( curFolder, SIZE );
   printf("\nThe current folder is: %s", curFolder);

   /* go back to previous folder */
   if( chdir(folder) )
   {
      puts("\nCannot change to old folder");
      exit(EXIT_FAILURE);
   }
```

```
   /* show the new folder or folder  */
   getcwd( curFolder, SIZE );
   printf("\nThe current folder is again: %s", curFolder);

   if (!folderExisted)
   {
   /* remove newly created directory  */
   if (rmdir(newFolder))
       {
        puts("\nCannot remove new folder");
        exit(EXIT_FAILURE);
       }
       else
        puts("\nNew folder removed");

    /* attempt to move to non-existant directory  */
       if (chdir(newFolder))
     puts("Cannot move to non-existant folder");
   }
   else       puts("\nPre-existing folder not removed");

   return 0;
}
```

```
Output
The current Folder is: Macintosh HD:C Reference:
The current folder is: Macintosh HD:C Reference:Sub:
The current folder is again: Macintosh HD:C Reference:
New folder removed
Cannot move to non-existant folder
For Windows, refer to "Example of rmdir() Usage" on page 530.
```

close

Close an open file.

```
#include <unistd.h>

int close(int fildes);
```

Table 41.4 close

fildes	int	The file descriptor

Remarks

The `close()` function closes the file specified by the argument. This argument is the value returned by `open()`. For example usage, see Listing 41.2.

`If` successful, `close()` returns zero. If unsuccessful, `close()` returns negative one and sets `errno`.

This function may not be implemented on all platforms.

See Also

"open, wopen" on page 121

"fclose" on page 304

"errno" on page 75

Listing 41.2 Example of close() Usage

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>


#define SIZE FILENAME_MAX
#define MAX 1024

char fname[SIZE] = "DonQ.txt";


int main(void)
{
   int fdes;
   char temp[MAX];
   char *Don = "In a certain corner of la Mancha, the name of\n\
which I do not choose to remember,...";
   char *Quixote = "there lived\none of those country\
 gentlemen, who adorn their\nhalls with rusty lance\
 and worm-eaten targets.";

   /* NULL terminate temp array for printf */
   memset(temp, '\0', MAX);

   /* open a file */
   if((fdes = open(fname, O_RDWR | O_CREAT ))== -1)
   {
      perror("Error ");
```

```
    printf("Can not open %s", fname);
    exit( EXIT_FAILURE);
}

/* write to a file  */
if( write(fdes, Don, strlen(Don)) == -1)
{
    printf("%s Write Error\n", fname);
    exit( EXIT_FAILURE );
}

/*   move back to over write ... characters */
if( lseek( fdes, -3L, SEEK_CUR ) == -1L)
{
    printf("Seek Error");
    exit( EXIT_FAILURE );
}

/* write to a file  */
if( write(fdes, Quixote, strlen(Quixote)) == -1)
{
    printf("Write Error");
    exit( EXIT_FAILURE );
}

/* move to beginning of file for read */
if( lseek( fdes, 0L, SEEK_SET ) == -1L)
{
    printf("Seek Error");
    exit( EXIT_FAILURE );
}

/* read the file  */
if( read( fdes, temp, MAX ) == 0)
{
    printf("Read Error");
    exit( EXIT_FAILURE);
}

/* close the file */
if(close(fdes))
{
    printf("File Closing Error");
    exit( EXIT_FAILURE );
}

 puts(temp);
```

```
   return 0;
}
```

```
Result
In a certain corner of la Mancha, the name of
which I do not choose to remember,there lived
one of those country gentlemen, who adorn their
halls with rusty lance  and worm-eaten targets.
```

cuserid

Retrieve the current user's ID.

```
#include <unistd.h>
```

```
char *cuserid(char *string);
```

Table 41.5 cuserid

string	char *	The user ID as a string

Remarks

The function `cuserid()` returns the user name associated with the current process. If the string argument is `NULL`, `the` file name is stored in an internal buffer. If it is not `NULL`, it must be at least `FILENAME_MAX` large. For example usage, see Listing 41.3.

For the MacOS, the login name is returned.

`cuserid()` returns a character pointer to the current user's ID.

For the MacOS, the users name is set using the sharing control panel

This function may not be implemented on all platforms.

Listing 41.3 Example of cuserid() Usage

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

int main(void)
{
 char *c_id = NULL;
 printf("The current user ID is %s\n",cuserid(c_id));
```

```
   return 0;
}
```

```
Result
The current user ID is CodeWarror
```

cwait

Wait for a process to terminate.

```
#include <unistd.h>
int cwait(int *termstat, int pid, int action);
int _cwait(int *termstat, int pid, int action);
```

Table 41.6 cwait

termstat	int	The termination status
pid	int	Process ID
action	int	The action is ignored

Remarks

The process exit code is returned.

Windows compatible function.

See Also

"exec functions" on page 521

"spawn functions" on page 533

dup

Duplicates a file handle.

```
#include <io.h>
int dup(int _a);
int _dup(int _a);
```

Table 41.7 dup

_a	int	A file handle to duplicate

Remarks

Creates a new file handle for an open file with the same attributes as the original file handle.

Anew file handle is returned upon success or a negative one otherwise.

This function may not be implemented on all platforms.

See Also

"dup2" on page 520

dup2

Duplicates a file handle unto an existing handle.

```
#include <io.h>
int dup2(int _a, int _b);
int _dup2(int _a, int _b);
```

Table 41.8 dup2

_a	int	A file handle to duplicate
_b	int	An existing file handle

Remarks

Associates a file handle for an open file with the same attributes as the original file handle. If the file associated with the second argument is open when _dup2 is called, the old file is closed.

Zero is returned upon success and a negative one upon failure.

This function may not be implemented on all platforms.

See Also

"dup" on page 520

exec functions

Load and execute a child process within the current program memory.

The suffix on the `exec` name determines the method that the child process operates.

- P will search the path variable
- L is used for known argument lists number
- V is for an unknown argument list number
- E allows you to alter an environment for the child process

exec

```
#include <unistd.h>
int exec(const char *path, ...);
int _exec(const char *path, ...);
```

execl

```
int execl(const char *path, ...);
int _execl(const char *path, ...);
```

execle

```
int execle(const char *path, ...);
int _execle(const char *path, ...);
```

execlp

```
int execlp(const char *path, ...);
int _execlp(const char *path, ...);
```

execv

```
 int execv(const char *path, ...);
int _execv(const char *path, ...);
```

execve

```
int execve(const char *path, ...);
int _execve(const char *path, ...);
```

excevp

```
 int excevp(const char *path, ...);
int _excevp(const char *path, ...);
```

Table 41.9 excevp

path	const char *	The commandline pathname to execute
...		A variable list of arguments

Remarks

Launches the application named and then quits upon successful launch. For example usage, see Listing 41.4.

If successful `exec()` returns zero. If unsuccessful `exec()` returns negative one and sets `errno` according to the error.

These functions may not be implemented on all platforms.

See Also

"Overview of errno.h" on page 75

Listing 41.4 Example of exec() Usage

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

```
#include <unistd.h>

#define SIZE FILENAME_MAX
char appName[SIZE] = "SimpleText";

int main(void)
{
   printf("Original Program\n");
   exec(appName);
   printf("program terminated"); /* not displayed */

   return 0;
}
```

```
result
Display "Original Program"
after the close of the program the SimpleText application is launched
```

getcwd

Get the current directory.

```
#include <unistd.h>
char *getcwd(char *buf, int size);
char *_getcwd(char *buf, int size);
```

Table 41.10 getcwd

buf	char	A buffer to hold the pathname of the current working directory
size	int	The size of the buffer

Remarks

The function `getcwd()` takes two arguments. One is a buffer large enough to store the full directory pathname, the other argument is the size of that buffer. For example usage, see Listing 41.1.

If successful, `getcwd()` returns a pointer to the buffer. If unsuccessful, `getcwd()` returns NULL and sets `errno`.

This function may not be implemented on all platforms.

See Also

"Overview of errno.h" on page 75

getlogin

Retrieve the username that started the process.

```
#include <unistd.h>
```

```
char *getlogin(void);
```

Remarks

The function `getlogin()` retrieves the name of the user who started the program. For example usage, see Listing 41.5.

The Mac doesn't have a login, so this function returns the Owner Name from the File Sharing Setup Control Panel

`getlogin()` returns a character pointer.

This function may not be implemented on all platforms.

Listing 41.5 Example of getlogin() Usage

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

int main(void)
{
  printf("The login name is %s\n", getlogin());

   return 0;
}
```

```
result
The login name is CodeWarrior
```

getpid

Retrieve the process identification number.

```
#include <unistd.h>
```

Table 41.11 getpid() Macros

Macro		Represents
#define getpid()		Process ID
#define getppid()		Parent process ID
#define getuid()		Real user ID
#define geteuid()		Effective user ID
#define getgid()		Real group ID
#define getegid()		Effective group ID
#define getpgrp()		Process group ID

Remarks

The `getpid()` function returns the unique number (`Process ID`) for the calling process. For example usage, see Listing 41.6.

`getpid()` returns an integer value.

These various related `getpid()` type functions don't really have any meaning on the Mac. The values returned are those that would make sense for a typical user process under UNIX.

`getpid()` always returns a value. There is no error returned.

This function may not be implemented on all platforms.

Listing 41.6 Example of getpid() Usage

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

int main(void)
{
  printf("The process ID is %d\n", getpid());
```

```
   return 0;
}
```

```
Result
The process ID is 9000
```

isatty

Determine a specified file_id

```
#include <unistd.h>
int isatty(int fildes);
int _isatty(int fildes);
```

Table 41.12 isatty

fildes	int	The file descriptor

Remarks

The function `isatty()` determines if a specified `file_id` is attached to the console, or if re-direction is in effect. For example usage, see Listing 41.7.

`isatty()` returns a non-zero value if the file is attached to the console. It returns zero if Input/Output redirection is in effect.

This function may not be implemented on all platforms.

See Also

"ccommand" on page 41

Listing 41.7 Example of isatty() ttyname() Usage

```
#include <console.h>
#include <stdio.h>
#include <unistd.h>
#include <unix.h>

int main(int argc, char **argv)
{
   int i;
   int file_id;
```

```
   argc = ccommand(&argv);

   file_id = isatty(fileno(stdout));
   if(!file_id )
   {
   for (i=0; i < argc; i++)
      printf("command line argument [%d] = %s\n",
            i, argv[i]);
   }
   else printf("Output to window");

   printf("The associated terminal is %s",
         ttyname(file_id) );

   return 0;
}
```

```
Result if file redirection is chosen using the command line arguments
Freescale CodeWarrior.
Written to file selected:

command line argument [0] = CRef
command line argument [1] = Freescale
command line argument [2] = CodeWarrior
The associated terminal is SIOUX
```

lseek

Seek a position on a file stream.

```
#include <unistd.h>

long lseek(int fildes, long offset, int origin);
```

Table 41.13 lseek

fildes	int	The file descriptor
offset	long	The offset to move in bytes
origin	int	The starting point of the seek

Remarks

The function `lseek()` sets the file position location a specified byte offset from a specified initial location. For example usage, see Listing 41.2.

The origin of the offset must be one of the positions listed in Table 41.14

Table 41.14 The lseek Offset Positions

Macro	Meaning
SEEK_SET	Beginning of file
SEEK_CUR	Current Position
SEEK_END	End of File

If successful, `lseek()` returns the absolute offset as the number of bytes from the beginning of the file after the seek has occurred. If unsuccessful, it returns a value of negative one long integer.

This function may not be implemented on all platforms.

See Also

"fseek" on page 341

"ftell" on page 344

"open, wopen" on page 121

read

Read from a file stream that has been opened in binary mode for unformatted Input/ Output.

```
#include <unistd.h>
int read(int fildes, char *buf, int count);
```

Table 41.15 read

fildes	int	The file descriptor

Table 41.15 read (*continued*)

buf	char *	A buffer to store the data read
count	int	The maximum size in bytes to read

Remarks

The function `read()` copies the number of bytes specified by the count argument, from the file to the character buffer. File reading begins at the current position. The position moves to the end of the read position when the operation is completed.

NOTE This function should be used in conjunction with `unistd.h:write()`, and `fcntl.h:open()` only.

`read()` returns the number of bytes actually read from the file. In case of an error a value of negative one is returned and errno is set.

This function may not be implemented on all platforms.

See Also

"fread" on page 331

"open, wopen" on page 121

For example of read() usage, see Listing 41.2.

rmdir

Delete a directory or folder.

```
#include <unistd.h>
int rmdir(const char *path);
```

Table 41.16 rmdir

path	const char *	The pathname of the directory being removed

Remarks

The `rmdir()` function removes the directory (folder) specified by the argument.

If successful, `rmdir()` returns zero. If unsuccessful, `rmdir()` returns negative one and sets `errno`.

This function may not be implemented on all platforms.

See Also

"mkdir" on page 275

"errno" on page 75

Listing 41.8 Example of rmdir() Usage

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

#define SIZE FILENAME_MAX
#define READ_OR_WRITE   0x0 /* fake a UNIX mode */

int main(void)
{
char folder[SIZE];
char curFolder[SIZE];
char newFolder[SIZE];
int folderExisted = 0;

   /* Get the name of the current folder or directory */
   getcwd( folder, SIZE );
   printf("The current Folder is: %s", folder);

   /*  create a new sub folder */
   /* note mode parameter ignored on Mac */
   sprintf(newFolder,"%s%s", folder, ".\\Sub" );
   if( mkdir(newFolder, READ_OR_WRITE ) == -1 )
   {
      printf("\nFailed to Create folder");
      folderExisted = 1;
   }

   /* change to new folder */
   if( chdir( newFolder) )
   {
      puts("\nCannot change to new folder");
      exit(EXIT_FAILURE);
   }

   /* show the new folder or folder  */
```

```
   getcwd( curFolder, SIZE );
   printf("\nThe current folder is: %s", curFolder);

   /* go back to previous folder */
   if( chdir(folder) )
   {
      puts("\nCannot change to old folder");
      exit(EXIT_FAILURE);
   }

   /* show the new folder or folder  */
   getcwd( curFolder, SIZE );
   printf("\nThe current folder is again: %s", curFolder);

   if (!folderExisted)
   {
   /* remove newly created directory  */
   if (rmdir(newFolder))
       {
        puts("\nCannot remove new folder");
        exit(EXIT_FAILURE);
       }
       else
        puts("\nNew folder removed");

    /* attempt to move to non-existant directory  */
       if (chdir(newFolder))
     puts("Cannot move to non-existant folder");
   }
   else       puts("\nPre-existing folder not removed");

   return 0;
}
Output
The current Folder is: C:\Programming\CW\Console
The current folder is: C:\Programming\CW\Console\Sub
The current folder is again: C:\Programming\CW\Console
New folder removed
Cannot move to non-existant folder
For Macintosh, refer to
```

"Example of chdir() Usage" on page 514.

sleep

Delay program execution for a specified number of seconds.

```
#include <unistd.h>
unsigned int sleep(unsigned int sleep);
```

Table 41.17 sleep

sleep	unsigned int	The length of time in seconds

Remarks

The function `sleep()` delays execution of a program for the time indicated by the unsigned integer argument. For the Macintosh system there is no error value returned. For example usage, see Listing 41.9.

The function `sleep()` returns zero.

This function may not be implemented on all platforms.

Listing 41.9 Example of sleep() Usage

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

int main(void)
{


   printf("Output to window\n");
   fflush(stdout); /* needed to force output */

   sleep(3);

   printf("Second output to window");

   return 0;
}
```

```
Result
Output to window
< there is a delay now >
```

```
Second output to window
```

spawn functions

The `spawn` family of functions create and run other processes (child processes). The suffix on the `spawn` name determines the method that the child process operates.

- P will search the path variable
- L is used for known argument lists number
- V is for an unknown argument list number
- E allows you to alter an environment for the child process

spawnl

```
int spawnl(int,const char *, ...);
int _spawnl(int,const char *, ...);
```

spawnv

```
int spawnv(int,const char *,const char *const*);
int _spawnv(int,const char *,const char *const*);
```

spawnle

```
int spawnle(int,const char *,...);
int _spawnle(int,const char *,...);
```

spawnve

```
int spawnve(int,const char *,const char *const*, const char
     *const*);
int _spawnve(int,const char *,const char *const*, const char
     *const*);
```

spawnlp

```
int spawnlp(int,const char *,...);
int _spawnlp(int,const char *,...);
```

spawnvp

```
int spawnvp(int,const char *,const char *const *);
int _spawnvp(int,const char *,const char *const *);
```

spawnlpe

```
int spawnlpe(int,const char *,...);
int _spawnlpe(int,const char *,...);
```

spawnvpe

```
int spawnvpe(int,const char *,const char *const *, const char
    *const*);
int _spawnvpe(int,const char *,const char *const *, const char
    *const*);
```

Remarks

All `spawn` functions take a variable argument list as a parameter.

The child process's exit status is returned.

These functions may not be implemented on all platforms.

See Also

"exec functions" on page 521

ttyname

Retrieve the name of the terminal associated with a file ID.

```
#include <unistd.h>
char *ttyname(int fildes);
```

Table 41.18 ttyname

fildes	int	The file descriptor

Remarks

The function `ttyname()` retrieves the name of the terminal associated with the file ID. For example usage, see Listing 41.7.

`ttyname()` returns a character pointer to the name of the terminal associated with the file ID, or NULL if the file ID doesn't specify a terminal.

This function may not be implemented on all platforms.

unlink

Delete (unlink) a file.

```
#include <unistd.h>
int unlink(const char *path);
```

Table 41.19 unlink

path	const char *	A pathname of the file to remove

Remarks

The function `unlink()` removes the specified file from the directory. For example usage, see Listing 41.10.

If successful, `unlink()` returns zero. If unsuccessful, it returns a negative one.

This function may not be implemented on all platforms.

See Also

"rmdir" on page 529

"mkdir" on page 275

Listing 41.10 Example of unlink() Usage

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

#define SIZE FILENAME_MAX

int main(void)
{
```

```
   FILE *fp;
   char fname[SIZE] = "Test.txt";

   /* create a file */
   if( (fp =fopen( fname,"w") ) == NULL )
   {
      printf("Can not open %s for writing", fname);
      exit( EXIT_FAILURE);
   }
   else printf("%s was opened for writing\n",fname);

   /* display it is available */
   if( !fclose(fp) ) printf("%s was closed\n",fname);

   /* delete file */
   if( unlink(fname) )
   {
      printf("%s was not deleted",fname);
      exit( EXIT_FAILURE );
   }

   /* show it can't be re-opened */
   if( (fp =fopen( fname,"r") ) == NULL )
   {
      printf("Can not open %s for reading it was deleted",
             fname);
      exit( EXIT_FAILURE);
   }
   else printf("%s was opened for reading\n",fname);

   return 0;
}
```

```
Result
Test.txt was opened for writing
Test.txt was closed
Can not open Test.txt for reading it was deleted
```

write

Write to a file stream that has been opened in binary mode for unformatted output.

```
#include <unistd.h>

int write(int fildes, const char* buf, size_t count)
```

Table 41.20 write

fildes	int	The file descriptor
buf	const char *	The address of the buffer being written
count	size_t	The size of the buffer being written

Remarks

The function `write()` copies the number of bytes in the `count` argument from the character array buffer to the file `fildes`. The file position is then incremented by the number of bytes written. For example usage, see Listing 41.2.

This function should be used in conjunction with "read" on page 528, and "open, wopen" on page 121 only.

`write()` returns the number of bytes actually written.

This function may not be implemented on all platforms.

See Also

"fwrite" on page 347

"read" on page 528

"open, wopen" on page 121

42

unix.h

The `unix.h` header file contains two global variables.

Overview of unix.h

The header file `unix.h` contains two global variables that are useful for console interface programs for setting a Mac OS file creator and file type.

The globals variables in unix.h are as follows:

- “ fcreator” on page 539 sets a Macintosh file creator.
- “ ftype” on page 540 sets a Macintosh file type.

UNIX Compatibility

Generally, you don’t want to use these functions in new programs. Instead, use their counterparts in the native API.

NOTE If you’re porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

_fcreator

To specify a Macintosh file creator.

```
#include <unix.h>
extern long _fcreator
```

Remarks

Use the global `_fcreator` to set the creator type for files created using the Standard C Libraries. For example usage, see Listing 42.1.

This global identifier is Macintosh Only

_ftype

To specify a Macintosh file type.

```
#include <unix.h>
extern long _ftype;
```

Remarks

Use the global _ftype to set the creator type for files created using the Standard C Libraries. For example usage, see Listing 42.1.

The value assigned to _fcreate and _ftype is a ResType (i.e. four character constant).

This global identifier is Macintosh only

Listing 42.1 Usage of Global Variables to Set file Creator and Type

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

#define oFile "test file"
const char *str = "CodeWarrior Software at Work";


int main(void)
{
   FILE *fp;
   _fcreator = 'ttxt';
   _ftype = 'TEXT';

   // create a new file for output and input
   if (( fp = fopen(oFile, "w+")) == NULL)
   {
      printf("Can't create file.\n");
      exit(1);
   }

   fwrite(str, sizeof(char), strlen(str), fp);
   fclose(fp);

   return 0;
}
```

```
   // output to the file using fwrite()
CodeWarrior Software at Work
```

43

utime.h

The `utime.h` header file contains several functions that are useful for porting a program from UNIX.

Overview of utime.h

The header file `utime.h` contains several functions that are useful for porting a program from UNIX. These functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you what the differences are.

This header file defines the facilities as follows:

- "utime" on page 543 sets a file modification time.
- "utimes" on page 546 sets a series of file modification times.

utime.h and UNIX Compatibility

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the native API.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

Table 1.1 for information on `POSIX` naming conventions.

---

utime

Sets a file's modification time.

```
#include <utime.h>
int utime(const char *path, /* Mac */ const struct utimbuf
```

```
     *buf);
int utime(const char *path, /* Windows */ struct utimbuf
     *buf);
```

Table 43.1 utime

path	const char *	The pathname as a string
buf	const struct utimbuf * struct utimbuf	The address of a struct that will hold a file's time information

Remarks

This function sets the modification time for the file specified in `path`. Since the Macintosh does not have anything that corresponds to a file's access time, it ignores the `actime` field in the `utimbuf` structure.

If `buf` is `NULL`, `utime()` sets the modification time to the current time. If `buf` points to a `utimbuf` structure, `utime()` sets the modification time to the time specified in the `modtime` field of the structure.

The `utimbuf` structures contains the fields in Table 43.2.

Table 43.2 The utimbuf Structure

This field...		is the...
time_t	actime	Access time for the file. Since the Macintosh has nothing that corresponds to this, `utime()` ignores this field.
time_t	modtime	The last time this file was modified.

If it is successful, `utime()` returns zero. If it encounters an error, `utime()` returns `-1` and sets `errno`.

This function may not be implemented on all platforms.

See Also

"ctime" on page 492

"mktime" on page 498

"fstat" on page 273

"stat" on page 276

"utimes" on page 546

Listing 43.1 Example for utime() Usage

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

int main(void)
{
   struct utimbuf timebuf;
   struct tm date;
   struct stat info;
   FILE * fp;

   fp = fopen("mytest", "w");
   if(!fp)
   {
      fprintf(stderr,"Could not open file");
      exit(EXIT_FAILURE);
   }
   fprintf(fp,"test");
   fclose(fp);
   /* Create a calendar time for
   Midnight, Apr. 4, 1994. */
   date.tm_sec=0;     /* Zero seconds    */
   date.tm_min=0;       /* Zero minutes    */
   date.tm_hour=0;    /* Zero hours      */
   date.tm_mday=4;    /* 4th day         */
   date.tm_mon=3;       /* .. of April     */
   date.tm_year=94;    /* ...in 1994      */
   date.tm_isdst=-1;    /* Not daylight savings */
   timebuf.modtime=mktime(&date);
      /* Convert to calendar time.     */

   /* Change modification date to  *
    * midnight, Apr. 4, 1994.    */
   utime("mytest", &timebuf);
   stat("mytest", &info);
   printf("Mod date is %s", ctime(&info.st_mtime));

   /* Change modification date *
    * to current time         */
   utime("mytest", NULL);
   stat("mytest", &info);
```

```
   printf("Mod date is %s", ctime(&info.st_mtime));

   return 0;
}
```

```
This program might display the following to standard output:
Mod date is Mon Apr  4 00:00:00 1994
Mod date is Mon Jul 10 17:45:17 2000
```

utimes

Sets a file's modification time

```
#include <utime.h>
int utimes(const char *path, struct timeval buf[2]);
```

Table 43.3 utimes

path	const char *	The pathname as a string
buf	timeval struct array	An array of time values used to set the modification dates

Remarks

This function sets the modification time for the file specified in path to the second element of the array `buf`. Each element of the array `buf` is a `timeval` structure, which has the fields in Table 43.4.

Table 43.4 The timeval Structure

This field		is the
int t	tv_sec	Seconds
int	tv_usec	Microseconds. Since the Macintosh does not use microseconds, `utimes()` ignores this.

The first element of `buf` is the access time.

Since the Macintosh does not have anything that corresponds to a file's access time, it ignores that element of the array.

If it is successful, `utimes()` returns `0`. If it encounters an error, `utimes()` returns `-1` and sets `errno`.

This function may not be implemented on all platforms.

See Also

"utime" on page 543

"ctime" on page 492

"mktime" on page 498

"fstat" on page 273

"stat" on page 276

Listing 43.2 Example for utimes() Usage

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

int main(void)
{
   struct tm date;
   struct timeval buf[2];
   struct stat info;


/* Create a calendar time for
Midnight, Sept. 9, 1965.*/
   date.tm_sec=0;       /* Zero seconds */
   date.tm_min=0;       /* Zero minutes */
   date.tm_hour=0;      /* Zero hours */
   date.tm_mday=9;      /* 9th day */
   date.tm_mon=8;       /* .. of September  */
   date.tm_year=65;     /* ...in 1965   */
   date.tm_isdst=-1;    /* Not daylight savings  */
   buf[1].tv_sec=mktime(&date);
   /* Convert to calendar time.  */

   /* Change modification date to   *
    * midnight, Sept. 9, 1965.      */
   utimes("mytest", buf);
   stat("mytest", &info);
   printf("Mod date is %s", ctime(&info.st_mtime));
```

```
   return 0;
}
```

```
This program prints the following to standard output:
Mod date is Thu Sep  9 00:00:00 1965
```

44

utsname.h

The `utsname.h` header file contains several functions that are useful for porting a program from UNIX.

Overview of utsname.h

These `utsname.h` functions are similar to the functions in many UNIX libraries. However, since the UNIX and Macintosh operating systems have some fundamental differences, they cannot be identical. The descriptions of the functions tell you the differences.

The header `utsname.h` has one function: "uname" on page 549, which retrieves information on the system you are using.

utsname.h and UNIX Compatibility

Generally, you don't want to use these functions in new programs. Instead, use their counterparts in the Macintosh Toolbox.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

Table 1.1 for information on `POSIX` naming conventions.

---

uname

Gets information about the system you are using.

```
#include <utsname.h>
int uname(struct utsname *name);
```

Table 44.1 uname

name	struct utsname *	A struct to store system information

Remarks

This function gets information on the Macintosh you're using and puts the information in the structure that `name` points to. The structure contains the fields listed in Table 44.2. All the fields are null-terminated strings.

Table 44.2 The utsname Structure

This field…	is…
sysnam	The operating system
nodename	The sharing node name.
release	The release number of system software.
version	The minor release numbers of the system software version.
machine	The type of the machine that you are using.

If it is successful, `uname()` returns zero. If it encounters an error, `uname()` returns `-1` and sets `errno`.

This function may not be implemented on all platforms.

See Also

"fstat" on page 273

"stat" on page 276

Listing 44.1 Example of uname() Usage

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

int main(void)
{
   struct utsname name;
```

```
    uname(&name);

    printf("Operating System: %s\n", name.sysname);
    printf("Node Name:        %s\n", name.nodename);
    printf("Release:          %s\n", name.release);
    printf("Version:          %s\n", name.version);
    printf("Machine:          %s\n", name.machine);

    return 0;
}
```

```
This application could print the following:
Operating System: MacOS
Node Name:        Ron's G4
Release:          9
Version:          3
Machine:            Power Macintosh
This machine is a Power Macintosh running Version 9 of the MacOS. The
Macintosh Name field of the File Sharing control panel contains "Ron's
G4"
```

45

wchar.h

The header file wchar.h contains defines and functions to manipulate wide character sets.

Overview of wchar.h

This header file defines the facilities as follows:

Input and Output Facilities

- "fgetwc" on page 558 behaves like `fgetc` for wide character arrays.
- "fgetws" on page 559 behaves like `fgets` for wide character arrays.
- "fputwc" on page 559 behaves like `fputc` for wide character arrays.
- "fputws" on page 560 behaves like `fputs` for wide character arrays.
- "fwprintf" on page 561 behaves like `fprintf` for wide character arrays.
- "fwscanf" on page 562 behaves like `fscanf` for wide character arrays.
- "getwc" on page 563 behaves like `getc` for wide character arrays.
- "getwchar" on page 563 behaves like `getchar` for wide character arrays.
- "putwc" on page 568 behaves like `putc` for wide character arrays.
- "putwchar" on page 568 behaves like `putchar` for wide character arrays.
- "swprintf" on page 569 behaves like `sprintf` for wide character arrays.
- "swscanf" on page 570 behaves like `sscanf` for wide character arrays.
- "wprintf" on page 605 behaves like `printf` for wide character arrays.
- "wscanf" on page 610 behaves like `scanf` for wide character arrays.
- "vfwprintf" on page 574, behaves like `vfprintf` for wide character arrays.
- "vfwscanf" on page 571 behaves like `vfscanf` for wide character arrays.
- "vswscanf" on page 572 behaves like `vsscanf` for wide character arrays.
- "vwprintf" on page 576 behaves like `vprintf` for wide character arrays.
- "vswprintf" on page 575 behaves like `fgetc` vsprintf for wide character arrays.
- "vwscanf" on page 573 behaves like `vscanf` for wide character arrays.

Time Facilities

- "wcsftime" on page 582 behaves like `csftime` for wide character arrays.
- "wctime" on page 600 behaves like `ctime` for wide character arrays.

String Facilities

- "watof" on page 576 behaves like `atof` for wide characters array.
- "wcscat" on page 578 behaves like `strcat` for wide character arrays.
- "wcschr" on page 579 behaves like `strchr` for wide character arrays.
- "wcscmp" on page 579 behaves like `strcmp` for wide character arrays.
- "wcscspn" on page 581 behaves like `strspn` for wide character arrays.
- "wcscoll" on page 580 behaves like `strcoll` for wide character arrays.
- "wcscpy" on page 581 behaves like `strcpy` for wide character arrays.
- "wcslen" on page 583 behaves like strlen for wide character arrays.
- "wcsncat" on page 584 behaves like `strncat` for wide character arrays.
- "wcsncmp" on page 585 behaves like `strncmp` for wide character arrays.
- "wcsncpy" on page 585 behaves like `strncpy` for wide character arrays.
- "wcspbrk" on page 586 behaves like `strbrk` for wide character arrays.
- "wcsrchr" on page 587 behaves like `strrchr` for wide character arrays.
- "wcsspn" on page 589 behaves like `strspn` for wide character arrays.
- "wcsstr" on page 589 behaves like `strstr` for wide character arrays.
- "wcstod" on page 590 behaves like `strtod` for wide character arrays.
- "wcstof" on page 591 behaves like `strtod` for wide character arrays.
- "wcstok" on page 592 behaves like `strtok` for wide character arrays.
- "wcstol" on page 593 behaves like `strtol` for wide character arrays.
- "wcstold" on page 594 behaves like `strtold` for wide character arrays.
- "wcstoll" on page 595 behaves like `strtoll` for wide character arrays.
- "wcstoul" on page 596 behaves like `strtoul` for wide character arrays.
- "wcstoull" on page 598 behaves like `strtoull` for wide character arrays.
- "wcsxfrm" on page 599 behaves like `strxfrm` for wide character arrays.
- "wmemchr" on page 601 behaves like `memchr` for wide character arrays.
- "wmemcpy" on page 603 behaves like `memcpy` for wide character arrays.
- "wmemcmp" on page 602 behaves like `memcmp` for wide character arrays.

- "wmemmove" on page 603 behaves like `memmove` for wide character arrays.
- "wmemset" on page 604 behaves like `memset` for wide character arrays.

Multibyte Character Functions

- "mbrlen" on page 564 behaves like strlen for multibyte character arrays
- "mbrtowc" on page 565 converts multibyte characters to wide character arrays
- "mbsinit" on page 566 determines the multibyte conversion status
- "mbsrtowcs" on page 567 converts multibyte strings to wide character strings
- "wcrtomb" on page 577 converts wide characters to multibyte character arrays.
- "wcsrtombs" on page 587 converts wide character strings to multibyte strings.

Conversion Functions

- "btowc" on page 557 converts byte characters to wide character arrays
- "wctob" on page 600 converts wide characters to byte character arrays

Wide Character and Byte Character Stream Orientation

There are two types of stream orientation for input and output, a wide character (wchar_t) oriented and a byte (char) oriented. A stream is un-oriented after that stream is associated with a file, until a byte or wide character input/output operation occurs.

Once any input/output operation is performed on that stream, that stream becomes oriented by that operation to be either byte oriented or wide character oriented and remains that way until the file has been closed and reopened.

Table 45.1 Byte Oriented Functions in Stdio.h

fgetc	fgets	fprintf	fputc	fputs
fread	fscanf	fwrite	getc	getchar
gets	printf	putc	putchar	puts
scanf	ungetc	vfprintf	vfscanf	vprintf

Table 45.2 Wide Character Oriented Functions

fgetwc	fgetws	fwprintf	fputwc	fputws	fwscanf
getwc	getwchar	putwc	putwchar	swprintf	swscanf
towctrans	vfwscanf	vswscanf	vwscanf	vfwprintf	vswprintf
vwprintf	wasctime	watof	wcscat	wcschr	wcscmp
wcscoll	wcscspn	wcscpy	wcslen	wcsncat	wcsncmp
wcsncpy	wcspbrk	wcsspn	wcsrchr	wcsstr	wcstod
wcstok	wcsftime	wcsxfrm	wctime	wctrans	wmemchr
wmemcmp	wmemcpy	wmemmove	wmemset	wprintf	wscanf

After a stream orientation is established, any call to an input/output function of the other orientation is not applied. For example, a byte-oriented input/output function does not have an effect on a wide-oriented stream.

Unicode

Unicode, also known as `UCS-2` (Universal Character Set containing 2 bytes) is a fixed-width encoding scheme that uses 16 bits per character. Characters are represented and manipulated in MSL as wide characters of type `wchar_t` and can be manipulated with the wide character functions defined in the C Standard.

Multibyte Characters

A Unicode character may be encoded as a sequence of one or more 8-bit characters, this is a multibyte character. There are two types of multibyte encoding, modal and non-modal. With modal encoding, a conversion state is associated with a multibyte string; this state is akin to the shift state of a keyboard. With non-modal encoding, no such state is involved and the first character of a multibyte sequence contains information about the number of characters in the sequence. The actual encoding scheme is defined in the LC_CTYPE component of the current locale.

In MSL, two encoding schemes are available, a direct encoding where only a single byte is used and the non-modal `UTF-8` (UCS Transformation Format -8) encoding scheme is used where each Unicode character is represented by one to three 8-bit characters. For Unicode characters in the range 0x0000 to 0x007F the encoding is direct and only a single byte is used.

Stream Orientation and Standard Input/Output

The three predefined associated streams, stdin, stdout, and stderr are un-oriented at program startup. If any of the standard input/output streams are closed it is not possible to reopen and reconnect that stream to the console. However, it is possible to reopen and connect the stream to a named file.

The C and C++ input/output facilities share the same stdin, stdout and stderr streams.

Definitions

The header wchar.h includes specific definitions for use with wide character sets.

Table 45.3 Wide Character Definitions

Defines	Definitions
mbstate_t	A value that can hold the conversion state for mode-dependent multibyte encoding
WCHAR_MIN	Minimum value of a wide char
WCHAR_MAX	Maximum value of a wide char
WEOF	A value that differs from any member of the wide character set and is used to denote end of file
win_t	An int type that can hold any wide character representation and WEOF

btowc

The function `btowc()` converts a single byte character to a wide character.

```
#include <wchar.h>
wint_t btowc(int c);
```

Table 45.4 btowc

int	c	The character to be converted

Returns

The function btowc() returns the wide character representation of the argument or WEOF is returned if c has the value `EOF` or if the current locale specifies that `UTF-8` encoding is to be used and unsigned char `c` does not constitute a valid single-byte `UTF-8` encoded character.

This function may not be implemented on all platforms.

See Also

"wctob" on page 600

fgetwc

Gets a wide character from a file stream.

```
#include <wchar.h>
wchar_t fgetwc(FILE * file);
```

Table 45.5 fgetwc

file	FILE *	The input stream to retrieve from

Remarks

Performs the same task as `fgetc` for wide character

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns the wide character or `WEOF` for an error

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"fgetc" on page 312

fgetws

The function `fgetws()` reads a wide character string from a file stream.

```
#include <wchar.h>
wchar_t *fgetws(wchar_t * s, int n, FILE * file);
```

Table 45.6 fgetws

s	wchar_t *	A wide char string to receive input
n	int	Maximum number of wide characters to be read
file	FILE *	A pointer to the input file stream

Remarks

Behaves like `fgets()` for wide characters.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns a pointer to `s` if successful or `NULL` for a failure.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"fgets" on page 316

fputwc

Inserts a single wide character into a file stream.

```
#include <wchar.h>
wchar_t fputwc(wchar_t c, FILE * file);
```

Table 45.7 fputwc

c	wchar_t	The wide character to insert
file	FILE *	A pointer to the output file stream

Remarks

Performs the same task as `fputc()` for a wide character type.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns the wide character written if it is successful, and returns `WEOF` if it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"fputc" on page 328

fputws

Inserts a wide character array into a file stream

```
#include <wchar.h>
int fputws(const wchar_t * s, FILE * file);
```

Table 45.8 fputws

s	wchar_t *	The string to insert
file	FILE *	A pointer to the output file stream

Remarks

Performs the same task as `fputs` for a wide character type.

If the file is opened in update mode (+) the file cannot be written to and then read from unless the write operation and read operation are separated by an operation that flushes the stream's buffer. This can be done with the `fflush()` function

or one of the file positioning operations (`fseek()`, `fsetpos()`, or `rewind()`).

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns a zero if successful, and returns a nonzero value when it fails.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"fputs" on page 330

fwprintf

Formatted file insertion

```
#include <wchar.h>
int fwprintf(FILE * file, const wchar_t * format, ...);
```

Table 45.9 fwprintf

file	FILE *	A pointer to the output file stream
format	wchar_t *	The format string
....		Variable arguments

Remarks

Performs the same task as `fprintf()` for a wide character type.

The `fwprintf()` function writes formatted text to `a wide character stream` and advances the file position indicator. Its operation is the same as `wprintf()` with the addition of the `stream` argument.

Refer to the "wprintf" on page 605 function for details of the format string.

On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns the number of arguments written or a negative number if an error occurs

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"wprintf" on page 605

fwscanf

Reads formatted text from a stream.

```
#include <wchar.h>
int fwscanf(FILE * file, const wchar_t * format, ...);
```

Table 45.10 fwscanf

file	FILE *	The input file stream
format	wchar_t *	The format string
....		Variable arguments

Remarks

Performs the same task as `fscanf` function for the wide character type.

The `fwscanf()` function reads programmer-defined, formatted text from a wide character stream. The function operates identically to the `wscanf()` function with the addition of the `stream` argument indicating the stream to read from.

Refer to the "wscanf" on page 610 function for details of the format string.

NOTE On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns the number of items read, which can be fewer than provided for in the event of an early matching failure. If the end of file is reached before any conversions are made, `EOF` is returned.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"wscanf" on page 610

getwc

Reads the next wide character from a wide character stream.

```
#include <wchar.h>
wchar_t getwc(FILE * file);
```

Table 45.11 getwc

file	FILE *	The file stream

Remarks

Performs the same task as `getc` for a wide character type.

Returns the next wide character from the stream or returns `WEOF` if the end-of-file has been reached or a read error has occurred.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"getc" on page 348

getwchar

Returns a wide character type from the standard input.

```
#include <wchar.h>
wchar_t getwchar(void);
```

Has no parameters.

Remarks

Performs the same task as `getchar` for a wide character type.

Returns the value of the next wide character from `stdin` as an `int` if it is successful. `getwchar()` returns `WEOF` if it reaches an end-of-file or an error occurs.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"getwchar" on page 563

mbrlen

Computes the length of a multibyte character encoded as specified in the `LC_CTYPE` component of the current locale. This function is essentially the same as `mblen()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

```
#include <stdlib.h>
int mbrlen(const char *s, size_t n, mbstate_t * ps);
```

Table 45.12 mbrlen

s	const char **	The multibyte array to measure
n	size_t	The Maximum size
ps	mbstate_t **	The current state of translation between multibyte and wide character, ignored if the encoding scheme is non-modal.

Remarks

The `mbrlen()` function returns the length of the multibyte character pointed to by `s`. It examines a maximum of `n` characters.

The MSL C implementation supports the `"C"` locale with `UTF-8` encoding only and returns the value of `mbrtowc(NULL, s, n, pc)`.

`mbrlen()` returns the value of `mbrtowc(NULL, s, n, pc)`.

This function may not be implemented on all platforms.

See Also

"wcslen" on page 583

"strlen" on page 468

mbrtowc

Translate a multibyte character to a `wchar_t` type according to the encoding specified in the `LC_CTYPE` component of the current locale. This function is essentially the same as `mbtowcs()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

```
#include <stdlib.h>

int mbrtowc(wchar_t *pwc,
      const char *s, size_t n, mbstate_t * ps);
```

Table 45.13 mbrtowc

pwc	wchar_t *	The wide character destination
s	const char *	The string to convert
n	size_t	The maximum wide characters to convert
ps	mbstate_t *	The current state of translation between multibyte and wide character, ignored if the encoding scheme is non-modal.

Remarks

If `s` a null pointer, this call is equivalent to mbrtowcs(NULL, “”, 1, ps);

If s is not a `null` pointer, the `mbrtowc()` function examines at most n bytes starting with the byte pointed to by s to determine how many bytes are needed to complete a valid encoding of a Unicode character. If this is less than or equal to `n` and `pwc` is not a null pointer, it converts the multibyte character, pointed to by `s`, to a character of type `wchar_t`, pointed to by `pwc` using the encoding scheme specified in the `LC_CTYPE` component of the current locale.

`mbrtowc()` returns the first of the following values that applies:

Zero, if s points to a `null` character, which is the value stored.

Greater than zero, if `s` points to a complete and valid multibyte character of `n` or fewer bytes, the corresponding Unicode wide character is stored (if `pwc` is not

NULL) and the value returned is the number of bytes in the complete multibyte character.

(size_t)(-2) if the next n bytes pointed to by s constitute an incomplete but potentially valid multibyte character. No value is stored.

(size_t)(-1) if the next n or fewer bytes pointed to by s do not constitute a complete and valid multibyte character. The value of EILSEQ is stored in errno but no wide character value is stored.

This function may not be implemented on all platforms.

See Also

"mbsrtowcs" on page 567

"wcrtomb" on page 577

"wcsrtombs" on page 587

mbsinit

Determines if the multi-byte state is the initial conversion state or not.

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Table 45.14 mbsinit

ps	const mbstate_t *	A pointer to a mbstate_t object

Remarks

If the status of the object pointed to is a null pointer or is in the initial converstion state a true values is returned otherwise zero is returned.

This function may not be implemented on all platforms.

See Also

"mbsrtowcs" on page 567

"wcrtomb" on page 577

"wcsrtombs" on page 587

mbsrtowcs

Convert a multibyte character array to a `wchar_t` array. This function is essentially the same as `mbstowcs()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

```
#include <stddlib.h>
size_t mbsrtowcs(wchar_t *pwcs,
const char **s, size_t n, mbstate_t * ps);
```

Table 45.15 mbsrtowcs

pwcs	wchar_t *	The wide character destination
s	const char **	Indirect pointer to the string to convert
n	size_t	The maximum wide characters to convert
ps	mbstate_t *	The current state of translation between multibyte and wide character, ignored if the encoding scheme is non-modal.

Remarks

The MSL C implementation of `mbsrtowcs()` converts a sequence of multibyte characters encoded according to the scheme specified in the LC_CTYPE component of the current locale from the wide character array indirectly pointed to by `s` and, if `pwcs` is not a null pointer, stores not more than n of the corresponding Unicode characters into the wide character array pointed to by `pwcs`. The function terminates prematurely if a null character is reached, in which case the null wide character is stored. or if an invalid multibyte encoding is detected. If conversion stops because a terminating null character is reached, a null pointer is assigned to the object pointed to by `s`, otherwise a pointer to the address just beyond the last multibyte character converted, if any.

If an invalidly encoded wide character is encountered, `mbsrtowcs()` stores the value of `EILSEQ` in `errno` and returns the value `(size_t)(-1)`. Otherwise

`mbsrtowcs()` returns the number of multibyte characters successfully converted, not including any terminating null wide character.

This function may not be implemented on all platforms.

See Also

"wcsrtombs" on page 587

"mbrtowc" on page 565

putwc

Write a wide character type to a stream.

```
#include <wchar.h>
wchar_t putwc(wchar_t c, FILE * file);
```

Table 45.16 putwc

c	wchar_t	The wide character to output
file	FILE	The output stream

Remarks

Performs the same task as `putc` for a wide character type.

Returns the wide character written when successful and returns `WEOF` when it fails

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"putc" on page 362

putwchar

Writes a wide character to standard output.

```
#include <wchar.h>
wchar_t putwchar(wchar_t c);
```

Table 45.17 putwchar

c	wchar_t	The wide character to write.

Remarks

Performs the same task as `putchar` for a wide character type.

Returns `c` if it is successful and returns `WEOF` if it fails

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"putchar" on page 363

swprintf

Formats text in a wide character type string.

```
#include <wchar.h>
int swprintf(wchar_t * S, size_t N, const wchar_t * format,
     ...);
```

Table 45.18 swprintf

s	wchar_t*	The string buffer to hold the formatted text
n	size_t	Number of characters allowed to be written
format	wchar_t*	The format string
....		Variable arguments

Remarks

Performs the same task as `sprintf` for a wide character type with an additional parameter for the maximum number of wide characters to be written. No more than `n` wide characters will be written including the terminating `NULL` wide character, which is always added unless the `n` is zero.

Refer to the "wprintf" on page 605 function for details of the format string.

Returns the number of characters assigned to `S`, not including the null character, or a This function may not be implemented on all platforms.

negative number if `n` or more characters were requested to be written.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"fwprintf" on page 561

"sprintf" on page 380

swscanf

Reads a formatted wide character string.

```
#include <wchar.h>
int swscanf(const wchar_t * s, const wchar_t * format, ...);
```

Table 45.19 swscanf

s	wchar_t*	The string being read
format	wchar_t*	The format string
....		Variable arguments

Remarks

Performs the same task as `sscanf` for a wide character type.

Returns the number of items successfully read and converted, which can be fewer than provided for in the event of an early matching failure. If the end of the input string is reached before any conversions are made, EOF is returned.

Refer to the "wscanf" on page 610 function for details of the format string.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"scanf" on page 370

vfwscanf

Read formatted text from a wide character stream.

```
#include <wchar.h>
int vfwscanf(FILE * file,
    const wchar_t * format_str, va_list arg);
```

Table 45.20 vfwscanf

file	FILE *	The stream being read
format_str	const wchar_t*	The format string
....		Variable arguments

Remarks

Performs the same task as `fscanf` for a wide character type.

Refer to the "wscanf" on page 610 function for details of the format string.

NOTE On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`vfwscanf()` returns the number of items assigned, which can be fewer than provided for in the case of an early matching failure. If an input failure occurs before any conversion, `vfwscanf()` returns EOF.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"fscanf" on page 335

vswscanf

Reads formatted text from a wide character string.

```
#include <wchar.h>
int __vswscanf(const wchar_t * s,
    const wchar_t * format, va_list arg);
```

Table 45.21 __vswscanf

s	wchar_t*	The string being read
format	wchar_t*	The format string
.arg	va_list	A variable argument list

Remarks

The `vswscantf()` function works identically to the `swscanf()` function. Instead of the variable list of arguments that can be passed to `swscanf()`, `vswscanf()` accepts its arguments in the array `arg` of type `va_list`, which must have been initialized by the `va_start()` macro (and possibly subsequent `va_arg` calls) from the stdarg.h header file. The `vswscanf()` function does not invoke the `va_end` macro.

Refer to the "wscanf" on page 610 function for details of the format string.

NOTE On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

`vswscanf()` returns the number of items assigned, which can be fewer than provided for in the case of an early matching failure. If an input failure occurs before any conversion, `vswscanf()` returns `EOF`.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"sscanf" on page 381

vwscanf

Reads formatted text from wide character oriented stdin.

```
#include <wchar.h>
int vwscanf(const wchar_t * format, va_list arg);
```

Table 45.22 vwscanf

s	wchar_t*	The string being read
format	wchar_t*	The format string
....		Variable arguments

Remarks

The `vwscantf()` function works identically to the `wscanf()` function. Instead of the variable list of arguments that can be passed to `wscanf()`, `vwscanf()` accepts its arguments in the array `arg` of type `va_list`, which must have been initialized by the `va_start()` macro (and possibly subsequent `va_arg` calls) from the stdarg.h header file. The `vwscanf()` function does not invoke the `va_end` macro.

Refer to the "wscanf" on page 610 function for details of the format string.

The `vwscanf()` function returns the number of items assigned, which can be fewer than provided for in the case of an early matching failure. If an input failure occurs before any conversion, `vwscanf()` returns `EOF`.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"vfwscanf" on page 571

"scanf" on page 370

vfwprintf

Write a formatted text to a file stream.

```
#include <wchar.h>
int vfwprintf(FILE * file,
const wchar_t * format_str, va_list arg);
```

Table 45.23 vfwprintf

file	FILE *	The stream being written
format_str	wchar_t*	The format string
arg	va_list	A variable argument list

Remarks

The `vfwprintf()` function works identically to the `fwprintf()` function. Instead of the variable list of arguments that can be passed to `fwprintf()`, `vfwprintf()` accepts its arguments in the array arg of type `va_list`, which must have been initialized by the `va_start()` macro (and possibly subsequent `va_arg` calls) from the stdarg.h header file. The `vfwprintf()` function does not invoke the `va_end` macro.

Refer to the "wprintf" on page 605 function for details of the format string.

NOTE On embedded/ RTOS systems this function only is implemented for stdin, stdout and stderr files.

Returns the number of wide characters written or a negative number if it failed.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"vfprintf" on page 387

vswprintf

Write a formatted output to a wide character string.

```
#include <stdio.h>
#include <wchar.h>
#include <stdarg.h>
int vswprintf(wchar_t * restrict s, size_t n,
const wchar_t * restrict format, va_list arg);
```

Table 45.24 vswprintf

s	wchar_t*	The string being read
n	size_t	Number of char to print
format	wchar_t*	The format string
arg	...	Variable arguments

Remarks

The `vswprintf()` function works identically to the `swprintf()` function. Instead of the variable list of arguments that can be passed to `swprintf()`, `vswprintf()` accepts its arguments in the array `arg` of type `va_list`, which must have been initialized by the `va_start()` macro (and possibly subsequent `va_arg` calls) from the stdarg.h header file. The `vfwprintf()` function does not invoke the `va_end` macro.

Refer to the "wprintf" on page 605 function for details of the format string.

The vswprintf function does not invoke the va_end macro.

Returns the number of characters written not counting the terminating null wide character. Otherwise a negative value if a failure occurs.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"swprintf" on page 569

"vsnprintf" on page 393

vwprintf

Write a formatted text to a wide character oriented stdout

```
#include <wchar.h>
int vwprintf(const wchar_t * format, va_list arg);
```

Table 45.25 vwprintf

format	wchar_t*	The format string
....		Variable arguments

Remarks

The `vwprintf()` function works identically to the `wprintf()` function. Instead of the variable list of arguments that can be passed to `wprintf()`, `vwprintf()` accepts its arguments in the array `arg` of type `va_list`, which must have been initialized by the `va_start()` macro (and possibly subsequent `va_arg` calls) from the stdarg.h header file. The `vwprintf()` function does not invoke the `va_end` macro.

Refer to the "wprintf" on page 605 function for details of the format string.

Returns the number of characters written or a negative value if it failed.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"vprintf" on page 391

watof

Convert a wide character string to a double type

```
#include <wchar.h>
double watof(wchar_t * str);
```

Table 45.26 watof

str	wchar_t	The wide character string to be converted

Remarks

Performs the same task as `atof` for a wide character type.

Returns the converted value or, if no conversion could be performed, zero.

This function may not be implemented on all platforms.

See Also

"atof" on page 410

wcrtomb

Translate a `wchar_t` type to a multibyte character according to the encoding scheme specified in the `LC_CTYPE` component of the current locale. This function is essentially the same as `wctomb()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

```
#include <stdlib.h>
int wcrtomb(char *s, wchar_t wchar, mbstate_t * ps);
```

Table 45.27 wcrtomb

s	char *	A multibyte string buffer
wchar	wchar_t	A wide character to convert
ps	mbstate_t *	The current state of translation between multibyte and wide character, ignored if the encoding scheme is non-modal.

Remarks

If s a null pointer, this call is equivalent to `wcrtomb(buf, L'\0', ps)` where `buf` is an internal buffer.

If s is not a null pointer, `wcrtomb()` determines the length of the `UTF-8` multibyte string that corresponds to the wide character `wchar` and stores that string in the multibyte string pointed to by `s`. At most `MB_CUR_MAX` bytes are stored.

`wcrtomb()` returns the number of bytes stored in the string `s`.

This function may not be implemented on all platforms.

See Also

"mbrtowc" on page 565

"wcsrtombs" on page 587

wcscat

Wide character string concatenation

```
#include <wchar.h>
wchar_t * wcscat(wchar_t * dst, const wchar_t * src);
```

Table 45.28 wcscat

dst	wchar_t *	The destination string
src	wchar_t *	The source string

Remarks

Performs the same task as `strcat` for a wide character type.

Returns a pointer to the destination string

This function may not be implemented on all platforms.

See Also

"strcat" on page 459

wcschr

Search for an occurrence of a wide character.

```
#include <wchar.h>
wchar_t * wcschr(const wchar_t * str, const wchar_t chr);
```

Table 45.29 wcschr

str	wchar_t *	The string to be searched
chr	wchar_t	The wide character to search for

Remarks

Performs the same task as `strchr` for a wide character type.

Returns a pointer to the successfully located wide character. If it fails, `wcschr()` returns a null pointer (`NULL`).

This function may not be implemented on all platforms.

See Also

"strchr" on page 460

wcscmp

Compare two wide character arrays.

```
#include <wchar.h>
int wcscmp(const wchar_t * str1, const wchar_t * str2);
```

Table 45.30 wcscmp

str1t	wchar_t *	Comparison string
str2	wchar_t *	Comparison string

Remarks

Performs the same task as `strcmp` for a wide character type.

Returns a zero if `str1` and `str2` are equal, a negative value if `str1` is less than `str2`, and a positive value if `str1` is greater than `str2`.

This function may not be implemented on all platforms.

See Also

"strcmp" on page 461

"wmemcmp" on page 602

wcscoll

Compare two wide character arrays according to the collating sequence defined in the LC_COLLATE component of the current locale.

```
#include <wchar.h>
int wcscoll(const wchar_t *str1, const wchar_t * str2);
```

Table 45.31 wcscoll

str1	wchar_t *	First comparison string
str2	wchar_t *	Second comparison string

Remarks

Performs the same task as `strcoll` for a wide character type.

Returns zero if `str1` is equal to `str2`, a negative value if `str1` is less than `str2`, and a positive value if `str1` is greater than `str2`.

This function may not be implemented on all platforms.

See Also

"strcoll" on page 462

"wcscmp" on page 579

"wmemcmp" on page 602

wcscspn

Find the first wide character in one wide character string that is also in another.

```
#include <wchar.h>
size_t wcscspn(const wchar_t * str, const wchar_t * set);
```

Table 45.32 wcscspn

str	wchar_t *	The string to be searched
set	wchar_t *	The set of characters to be searched for

Remarks

The `wcscspn()` function finds the first wide character in the `null` terminated wide character string `s1` that is also in the `null` terminated wide character string `s2`. for this purpose, the `null` terminators are considered part of the strings. The function starts examining characters at the beginning of `s1` and continues searching until a wide character in `s1` matches a wide character in `s2`.

`wcscspn()` returns the index of the first wide character in `s1` that matches a wide character in `s2`.

This function may not be implemented on all platforms.

See Also

"strcspn" on page 465

wcscpy

Copy one wide character array to another.

```
#include <wchar.h>
wchar_t * (wcscpy)(wchar_t * dst, const wchar_t * src);
```

Table 45.33 wcscpy

dst	wchar_t *	The destination string
src	wchar_t *	The source being copied

Remarks

The `wcscpy()` function copies the wide character array pointed to by `src` to the wide character array pointed to `dst`. The `src` argument must point to a null terminated wide character array. The resulting wide character array at `dest` is null terminated as well.

If the arrays pointed to by `dest` and `source` overlap, the operation of strcpy() is undefined.

Returns a pointer to the destination string.

This function may not be implemented on all platforms.

See Also

"strcpy" on page 464

wcsftime

Formats a wide character string for time.

```
#include <wchar.h>
size_t wcsftime(wchar_t * str, size_t max_size,
const wchar_t * format_str, const struct tm * timeptr);
```

Table 45.34 wcsftime

str	wchar_t *	The destination string
max_size	size_t	Maximum string size
format_str	const wchar_t *	The format string
timeptr	const struct tm *	The time structure containing the calendar time

Remarks

Performs the same task as `strftime` for a wide character type.

The `wcsftime` function returns the total number of characters in the argument `str` if the total number of characters including the null character in the string argument `str` is less than the value of `max_size` argument. If it is greater, `wcsftime` returns 0

This function may not be implemented on all platforms.

See Also

"strftime" on page 499

wcslen

Compute the length of a wide character array.

```
#include <wchar.h>
size_t (wcslen)(const wchar_t * str);
```

Table 45.35 wcslen

str	wchar_t *	The string to compute

Remarks

The `wcslen()` function computes the number of characters in a null terminated wide character array pointed to by `str.` The null character `(L'\0')` is not added to the wide character count.

Returns the number of characters in a wide character array not including the terminating `null` character.

This function may not be implemented on all platforms.

See Also

"strlen" on page 468

wcsncat

Append a specified number of characters to a wide character array.

```
#include <wchar.h>
wchar_t * wcsncat(wchar_t * dst, const wchar_t * src, size_t
     n);
```

Table 45.36 wcsncat

dst	wchar_t *	The destination string
src	wchar_t *	The string to be appended
n	size_t	The number of characters to copy

Remarks

The `wcsncat()` function appends a maximum of n characters from the wide character array pointed to by source to the wide character array pointed to by `dest`. The `dest` argument must point to a null terminated wide character array. The `src` argument does not necessarily have to point to a null terminated wide character array.

If a null wide character is reached in `src` before `n` characters have been appended, `wcsncat()` stops.

When done, `wcsncat()` terminates dest with a null wide character `(L'\0')`.

Returns a pointer to the destination string.

This function may not be implemented on all platforms.

See Also

"strncat" on page 469

wcsncmp

Compare not more than a specified number of wide characters.

```
#include <wchar.h>
int csncmp(const wchar_t * str1,
const wchar_t * str2, size_t n);
```

Table 45.37 csncmp

str1	wchar_t *	First comparison string
str2	wchar_t *	Second comparison string
n	size_t	Maximum number of characters to compare

Remarks

Performs the same task as `strncmp` for a wide character type.

Returns a zero if the first `n` characters of `str1` and `str2` are equal, a negative value if `str1` is less than `str2`, and a positive value if `str1` is greater than `str2`.

This function may not be implemented on all platforms.

See Also

"strncmp" on page 470

wcsncpy

Copy a specified number of wide characters.

```
#include <wchar.h>
wchar_t * wcsncpy(wchar_t * dst,
const wchar_t * src, size_t n);
```

Table 45.38 wcsncpy

dst	wchar_t *	Destination string
src	wchar_t *	Source to be copied
n	size_t	Number of characters to copy

Remarks

Performs the same task as `strncpy` for a wide character type.

Returns a pointer to the destination string.

This function may not be implemented on all platforms.

See Also

"strncpy" on page 472

"wcscpy" on page 581

wcspbrk

Look for the first occurrence of an element of an array of wide characters in another.

```
#include <wchar.h>
wchar_t * wcspbrk(const wchar_t * str, const wchar_t * set);
```

Table 45.39 wcspbrk

str	wchar_t*	The string being searched
set	wchar_t *	The search set

Remarks

Performs the same task as `strpbrk` for a wide character type.

Returns a pointer to the first wide character in `str` that matches any wide character in `set`, and returns a null pointer (`NULL`) if no match was found.

This function may not be implemented on all platforms.

See Also

"strpbrk" on page 473

wcsrchr

Search a wide character string for the last occurrence of a specified wide character.

```
#include <wchar.h>
wchar_t * wcsrchr(const wchar_t * str, wchar_t chr);
```

Table 45.40 wcsrchr

str	const wchar_t *	The string being searched
chr	wchar_t	The wide character to search for

Remarks

Performs the same task as `strrchr` for a wide character type.

Returns a pointer to the wide character found or returns a null pointer (`NULL`) if it fails.

This function may not be implemented on all platforms.

See Also

"strchr" on page 460

wcsrtombs

Translate a `wchar_t` type character array to a multibyte character array. This function is essentially the same as `wcstombs()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

```
#include <wchar.h>
size_t wcsrtombs(char *dst, const wchar_t **src, size_t n,
     mbstate_t * ps);
```

Table 45.41 wcsrtombs

dst	char *	The character string destination
src	const wchar_t *	Indirect pointer to the wide character string to be converted
n	size_t	The maximum length to convert
ps	mbstate_t *	The current state of translation between multibyte and wide character, ignored if the encoding scheme is non-modal.

Remarks

The MSL implementation of the `wcsrtombs()` function converts a character array containing `wchar_t` type Unicode characters indirectly pointed to by src to a character array containing `UTF-8` multibyte characters. If dst is not a null pointer, these multibyte characters are stored in the array pointed to by dst. Conversion continues until either a terminating null wide character is encountered or (if dst is not a null pointer) if the translation of the next wide character would cause the total number of bytes to be stored to exceed `n`.

If `dst` is not a null pointer, the `wchar_t *` object pointed to by `src` is assigned either a `null pointer` if conversion ended because a `null` wide character was reached or the address just past the last wide character converted.

`wcsrtombs()` returns the number of bytes modified in the character array pointed to by `s`, not including a terminating `null` character, if any.

This function may not be implemented on all platforms.

See Also

"wcrtomb" on page 577

"mbsrtowcs" on page 567

wcsspn

Count the number of wide characters in one wide character array that are in another.

```
#include <wchar.h>
size_t wcsspn(const wchar_t * str, const wchar_t * set);
```

Table 45.42 wcsspn

str	wchar_t *	The searched string
set	const wchar_t *	The search set

Remarks

Performs the same task as `strspn` for a wide character type.

Returns the number of characters in the initial segment of `str` that contains only characters that are elements of `set`.

This function may not be implemented on all platforms.

See Also

"strspn" on page 475

wcsstr

Search for a wide character array within another.

```
#include <wchar.h>
wchar_t * wcsstr(const wchar_t * str, const wchar_t * pat);
```

Table 45.43 wcsstr

str	const wchar_t *	The string to search
pat	const wchar_t *	The string being searched for

Remarks

Performs the same task as `strstr` for a wide character type.

Returns a pointer to the first occurrence of `s2` in `s1` and returns a null pointer (`NULL`) if `s2` cannot be found.

This function may not be implemented on all platforms.

See Also

"strstr" on page 476

"wcschr" on page 579

wcstod

Converts a wide character array to double values.

```
#include <wchar.h>
double wcstod(wchar_t * str, char ** end);
```

Table 45.44 wcstod

str	wchar_t *	The string being converted
end	char **	If not null, a pointer to the first position not convertible.

Remarks

The `wcstod()` function converts a wide character array, pointed to by nptr, to a floating point value of type `double`. The wide character array can be in either decimal or hexadecimal floating point constant notation (e.g. `103.578`, `1.03578e+02`, or `0x1.9efef9p+6`).

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to a value of type double.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

Returns a floating point value of type `double`. If `str` cannot be converted to an expressible double value, `wcstod()` returns `HUGE_VAL`, defined in `math.h`, and sets `errno` to `ERANGE`

This function may not be implemented on all platforms.

See Also

"wcstof" on page 591
"strtod" on page 435
"wcstold" on page 594
"errno" on page 75

wcstof

Wide character array conversion to floating point value of type float.

```
#include <wchar.h>
float wcstof(const wchar_t * restrict nptr,
wchar_t ** restrict endptr);
```

Table 45.45 wcstof

nptr	const wchar_t *	A Null terminated wide character array to convert
endptr	wchar_t **	A pointer to a position in nptr that follows the converted part.

Remarks

The `wcstof()` function converts a wide character array, pointed to by nptr, to a floating point value of type float. The wide character array can be in either decimal or hexadecimal floating point constant notation (e.g. `103.578`, `1.03578e+02`, or `0x1.9efef9p+6`).

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to a value of type float.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`wcstof()` returns a floating point value of type `float`. If `nptr` cannot be converted to an expressible float value, `wcstof()` returns `HUGE_VAL`, defined in `math.h`, and sets `errno` to `ERANGE`.

This function may not be implemented on all platforms.

See Also

"wcstod" on page 590
"wcstold" on page 594
"strtof" on page 437
"errno" on page 75

wcstok

Extract tokens within a wide character array.

```
#include <wchar.h>
wchar_t * wcstok(wchar_t * str,
const wchar_t * set, wchar_t ** ptr););
```

Table 45.46 wcstok

str	wchar_t *	The string to be modified
set	wchar_t *	The list of wide character to find
ptr	wchar_t *	Continuation information

Remarks

Performs the same task as `strtok` for a wide character type however, it makes use of a third argument to contain sufficient information to continue the tokenization process.

When first called, the first argument is non-null. `wcstok()` returns a pointer to the first token in `str` or returns a null pointer if no token can be found.

Subsequent calls to `wcstok()` with a `NULL str` argument causes wcstok() to return a pointer to the next token or return a null pointer (NULL) when no more tokens exist. When called with a NULL str argument, the value of the ptr argument must have been set by a previous call to `wcstok()`.

The `wcstok` function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

This function may not be implemented on all platforms.

See Also

"strtok" on page 477

"errno" on page 75

wcstol

Wide character array conversion to floating point value of type long int.

```
#include <wchar.h>
long int wcstol(const wchar_t * restrict nptr,
wchar_t ** restrict endptr, int base);
```

Table 45.47 wcstol

nptr	const wchar_t *	A Null terminated wide character array to convert
endptr	wchar_t **	A pointer to a position in nptr that follows the converted part
base	int	A numeric base between 2 and 36

Remarks

The `wcstol()` function converts a wide character array, pointed to by nptr, to an integer value of type long. The base argument in `wcstold()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then `wcstold()` converts the wide character array based on its format. Wide character arrays beginning with `'0'` are assumed to be octal, number strings beginning with `'0x'` or `'0X'` are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to a value of type long int.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`wcstol()` returns an signed integer value of type `long int`. If the converted value is less than `LONG_MIN`, `wcstol()` returns `LONG_MIN` and sets `errno` to `ERANGE`. If the converted value is greater than `LONG_MAX`, `wcstol()` returns `LONG_MAX` and sets `errno` to `ERANGE`. The `LONG_MIN` and `LONG_MAX` macros are defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"strtol" on page 438

"errno" on page 75

"wcstoll" on page 595

"wcstoul" on page 596

wcstold

A wide character array conversion to an floating point value of type long double.

```
#include <wchar.h>
long double wcstold(const wchar_t * restrict nptr,
wchar_t ** restrict endptr);
```

Table 45.48 wcstold

nptr	const wchar_t *	A Null terminated wide character array to convert
endptr	wchar_t **	A pointer to a position in nptry that is not convertible.

Remarks

The `wcstold()` function converts a wide character array, pointed to by `nptr`, expected to represent an integer expressed in radix base, to an integer value of type `long int`. A plus or minus sign (+ or -) prefixing the number string is optional. The wide character array can be in either decimal or hexadecimal floating point constant notation (e.g. 103.578, 1.03578e+02, or 0x1.9efef9p+6).

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to a double value.

In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

Returns a floating point value of type long double. If nptr cannot be converted to an expressible double value, wcstold() returns HUGE_VAL, defined in math.h, and sets errno to ERANGE

This function may not be implemented on all platforms.

See Also

"wcstod" on page 590

"wcstof" on page 591

"strtold" on page 441

wcstoll

Wide character array conversion to integer value of type long long int.

```
#include <wchar.h>
long long int wcstoll(const wchar_t * restrict nptr,
wchar_t ** restrict endptr, int base);
```

Table 45.49 wcstoll

nptr	const wchar_t *	A Null terminated wide wide character array to convert
endptr	wchar_t **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The wcstoll() function converts a wide character array, pointed to by nptr, expected to represent an integer expressed in radix base to an integer value of type long long int. A plus or minus sign (+ or -) prefixing the number string is optional.

The base argument in `wcstoll()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then `wcstoll()` converts the wide character array based on its format. Wide character arrays beginning with'0' are assumed to be octal, number strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to a long long int value.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`wcstoll()` returns an `integer` value of type `long long int`. If the converted value is less than `LLONG_MIN`, `wcstoll()` returns `LLONG_MIN` and sets `errno` to `ERANGE`. If the converted value is greater than `LLONG_MAX`, `wcstoll()` returns `LLONG_MAX` and sets `errno` to `ERANGE`. The `LLONG_MIN` and `LLONG_MAX` macros are defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"strtoll" on page 442

"wcstol" on page 593

"wcstoul" on page 596

"errno" on page 75

wcstoul

Wide character array conversion to integer value of type unsigned long int.

```
#include <wchar.h>
unsigned long int wcstoul(const wchar_t * restrict nptr,
wchar_t ** restrict endptr, int base);
```

Table 45.50 wcstoul

nptr	const wchar_t *	A Null terminated wide character array to convert
endptr	wchar_t **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `wcstoul()` function converts a wide character array, pointed to by `nptr`, to an integer value of type `unsigned long int`, in base. A plus or minus sign prefix is ignored.

The `base` argument in `wcstoul()` specifies the base used for conversion. It must have a value between 2 and 36, or 0.The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If `base` is 0, then `strtol()` and `wcstoul()` convert the wide character array based on its format. Wide character arrays beginning with `'0'` are assumed to be octal, number strings beginning with `'0x'` or `'0X'` are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a null pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr`. This position marks the first wide character that is not convertible to the functions' respective types.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`wcstoul()` returns an unsigned integer value of type `unsigned long int`. If the converted value is greater than `ULONG_MAX`, `wcstoul()` returns `ULONG_MAX` and sets `errno` to `ERANGE`. The `ULONG_MAX` macro is defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"wcstol" on page 593

"wcstoll" on page 595

"wcstoull" on page 598

"strtoul" on page 443

"errno" on page 75

wcstoull

Wide character array conversion to integer value of type unsigned long long int.

```
#include <wchar.h>
unsigned long long int wcstoull(
const wchar_t * restrict nptr,
wchar_t ** restrict endptr, int base);
```

Table 45.51 wcstoull

nptr	const wchar_t *	A Null terminated wide character array to convert
endptr	wchar_t **	A pointer to a position in nptry that is not convertible.
base	int	A numeric base between 2 and 36

Remarks

The `wcstoull()` function converts a wide character array, pointed to by `nptr,` expected to represent an integer expressed in radix base to an integer value of type `unsigned long long int.` A plus or minus sign prefix is ignored.

The base argument in `wcstoull()` specifies the base used for conversion. It must have a value between 2 and 36, or 0. The letters a (or A) through z (or Z) are used for the values 10 through 35; only letters and digits representing values less than base are permitted. If base is 0, then `wcstoull()` converts the wide character array based on its format. Wide character arrays beginning with'0' are assumed to be octal, number strings beginning with'0x' or'0X' are assumed to be hexadecimal. All other number strings are assumed to be decimal.

If the `endptr` argument is not a `null` pointer, it is assigned a pointer to a position within the wide character array pointed to by `nptr.` This position marks the first wide character that is not convertible to a `long long` value.

In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

This function skips leading white space.

`wcstoull()` returns an `unsigned integer` value of type `unsigned long long int`. If the converted value is greater than `ULLONG_MAX`, `wcstoull()` returns `ULLONG_MAX` and sets `errno` to `ERANGE`. The `ULLONG_MAX` macro is defined in `limits.h`

This function may not be implemented on all platforms.

See Also

"wcstol" on page 593

"wcstoll" on page 595

"wcstoul" on page 596

"strtoul" on page 443

"errno" on page 75

wcsxfrm

Transform a wide character array as specified in the `LC_COLL` component of the current locale.

```
#include <wchar.h>
size_t wcsxfrm(wchar_t * str1, const wchar_t * str2, size_t
     n);
```

Table 45.52 wcsxfrm

str1	wchar_t *	The destination string
str2	wchar_t *	The source string
n	size_t	Maximum number of characters

Remarks

Performs the same task as `strxfrm` for a wide character type.

Returns the length of the transformed wide string in `str1`, not including the terminating `null` wide character. If the value returned is `n` or greater, the contents of `str1` are indeterminate.

This function may not be implemented on all platforms.

See Also

"strxfrm" on page 479

wctime

Convert a `time_t` type to a wide character array

```
#include <wchar.h>
wchar_t * wctime(const time_t * timer);
```

Table 45.53 wctime

timer	const time_t *	The Calendar Time

Remarks

Performs the same task as `ctime` for a wide character type.

Returns a pointer to wide character array containing the converted `time_t` type

This function may not be implemented on all platforms.

See Also

"ctime" on page 492

wctob

The function `wctob()` converts a wide character to a byte character.

```
#include <wchar.h>
int wctob(wint_t wc);
```

Table 45.54 wctob

int	wchar_t *	The wide character to be converted

Returns

The function `wctob()` returns the single byte representation of the argument `wc` as an unsigned char converted to an int or `EOF` is returned if `wc` does not correspond to a valid multibyte character.

This function may not be implemented on all platforms.

See Also

"wcrtomb" on page 577

wmemchr

Search for an occurrence of a specific wide character.

```
#include <wchar.h>
void * wmemchr(const void * src, int val, size_t n);
```

Table 45.55 wmemchr

src	const void *	The string to be searched
val	int	The value to search for
n	size_t	The maximum length of a search

Remarks

Performs the same task as `memchr()` for a wide character type.

Returns a pointer to the found wide character, or a null pointer (`NULL`) if `val` cannot be found.

This function may not be implemented on all platforms.

See Also

"memchr" on page 454

"wcschr" on page 579

wmemcmp

Compare two blocks of memory, treated as wide characters.

```
#include <wchar.h>
int wmemcmp(const void * src1, const void * src2, size_t n);
```

Table 45.56 wmemcmp

src1	const void *	First memory block to compare
src2	const void *	Second memory block to compare
n	size_t	Maximum number of wide characters to compare

Remarks

Performs the same task as `memcmp()` for a wide character type.

The function `wmemcmp` returns a zero if all `n` characters pointed to by `src1` and `src2` are equal.

The function `wmemcmp` returns a negative value if the first non-matching wide character pointed to by `src1` is less than the wide character pointed to by `src2`.

The function `wmemcmp` returns a positive value if the first non-matching wide character pointed to by `src1` is greater than the wide character pointed to by `src2`.

This function may not be implemented on all platforms.

See Also

"memcmp" on page 456

"wcscmp" on page 579

wmemcpy

Copy a contiguous memory block.

```
#include <wchar.h>
void * (wmemcpy)(void * dst, const void * src, size_t n);
```

Table 45.57 wmemcpy

dst	void *	The destination string
src	const void *	The source string
n	size_t	Maximum length to copy

Remarks

Performs the same task as `memcpy()` for a wide character type.

Returns a pointer to the destination string.

This function may not be implemented on all platforms.

See Also

"memcpy" on page 457

wmemmove

Copy an overlapping contiguous memory block.

```
#include <wchar.h>
void * (wmemmove)(void * dst, const void * src, size_t n);
```

Table 45.58 wmemmove

dst	void *	The destination string
src	const void *	The source string
n	size_t	The maximum length to copy

Remarks

Performs the same task as `memmove()` for a wide character type.

Returns a pointer to the destination string.

This function may not be implemented on all platforms.

See Also

"memmove" on page 458

"wcscpy" on page 581

wmemset

Clear the contents of a block of memory.

```
#include <wchar.h>
void * wmemset(void * dst, int val, size_t n);
```

Table 45.59 wmemset

dst	void *	The destination string
val	int	The value to be set
n	size_t	The maximum length

Remarks

Performs the same task as `memset()` for a wide character type.

Returns a pointer to the destination string

This function may not be implemented on all platforms.

See Also

"memset" on page 458

wprintf

Send formatted wide character text to a standard output.

```
#include <wchar.h>
int wprintf(const wchar_t * format, ...);
```

Table 45.60 wprintf

format	wchar_t*	The format string
....		Variable arguments

Remarks

Performs the same task as `printf()` for a wide character type.

Returns the number of arguments written or a negative number if an error occurs.

Table 45.61 Length Modifiers for Formatted Output Functions

Modifier	Description
h	The h flag followed by d, i, o, u, x, or X conversion specifier indicates that the corresponding argument is a short int or unsigned short int.
l	The lower case L followed by d, i, o, u, x, or X conversion specifier indicates the argument is a long int or unsigned long int. The lower case L followed by a `c` conversion specifier indicates that the argument is of type `wint_t`. The lower case L followed by an `s` conversion specifier indicates that the argument is of type `wchar_t`.
ll	The double l followed by d, i, o, u, x, or X conversion specifier indicates the argument is a long long or unsigned long long
L	The upper case L followed by e, E, f, g, or G conversion specifier indicates a long double.

Table 45.61 Length Modifiers for Formatted Output Functions (*continued*)

Modifier	Description
v	**AltiVec**: `A vector bool char, vector signed char or vector unsigned char` when followed by `c, d, i, o, u, x or X` A vector float, when followed by `f`.
vh hv	**AltiVec**: A `vector short, vector unsigned short, vector bool short or vector pixel` when followed by `c, d, i, o, u, x or X`
vl lv	**AltiVec**: A `vector int, vector unsigned int or vector bool int` when followed by `c, d, i, o, u, x or X`

Table 45.62 Flag Specifiers for Formatted Output Functions

Modifier	Description
-	The conversion will be left justified.
+	The conversion, if numeric, will be prefixed with a sign (+ or -). By default, only negative numeric values are prefixed with a minus sign (-).
space	If the first character of the conversion is not a sign character, it is prefixed with a space. Because the plus sign flag character (+) always prefixes a numeric value with a sign, the space flag has no effect when combined with the plus flag.

Table 45.62 Flag Specifiers for Formatted Output Functions (*continued*)

Modifier	Description
#	For c, d, i, and u conversion types, the # flag has no effect. For s conversion types, a pointer to a Pascal string, is output as a character string. For o conversion types, the # flag prefixes the conversion with a 0. For x conversion types with this flag, the conversion is prefixed with a 0x. For e, E, f, g, and G conversions, the # flag forces a decimal point in the output. For g and G conversions with this flag, trailing zeroes after the decimal point are not removed.
0	This flag pads zeroes on the left of the conversion. It applies to d, i, o, u, x, X, e, E, f, g, and G conversion types. The leading zeroes follow sign and base indication characters, replacing what would normally be space characters. The minus sign flag character overrides the 0 flag character. The 0 flag is ignored when used with a precision width for d, i , o, u, x, and X conversion types.
@	**AltiVec**: This flag indicates a pointer to a string specified by an argument. This string will be used as a separator for vector elements.

Table 45.63 Conversion Specifiers for Formatted Output Functions

Modifier	Description
d	The corresponding argument is converted to a signed decimal.
i	The corresponding argument is converted to a signed decimal.
o	The argument is converted to an unsigned octal.
u	The argument is converted to an unsigned decimal.

Table 45.63 Conversion Specifiers for Formatted Output Functions (*continued*)

Modifier	Description
x, X	The argument is converted to an unsigned hexadecimal. The x conversion type uses lowercase letters (abcdef) while X uses uppercase letters (ABCDEF).
n	This conversion type stores the number of items output by printf() so far. Its corresponding argument must be a pointer to an int.
f, F	The corresponding floating point argument (float, or double) is printed in decimal notation. The default precision is 6 (6 digits after the decimal point). If the precision width is explicitly 0, the decimal point is not printed. For the `f` conversion specifier, a double argument representing infinity produces `[-]inf;` a double argument representing a `NaN` (Not a number) produces `[-]nan.` For the *F* conversion specifier, `[-]INF` or `[-]NAN` are produced instead.
e, E	The floating point argument (float or double) is output in scientific notation: `[-]b.aaae±Eee.` There is one digit (*b*) before the decimal point. Unless indicated by an optional precision width, the default is 6 digits after the decimal point (`aaa`). If the precision width is 0, no decimal point is output. The exponent (*ee*) is at least 2 digits long. The e conversion type uses lowercase e as the exponent prefix. The E conversion type uses uppercase E as the exponent prefix.
g, G	The g conversion type uses the f or e conversion types and the G conversion type uses the f or E conversion types. Conversion type e (or E) is used only if the converted exponent is less than -4 or greater than the precision width. The precision width indicates the number of significant digits. No decimal point is output if there are no digits following it.

Table 45.63 Conversion Specifiers for Formatted Output Functions (*continued*)

Modifier	Description
c	The corresponding argument is output as a character.
s	The corresponding argument, a pointer to a character array, is output as a character string. Character string output is completed when a null character is reached. The null character is not output.
p	The corresponding argument is taken to be a pointer. The argument is output using the X conversion type format.

Table 45.64 CodeWarrior Extensions for Formatted Output Functions

Modifier	Description
#s	The corresponding argument, a pointer to a Pascal string, is output as a character string. A Pascal character string is a length byte followed by the number characters specified in the length byte. **Note**: This conversion type is an extension to the ANSI C library but applied in the same manner as for other format variations.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"printf" on page 353

"fwprintf" on page 561

wscanf

Reads a wide character formatted text from standard input

```
#include <wchar.h>
int wscanf(const wchar_t * format, ...);
```

Table 45.65 wscanf

format	wchar_t*	The format string
....		Variable arguments

Remarks

Performs the same task as `scanf()` for a wide character type.

Returns the number of items successfully read and returns `WEOF` if a conversion type does not match its argument or and end-of-file is reached.

Table 45.66 Length Specifiers for Formatted Input

Modifier	Description
hh	The `hh` flag indicates that the following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument that is of type `char` or `unsigned char`.
h	The `h` flag indicates that the following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument that is of type `short int` or `unsigned short int`.
l	When used with integer conversion specifier, the `l` flag indicates `long int` or an `unsigned long int` type. When used with floating point conversion specifier, the `l` flag indicates a `double`. When used with a `c` or `s` conversion specifier, the `l` flag indicates that the corresponding argument with type pointer to `wchar_t`.

Table 45.66 Length Specifiers for Formatted Input (*continued*)

Modifier	Description
ll	When used with integer conversion specifier, the ll flag indicates that the corresponding argument is of type long long or an unsigned long long.
L	The L flag indicates that the corresponding float conversion specifier corresponds to an argument of type long double.
v	**AltiVec**: `A vector bool char, vector signed char or vector unsigned char` when followed by `c, d, i, o, u, x or X` A vector float, when followed by `f`.
vh hv	**AltiVec**: `vector short, vector unsigned short, vector bool short or vector pixel` when followed by `c, d, i, o, u, x or X`
vl lv	**AltiVec**: `vector long, vector unsigned long or vector bool` when followed by `c, d, i, o, u, x or X`

Table 45.67 Conversion Specifiers for Formatted Input

Modifier	Description
d	A decimal integer is read.
i	A decimal, octal, or hexadecimal integer is read. The integer can be prefixed with a plus or minus sign (+, -), 0 for octal numbers, 0x or 0X for hexadecimal numbers.
o	An octal integer is read.
u	An unsigned decimal integer is read.
x, X	A hexadecimal integer is read.

Table 45.67 Conversion Specifiers for Formatted Input (*continued*)

Modifier	Description
e, E, f, g, G	A floating point number is read. The number can be in plain decimal format (e.g. 3456.483) or in scientific notation `([-]b.aaae[-]dd)`.
s	If the format specifier s is preceded with an l (el) length modifier, then the corresponding argument must be a pointer to an array of wchar_t. This array must be large enough to accept the sequence of wide_characters being read, including the terminating null character, automatically appended. If there is no preceding l length modifier then the corresponding argument must be an array of characters. The wide-characters read from the input field will be converted to a sequence of multibyte characters before being assigned to this array. That array must be large enough to accept the sequence of multibyte characters including the terminating null character automatically appended.
c	A character is read. White space characters are not skipped, but read using this conversion specifier.
p	A pointer address is read. The input format should be the same as that output by the p conversion type in printf().
n	This conversion type does not read from the input stream but stores the number of characters read so far in its corresponding argument.
[scanset]	Input stream characters are read and filtered determined by the `scanset`. See "wscanf" for a full description.

This function may not be implemented on all platforms.

See Also

"Wide Character and Byte Character Stream Orientation" on page 555

"scanf" on page 370

"fwscanf" on page 562

Non Standard <wchar.h> Functions

Various non standard functions are included in the header `wchar.h` for legacy source code and compatibility with operating system frameworks and application programming interfaces.

- For the function `wcsdup`, see "wcsdup" on page 107 for a full description.
- For the function `wcsicmp`, see "wcsicmp" on page 108 for a full description.
- For the function `wcslwr`, see "wcslwr" on page 109 for a full description.
- For the function `wcsincmp`, see "wcsnicmp" on page 111 for a full description.
- For the function `wcsnset`, see "wcsnset" on page 112 for a full description.
- For the function `wcsrev`, see "wcsrev" on page 113 for a full description.
- For the function `wcsset`, see "wcsset" on page 113 for a full description.
- For the function `wcspnp`, see "wcsspnp" on page 114 for a full description.
- For the function `wcsupr`, see "wcsupr" on page 114 for a full description.
- For the function `wtoi`, see "wtoi" on page 115 for a full description.

46

wctype.h

The `ctype.h` header file supplies macros and functions for testing and manipulation of wide character type.

Overview of wctype.h

This header file defines the facilities as follows:

- "wctype.h Types" on page 616 defines two types used for wide character values.
- "iswalnum" on page 616 tests for alpha-numeric wide characters.
- "iswalpha" on page 616 tests for alphabetical wide characters.
- "iswblank" on page 617 tests for a blank space or space holder.
- "iswcntrl" on page 618 tests for control wide characters.
- "iswdigit" on page 618 tests for digital wide characters.
- "iswgraph" on page 619 tests for graphical wide characters.
- "iswlower" on page 619 tests for lower wide characters.
- "iswprint" on page 620 tests for printable wide characters.
- "iswpunct" on page 620 tests for punctuation wide characters.
- "iswspace" on page 621 tests for whitespace wide characters.
- "iswupper" on page 621 tests for uppercase wide characters.
- "iswxdigit" on page 622 tests for a hexadecimal wide character type.
- "towlower" on page 623 converts wide characters to lower case.
- "towupper" on page 624 converts wide characters to upper case.

Mapping Facilities

- "towctrans" on page 623 maps a wide character type to another wide character type.
- "wctrans" on page 624 constructs a value that represents a mapping between wide characters.

Types

The `wctype.h` header file contains two types described in Table 46.1 used for wide character manipulations.

Table 46.1 wctype.h Types

wctrans_t	A scalar type which can represent locale specific character mappings
wctype_t	A scalar type that can represent locale specific character classifications

iswalnum

Tests for alpha-numeric wide characters.

```
#include <wctype.h>
int iswalnum(wchar_t wc);
```

Table 46.2 iswalnum

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isalnum for wide character type.

In the `"C"` locale `true` is returned for an alphanumeric: `[a-z]`, `[A-Z]`, `[0-9]`

This function may not be implemented on all platforms.

See Also

"isalnum" on page 55

iswalpha

Tests for alphabetical wide characters.

```
#include <wctype.h>
int iswalpha(wchar_t wc);
```

Table 46.3 iswalpha

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isalpha for wide character type.

In the "C" locale `true` is returned for an alphabetic: `[a-z]`, `[A-Z]`

This function may not be implemented on all platforms.

See Also

"isalpha" on page 57

iswblank

Tests for a blank space or a word separator dependent upon the locale usage.

```
#include <wctype.h>
int isblank(win_t c);
```

Table 46.4 isblank

c	win_t	character being evaluated

Remarks

This function determines if a wide character is a blank space or tab or if the wide character is in a locale specific set of wide characters for which `iswspace` is `true` and is used to separate words in text.

In the "C" locale, isblank returns true only for the space and tab characters.

Return

True is returned if the criteria are met.

This function may not be implemented on all platforms.

See Also

"iswspace" on page 621

iswcntrl

Tests for control wide characters.

```
#include <wctype.h>
int iswcntrl(wchar_t wc);
```

Table 46.5 iswcntrl

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as iscntrl for wide character type.

True for the delete character (`0x7F`) or an ordinary control character from `0x00` to `0x1F`.

This function may not be implemented on all platforms.

See Also

"iscntrl" on page 58

iswdigit

Tests for digital wide characters.

```
#include <wctype.h>
int iswdigit(wchar_t wc);
```

Table 46.6 iswdigit

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isdigit for wide character type.

In the `"C"` locale `true` is returned for a numeric character: `[0-9]`.

This function may not be implemented on all platforms.

See Also

"isdigit" on page 58

iswgraph

Tests for graphical wide characters.

```
#include <wctype.h>
int iswgraph(wchar_t wc);
```

Table 46.7 iswgraph

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isgraph for wide character type.

In the "C" locale `true` is returned for a non-space printing character from the exclamation (`0x21`) to the tilde (`0x7E`).

This function may not be implemented on all platforms.

See Also

"isgraph" on page 59

iswlower

Tests for lowercase wide characters.

```
#include <wctype.h>
int iswlower(wchar_t wc);
```

Table 46.8 iswlower

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as islower for wide character type.

In the "C" locale `true` is returned for a lowercase letter: `[a-z]`.

This function may not be implemented on all platforms.

See Also

"islower" on page 59

"iswupper" on page 621

iswprint

Tests for printable wide characters.

```
#include <wctype.h>
int iswprint(wchar_t wc);
```

Table 46.9 iswprint

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isprint for wide character type.

In the `"C"` locale `true` is returned for a printable character from space (`0x20`) to tilde (`0x7E`).

This function may not be implemented on all platforms.

See Also

"isprint" on page 60

iswpunct

Tests for punctuation wide characters.

```
#include <wctype.h>
int iswpunct(wchar_t wc);
```

Table 46.10 iswpunct

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as ispunct for wide character type.

True for a punctuation character. A punctuation character is neither a control nor an alphanumeric character.

This function may not be implemented on all platforms.

See Also

"ispunct" on page 60

iswspace

Tests for whitespace wide characters.

```
#include <wctype.h>
int iswspace(wchar_t wc);
```

Table 46.11 iswspace

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isspace for wide character type.

In the "C" locale true is returned for a space, tab, return, new line, vertical tab, or form feed.

This function may not be implemented on all platforms.

See Also

"isspace" on page 61

iswupper

Tests for uppercase wide characters.

```
#include <wctype.h>
int iswupper(wchar_t wc);
```

Table 46.12 iswupper

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isupper for wide character type.

In the "C" locale true is returned for an uppercase letter: [A-Z].

This function may not be implemented on all platforms.

See Also

"isupper" on page 61

"iswlower" on page 619

iswxdigit

Tests for a hexadecimal wide character type.

```
#include <wctype.h>
int iswxdigit(wchar_t wc);
```

Table 46.13 iswxdigit

wc	wchar_t	The wide character to test

Remarks

Provides the same functionality as isxdigit for wide character type.

True for a hexadecimal digit [0-9], [A-F], or [a-f].

This function may not be implemented on all platforms.

See Also

"isxdigit" on page 62

towctrans

Maps a wide character type to another wide character type.

```
#include <wchar.h>
wint_t towctrans(wint_t c, wctrans_t value);
```

Table 46.14 towctrans

c	wint_t	The character to remap
value	wctrans_t	A value retuned by wctrans

Remarks

Maps the first argument to an upper or lower value as specified by value.

Returns the remapped character.

This function may not be implemented on all platforms.

See Also

"wctrans" on page 624

towlower

Converts wide characters from upper to lowercase.

```
#include <wctype.h>
wchar_t towlower(wchar_t wc);
```

Table 46.15 towlower

wc	wchar_t	The wide character to convert

Remarks

Provides the same functionality as tolower for wide character type.

The lowercase equivalent of a uppercase letter and returns all other characters unchanged

This function may not be implemented on all platforms.

See Also

"tolower" on page 62

"towupper" on page 624

towupper

Converts wide characters from lower to uppercase.

```
#include <wctype.h>
wchar_t towupper(wchar_t wc);
```

Table 46.16 towupper

wc	wchar_t	The wide character to convert

Remarks

Provides the same functionality as toupper for wide character type.

The uppercase equivalent of a lowercase letter and returns all other characters unchanged.

This function may not be implemented on all platforms.

See Also

"toupper" on page 63

"towlower" on page 623

wctrans

Constructs a property value for `"toupper"` and `"tolower"` for character remapping.

```
#include <wchar.h>
wctrans_t wctrans(const char *name);
```

Table 46.17 wctrans

name	const char *	toupper or tolower property

Remarks

Constructs a value that represents a mapping between wide characters. The value of name can be either `toupper` or `tolower`.

A `wctrans_t` type

This function may not be implemented on all platforms.

See Also

"towctrans" on page 623

47

WinSIOUX.h

The SIOUX and WinSIOUX (Simple Input and Output User eXchange) libraries handle Graphical User Interface issues. Such items as menus, windows, and events are handled so your program doesn't need to for C, Pascal and C++ programs.

Overview of WinSIOUX

The following section describes the Windows versions of the console emulation interface known as WinSIOUX. The facilities and structure members for the Windows Standard Input Output User eXchange console interface are as follows:

- "Using WinSIOUX" on page 627 explains SIOUX properties.
- "WinSIOUX for Windows" on page 628 explains the SIOUX library for Windows 95 and Windows NT.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

See Also

"MSL Extras Library Headers" on page 4, for information on `POSIX` naming conventions.

Using WinSIOUX

Sometimes you need to port a program that was originally written for a command line interface such as DOS or UNIX. Or you need to write a new program quickly and don't have the time to write a complete Graphical User Interface that handles windows, menus, and events.

To help you, CodeWarrior provides you with the WinSIOUX libraries, which handles all the Graphical User Interface items such as menus, windows, and titles so your program doesn't need to. It creates a window that's much like a dumb terminal or TTY but with scrolling. You can write to it and read from it with the standard C functions and C++ operators, such as printf(), scanf(), getchar(), putchar() and the C++ inserter and extractor operators << and >>. The SIOUX and WinSIOUX libraries also creates a File menu that lets you save and print the contents of the window. There is also an Edit menu that lets you cut, copy, and paste the contents in the window.

This function may not be implemented on all platforms.

See Also

"Overview of console.h" on page 41.

NOTE If you're porting a UNIX or DOS program, you might also need the functions in other UNIX compatibility headers.

WinSIOUX for Windows

The WinSIOUX window is a re-sizable, scrolling text window, where your program reads and writes text.

With the commands from the Edit menu, you can cut and copy text from the WinSIOUX window and paste text from other applications into the WinSIOUX window. With the commands in the File menu, you can print or save the contents of the WinSIOUX window

- "Creating a Project with WinSIOUX" on page 628 basic steps to create a WinSIOUX program.
- "Customizing WinSIOUX" on page 629 settings used to create the WinSIOUX console
- "clrscr" on page 630 is used to clear the WinSIOUX console screen and buffer

Creating a Project with WinSIOUX

To use the WinSIOUX library, create a project from a project stationery that creates a WinSIOUX Console style project.

A Win SIOUX project must contain at least these libraries:

- ANSIC_WINSIOUX.LIB
- ANSIC_WINSIOUXD.LIB
- Mwcrt.lib
- MWCRTD.lib

The Win32SDK libraries:

- Winspool.lib
- Comdlg32.lib
- Gdi32.lib
- Kernel32.lib
- User32.lib

And the resource file:

WinSIOUX.rc

Customizing WinSIOUX

WinSIOUX offers the user a limited ability to customize the WinSIOUX window display. The following sections describe how you achieve this customization by modifying the structure SIOUXSettings, of type tSIOUXSettings. WinSIOUX examines some of the data fields of SIOUXSettings to determine how to create the WinSIOUX window and environment.

NOTE To customize WinSIOUX, you must modify SIOUXSettings before you call any function that uses standard input or output. If you modify SIOUXSettings afterwards, WinSIOUX does not change its window.

Table 47.1 The SIOUX Settings Structure

This field…		Specifies…
char	initializeTB	Not applicable to WinSIOUX.
char	standalone	Not applicable to WinSIOUX.
char	setupmenus	Not applicable to WinSIOUX.
char	autocloseonquit	Whether to close the window and quit the application automatically when the program has completed.
char	asktosaveonclose	Query the user whether to save the WinSIOUX output as a file, when the program is done.
char	showstatusline	Not applicable to WinSIOUX.

Table 47.1 The SIOUX Settings Structure (*continued*)

This field…		Specifies…
short	tabspaces	If greater than zero, substitute a tab with that number of spaces. If zero, print the tabs.
short	column	The number of characters per line that the SIOUX window will contain.
short	rows	The number of lines of text that the SIOUX window will contain.
short	toppixel	Not applicable to WinSIOUX.
short	leftpixel	Not applicable to WinSIOUX.
short	fontname[32]	The font to be used in the WinSIOUX window.
short	fontsize	The size of the font to be used in the WinSIOUX window.
short	fontface	Not applicable to WinSIOUX.

clrscr

Clears the WinSIOUX window and flushes the buffers.

```
#include <WinSIOUX.h>
void clrscr(void);
```

Remarks

This function simply calls the function WinSIOUXclrscr, that clears the screen.

This function is not implemented directly on Windows console.

Windows compatible, only.

48

MSL Flags

This appendix contains a description of the macros and defines that are used as switches or flags in the MSL C library.

Overview of the MSL Switches, Flags and Defines

The MSL C library has various flags that may be set to customize the library to users specifications. The flags are explained as follows:

__ANSI_OVERLOAD__

When defined (and when using the C++ compiler) the math functions are overloaded with float and long double versions as specified by 26.5 paragraph 6 of the C++ standard. Disabling this flag removed the overloaded functions.

_MSL_C_LOCALE_ONLY

The flag _MSL_C_LOCALE_ONLY provides for disabling the locale mechanism in the library.

The MSL C library should be recompiled after changing the value of the flag.

When off, the locale mechanism works as described by the ANSI C standard. When on, there are no locales, and anything which is dependent upon locales will function as if the “C” locale is in place. This saves code size and increases execution speed slightly.

_MSL_IMP_EXP

This macro determines how the standard headers are decorated for importing and exporting symbols from/to a shared version of the standard runtime/C/C++ library.

If this macro is defined to nothing then the headers are configured for linking against static libraries.

In the header file `UseDLLPrefix.h` the macro is defined to `__declspec(dllimport)`. This will allow you to link against one of the supplied shared libraries. Other related macros

`_MSL_IMP_EXP_C`

`_MSL_IMP_EXP_SIOUX`

`_MSL_IMP_EXP_RUNTIME`

allow you to link with “part” of the shared library and an individual static library.

By default the previous macros are set to whatever _MSLIMP_EXP is set to (see ansi_parms.h). As an example, if you wish to link in the static version of the SIOUX library then you would edit `ansi_parms.h` and define `_MSL_IMP_EXP_SIOUX` to nothing and link in the appropriate static `SIOUX lib`.

_MSL_INTEGRAL_MATH

When defined (and when `__ANSI_OVERLOAD__` is defined and when using the C++ compiler), additional overloads to the math functions with integral arguments are created. This flag is meant to prevent compile time errors for statements like:

```
double c = cos(0);
// ambiguous with __ANSI_OVERLOAD__
```

```
// and not _MSL_INTEGRAL_MATH
```

_MSL_MALLOC_0_RETURNS_NON_NULL

This flag determines the implementation of a null allocated malloc statement. If not defined `malloc(0)` returns 0. If defined `malloc(0)` returns non-zero.

The C lib must be recompiled when flipping this switch.

_MSL_NEEDS_EXTRAS

Macintosh and Windows programs have the ability to use extra C functions when they have the MSL Extras Library linked into their project. This flag determines if they can be accessed from a C standard header or not. The flag `_MSL_NEEDS_EXTRAS` default setting is `true` for Windows and `false` for the Macintosh.

Macintosh developers must specify a non standard header to access the non standard extra functions. If Macintosh programmers wishes to access non standard functions via a standard header in legacy code, they need to do is to set `_MSL_NEEDS_EXTRAS` to `true` in `ansi_prefix.mac.h`

Windows users can access the functions in a standard header to be legacy compatible as well as the actual header where they are declared. If a Windows programmer does not want to access non standard functions via a standard header then they need to turn off `_MSL_NEEDS_EXTRAS` in `ansi_prefix.Win32.h`.

_MSL_OS_DIRECT_MALLOC

If defined `malloc` will call straight through to the OS without forming memory pools. This will likely degrade performance but may be valuable for debugging purposes.

The C lib must be recompiled when flipping this switch.

_MSL_CLASSIC_MALLOC

Enables the version of malloc that was in Pro 4 (for backward compatibility).

The C lib must be recompiled when flipping this switch.

_MSL_USE_NEW_FILE_APIS

You can use this flags to turn on and off the new-style use of the new HFS+ file system APIs, present starting with Mac OS 9.0. Using the new file system APIs gets filenames greater than 32 characters and file sizes of greater than 2GB. However, due to the standard interfaces for C, you can only access files in 2GB chunks.

Prototypes for `fread()`, for example, take a `long` as parameter for the amount of data to be read from a file, so to access more than `2GB` in a file, you must make successive calls to `fread()`.

The C lib must be recompiled when flipping this switch.

_MSL_USE_OLD_FILE_APIS

You can use this flags to turn on and off the old-style use of the Mac file system `APIs`. Using the old file system APIs limits you to filenames of 32 characters or less and file sizes of 2GB or less.

The C lib must be recompiled when flipping this switch.

_MSL_POSIX

This flag adds the `POSIX` function `stat` to the global namespace. This is on by default.

Turning this off should allow the user to link against a third party `POSIX` library with the same names. The `POSIX` names preceded by a leading underscore are always available in MSL. For example, `_open` and `_stat`.

Remarks

This macro is located in the prefix files `ansi_prefix.win32.h` or `ansi_prefix.macos.h`.

Commenting this out does not require recompilation of the library.

_MSL_STRERROR_KNOWS_ERROR_NAMES

The flag _MSL_STRERROR_KNOWS_ERROR_NAMES controls what happens when the strerror() call is made.

When the flag is on, strerror() will return robust error messages. When the flag is off, strerror() will always return with an “unknown” error. This provides for a huge savings in code/data size in the executable.

__SET_ERRNO__

If this flag is defined it will cause the standard math functions to set the global errno to either `EDOM` or `ERANGE` as specified in the 1989 C standard. The modern C standard specifies that setting errno for the mathlib is optional. So in the spirit of the new standard on x86 it is now optional.

Turning this off will improve performance by reducing checks for incorrect input values but will not set errno. Most of the standard math functions are now in the header `math_x87.h`.

Remarks

Since these definitions are in a header file they can be configured when building your application and therefore you do NOT need to rebuild the library. This switch is on by default.

This flag is only for Win32 x86 libraries.

49

Secure Library Functions

This chapter contains a description of the secure library functions that are an extension to the C99 Standard. These functions promote safer, more secure programming. The functions verify that output buffers are large enough for intended results and return a failure indicator if not. Data is never written past the end of an array and all string results are null terminated.

The secure library functions are only accessed by their respective headers if the `__USE_SECURE_LIB__` macro is defined in the source file where the header file is included. If a header file is included more than once in a given scope, this will result in undefined behavior if `__USE_SECURE_LIB__` is defined for some inclusions and not others.

Input/Output

If the `__USE_SECURE_LIB__` macro is defined in the source file where <stdio.h> is included, then stdio.h defines the following macros.

`L_tmpnam_s`

This macro expands to an integer constant expression that is the size needed for an array of char large enough to hold a temporary file name string generated by the `tmpnam_s` function.

`TMP_MAX_S`

This macro expands to an integer constant expression that is the maximum number of unique file names that can be generated by the `tmpnam_s` function.

File Operations

tmpnam_s

The `tmpnam_s` function generates a unique string that represents a valid file name. File names created by the `tmpnam_s` function are temporary only in the sense that their names should not collide with those generated by conventional naming rules. It is still

necessary to use the `remove` function to remove such files when their use is ended, and before program termination.

The function is capable of generating `TMP_MAX_S` different strings. Any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings shall be less than the value of the `L_tmpnam_s` macro. The `tmpnam_s` function generates a different string each time it is called.

```
#define __USE_SECURE_LIB__

#include <stdio.h>

int tmpnam_s(char *s, size_t maxsize);
```

Remarks

If no suitable string is generated, or if the length of the string is not less than the value of maxsize, the `tmpnam_s` function writes a null character to s[0] (if maxsize is greater than zero) and returns `ERANGE`.

Otherwise, the `tmpnam_s` function writes the string in the array pointed to by `s` and returns zero.

The value of the `TMP_MAX_S` macro should be at least 25.

Formatted input/output functions

fscanf_s

The `fscanf_s` function is equivalent to `fscanf` except that the `c`, `s`, and `[` conversion specifiers apply to a pair of arguments, unless assignment is suppressed by an asterisk (*). The first of these arguments is the same as for `fscanf`. That argument is followed by the second argument in the argument list, which has a type of `size_t` and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.

If the format is known, an implementation may issue a diagnostic for any argument used to store the result from a `c`, `s`, or `[` conversion specifier if that argument is not followed by an argument of type `size_t`. Limited checking may be done if the format is not known at translation time. For example, a diagnostic may be issued for each argument after a format of type `pointer` to one of char, signed char, unsigned char, or void, if not followed by an argument of type `size_t`. The diagnostic could provide a warning that unless the pointer is being used with a conversion specifier using the `hh` length modifier, a length argument must follow the pointer argument. Another useful diagnostic could flag any non-pointer argument that follows a format that was not a `size_t` type.

A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

```
#define __USE_SECURE_LIB__

#include <stdio.h>

int fscanf_s(FILE * restrict stream, const char * restrict
     format, ...);
```

Remarks

The `fscanf_s` function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the `fscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or zero if an early match failure occurs.

Example

```
#define __USE_SECURE_LIB__

#include <stdio.h>

/* ... */

int n; char s[5];

n = fscanf_s(stdin, "%s", s, sizeof s);
```

Providing hello as the input line will assign 0 to `n` since a matching failure will occur. The end of line character attached to hello (hello\0) requires that the total array size can hold six characters. No assignment to `s` occurs.

scanf_s

The `scanf_s` function is equivalent to `fscanf_s` with the stdin argument inserted before the arguments to `scanf_s`.

```
#define __USE_SECURE_LIB__

#include <stdio.h>

int scanf_s(const char * restrict format, ...);
```

Remarks

The `scanf_s` function returns the value of the EOF macro if an input failure occurs before conversion. Otherwise, the `scanf_s` function returns the number of input items assigned, which can be fewer than provided for or zero if an early match failure occurs.

sscanf_s

The sscanf_s function is equivalent to fscanf_s, except that input is obtained from a string rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the fscanf_s function. If copying takes place between objects that overlap, the behavior is undefined.

```
#define __USE_SECURE_LIB__

#include <stdio.h>

int sscanf_s(const char * restrict s, const char * restrict
     format, ...);
```

Remarks

The sscanf_s function returns the value of the EOF macro if an input failure occurs before conversion. Otherwise, the number of input items assigned are returned, which can be fewer than provided for, or zero if an early match failure occurs.

vfscanf_s

The vfscanf_s function is equivalent to fscanf_s, although the variable argument list is replaced by arg, which is initialized by the va_start macro (and possibly subsequent va_arg calls). The vfscanf_s function does not invoke the va_end macro.

Since the functions vfscanf_s, vscanf_s, and vsscanf_s invoke the va_arg macro, the value of arg will vary after the return.

```
#define __USE_SECURE_LIB__

#include <stdarg.h>

#include <stdio.h>

int vfscanf_s(FILE * restrict stream, const char * restrict
     format, va_list arg);
```

Remarks

The vfscanf_s function returns the value of the EOF macro if an input failure occurs before conversion. Otherwise, it returns the number of input items assigned, which can be fewer than provided for, or zero if an early match failure occurs.

vscanf_s

The vscanf_s function is equivalent to scanf_s, although the variable argument list is replaced by arg, which is initialized by the va_start macro (and possibly subsequent va_arg calls). The vscanf_s function does not invoke the va_end macro.

Since the functions vfscanf_s, vscanf_s, and vsscanf_s invoke the va_arg macro, the value of arg will vary after the return.

```
#define __USE_SECURE_LIB__

#include <stdarg.h>

#include <stdio.h>

int vscanf_s(const char * restrict format, va_list arg);
```

Remarks

The vscanf_s function returns the value of the EOF macro if an input failure occurs before conversion. Otherwise, the vscanf_s function returns the number of input items assigned, which can be fewer than provided for, or zero if an early match failure occurs.

vsscanf_s

The vsscanf_s function is equivalent to sscanf_s, although the variable argument list is replaced by arg, which is initialized by the va_start macro (and possibly subsequent va_arg calls). The vsscanf_s function does not invoke the va_end macro.

Since the functions vfscanf_s, vscanf_s, and vsscanf_s invoke the va_arg macro, the value of arg will vary after the return.

```
#define __USE_SECURE_LIB__

#include <stdarg.h>

#include <stdio.h>

int vsscanf_s(const char * restrict s, const char * restrict
     format, va_list arg);
```

Remarks

The vsscanf_s function returns the value of the EOF macro if an input failure occurs before conversion. Otherwise, it returns the number of input items assigned, which can be fewer than provided for, or zero if an early match failure occurs.

Character input/output functions

gets_s

Index

Symbols

Numerics

A

B

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W