EWL C Reference Manual

Document Number: CWEWLCREF
Rev. 10.x, 02/2014

Contents

Section number **Title** **Page**

Chapter 11
locale.h

Chapter 12
limits.h

Chapter 13
maths.h

Chapter 21
stdlib.h

Chapter 22
string.h

Chapter 26
wctype.h

Chapter 27
EWL Extra Library

Chapter 1
Introduction

The CodeWarrior C and C++ compilers use the Embedded Warrior Library (EWL) for C to provide and extend the libraries documented in the ISO/IEC standards for C. EWL C also offers facilities described in the POSIX specification and some common application programming iwnterfaces (APIs) from UNIX operating systems.

1.1 ISO/IEC Standards

The Embedded Warrior Library for C conforms to the library described in the ISO/IEC9899:1999 ("C99") standard. The EWL also conforms to the previous standard, ISO/IEC9899-1990 ("C90"). EWL uses a macro, EWL_ C99, to separate C90 features from C99features. If EWL_ C99 is defined to have the value 0 before building the EWL C library, only those parts of the library that were defined in C90 are compiled, yielding a smaller library. If EWL_ C99 is defined to have a non-zero value before building the library, the full EWL C library is compiled to conform to C99.

1.2 Intrinsic Functions

Intrinsic functions generate in-line assembly code instead of making a call to a library function. Intrinsic functions generate less object code and perform faster than their regular counterparts. In some cases these functions generate just a single assembly instruction.

For example, for target processors that have built-in floating-point facilities, the compiler generates a single assembly instruction when it encounters this intrinsic function:

```
long __labs(long);
```

The compiler generates a sequence of instructions that are highly optimized for the target processor when it encounters this intrinsic function:

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

Because an intrinsic function is not a real function, a program cannot use a pointer to an intrinsic function. These statements give an example:

```
#include <math.h>

typedef long (*functype)(long);

functype f1 = labs; /* OK: non-intrinsic function in EWL. */

functype f2 = __labs; /* Error: intrinsic function. */
```

Chapter 2
Configuring EWL

EWL uses the definitions of preprocessor macros to specify the capabilities of the platform on which EWL runs and to choose the policies that EWL uses to manage its resources. EWL provides a prefix file for each build target that EWL runs on. An EWL prefix file is a C header file that contains a list of macro definitions. To compile a customized version of EWL for your build target, copy your build target's prefix file then customize its macro definitions.

For example, to compile a custom version of EWL C for the SockSort build target, find and copy the header file named `ansi_prefix.SockSort.h`. Edit the contents of your copy of this prefix file, then add this customized file to your project to build a new, custom EWL C library. Add this customized library and your prefix file to your projects to use your custom standard library.

2.1 Configuring Memory Management

EWL has a flexible memory management system.

In most cases, the default configuration should be sufficient. If the operating system of the build target has its own memorymanagement, simply complete the `sys_alloc(),sys_free()`, and `sys_pointer_size()` routines in the `pool_alloc_target.c` file, where target represents the build target on which EWL runs.

EWL calls `sys_alloc()` with the size of a desired memory block whenever EWL needs more heap memory from the system to satisfy calls from `_malloc()`. If the request succeeds, the `sys_alloc()` routine must return a pointer to a block of memory of the requested size. EWL calls the `sys_free()` routine to release a block of memory allocated with `sys_alloc()`. An implementation of `sys_free()` should be return the memory block to the operating system.

EWL calls the sys_pointer_size() routine with a pointer to a previously allocated memory block obtained from sys_alloc() when EWL needs to determine the size of the memory block (the value of the size spassed to the sys_alloc() call that obtained the memory).

If the build target does not have an operating system, or the system does not support its own memory management, EWL can still provide the _ malloc() routine by using a block of RAM in the program heap as a memory pool. The EWL_OS_ALLOC_SUPPORT macro must be turned off in the platform prefix file. The EWL_HEAP_EXTERN_PROTOTYPES, EWL_HEAP_START, and __EWL_HEAP_SIZE macros must also be defined in order for EWL to find the heap block to use as a memory pool. Generally, the linker sets aside a block of memory for this pool. But the user program can also set aside a memory pool in the form of a large array.

EWL provides a minimalist memory allocator with low space and time overhead. It coalesces freed blocks when possible providing a linear and small implementation. To use this alternative define _EMBEDDED_WARRIOR_MALLOC. The linker must define the symbols : HEAP_START, SP_INIT, stack_safety (minimal distance between stack and heap, assuming they grow toward each other), mem_limit (maximum address).

EWL provides an alternate memory allocation system for build targets with no operating system. This allocation system may perform better for small heaps of about 4Kb. It also works with arbitrary pointer sizes and alignments. To use this alternative, define EWL_BAREBOARD_MALLOC. If you need alignment for this version of _ malloc() to be different than the value returned by sizeof(void*), then define EWL_ALIGNMENT to whatever multiple of sizeof(void*) you need.

NOTE

The EWL memory management is configured with _EMBEDDED_WARRIOR_MALLOC by default.

The table below lists the macros that configure the EWL memory management system:

Table 2-1. Macros for EWL Memory Management

Macro	Effect
EWL_BAREBOARD_MALLOC	Defined if the alternative small-memory allocator is to be used. Available only for bareboard systems. This allocator allows arbitrary pointer sizes and alignments
_EMBEDDED_WARRIOR_MALLOC	Defined if the minimalistic allocator is to be used. Available only for bareboard systems. This allocator allows arbitrary pointer sizes and alignments
EWL_MALLOC_0_RETURNS_NON_NULL	Defined as 1 if EWL returns a non-NULL value for zero-sized _ malloc() requests. Defined as 0 if EWL returns NULL for zerosized _ malloc() requests.
EWL_OS_DIRECT_MALLOC	Defined as 1 if EWL should make operating system requests for memory every time the user program calls _ malloc(). Defined as 0 if EWL uses its internal memory pools to satisfy _ malloc()requests. Generally, EWL memory pools give

Table continues on the next page...

Table 2-1. Macros for EWL Memory Management (continued)

Macro	Effect
	better performance than most operating system memory allocators. Using EWL_OS_DIRECT_MALLOC can sometimes provide some help for debugging. EWL_OS_ALLOC_SUPPORT must be defined as 1 for EWL_OS_DIRECT_MALLOC to take effect.
EWL_CLASSIC_MALLOC(old name: EWL_PRO4_MALLOC)	Defined if EWL is to use the original EWL memory pool scheme. Undefined if EWL uses its more modern pooling scheme.
EWL_ALLOCATE_SIZE	Defined to the routine name that returns the size of an allocated memory block. Default routine name is allocate_size
EWL_ALLOCATE	Defined to the internal EWL routine name that allocates a memory block. Used only with the modern memory pooling scheme. Default routine name is allocate.
EWL_ALLOCATE_RESIZE	Defined to the internal EWL 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
EWL_ALLOCATE_EXPAND	Defined to the internal EWL 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.
EWL_OS_ALLOC_SUPPORT(old name: No_Alloc_OS_Support)	Defined to 1 if the target operating system provides memory allocation. Defined to 0 if the operating system does not support memory allocation. When defined to 1, the programmer must supply sys_alloc(), sys_free(), and sys_pointer_size() functions in the pool_alloc_target .c file. When defined to 0, the programmer must define the EWL_HEAP_EXTERN_PROTOTYPES, EWL_HEAP_START, and _EWL_HEAP_SIZE macros. There must also be writable space provided at link time for EWL to use as a memory pool.
EWL_HEAP_EXTERN_PROTOTYPES	When EWL_OS_ALLOC_SUPPORT is off, the EWL 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 EWL_HEAP_EXTERN_PROTOTYPES so it expands to appropriate external prototypes.
EWL_POOL_ALIGNMENT	Specifies the alignment requirements of _malloc() and _free() only when using the original allocator. The alignment is a mask used to ensure that blocks allocated always have sizes that are multiples of a given power-of-two. This exponent is 4. The alignment factor must be a multiple of four and must also be a multiple of sizeof(long).
EWL_USE_FIX_MALLOC_POOLS	For tiny allocations, fixed sized pools help significantly speed allocation and deallocation. Used only with the modern memory pooling scheme. You can reserve a pool for a small range of sizes. Disable fixed-size pools by setting EWL_USE_FIX_MALLOC_POOLS to 0. The default value is 1. Use of fixed size pools requires further configuration. With the default configuration, each pool will handle approximately

Table continues on the next page...

Table 2-1. Macros for EWL Memory Management (continued)

Macro	Effect
	4000 bytes worth of requests before asking for more memory. There are 4 pool types. Each type is responsible for a different range of requests: 0-12 bytes, 13-20 bytes, 21-36 bytes, and 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.
EWL_HEAP_EXTERN_PROTOTYPES	When EWL_OS_ALLOC_SUPPORT is off, the EWL 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 EWL_HEAP_EXTERN_PROTOTYPES so it expands to appropriate external prototypes.
EWL_HEAP_START	When EWL_OS_ALLOC_SUPPORT is off, the EWL alloc.c file must be able to find the start of the writable memory pool area. The EWL_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.
_EWL_HEAP_SIZE	When EWL_OS_ALLOC_SUPPORT is off, the EWL alloc.c file must be able to determine the size of the writable memory pool. The _EWL_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 EWL _CALLOC() routine is simply _CALLOC. Otherwise, if _CALLOCis defined, the EWL _CALLOC() routine is named whatever the _CALLOC macro is defined to. This is useful in case a target platform has its own implementation of _CALLOC() and the EWL name conflicts with the target's name.
_FREE	If _ FREE is undefined, the name of the EWL free() routine is simply _ free. Otherwise, if _ FREE is defined, the EWL _free() routine is named whatever the _FREE macro is defined to. This is useful in case a target platform has its own implementation of _ free() and the EWL conflicts with the target's name.
_MALLOC	If _ MALLOC is undefined, the name of the EWL _ malloc() routine is simply _ malloc. Otherwise, if _ MALLOC is defined, the EWL _ malloc() routine is named whatever the _ MALLOC macro is defined to. This is useful in case a target platform has its own implementation of _ malloc() and the EWL name conflicts with the target's name.
_REALLOC	If _ REALLOC is undefined, the name of the EWL _ realloc() routine is simply Kimono. Otherwise, if Kimono is defined, the EWL Kimono routine is named whatever the Kimono macro is defined to. This is useful in case a target platform has its own implementation of Kimono and the EWL name conflicts with the target's name.

2.2 Configuring Time and Date

EWL comes configured by default to take advantage of a build target's features to determine the time of day and return an internal clock tick.

For EWL to provide these facilities, a build target must provide four simple functions. Time and clock stub functions are in the `time_target.c`, where `target` is the name of the build target.

- For build targets that have an internal clock tick, the `get_clock()` function must obtain the current clock tick and return its value to EWL. 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 EWL. Depending on the value of `EWL_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 EWL. If the conversion fails, return 0 to EWL.
- 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 EWL. If the conversion fails, return 0 to EWL. The `to_local_time()` function is only used when `EWL_TIME_T_IS_LOCALTIME` is off.
- 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 table below lists the macros that configure EWL C's time and clock facilities:

Table 2-2. EWL Date and Time Management Macros

Macro	Effect
`Kimono`	Defined to 1 if the EWL platform supports retrieving the time. Defined to 0 if the EWL platform does not support retrieving the time.
`EWL_CLOCK_T_AVAILABLE`	Defined to 1 if the EWL platform supports the `clock_t` type. Defined to 0 if the EWL platform does not support the `clock_t` type. The `EWL_OS_TIME_SUPPORT` macro must be on before the `EWL_CLOCK_T_AVAILABLE` macro is recognized.
`EWL_CLOCK_T_DEFINED`	Defined to 1 if the EWL platform defined the `clock_t` type. Defined to 0 if the EWL platform does not define the `clock_t` type.

Table continues on the next page...

Table 2-2. EWL Date and Time Management Macros (continued)

Macro	Effect
EWL_CLOCK_T	Set to the clock_t type. Default value is unsigned long int.
EWL_TIME_T_AVAILABLE	Defined to 1 if the EWL platform supports the time_t type. Defined to 0 if the EWL platform does not support the time_t type. The EWL_OS_TIME_SUPPORTmacro must be on before the EWL_TIME_T_AVAILABLE macro is recognized.
EWL_TIME_T_DEFINED	Defined to 1 if the EWL platform defined the time_t type. Defined to 0 if the EWL platform does not define the time_t type.
EWL_TIME_T_IS_LOCALTIME	Defined to 1 if the EWL platform value for time_t represents local time, preadjusted for any time zone and offset from UTC (GMT). Defined to 0 if the EWL platform value for time_t represents Universal Time Coordinated. The default value is 1.
EWL_CLOCKS_PER_SEC	Set to the number of clock ticks per second. The default value is 60.
EWL_TM_STRUCT_AVAILABLE	Defined to 1 if the EWL platform supports the tm structure. Defined to 0 if the EWL platform does not support the tm structure.

2.3 Configuring Input and Output

This section describes configuring input and output options.

2.3.1 Configuring File Input/Output

Setting up EWL 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 EWL not know about it by defining EWL_OS_DISK_FILE_SUPPORT to 0.

In that mode, EWL does not know about any routines requiring file manipulation such as fopen(), fread(), fwrite(), fclose(), and so on.

When EWL_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 EWL_FILENAME_MAX properly reflects the maximum length of a file name. Also, if the default internal EWL file buffer size is not appropriate, choose a new value for EWL_BUFSIZ. Once all the macros are properly defined, the following routines in file_io_target.c (where target represents the build target on which EWL will run) must be completed.

2.3.2 Routines

The `open_file()` routine is perhaps the most complicated of all the low level file I/O routines. It takes a filename and some 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 operation 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. This function must 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 call to this function should then store zero in the count argument and a result code of `io_EOF`. If the read operation succeeds, return `no_io_error`. If the read fails, 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 a variable of type unsigned long due to certain internal constraints of EWL. The value should actually be treated as type 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 end-of-file 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 operation is successful, return no_io_error. If the close failed, return io_error.

The temp_file_name() routine creates a new unique file name suitable for a temporary file. Function tmpnam() uses this function to perform the low-level work.

The delete_file() routine deletes an existing file, given its file name. If the delete operation is successful, return no_io_error. If the delete failed, return io_error.

The rename_file() routine renames an existing file, given its existing file name and a desired new file name. If the rename is 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 EWL_WIDE_CHAR is defined as 1, then also define EWL_WFILEIO_AVAILABLE as 1. The following stub routines must also be completed in the file_io_plat.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 table below lists the macros that configure the EWL file I/O system.

Table 2-3. EWL File Input/Output Macros

Macro	Effect
EWL_OS_DISK_FILE_SUPPORT	Defined to 1 if the EWL platform supports disk file I/O. Defined to 0 if the EWL platform does not support disk file I/O. Default value is 0.
EWL_FILENAME_MAX	Set to the maximum number of characters allowed in a file name. The default value is 256.
EWL_BUFSIZ	Set to the file I/O buffer size in bytes used to set the BUFSIZ macro. The default value is 4096.
EWL_WFILEIO_AVAILABLE	Defined to 1 if EWL has wchar_t extensions for file I/O calls needing filenames, such as wfopen(). Defined to 0 if EWL only has traditional C file I/O calls. Default value is 0. It is an error to have EWL_WIDE_CHAR defined as 0 and EWL_WFILEIO_AVAILABLE defined as 1.

2.3.3 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 EWL_CONSOLE_SUPPORT is defined as 0, EWL does not generate source code to construct the stdin, stdout, and stderr streams. Also, calls such as printf() are not placed in the standard C library.

When EWL_CONSOLE_SUPPORT is 1, the console access can be provided in the following ways:

- The first way is to have EWL automatically throw away all items read and written to the console by defining EWL_NULL_CONSOLE_ROUTINES.
- The second way is to have EWL treat all console I/O as if it were file I/O by defining EWL_FILE_CONSOLE_ROUTINES. Treating the console as file I/O requires configuring the file I/O portion of EWL as described in the previous section. Input and output go through the read_file(), write_file(), and close_file() bottlenecks instead of read_console(), write_console(), and close_console(), respectively.
- The third way to provide console access is to define EWL_CONSOLE_SUPPORT as 1 and leave the remainder of the EWL_CONSOLE_FILE_IS_DISK_FILE and EWL_FILE_CONSOLE_ROUTINES flags in their default state, defined as 0. EWL 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_plat.c file.

The macros listed in the table below configure the EWL console input/output capabilities.

Table 2-4. EWL Console Input/Output Management Macros

Macro	Behavior
EWL_CONSOLE_SUPPORT	Defined to 1 if the EWL platform supports console I/O. Defined to 0 if the EWL platform does not support console I/O. Default value is 1.
EWL_BUFFERED_CONSOLE	Defined to 1 if the EWL platform console I/O is buffered. Defined to 0 if the EWL platform console I/O is not buffered. Default value is 1.
EWL_CONSOLE_FILE_IS_DISK_FILE	Defined to 1 if the EWL platform has console I/O, but it is really in a file. Defined to 0 if the EWL platform has traditional console I/O. Default value is 0.
EWL_CONSOLE_FILE_IS_DISK_FILE	Defined to 1 if the EWL platform does not perform console I/O. Defined to 0 if the EWL platform performs console I/O. This flag may be set independently of EWL_CONSOLE_SUPPORT. However, when EWL_CONSOLE_SUPPORT is defined as 0, EWL_NULL_CONSOLE_ROUTINES must always be defined as 1. When EWL_CONSOLE_SUPPORT is 1 and EWL_NULL_CONSOLE_ROUTINES is also 1, all console I/O is ignored. Default value is 0.
EWL_FILE_CONSOLE_ROUTINES	Defined to 1 if the EWL 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 EWL platform uses the special console read/write/close routines.

Table 2-4. EWL Console Input/Output Management Macros

Macro	Behavior
	When EWL_CONSOLE_FILE_IS_DISK_FILE is 1, EWL_FILE_CONSOLE_ROUTINES must always be 1. Default value is 0.

2.4 Configuring Threads

EWL is highly adaptive when it comes to multithreading.

The C standard makes no mention of threading. It even has points where global data is accessed directly, for example errno, asctime(). EWL can be configured to know about multithreaded systems and be completely reentrant. There are essentially three ways to configure the EWL thread support:

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

With EWL_THREADSAFE defined to 1, EWL 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 EWL for a single threaded environment. Operations such as file input/output and memory allocations will be quicker because 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 EWL_THREADSAFE defined to 1, EWL 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 defining EWL_THREADSAFE and EWL_PTHREADS is enough for EWL to fully operate. No other custom code is necessary.

With EWL_THREADSAFE defined to 1 and EWL_PTHREADS defined to 0, 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_platform .c file, and then by completing the four critical region functions in the critical_regions_platform .h header file (where platform represents the target platform which EWL runs on). The compiler runtime library must make a call to init_critical_regions() before calling main().

With EWL_THREADSAFE on, the EWL_LOCALDATA_AVAILABLE flag controls whether or not the EWL library is completely reentrant or not. When EWL_LOCALDATA_AVAILABLE is off, the EWL 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 EWL_LOCALDATA_AVAILABLE is on, the EWL library uses thread local storage to maintain state information. Each thread has its own copy of some dynamic memory that gets used.

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

```
#define _EWL_LOCALDATA(_a) __ewl_GetThreadLocalData()->_a
```

With EWL_LOCALDATA_AVAILABLE on and EWL_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 EWL header (ewl_thread_local_data.h) must be modified to include the platform header (thread_local_data_platform.h) based on its dest_os value. The EWL_LOCALDATA macro is used to access items in thread local storage. So, for example, if the random number seed needs to be obtained, the EWL code will invoke

```
_EWL_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 EWL_PTHREADS even if the underlying platform does not have built-in pthread support. Instead of writing custom code to support the EWL threading model, it may be easier to turn on EWL_PTHREADS and then write comparable pthread routines. When EWL_PTHREADS is on and _EWL_THREADSAFE is on, four pthread routines in the pthread_platform .c file are used by EWL to implement critical regions.

2.4.1 Pthread Functions

The pthread_mutex_init() routine creates a single mutual exclusion lock (mutex). EWL 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 `EWL_LOCALDATA_AVAILABLE` is on, four more pthread routines in the `pthread_platform.c` file are used by EWL 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 following macros, listed in table below are used to configure and use the EWL threading support:

Table 2-5. EWL Thread Management Macros

Macro	Behavior
`EWL_THREADSAFE`	Defined to 0 if there is no multithread support in EWL. Defined to 1 if there should be multithread support in EWL. When defined to 1, many internal aspects of EWL are guarded by critical regions. Having critical regions inside EWL will slow down the execution time for the tradeoff of working correctly on a multithreaded system. Also, many EWL functions will use thread local storage for maintaining state information.
`EWL_PTHREADS`	Defined to 1 if the EWL platform supportsthe POSIX threading model. Defined to 0 if the EWL platform does not support the POSIX threading model. It is an error to define `EWL_PTHREADS` to 1 and `EWL_THREADSAFE` to 0. EWL has generic support for the POSIX thread model, so turning on `EWL_THREADSAFE` and `EWL_THREADS` is enough to properly support a multithreaded system without the need to write any additional support code.
`EWL_LOCALDATA`	Internal EWL flag for accessing thread local data when `EWL_THREADSAFE` is 1. Accesses static global data when `EWL_THREADSAFE`is 0.
`EWL_LOCALDATA_AVAILABLE`	Defined to 1 if the EWL platform supports thread local data, accessible using the `EWL_LOCALDATA` macro. Defined to 0 if the EWL platform does not support thread local data.

2.5 Configuring Assertions

Use the EWL_ASSERT_DISPLAYS_FUNC macro to specify whether or not the assert() facility also reports the name of the function in which an assertion failed.

When compiling a custom version of EWL with EWL_ASSERT_DISPLAYS_FUNC defined as 1, EWL compiles the assert() facility to report the file, line, test expression, and name of the function in which an active assertion fails. When compiling EWL with EWL_ASSERT_DISPLAYS_FUNC defined as 0, the assert() facility only reports the file, line, and test expression.

The EWL_OS_DISK_FILE_SUPPORT and EWL_CONSOLE_SUPPORT macros determine the assert() macro's ability to report an error.

2.6 Configuring Complex Number Facilities

Use the EWL_COMPLEX macro to specify whether or not EWL provides standard facilities for complex number manipulation described by the C99 standard (ISO/IEC 9899-1999).

When compiling a custom version of EWL with EWL_COMPLEX defined as 1 and the EWL_C99 macro is defined as 1 the library provides the facilities for complex number manipulation.

2.7 Configuring C99 Features

Use the EWL_C99 macro to specify whether or not EWL provides a C library that is compliant with the C90 (ISO/IEC 9899-1990) standard or the C99 (ISO/IEC 9899-1999) standard.

When compiling a custom version of EWL with EWL_C99 defined as 0, the library conforms to the C90 specification. If EWL_C99 is defined as 1 the library conforms to the C99 specification.

2.8 Configuring Locale Features

Use the EWL_C_LOCALE_ONLY macro to specify whether or not EWL provides only the "C" locale or the entire locale mechanism specified in the C99 (ISO/IEC 9899-1999) standard.

When compiling a custom version of EWL with EWL_C_LOCALE_ONLY defined as 1, the library provides only the "C" locale. This configuration also reduces the sizes of the library's object code. If EWL_C_LOCALE_ONLY is defined as 0 the library conforms to the locale facilities described in the C99 specification.

When EWL_C_LOCALE_ONLY is defined as 0, the library uses the EWL_DEFAULT_LOCALE macro to determine which locale is in effect when target program starts. When compiling a custom version of EWL with EWL_DEFAULT_LOCALE defined as 0, the library starts with the "" locale. When defined as 1, the library starts with the "C" locale. When 2, the library starts with the "C-UTF-8" locale. The library ignores this option when EWL_C_LOCALE_ONLY is defined as 1.

2.9 Configuring Floating-Point Math Features

Use the EWL_FLOATING_POINT macro to specify whether or not EWL provides floating-point math features.

When compiling a custom version of EWL with EWL_FLOATING_POINT defined as 0, the library does not provide facilities for floating point operations. This setting also reduces the size of the library's object code.

If EWL_FLOATING_POINT is defined as 1 then the library provides floating point facilities.

2.10 Configuring the EWL Extras Library

Use the EWL_NEEDS_EXTRAS macro to specify whether or not EWL should extend its standard behavior with the facilities that EWL Extras provides.

When building a custom version of EWL with EWL_NEEDS_EXTRAS defined as 1, EWL extends its standard library with the features that EWL Extras provides. When using EWL, EWL Extras features can then be accessed through standard header files.

The table below lists the standard header files that include EWL Extras facilities when EWL_NEEDS_EXTRAS is defined as 1.

When building EWL with EWL_NEEDS_EXTRAS defined as 0, EWL does not extend its standard library. EWL Extras facilities must be accessed directly through the EWL Extras non-standard header files.

Table 2-6. Extending Standard Header Files with EWL Extras

This standard header file	includes this EWL Extras header file
stdlib.h	extras_stdlib.h
string.h	extras_string.h
time.h	extras_time.h
wchar.h	extras_wchar.h

2.11 Configuring Wide-Character Facilities

Use the EWL_WIDE_CHAR macro to specify whether or not EWL provides wide character facilities.

When compiling a custom version of EWL with EWL_WIDE_CHAR defined as 1 the library provides facilities for wide-character manipulation.

When compiling the library with EWL_WIDE_CHAR is defined as 0, the library does not provide wide-character facilities. This setting also reduces the size of the library's object code.

2.12 Porting EWL to an Embedded OS

EWL needs to be modified to support an embedded OS. Use the following steps to port EWL to an embedded OS.

1. Create a copy of EWL_C\PPC_EABI\Project \EWL_C.PPCEABI.stub.mcp and EWL_C\PPC_EABI \Include \PREFIX_EPPC_STUB.h. Rename the files to reflect the name of the OS. For example, if your OS is called MyOS then rename the files to EWL_C.PPCEABI.MyOS.mcp and PREFIX_EPPC_MyOS.h. Note that it is a good practice to retain the copy of the original files.
2. Select a particular target among the many targets present in the project.

- In the Target Settings Preference panel, rename the Target Name to reflect the name of the OS. For example if you select the target `EWL_C.PPCEABI.stub.S.UC` then rename the selected target to `EWL_C.PPCEABI.MyOS.S.UC`.
- In the EPPC Target Preference panel, rename the File Name to reflect the name of the OS. For example, if you select the target `EWL_C.PPCEABI.stub.S.UC` then rename the file name to EWL_C.PPCEABI.MyOS.S.UC.a

3. In the C/C++ Language Preference panel of `EWL_C.PPCEABI.MyOS.mcp`, replace the prefix file with `PREFIX_EPPC_MyOS.h`.
4. Replace the definition for `PREFIX_EPPC_STUB` in `PREFIX_EPPC_MyOS.h` with a definition to represent your OS, such as `PREFIX_EPPC_MyOS`.
5. Read your OS documentation and determine whether it has support for providing time, memory allocation, and disk files.
6. Adjust the following macros in `PREFIX_EPPC_MyOS.h` depending on the services available in your OS.
   - `EWL_OS_DISK_FILE_SUPPORT`
   - `EWL_OS_ALLOC_SUPPORT`
   - `EWL_OS_TIME_SUPPORT`
   - `EWL_CLOCK_T_AVAILABLE`
   - `EWL_TIME_T_AVAILABLE`

   The prefix file `PREFIX_EPPC_MyOS.h` includes the header file `ansi_prefix.PPCEABI.bare.h` and the following services:

   - `EWL_CONSOLE_SUPPORT`
   - `EWL_BUFFERED_CONSOLE`

   Determine if you want to send any information on the console during development. If you do not want to send debugging information to a console window on the host machine, reset `_EWL_CONSOLE_SUPPORT` to 0. If you are using either `SMC1_UART_PPCE_24.a` or `SMC2_UART_PPCE_24.a` to read from or write to the console window, you need to set `EWL_BUFFERED_CONSOLE` to 1.
7. Duplicate and rename EWL files based on OS-specific features . The files are `time.stub.c`, `pool_alloc.stub.c` and `file_io.stub.c`. When you rename these files, replace *stub* with *MyOS*.
8. Insert OS system calls into the stub routines of these files to fetch the relevant information.
9. Remove the stub files from `EWL C.PPCEABI.MyOS.mcp` and insert the OS specific files, only if support is provided.
10. In the `pool_alloc_MyOS.c` file replace `#ifdef_No_Alloc_OS_Support` with `#if ! _EWL_OS_ALLOC_SUPPORT`.
11. In the prefix file `PREFIX_EPPC_MyOS.h`, define a macro to represent your OS, such as `MyOS_PPC_EABI`.

12. In `EWL_C.PPCEABI.MyOS.mcp` search for `Generic_PPC_EABI_OS`. At all instances, add an `#elif_MyOS_PPC_EABI` and enter your OS-specific information.
13. Modify `ExitProcess` in `__ppc_eabi_init.c [pp]` to safely return your application to the OS. `__ppc_eabi_init.c[pp]` is found in the *PowerPC EABI Support* directory.
14. You might need to modify the `critical_regions.ppc_eabi.c` and `sysenv.c` files.
15. Review the other EWL components and identify `__va_arg`, `__mem`, `runtime` and the OS-specific modifications to be made to them.

Chapter 3
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.

3.1 Macros in assert.h

The assert.h header files provides following macros.

3.1.1 assert()

Stops execution if a test fails.

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

Parameter

`expression`

The result of a boolean expression.

Remarks

If expression is false the `assert()` macro outputs a diagnostic message and calls `abort()`.

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

Because calls to `assert()` may be disabled, make sure that your program does not depend on the effects of computing the value of expression.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

Listing: Example of assert()

```
/* Make sure that assert() is enabled. */
#undef NDEBUG
#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;
}
```

Chapter 4
complex.h

Facilities for mathematical functions that operate on complex numbers. This header file defines some convenient macros for manipulating complex values. The table below lists these macros.

Table 4-1. Definitions in Complex.h

Macro	Action
`complex`	Define variables of complex type. Expands to the built-in type, `Complex`.
`Complex_I`	Expands to the square root of -1, of type `const float_Complex`.
`imaginary`	Define variables of imaginary type. Expands to the built-in type, `Imaginary`.
`Imaginary_I`	Expands to the square root of -1, of type `const float_Imaginary`.
`I`	Expands to `Imaginary_I`.

The facilities in this header file are only available when the compiler is configured to compile source code for C99 (ISO/IEC 9899:1999). Refer to the Build Tools Reference for information on compiling C99 source code.

4.1 Hyperbolic Trignomatry

Compute hyperbolic trigonometric values.

4.1.1 cacos()

Computes the arc value of cosine.

```
#include <math.h>
double complex cacos(double complex x);
float complex cacosf(float complex x);
long double complex cacosl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the arccosine of the argument x, in radians. If x is out of range then these functions set `errno` to `EDOM` and

```
fpclassify(cacos(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.2 cacosh()

Computes the inverse hyperbolic cosine.

```
#include <math.h>
double complex cacosh(double complex x);
float complex cacoshf(float complex x);
long double complex cacoshl(long double complex x);
```

Parameter

`x`

A complex value from which to compute the inverse hyperbolic cosine.

Remarks

These functions return the non-negative inverse hyperbolic cosine of x. If x is out of range then these functions set `errno` to `EDOM` and

```
fpclassify(cacosh(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.3 casin()

Computes the arc value of sine.

```
#include <math.h>
double complex casin(double complex x);
float complex casinf(float complex x);
long double complex casinl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the arcsine of the argument x, in radians. If x is out of range then these functions set `errno` to `EDOM` and

```
fpclassify(casin(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.4 casinh()

Computes the arc value of hyperbolic sine.

```
#include <math.h>
double complex casinh(double complex x);
float complex casinhf(float complex x);
long double complex casinhl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the hyperbolic arcsine of the argumentargument x. If result of the computation is out of range, these functions set `errno` to `EDOM` and

```
fpclassify(casinh(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.5 catan()

Computes the arc value of tangent.

```
#include <math.h>
  double complex catan(double complex x);
   float complex catanf(float complex x);
    long double complex catanl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the arctangent of the argument x, in radians.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.6 catanh()

Computes the arc value of hyperbolic tangent.

```
#include <math.h>
  double complex catanh(double complex x);
  float complex catanhf(float complex x);
  long double complex catanhl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the hyperbolic arctangent of the argument x, in radians. If x is out of range then these functions set `errno` to `EDOM` and

```
fpclassify(catanh(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.7 ccos()

Computes the cosine.

```
#include <math.h>
double complex ccos(double complex x);
float complex ccosf(float complex x);
long double complex ccosl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the cosine of x, in radians.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.8 ccosh()

Computes the hyperbolic cosine.

```
#include <math.h>
  double complex ccosh(double complex x);
 float complex ccoshf(float complex x);
 long double complex ccoshl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the hyperbolic cosine of x, in radians.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.9 csin()

Computes the sine.

```
#include <math.h>
  double complex csin(double complex x);
float complex csinf(float complex x);
 long double complex csinl(long double complex x);
```

Parameter

x

A complex value.

Remarks

These functions return the sine of the argument x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.10 csinh()

Computes the arc value of hyperbolic tangent.

```
#include <math.h>

  double complex csinh(double complex x);

float complex csinhf(float complex x);

long double complex csinhl(long double complex x);
```

Parameter

x

A complex value.

Remarks

These functions return the hyperbolic sine of the argument x. These functions may assign `ERANGE` to `errno` if x is out of range. Use `fpclassify()` to check the validity of the results returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.1.11 ctan()

Computes the tangent.

```
#include <math.h>

 double complex ctan(double complex x);

float complex ctanf(float complex x);

long double complex ctanl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return the tangent of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.2 Exponent and Logarithms

Compute exponential and logarithmic values.

4.2.1 cexp()

Computes a power of e.

```
#include <math.h>

double complex cexp(double complex x);

float complex cexpf(float complex x);

long double complex cexpl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return e x, where e is the natural logarithm base value.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.2.2 clog()

Computes natural logarithms.

```
#include <math.h>
 double complex clog(double complex x);
  float complex clogf(float complex x);
 long double complex clogl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

These functions return `log` e x. If x < 0 , `clog()` assigns EDOM to errno. Use `fpclassify()` to check the validity of the result returned by `clog()`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.3 Power and Absolute Values

Compute powers, roots, and absolute values.

4.3.1 cabs()

Computes an absolute value.

```
#include <math.h>
 double complex cabs(double complex x);
 float complex cabsf(float complex x);
 long double complex cabsl(long double complex x);
```

Parameter

x

A complex value.

Remarks

These functions return the absolute value of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.3.2 cpow()

Computes the power of a base number.

```
#include <math.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
```

Parameter

x

A complex value to use as base.

y

A complex value to use as exponent.

Remarks

These functions computes xy.

These functions assign EDOM to errno if they cannot compute a value. Use fpclassify() to check the validity of the results returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.3.3 csqrt()

Computes the square root.

```
#include <math.h>
double complex csqrt(double complex x);
float complex csqrtf(float complex x);
long double complex csqrtl(long double complex x);
```

Parameter

x

A complex value.

Remarks

These functions return the square root of x. If x < 0 , csqrt() assigns EDOM to errno. Use fpclassify() to check the validity of the result returned by clog().

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.4 Manipulation

Change and retrieve the properties of complex values.

4.4.1 carg()

Computes the phase angle.

```
#include <math.h>
double carg(double complex x);
float cargf(float complex x);
long double cargl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.4.2 cimag()

Computes the imaginary part of a complex number.

```
#include <math.h>
  double cimag(double complex x);
  float cimag(float complex x);
  long double cimagl(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.4.3 conj()

Computes the complex conjugate of a complex number.

```
#include <math.h>
  double complex conj(double complex x);
  float complex conjl(float complex x);
  long double complex conjl(long double complex x);
```

Parameter

x

A complex value.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.4.4 cproj()

Computes the projection onto the Riemann sphere.

```
#include <math.h>
double complex cproj(double complex x);
float complex cprojl(float complex x);
long double complex cprojl(long double complex x);
```

Parameter

x

A complex value.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

4.4.5 creal()

Computes the real part of a complex number.

```
#include <math.h>
double creal(double complex x);
float crealf(float complex x);
long double creall(long double complex x);
```

Parameter

`x`

A complex value.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Chapter 5
ctype.h

Macros for testing the kind of character and for converting alphabetic characters to uppercase or lowercase.

5.1 Macros in ctype.h

This section lists macros available in ctype.h header files.

5.1.1 isalnum(), isalpha(), isblank(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit()

Tests for membership in subsets of the character set.

```
#include <ctype.h>
 int isalnum(int c);
 int isalpha(int c);
 int isblank(int c);
 int iscntrl(int c);
 int isdigit(int c);
 int isgraph(int c);
 int islower(int c);
 int isprint(int c);
 int ispunct(int c);
 int isspace(int c);
 int isupper(int c);
 int isxdigit(int c);
```

Parameter

`c`

A character value to test.

Remarks

These functions test ASCII characters (`0x00` to `0x7F`) and the `EOF` value. The results of these functions are not defined for character values in the range `0x80` to `0xFF`.

The c argument is of type `int` so that the `EOF` value, which is outside the range of the `char` type, may also be tested.

The table below lists the behaviors of these functions.

Table 5-1. Character Testing Functions

Function	returns true if c is one of these values.
`isalnum(c)`	Alphanumeric characters. For the `"C"`locale, "a" to "z" , "A" to "Z" , or "0" to "9".
`isalpha(c)`	Alphabetic characters. For the `"C"`locale, "a" to "z" or "A" to "Z".
`isblank(c)`	Word-separating whitespace. For the `"C"`locale, the space and tab characters.
`iscntrl(c)`	Non-printing control characters. The delete character (0x7F) or an ordinary control character from 0x00 to 0x1F.
`isdigit(c)`	Numeric characters.
`isgraph(c)`	Non-space printing character.
`islower(c)`	A lowercase alphabetic character, from"a" to "z".
`isprint(c)`	A printable character, including the space character. The compliment of the subset that `iscntrl()`tests for.
`ispunct(c)`	A punctuation character. A punctuation character is not in the subsets for which `isalnum()`, `isblank()`, or `iscntrl()` return true.
`isspace(c)`	A whitespace character. A space, tab, return, new line, vertical tab, or form feed.
`isupper(c)`	An uppercase alphabetic character, from "A" to "Z".
`isxdigit(c)`	A hexadecimal character. From "0" to "9" , "a" to "f" , or "A" to "F" .

Listing: Character Testing Function Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
char *test = "Fb6# 9,";
isalnum(test[0]) ?
printf("%c is alphanumerical\n", test[0]) :
printf("%c is not alphanumerical\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 space\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 lowercase \n", test[1]) :
printf("%c is not lowercase\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 uppercase \n", test[1]) :
printf("%c is not uppercase\n", test[1]);
isxdigit(test[5]) ?
printf("%c is a hexadecimal digit\n", test[5]) :
printf("%c is not a hexadecimal digit\n", test[5]);
return 0;
}
Output:
F is alphanumerical
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
F is uppercase
9 is a hexadecimal digit
```

5.1.2 tolower, toupper

Converts alphabetical characters to lowercase or uppercase.

```
#include <ctype.h>

int tolower(int c);

 int toupper(int c);
```

Parameter

c

A character value to convert.

Remarks

The `tolower()` function converts an uppercase alphabetic character to its equivalent lowercase character. It returns all other characters unchanged. The `toupper()` function converts a lowercase alphabetic character to its uppercase equivalent. It returns all other characters unchanged.

Listing: Character Conversion Example

```
#include <stdio.h>

#include <ctype.h>

int main(void)

{

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 **
```

Chapter 6
errno.h

This header file provides the global error code which some functions in the library use to store the status of their computations.

6.1 errno()

A global variable for storing error result.

```
#include <errno.h>
extern int errno;
```

Remarks

Many 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 more detailed error information by assigning a value to the global variable `errno`. A value of zero specifies that there is no error. A non-zero value specifies that a function encountered an error.

An EWL function only assigns a value to `errno` when an error occurs. It is the programmer's responsibility to assign 0 to `errno` before calling a function that uses it.

The table below lists the values that EWL functions use to report an error condition.

Table 6-1. EWL Date and Time Management Macros

Error Code	Description
EDOM	A numerical argument is not within the acceptable range of values that a function requires.
EILSEQ	An operation to convert to or from a multibyte character sequence could not be completed.
ERANGE	A result could not be computed because it would be beyond the range of values that can be stored in its data type.

Listing: Example of errno() Usage

```
#include <errno.h>
#include <stdio.h>
#include <extras.h>
int main(void)
{
char *num = "99999999999999999999999999999999";
long result;
errno = 0; /* Assume no error. */
result = strtol( num, 0, 10);
if (errno == 0)
printf("The string as a long is %ld", result);
else
printf("Conversion error\n");
return 0;
}
Output:
Conversion error
```

Chapter 7
fenv.h

This header file declares data types, macros, and functions for querying and modifying the behavior of the floating-point environment. The floating-point environment comprises these aspects:

- Exception flags describe the state of the most recent floating-point operation. This header file offers facilities to set and retrieve the values of these flags.
- Control modes specify to the environment how to modify the behavior of its operations. This header file offers facilities for modifying these control modes.

7.1 Data Types and Pragma for the Floating-Point Environment

The fenv.h header file provides following data types and pragma for the floating-point environment.

7.1.1 fenv_t, fexcept_t

Data types for querying and manipulating the floating-point environment.

```
#include <fenv.h>
 typedef /* ... */ fenv_t;
  typedef /* ... */ fexcept_t;
```

Remarks

The `fenv_t` data type represents the target system's floating point environment.

The `fexcept_t` data type represents the status flags in the environment

7.1.2 FENV_ACCESS

Allows access to the floating-point environment.

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

Remarks

This pragma must be set to `on` to specify to the compiler that the program will use the facilities in the `fenv.h` header file.

Place this directive before external declarations to specify that subsequent object declarations will access the floating-point environment. This placement's effect spans until the next directive or the end of the source file. Place this directive at the beginning of compound statement to specify that the statements within the compound statement will access the environment. This placement's effect lasts until the next directive or the end of the compound statement.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

7.1.3 fegetenv()

Retrieves the current floating-point environment.

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

Parameter

`envp`

Pointer to a floating-point environment object.

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 floatingpoint exceptions raised during an interim calculation.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fegetenv()

```
#include <fenv.h>

#pragma STDC FENV_ACCESS on

int main(void)

{

float x = 0.0, y = 0.0;

fenv_t env1, env2;

feclearexcept(FE_ALL_EXCEPT);

fesetenv(FE_DFL_ENV);

x = x + y; /* Might raise exception. */

fegetenv(&env1);

y = y * x; /* Might raise exception. */

fegetenv(&env2);

return 0;

}
```

7.2 Exceptions for the Floating-Point Environment

Facilities for retrieving and setting the floating-point environment's status flags.

The floating-point environment has exception flags. When a floating-point operation gives an extraordinary result, the environment raises an exception. Raising a floatingpoint exception does not affect a program's execution. Instead, these exceptions act as status flags. The `fenv.h` header file offers facilities to clear, set, and test the state of these exceptions, allowing you to choose how your program manages and reacts to them.

The table below lists the macros that define the exceptions that affect these flags. To specify more than one exception, combine these macros with the bitwise-OR operator (|).

Table 7-1. Floating-point Exceptions

Macro	Status Flag
FE_DIVBYZERO	division by zero
FE_INEXACT	inexact value

Table continues on the next page...

Table 7-1. Floating-point Exceptions (continued)

Macro	Status Flag
FE_INVALID	invalid value
FE_OVERFLOW	overflow
FE_UNDERFLOW	underflow
FE_ALL_EXCEPT	all exceptions combined

7.2.1 feclearexcept()

Clears the specified floating-point environment flags

```
#include <fenv.h>
  void feclearexcept(int e);
```

Parameter

e

Zero or more exceptions to reset.

Remarks

This facility may not be available on configurations of the EWL that run on platformsthat do not have floating-point math capabilities.

Listing: Example of feclearexcept()

```
#include <fenv.h>
#include <stdio.h>
#pragma STDC FENV_ACCESS on
int main(void)
{
double x = 123.0;
double y = 0.0;
double result;
/* Reset flags before starting the calculation. */
feclearexcept(FE_ALL_EXCEPT);
result = x / y; /* Should set the FE_DIVBYZERO flag. */
if (fetestexcept(FE_DIVBYZERO))
```

```
printf("Division by zero.\n");
return 0;
}
Output:
Division by zero.
```

7.2.2 fegetexceptflag()

Stores a representation of the states of the floating-point exception flags.

```
#include <fenv.h>
void fegetexceptflag(fexcept_t *f, int excepts);
```

Parameter

`f`

A pointer to an exception flag variable.

`excepts`

Zero or more exceptions to reset.

Remarks

This function saves the states of floating-point exception flags to memory.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fegetexceptflag()

```
#include <fenv.h>
#include <stdio.h>
#pragma STDC FENV_ACCESS on
int main(void)
{
double x = 123.0;
double y = 0.0;
double result;
fexcept_t flags;
```

```
/* Reset flags before starting the calculation. */

feclearexcept(FE_ALL_EXCEPT);

result = x / y; /* Should set the FE_DIVBYZERO flag. */

fegetexceptflag(&flags, FE_ALL_EXCEPT);

if (flags & FE_DIVBYZERO)

printf("Division by zero.\n");

if (flags & FE_INEXACT)

printf("Inexact value.\n");

return 0;

}

Output:

Division by zero.
```

7.2.3 feraiseexcept()

Sets floating-point environment flags.

```
#include <fenv.h>
void feraiseexcept(int e);
```

Parameter

e

Zero or more exceptions to set.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of feraiseexcept()

```
#include <fenv.h>

#include <stdio.h>

#pragma STDC FENV_ACCESS on

int main(void)

{

double x = 123.0;
```

```
double y = 3.0;

double result;

/* Reset flags before starting the calculation. */

feclearexcept(FE_ALL_EXCEPT);

result = x / y; /* Should not set the FE_DIVBYZERO flag. */

if (fetestexcept(FE_DIVBYZERO | FE_INVALID))

printf("Division by zero and invalid operation.\n");

feraiseexcept(FE_DIVBYZERO);

if (fetestexcept(FE_DIVBYZERO))

printf("Division by zero.\n");

return 0;

}

Output:

Division by zero.
```

7.2.4 fesetexceptflag()

Sets the floating-point environment's flags to the settings in contained a variable of type `fexcept_t`.

```
#include <fenv.h>
void fesetexceptflag(const fexcept_t *f, int excepts);
```

Parameter

`f`

A pointer to a constant exception flag variable.

`excepts`

Zero or more exceptions to copy to the floating-point environment.

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fesetexceptflag()

```
#include <fenv.h>
#include <stdio.h>
#pragma STDC FENV_ACCESS on
int main(void)
{
double result;
fexcept_t flags = 0;
/* Save the divisdivisionon-by-zero and overflow flags. */
fegetexceptflag(&flags, FE_DIVBYZERO | FE_OVERFLOW);
result = 0.0 / 0.0; /* Division by zero! */
/* Restore our flags. */
fesetexceptflag(&flags, FE_DIVBYZERO | FE_OVERFLOW);
return 0;
}
```

7.2.5 fetestexcept()

Test if a floating-point exception has been raised.

```
#include <fenv.h>
void fetestexcept(int e);
```

Parameter

e

Zero or more exceptions to test.

Remarks

This function returns true if one or more of the exceptions specified by e have been raised.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

7.3 Rounding Modes for the Floating-Point Environment

Facilities for retrieving and setting the floating-point environment's modes for rounding off numbers.

The floating point environment has a control mode to specify how to perform rounding operations. The table below lists the macros that define the modes that specify how to round off numbers.

Table 7-2. Rounding Modes

Macro	Rounding Mode
FE_DOWNWARD	To the smallest integer
FE_TONEAREST	To the nearest integer
FE_TOWARDZERO	To the largest integer when negative, to the smallest integer when positive
FE_UPWARD	To the largest integer

7.3.1 fegetround()

Returns the floating-point environment's current rounding direction.

```
#include <fenv.h>
int fegetround(void);
```

Remarks

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fegetround()

```
#include <fenv.h>
#include <stdio.h>
#pragma STDC FENV_ACCESS on
int main(void)
{
int direction;
double x_up = 0.0;
double a = 5.0;
double b = 2.0;
double c = 3.0;
```

```
double d = 6.0;
double f = 2.5;
double g = 0.5;
double ubound = 0.0;
feclearexcept(FE_ALL_EXCEPT);
/* Calculate denominator. */
fesetround(FE_DOWNWARD);
x_up = f + g;
direction = fegetround(); /* Save the direction. */
fesetround(FE_UPWARD);
ubound = (a * b + c * d) / x_up;
fesetround(direction); /* Restore original direction. */
printf("(a * b + c * d) / (f + g) = %g\n", ubound);
return 0;
}
Output:
9.3333
```

7.3.2 fesetround()

Sets the floating-point environment's rounding direction.

```
#include <fenv.h>
int fesetround(int direction);
```

Parameter

`direction`

A valid rounding direction.

Remarks

This function returns true if the rounding mode specified by direction is valid, false otherwise.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Chapter 8
float.h

This header file defines macros that specify the characteristics of the floating point data types `float`, `double`, and `long double`. The table below lists the macros that describe the properties of the floating point types.

The prefix of each macro's name specifies the data type that the macro applies to:

- `FLT`: the float type
- `DBL`: the double type
- `LDBL`: the long double type

Table 8-1. Floating Point Characteristics

Macro	Characteristics
`FLT_RADIX`	A numerical argument is not within the acceptable range of values that a function requires.
`F LT_ROUNDS`	An operation to convert to or from a multibyte character sequence could not be completed.
`FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG`	A result could not be computed because it would be beyond the range of values that can be stored in its data type.
`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.
`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 expressible in the data type.
`FLT_MAX, DBL_MAX, LDBL_MAX`	The largest floating point value expressible in the data type.
`FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON`	The smallest fraction expressible in the data type.

Chapter 9
inttypes.h

Defines types, macros, and functions for manipulating integer values.

This header file also includes the `stdint.h` header file.

9.1 Integer Input Scanning

Defines macros for converting formatted text to integer types.

The `inttypes.h` header file defines macros that expand to a literal character string that forms a conversion specifier suitable for use with formatted input functions.

The table below lists the macros that expand to conversion specifiers for input.

Table 9-1. Integer Conversion Specifiers for Input

Format	Type	Macro
decimal	int8_t	SCNd8
decimal	int16_t	SCNd16
decimal	int32_t	SCNd32
decimal	int64_t	SCNd64
decimal	int_least8_t	SCNdLEAST8
decimal	int_least16_t	SCNdLEAST16
decimal	int_least32_t	SCNdLEAST32
decimal	int_least64_t	SCNdLEAST64
decimal	int_fast8_t	SCNdFAST8
decimal	int_fast16_t	SCNdFAST16
decimal	int_fast32_t	SCNdFAST32
decimal	int_fast64_t	SCNdFAST64
decimal	intmax_t	SCNdMAX

Table continues on the next page...

Table 9-1. Integer Conversion Specifiers for Input (continued)

Format	Type	Macro
decimal	intptr_t	SCNdPTR
octal, decimal, or hexadecimal	int8_t	SCNi8
octal, decimal, or hexadecimal	int16_t	SCNi16
octal, decimal, or hexadecimal	int32_t	SCNi32
octal, decimal, or hexadecimal	int64_t	SCNi64
octal, decimal, or hexadecimal	int_least8_t	SCNiLEAST8
octal, decimal, or hexadecimal	int_least16_t	SCNiLEAST16
octal, decimal, or hexadecimal	int_least32_t	SCNiLEAST32
octal, decimal, or hexadecimal	int_least64_t	SCNiLEAST64
octal, decimal, or hexadecimal	int_fast8_t	SCNiFAST8
octal, decimal, or hexadecimal	int_fast16_t	SCNiFAST16
octal, decimal, or hexadecimal	int_fast32_t	SCNiFAST32
octal, decimal, or hexadecimal	int_fast64_t	SCNiFAST64
octal, decimal, or hexadecimal	intmax_t	SCNiMAX
octal, decimal, or hexadecimal	intptr_t	SCNiPTR
unsigned octal	int8_t	SCNo8
unsigned octal	int16_t	SCNo16
unsigned octal	int32_t	SCNo32
unsigned octal	int64_t	SCNo64
unsigned octal	int_least8_t	SCNoLEAST8
unsigned octal	int_least16_t	SCNoLEAST16
unsigned octal	int_least32_t	SCNoLEAST32
unsigned octal	int_least64_t	SCNoLEAST64
unsigned octal	int_fast8_t	SCNoFAST8
unsigned octal	int_fast16_t	SCNoFAST16
unsigned octal	int_fast32_t	SCNoFAST32
unsigned octal	int_fast64_t	SCNoFAST64
unsigned octal	intmax_t	SCNoMAX
unsigned octal	intptr_t	SCNoPTR
unsigned decimal	int8_t	SCNu8
unsigned decimal	int16_t	SCNu16
unsigned decimal	int32_t	SCNu32
unsigned decimal	int64_t	SCNu64
unsigned decimal	int_least8_t	SCNuLEAST8
unsigned decimal	int_least16_t	SCNuLEAST16
unsigned decimal	int_least32_t	SCNuLEAST32
unsigned decimal	int_least64_t	SCNuLEAST64
unsigned decimal	int_fast8_t	SCNuFAST8
unsigned decimal	int_fast16_t	SCNuFAST16

Table continues on the next page...

Table 9-1. Integer Conversion Specifiers for Input (continued)

Format	Type	Macro
unsigned decimal	int_fast32_t	SCNuFAST32
unsigned decimal	int_fast64_t	SCNuFAST64
unsigned decimal	intmax_t	SCNuMAX
unsigned decimal	intptr_t	SCNuPTR
unsigned hexadecimal	int8_t	SCNx8
unsigned hexadecimal	int16_t	SCNx16
unsigned hexadecimal	int32_t	SCNx32
unsigned hexadecimal	int64_t	SCNx64
unsigned hexadecimal	int_least8_t	SCNxLEAST8
unsigned hexadecimal	int_least16_t	SCNxLEAST16
unsigned hexadecimal	int_least32_t	SCNxLEAST32
unsigned hexadecimal	int_least64_t	SCNxLEAST64
unsigned hexadecimal	int_fast8_t	SCNxFAST8
unsigned hexadecimal	int_fast16_t	SCNxFAST16
unsigned hexadecimal	int_fast32_t	SCNxFAST32
unsigned hexadecimal	int_fast64_t	SCNxFAST64
unsigned hexadecimal	intmax_t	SCNxMAX
unsigned hexadecimal	intptr_t	SCNxPTR

Listing: Example of Integer Input

```
#include <stdio.h>
#include <inttypes.h>
int main(void)
{
int8_t i8;
intmax_t im;
printf("Enter an integer surrounded by ! marks.\n");
scanf("!%" SCNd8 "!", &i8);
printf("Enter a large integer\n");
printf("in hexadecimal, octal, or decimal.\n");
scanf("%" SCNiMAX, &im);
return 0;
}
Output:
Enter an 8-bit integer surrounded by ! marks.
!63!
Enter a large integer
in hexadecimal, octal, or decimal.
175812759
```

9.2 Integer Output Scanning

Defines macros for converting integer types to formatted text.

The `inttypes.h` header file defines macros that expand to a literal character string that forms a conversion specifier suitable for use with formatted output functions.

The table below lists the macros that expand to conversion specifiers for output.

Table 9-2. Integer Conversion Specifiers for Output

Type	Format	Macro
int8_t	decimal	PRId8 or PRIi8
int16_t	decimal	PRId16 or PRIi16
int32_t	decimal	PRId32 or PRIi32
int64_t	decimal	PRId64 or PRIi64
int_least8_t	decimal	PRIdLEAST8 or PRIiLEAST8
int_least16_t	decimal	PRIdLEAST16 or PRIiLEAST16
int_least32_t	decimal	PRIdLEAST32 or PRIiLEAST32
int_least64_t	decimal	PRIdLEAST64 or PRIiLEAST64
int_fast8_t	decimal	PRIdFAST8 or PRIiFAST8
int_fast16_t	decimal	PRIdFAST16 or PRIiFAST16
int_fast32_t	decimal	PRIdFAST32 or PRIiFAST32
int_fast64_t	decimal	PRIdFAST64 or PRIiFAST64
intmax_t	decimal	PRIdMAX or PRIiMAX
intptr_t	decimal	PRIdPTR or PRIiPTR
int8_t	unsigned octal	PRIo8
int16_t	unsigned octal	PRIo16
int32_t	unsigned octal	PRIo32
int64_t	unsigned octal	PRIo64
int_least8_t	unsigned octal	PRIoLEAST8
int_least16_t	unsigned octal	PRIoLEAST16
int_least32_t	unsigned octal	PRIoLEAST32
int_least64_t	unsigned octal	PRIoLEAST64
int_fast8_t	unsigned octal	PRIoFAST8
int_fast16_t	unsigned octal	PRIoFAST16
int_fast32_t	unsigned octal	PRIoFAST32
int_fast64_t	unsigned octal	PRIoFAST64
intmax_t	unsigned octal	PRIoMAX
intptr_t	unsigned octal	PRIoPTR
int8_t	unsigned decimal	PRIu8
int16_t	unsigned decimal	PRIu16
int32_t	unsigned decimal	PRIu32
int64_t	unsigned decimal	PRIu64
int_least8_t	unsigned decimal	PRIuLEAST8
int_least16_t	unsigned decimal	PRIuLEAST16
int_least32_t	unsigned decimal	PRIuLEAST32

Table continues on the next page...

Table 9-2. Integer Conversion Specifiers for Output (continued)

Type	Format	Macro
int_least64_t	unsigned decimal	PRIuLEAST64
int_fast8_t	unsigned decimal	PRIuFAST8
int_fast16_t	unsigned decimal	PRIuFAST16
int_fast32_t	unsigned decimal	PRIuFAST32
int_fast64_t	unsigned decimal	PRIuFAST64
intmax_t	unsigned decimal	PRIuMAX
intptr_t	unsigned decimal	PRIuPTR
int8_t	unsigned hexadecimal	PRIx8
int16_t	unsigned hexadecimal	PRIx16
int32_t	unsigned hexadecimal	PRIx32
int64_t	unsigned hexadecimal	PRIx64
int_least8_t	unsigned hexadecimal	PRIxLEAST8
int_least16_t	unsigned hexadecimal	PRIxLEAST16
int_least32_t	unsigned hexadecimal	PRIxLEAST32
int_least64_t	unsigned hexadecimal	PRIxLEAST64
int_fast8_t	unsigned hexadecimal	PRIxFAST8
int_fast16_t	unsigned hexadecimal	PRIxFAST16
int_fast32_t	unsigned hexadecimal	PRIxFAST32
int_fast64_t	unsigned hexadecimal	PRIxFAST64
intmax_t	unsigned hexadecimal	PRIxMAX
intptr_t	unsigned hexadecimal	PRIxPTR
int8_t	unsigned hexadecimal with uppercase letters	PRIX8
int16_t	unsigned hexadecimal with uppercase letters	PRIX16
int32_t	unsigned hexadecimal with uppercase letters	PRIX32
int64_t	unsigned hexadecimal with uppercase letters	PRIX64
int_least8_t	unsigned hexadecimal with uppercase letters	PRIXLEAST8
int_least16_t	unsigned hexadecimal with uppercase letters	PRIXLEAST16
int_least32_t	unsigned hexadecimal with uppercase letters	PRIXLEAST32
int_least64_t	unsigned hexadecimal with uppercase letters	PRIXLEAST64
int_fast8_t	unsigned hexadecimal with uppercase letters	PRIXFAST8
int_fast16_t	unsigned hexadecimal with uppercase letters	PRIXFAST16
int_fast32_t	unsigned hexadecimal with uppercase letters	PRIXFAST32

Table continues on the next page...

Table 9-2. Integer Conversion Specifiers for Output (continued)

Type	Format	Macro
int_fast64_t	unsigned hexadecimal with uppercase letters	PRIXFAST64
intmax_t	unsigned hexadecimal with uppercase letters	PRIXMAX
intptr_t	unsigned hexadecimal with uppercase letters	PRIXPTR

Listing: Example of Integer Output

```
#include <stdio.h>
#include <inttypes.h>
int main(void)
{
intmax_t im = 175812759L;
printf("%" PRId8 "\n", 63);
printf("%" PRIxMAX, im);
return 0;
}
Output:
63
0xa7ab097
```

9.3 imaxabs(), imaxdiv(), strtoimax(), strtoumax(), wcstoimax(), wcstoumax()

These functions operate on the largest integer types, intmax_t and uintmax_t.

```
#include <inttypes.h>
  intmax_t imaxabs(intmax_t j);
typedef struct {
 intmax_t quot;
  intmax_t rem;
  } imaxdiv_t;
 imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
 intmax_t strtoimax(const char * restrict nptr,
  char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr,
 char ** restrict endptr, int base);
 intmax_t wcstoimax(const wchar_t * restrict nptr,
  wchar_t ** restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr,
  wchar_t ** restrict endptr, int base);
```

Remarks

The table below matches these functions to an equivalent integer-based functions in stdlib.h and wchar.h.

Table 9-3. Greatest-Width Integer Functions

Function	Greatest-width Integer Equivalent Function
imaxabs()	abs()
imaxdiv()	div()
strtoimax()	strtol()
strtoumax()	strtoul()
wcstoimax()	wcstol()
wcstoumax()	wcstoul()

Chapter 10
iso646.h

Defines alternative keywords for operators that use characters that are not available in some character sets.

The ISO/IEC 646 standard defines character sets based on the ASCII character set that replaces some punctuation characters with other characters to accommodate regional requirements. Some of these character sets do not have some of the characters that the C language uses in its operators and keywords. For example, the bitwise-and operator uses the ampersand, "&" , which is not available in some character sets. Use the preprocessor macros defined in `iso646.h` to use operators which are composed of characters that are available in all character sets specified by the ISO/IEC 646 standard.

The table below lists the preprocessor macros that define replacements for operators that use characters that are not available in some character sets.

Table 10-1. Alternative Keywords

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

Chapter 11
locale.h

Provides facilities for handling different character sets and numeric formats.

11.1 Functions in locale.h

This section lists the functions available in the locale.h header file.

11.1.1 lconv

Specifies formatting characteristics.

```
#include <locale.h>
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;
};
```

Parameter

`decimal_point`

Character string containing the non-monetary decimal point.

`thousands_sep`

Character string containing the non-monetary separator for digit grouping.

`grouping`

Non-monetary digit grouping sequence, encoded in a character string. Each character in the sequence is actually a number specifying the number of digits to group left of the decimal point. The null character, 0, specifies that the preceding group should be repeated. The number `CHAR_MAX` specifies that grouping is no longer required.

`int_curr_symbol`

Character string composed of a 3-character international currency symbol followed by separation character between the currency symbol and the currency number.

`currency_symbol`

Character string containing the local currency symbol.

`mon_decimal_point`

Character string containing the monetary decimal point.

`mon_thousands_sep`

Character string containing the monetary separator for digit grouping.

`mon_grouping`

Monetary digit grouping sequence. Uses same format as `grouping`.

`positive_sign`

Character string containing the sign for positive monetary numbers.

`negative_sign`

Character string containing the sign for negative monetary numbers.

`int_frac_digits`

Number of digits that may appear to the right of the decimal point for international monetary numbers.

`frac_digits`

Number of digits that may appear to the right of the decimal point for non-international monetary numbers.

`p_cs_precedes`

Contains 1 if `currency_symbol` should precede positive monetary numbers, 0 otherwise.

`p_sep_by_space`

Contains 1 if `currency_symbol` should be separated from positive monetary values by a space, 0 otherwise.

`n_cs_precedes`

Contains 1 if `currency_symbol` should precede negative monetary numbers, 0 otherwise.

`n_sep_by_space`

Contains 1 if `currency_symbol` should be separated from negative monetary values by a space, 0 otherwise.

`p_sign_posn`

Specifies how to use the `positive_sign` character string. Contains 0 when parentheses should surround the number and `currency_symbol`, 1 when the sign should precede the number and `currency_symbol`, 2 when the sign should follow the number and `currency_symbol`, 3 when the sign precedes `currency_symbol`, 4 when the sign follows `currency_symbol`.

`n_sign_posn`

Specifies how to use the `negative_sign` character string. Follows the same convention as `p_sign_posn`.

Remarks

The `lconv` structure specifies numeric formatting requirements. Use `localeconv()` to retrieve a pointer to an `lconv` structure for the current locale.

11.1.2 localeconv()

Returns conversion settings for the current locale.

```
#include <locale.h>

struct lconv* localeconv(void);
```

Remarks

The ISO/IEC C standard specifies that aspects of the C compiler should be adaptable to geographic locales. The `locale.h` header file provides facilities for handling different character sets and numeric formats. CodeWarrior C compilers and EWL use the `"C"` locale by default and a native locale named `""` (the empty string).

11.1.3 setlocale()

Queries or sets locale properties.

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

Parameter

`cat`

The property to query or set.

`locale`

A pointer to the locale.

Remarks

The cat argument specifies the locale property to query or set. The following table lists the values recognized by `setlocale()`.

If locale is a null pointer then `setlocale()` makes a query. It returns a pointer to a character string that indicates which locale that the cat property is set to. Your program must not modify this character string. Subsequent calls to the function might alter this character string.

If locale is not a null pointer then `setlocale()` modifies the locale property specified by cat. The character string that locale points to must be a result returned by a previous query to `setlocale()`. The function returns a pointer to a character string that describes the newly-set locale.

CodeWarrior C compilers and EWL use the `"C"` locale by default and a native locale named `""` (the empty string).

The function returns NULL if it cannot perform a query or set operation. This facility may not be available on some configurations of the EWL.

Table 11-1. Locale Properties for setlocale()

Value	Locale Property
LC_ALL	All properties.
LC_COLLATE	Behaviors for strcoll() and strxfrm.
LC_CTYPE	Character manipulation behaviors of facilities in ctype.h, wctype.h, and stdlib.h.
LC_MONETARY	Monetary formatting properties returned by localeconv().
LC_NUMERIC	Non-monetary numeric formatting properties returned by localeconv().
LC_TIME	Behavior for strftime().

Listing: Example of setlocal()

```
#include <locale.h>
#include <stdlib.h>
#include <string.h>
char* copylocale(int cat);
char* copylocale(int cat)
{
char* loc;
char* copy;
/* Make query. */
if ((loc = setlocale(cat, NULL)) == NULL)
return NULL; /* Query failure. */
/* Allocate memory, including null character. */
copy = (char*)malloc(strlen(loc) + 1);
if (copy == NULL) /* Memory failure. */
return NULL;
return strcpy(copy, loc);
}
int main(void)
{
char* save;
if ((save = copylocale(LC_ALL)) == NULL)
return 1;
setlocale(LC_ALL, ""); /* Set native locale. */
/* ... */
setlocale(LC_ALL, save); /* Restore locale. */
/* ... */
_free(save);
return 0;
}
```

Chapter 12
limits.h

Defines maximum and minimum values that integral data types may contain.

This header file defines macros that describe the maximum and minimum values that objects with integral data types may accurately represent. Assigning values beyond the range of these limits will give undefined results at runtime.

The `limits.h` header file does not define macros to describe the minimum values for unsigned data types. The minimum value of an unsigned data type is always 0. The table below lists the preprocessor macros that define replacements for operators that use characters that are not available in some character sets.

Table 12-1. Integral Limits

Macro	Value
`CHAR_BIT`	Number of bits in the smallest data type that is not a bit field.
`CHAR_MAX, CHAR_MIN`	Maximum and minimum values, respectively, for `char`.
`SCHAR_MAX, SCHAR_MIN`	Maximum and minimum values, respectively, for `signed char`.
`UCHAR_MAX`	Maximum value for `unsigned char`.
`SHRT_MAX, SHRT_MIN`	Maximum and minimum values, respectively, for `short int`.
`USHRT_MAX`	Maximum value for `unsigned short int`.
`INT_MAX, INT_MIN`	Maximum and minimum values, respectively, for `int`.
`UINT_MAX`	Maximum value for `unsigned int`.
`LONG_MAX, LONG_MIN`	Maximum and minimum values, respectively, for `long int`.
`ULONG_MA`	Maximum value for `unsigned long int`.
`MB_LEN_MAX`	Maximum number of bytes in a multibyte character.
`LLONG_MAX, LLONG_MIN`	Maximum and minimum values, respectively, for `long long int`. This facility is not part of the ISO/IEC standard for C. It is an EWL extension of the C Standard Library.
`ULLONG_MAX`	Maximum value for `unsignedlong long int`. This facility is not part of the ISO/IEC standard for C. It is an EWL extension of the C Standard Library.

Chapter 13
maths.h

Provides floating point mathematical and conversion functions.The functions and data types of the EWL implementation of the `math.h` header file follow provide facilities to perform mathematical operations.

13.1 Predefined Values

Some useful values that describe properties of the floating point environment.

```
#include <math.h>

#define
HUGE_VAL /*...*/

#define
HUGE_VALF /*...*/

#define
HUGE_VALL /*...*/

#define
NAN /*...*/

#define
INFINITY /*...*/
```

If the floating-point result of a function is too large to be represented as a value by the return type, the function will return `HUGE_VAL` if the return type is double, `HUGE_VALF` if the type is float, or `HUGE_VALL` if long double.

The NAN macro defines the quiet `NaN` of type `float`.

The `INFINITY` macro defines the value of infinity of type `float`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.2 Floating Point Math Errors

Mechanism for revealing floating-point math errors.

The functions declared in the math.h header file are not fully compliant with the ISO/IEC C standard's specifications for reporting errors. The standard requires floating point functions to set the errno variable to report an error condition. This mechanism is inefficient and un-informative.

EWL provides `fpclassify()` to simplify error checking. To check the validity of a computation returned by a function in `math.h`, call `fpclassify()` with the math function's result.

The following two listings compare these two error-checking approaches.

Listing: Using errno for Error Checking

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
int main(void)
{
double x;
errno = 0;
x = log(0);
if (errno)
puts("error");
return 0;
}
```

Listing: Using fpclassify() for Error Checking

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x;
x = log(0);
```

```
if (fpclassify(x) != FP_NORMAL)

puts("error");

return 0;

}
```

13.3 NaN

A floating-point value that represents the result of a uncomputable math operation.

NaN has no relationship with any other number. NaN is neither greater than, less than, or equal to any other number. There are two types of NaN:

- Quiet NaN
- Signalling NaN

13.3.1 Quiet NaN

A quiet NaN is the result of an indeterminate calculation. For example, zero divided by zero and infinity minus infinity are examples of computations that result in NaN. A quiet NaN's binary representation has a full exponent, its most significant bit is 0, and its 2nd-most significant bit is 1.

13.3.2 Signalling NaN

A signalling NaN does not occur as a result of an arithmetic computation. A signalling NaN occurs when you load a bad value from memory into a floating-point register that happens to has the same bit pattern as a signalling NaN. IEEE 754 requires that in such a situation the processor should raise an exception. It should then convert the signalling NaN to a quiet NaN so the lifetime of a signalling NaN will be brief.

A signalling NaN's binary representation has a full exponent and its 2 most significant bits are 1

13.4 Floating-Point Classification

Facilities to query properties of floating-point values.

13.4.1 fpclassify()

Classifies floating-point numbers.

```
#include <math.h>
int fpclassify(x);
```

Parameter

`x`

A value of type float, double, or long double.

Remarks

This macro returns the type of a floating-point value as an integral value:

- `FP_NAN`: NaN ("Not a Number")
- `FP_INFINITE`: positive or negative infinity
- `FP_ZERO`: zero
- `FP_NORMAL`: a normalized floating point value
- `FP_SUBNORMAL`: a denormalized floating point value

Use this macro to check for errors in floating point computations. This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fpclassify()

```
#include <math>

#include <stdio.h>

void mypow(double x, double y)

{

double p;

p = pow(x, y);

switch (fpclassify(p))
```

```
{
case FP_ZERO:
case FP_NORMAL:
case FP_SUBNORMAL:
printf("%f", p);
break;
default:
printf("error in pow()");
break;
}
}
```

13.4.2 isfinite()

Tests if a value is a finite number.

```
#include <math.h>
int isfinite(x);
```

Parameter

`x`

A `float`, `double`, or `long double` value to test.

Remarks

The function returns true if the value tested is finite. Otherwise it returns false. A finite number is not infinite and is not NaN.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.4.3 isnan()

Tests if a value is a computable number.

```
#include <math.h>
int isnan(x);
```

Parameter

`x`

A `float`, `double`, or `long double` value to test.

Remarks

This function returns true if x is not a number.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.4.4 isnormal()

Tests if a value is a computable number.

```
#include <math.h>
int isnormal(x);
```

Parameter

`x`

A `float`, `double`, or `long double` value to test.

Remarks

This macro returns true if the argument is a normalized number. A normalized number is not zero, not subnormal, not infinite, and not NaN.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.4.5 signbit()

Tests if a value is a computable number.

```
#include <math.h>
int signbit(x);
```

Parameter

x

A float, double, or long double value to test.

Remarks

This macro returns true if x is negative.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.5 Trigonometry

Compute sine, cosine, and other trigonometric values.

13.5.1 acos()

Computes the arc value of cosine.

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

Parameter

x

A floating-point value, in the range of -1.0 to 1.0.

Remarks

These functions return the arccosine of the argument x, in radians from 0 to pi. If x is not in the range of -1.0 to 1.0 then these functions set errno to EDOM and

```
fpclassify(acos(x))
```

returns FP_NAN.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.5.2 asin()

Computes the arc value of sine.

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

Parameter

x

A floating-point value, in the range of -1.0 to 1.0.

Remarks

These functions return the arcsine of the argument x, in radians from -pi /2 to pi / 2. If x is not in the range of -1.0 to 1.0 then these functions set `errno` to `EDOM` and

```
fpclassify(asin(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.5.3 atan()

Computes the arc value of tangent.

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

Parameter

x

A floating-point value.

Remarks

These functions return the arctangent of the argument x, in radians from -pi / 2 to pi / 2.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.5.4 atan2()

Computes the value of a tangent.

```
#include <math.h>
double atan2(double y, double x);
float atan2f(float, float);
long double atan2l(long double, long double);
```

Remarks

These functions return the arctangent of y / x, in radians from -pi to pi. If y and x are 0, then these functions set `errno` to `EDOM` and

```
fpclassify(atan2(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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
```

13.5.5 cos()

Computes the cosine.

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

Parameter

x

A value from which to compute.

Remarks

These functions return the cosine of x, which is measured in radians.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.5.6 sin()

Computes the sine of a radian value.

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

Parameter

x

A floating point value, in radians.

Remarks

These functions return the sine of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of sin()

```
#include <math.h>
#include <stdio.h>
#define DegtoRad (2.0*pi/360.0)
int main(void)
{
double x = 57.0;
double xRad = x*DegtoRad;
printf("The sine of %.2f degrees is %.4f.\n",x, sin(xRad));
return 0;
}
Output:
The sine of 57.00 degrees is 0.8387.
```

13.5.7 tan()

Computes the tangent of a radian value.

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

Parameter

`x`

A floating point value, in radians.

Remarks

These functions compute the tangent of x. If x is close to an odd multiple of pi divided by 2, these functions assign `errno` to `EDOM`. Use `fpclassify()` to check the validity of the results returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of tan()

```
#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.
```

13.6 Hyperbolic Trigonometry

Compute hyperbolic trigonometric values.

13.6.1 acosh()

Computes the inverse hyperbolic cosine.

```
#include <math.h>
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Parameter

`x`

A value from which to compute the inverse hyperbolic cosine.

Remarks

These functions return the non-negative inverse hyperbolic cosine of x. If x is not greater than 1 then these functions set `errno` to `EDOM` and

```
fpclassify(acosh(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of acosh()

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double a = 3.14;
printf("The arc hyperbolic cosine of %f is %f.\n", a, acosh(a));
return 0;
```

```
}
Output:
The arc hyperbolic cosine of 3.140000 is 1.810991.
```

13.6.2 asinh()

Computes the inverse hyperbolic sine.

```
#include <math.h>
double asinh(double x);
float asinhf(float x);
long double acsinhl(long double x);
```

Parameter

x

A floating-point value.

Remarks

These functions return the hyperbolic arcsine of the argument x. If the result of the computation is out of range, these functions set `errno` to `EDOM` and

```
fpclassifyfpclassify(asinh(x))
```

returns `FP_NAN`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.6.3 atanh()

Computes the inverse hyperbolic tangent.

```
#include <math.h>
double atanh(double x);
float atanhf(float);
long double atanhl(long double);
```

Parameter

x

A floating-point value.

Remarks

These functions return the hyperbolic arcsine of the argument x. If x is greater than 1 or less than -1, these functions set set errno to EDOM and

```
fpclassify(atanh(x))
```

returns FP_NAN.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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", c, atanh(c));
return 0;
}
Output:
The arc hyperbolic tan of 0.500000 is 0.549306.
```

13.6.4 cosh()

Computes the hyperbolic cosine.

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

Parameter

x

A value from which to compute.

Remarks

These functions return the hyperbolic cosine of I, which is measured in radians.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of cosh()

```
#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.
```

13.6.5 sinh()

Computes the hyperbolic sine.

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

Parameter

x

A floating-point value.

Remarks

These functions compute the hyperbolic sine of x. These functions may assign `ERANGE` to `errno` if x is out of range. Use `fpclassify()` to check the validity of the results returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of sinh() Usage

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

13.6.6 tanh()

Computes the hyperbolic tangent.

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

Parameter

x

A floating-point value.

Remarks

These functions compute the hyperbolic tangent of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.7 Exponents and Logarithms

Exponential and logarithmic values.

13.7.1 exp()

Computes the power of e.

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

Parameter

x

A value from which to compute.

Remarks

These functions returns ex, where e is the natural logarithm base value.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.7.2 exp2()

Computes the power of 2.

```
#include <math.h>

double exp2(double x);

float exp2f(float x);

long double exp2l(long double x);
```

Parameter

`x`

A value from which to compute.

Remarks

These functions returns 2x.

If x is too large, the function sets `errno` to `ERANGE` and `fpclassify(exp2(x))` does not return `FP_NORMAL`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of exp2() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double i = 12;
printf("2^%f = %f.\n",i,i, exp2(i));
return 0;
}
Output:
2^(12.000000) = 4096.000000.
```

13.7.3 expm1()

Computes a power of e minus 1.

```
#include <math.h>
double expm1(double x);
float expm1l(float x);
long double expm1l(long double x);
```

Parameter

x

A value from which to compute.

Remarks

This function returns e^X - 1. This function may be more accurate than calling `exp(x)` and subtracting 1.0 from its result.

If x is too large, the function sets `errno` to `ERANGE` and `fpclassify(exp2(x))` does not return `FP_NORMAL`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of expm1() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double j = 12;
printf("e^%f - 1 = %f\n",j,expm1(j));
return 0;
}
Output:
e^12.000000 - 1 = 162753.791419
```

13.7.4 frexp()

Extracts the mantissa and exponent from a floating point number's binary representation.

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

Parameter

x

The floating point value to extract from.

e

A pointer to an integer in which to store the exponent.

Remarks

A floating point number's binary representation follows this formula:

```
m 2 e
```

where m is the mantissa and e is the exponent, 0.5 < m < 1.0 and e is an integer value.

These functions return, when possible, the value of a floating-point number's exponent ine and returns the mantissa.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.7.5 ilogb()

Compute the exponent of a value as a signed integer.

```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int double ilogbl(long double x);
```

Parameter

x

A floating point value.

Remarks

These functions return the natural exponent of x as a signed int value. If x is zero they return the value FP_ILOGB0. If x is infinite they return the value INT_MAX. If x is a NaN they return the value FP_ILOGBNAN. Otherwise, these functions are equivalent to calling the

corresponding `logb()` functions and casting the returned value to type `int`. A range error may occur if x is 0. Use `fpclassify()` to check the validity of the result returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.7.6 ldexp()

Constructs a floating point value from a mantissa and exponent.

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

Parameter

`x`

A mantissa value.

`exp`

A exponent value.

Remarks

The `ldexp()` functions compute x * 2 exp. These functions can be used to construct a floating point value from the values returned by the `frexp()` functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.7.7 log()

Compute natural logarithms.

```
#include <math.h>

double log(double x);

float logf(float x);

long double logl(long double x);
```

Parameter

x

A floating point value.

Remarks

This function returns log e x.

If x < 0, `log()` assigns `EDOM` to `errno`. Use `fpclassify()` to check the validity of the result returned by `log()`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of log() 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.
```

13.7.8 log1p()

Compute base-e logarithms.

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Parameter

`x`

The value being computed.

Remarks

These functions computes the base-e logarithm of x, denoted as `loge(1.0 + x)`.

The value of x must be greater than -1. Use `fpclassify()` to check the validity of the result returned by `log1p()`.

For small magnitude x, these functions are more accurate than `log(x+1.0)`. The functions return base-e logarithm of `(1 + x)`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of log1p() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
float u = 5.0;
```

```
printf("log1p(%f) = %f\n", u, log1pf(u));
return 0;
}
Output:
log1p(5.000000) = 1.791759.
```

13.7.9 log2()

Compute base-2 logarithms.

```
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Parameter

`x`

The value being computed.

Remarks

These functions computes the base-2 logarithm of x, denoted as `loge(1.0 + x)`.

The value of x must be greater than 0. Use `fpclassify()` to check the validity of the result returned by `log1p()`.

For small magnitude x, these functions are more accurate than `log(x+1.0)`. The functions return base-2 logarithm of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of log2() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
float u = 5.0;
printf("log2(%f) = %f\n", u, log2f(u));
```

```
return 0;
}
Output:
log2(5.000000) = 2.321928.
```

13.7.10 logb()

Compute a logarithm with the floating-point representation's radix as base.

```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Parameter

`x`

The value being computed.

Remarks

These functions compute the exponent of x in the target system's binary representation.

The value of `x` must be not be 0. Use `fpclassify()` to check the validity of the result returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of logb() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
float u = 5.0;
printf("logb(%f) = %f\n", u, logbf(u));
return 0;
}
```

```
Output:

log2(5.000000) = 2.000000
```

13.7.11 scalbn(), scalbln()

Computes x *

Computes x * `FLT_RADIX^n` efficiently.

```
#include <math.h>

double scalbn(double x, int n);

float scalbnf(float x, int n);

long double scalblnl(long double x, int n);

double scalbln(double x, long int n);

float scalblnf(float x, long int n);

long double scalblnl(long double x, long int n);
```

Parameter

`x`

The original value.

`n`

The original value.

Remarks

These functions return `* FLT_RADIX`.

A range error may occur. Use `fpclassify()` to check the validity of the result returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.8 Powers and Absolute Values

Compute powers, roots, and absolute values.

13.8.1 cbrt()

Computes the cube root.

```
#include <math.h>
double cbrt(double x);
float cbrtf(float fx);
long double cbrtl(long double lx);
```

Parameter

`x`

A value from which to compute a cube root.

Remarks

This function computes the cube root of its argument.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.8.2 fabs()

Computes an absolute value.

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

Parameter

`x`

A value from which to compute.

Remarks

These functions return the absolute value of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.8.3 hypot()

Computes the length of a hypotenuse in a right-angle triangle.

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Parameter

`x`

A value representing the length of one side that is not the hypotenuse.

`y`

A value representing the length of other side that is not the hypotenuse.

Remarks

These functions compute the square root of the sum of the squares of x and y. These functions may be more accurate than the expression `sqrt(pow(x) + pow(y))`.

If these functions cannot compute a value they set `errno` to `ERANGE`. Use `fpclassify()` to check the validity of the result returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of hypot() usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double r = 3.0;
double s = 4.0;
printf("(%f^2 + %f^2)^(.5) = %f\n",r,s,hypot(r,s));
return 0;
}
Output:
(3.000000^2 + 4.000000^2)^(.5) = 5.000000.
```

13.8.4 pow()

Compute the power of a base number.

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

Parameter

`x`

A floating point value to use as base.

`y`

A floating point value to use as exponent.

Remarks

These functions compute xy .

These functions 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. Use `fpclassify()` to check the validity of the results returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.8.5 sqrt()

Compute square root.

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

Parameter

`x`

A floating point value

Remarks

These functions return the square root of x. These functions assign `EDOM` to `errno` if x < 0. Use `fpclassify()` to check the validity of the results returned by these functions.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.9 Statistical Errors and Gamma

Compute statistical and probability values.

13.9.1 erf()

Computes the Gauss error function.

```
#include <math.h>
double erf(double x);
```

Parameter

x

The value to compute.

Remarks

This function is defined as

```
erf(x) = 2/sqrt(pi) * ( integral from
0 to
x of
exp(pow(-t, 2))
dt
)
```

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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", g, erf(g));

return 0;

}

Output:

The error function of (10.000000) = 1.000000.
```

13.9.2 erfc()

Computes the Gauss complementary error function.

```
#include <math.h>
double erfc(double x);
```

Parameter

x

The value to compute.

Remarks

This function is defined as

```
erfc(x) = 1 - erf(x)
```

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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", h, erfc(h));
return 0;
}
Output:
The inverse error function of (10.000000) = 0.000000.
```

13.9.3 gamma()

Computes logeG(x).

```
#include <math.h>
double gamma(double x);
extern int signgam;
```

Parameter

x

The value from which to compute.

Remarks

This function computes logeG(x), where G(x) is defined as the integral of e-t * t x-1 dt from 0 to infinity. The function returns the sign of G(x) the external variable signgam. The argument x need not be a non-positive integer, (G(x) is defined over the real numbers, except the non-positive integers).

If this function cannot compute a value it sets errno to a non-zero value, either EDOM or ERANGE, and returns either HUGE_VAL or NAN. Use fpclassify() to check the validity of the result returned by gamma().

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

13.9.4 lgamma()

Computes loge| G(x) |.

```
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
extern int signgam;
```

Parameter

x

The value from which to compute.

Remarks

This function computes loge of the absolute value of G(x), where G(x) is defined as the integral of e-t * t x-1 dt from 0 to infinity. The function returns the sign of G(x) the external variable signgam. The argument x need not be a non-positive integer, (G(x) is defined over the real numbers, except the non-positive integers).

If this function cannot compute a value it sets `errno` to a non-zero value, either `EDOM` or `ERANGE`, and returns either `HUGE_VAL` or `NAN`. Use `fpclassify()` to check the validity of the result returned by `gamma()`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

13.10 Rounding

Compute the closest integral values of real numbers.

13.10.1 ceil()

Rounds a number up to the closest whole number.

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

Parameter

`x`

A number to round up.

Remarks

These functions return the smallest integer that is not less than x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.10.2 floor()

Rounds a number up to the closest whole number.

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

Parameter

x

A number to round down.

Remarks

These functions return the largest integer that is not greater than x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.10.3 lrint(), llrint()

Rounds off to an integral value according to the current rounding direction.

```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long int lrint(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrint(long double x);
```

Parameter

x

The value to be computed.

Remarks

These functions round x to an integral value in floating-point format using the current rounding direction. Unlike the `rint()` functions, these functions return their results as integer values.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.10.4 lround(), llround()

Rounds to an integral value.

```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Parameter

`x`

The value to be rounded.

Remarks

These functions round x to the neareset integer value. These functions ignore the current rounding direction; halfway values are rounded away from zero.

Unlike `round()`, these functions return their results as values of integer type.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of lround() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x = 2.4;
printf("lround(%f) = %f\n", x, lround(x));
return 0;
}
Output:
round(2.400000) = 2.000000
```

13.10.5 nearbyint()

Rounds off its argument to an integral value.

```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

Parameter

x

The value to be computed.

Remarks

These functions compute the closest integer value but do not raise an inexact exception.

The argument is returned as an integral value in floating point format.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of nearbyint() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x = 5.7;
printf("nearbyint(%f) = %f\n", x, nearbyint(x));
return 0;
}
Output:
nearbyint(5.700000) = 6.000000
```

13.10.6 rint()

Rounds off to an integral value.

```
#include <math.h>
double rint(double x);
```

```
float rintf(float x);
long double rint(long double x);
```

Parameter

x

The value to be compued.

Remarks

These functions round x to an integral value in floating-point format using the current rounding direction.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of rint() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x = 2.5;
printf("rint(%f) = %f\n", x, rint(x));
return 0;
}
Output:
rint(2.500000) = 2.000000
```

13.10.7 round()

Rounds to an integral value, rounding halfway values furthest from zero.

```
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Parameter

x

The value to be rounded.

Remarks

These functions round x to the nearest integer value. These functions ignore the current rounding direction; halfway values are rounded away from zero.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of round() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x = 2.5;
printf("round(%f) = %f\n", x, round(x));
return 0;
}
Output:
round(2.500000) = 2.000000
```

13.10.8 trunc()

Rounds to an integral value nearest to but not larger in magnitude than the argument.

```
#include <math.h>
double trunc(double x);
double truncf(double x);
double truncl(double x);
```

Parameter

x

The value to be truncated.

Remarks

This function returns an argument to an integral value in floating-point format.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of trunc() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x = 2.4108;
printf("trunc(%f) = %f\n", x, trunc(x));
return 0;
}
Output:
trunc(2.410800) = 2.000000
```

13.11 Remainders

Compute modulo, remainder, and quotient values.

13.11.1 fmod()

Computes a remainder after division.

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

Parameter

x

The dividend.

y

The divisor.

Remarks

These functions return, when possible, the value r, where x = i y + r for some integer i that allows |r | < |y|. The sign of r matches the sign of x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.11.2 modf()

Compute the integer and fraction parts of a floating point number.

Compute the integer and fraction parts of a floating point number.

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

Parameter

x

A floating-point value.

iptr

A pointer to the floating-point value.

Remarks

These functions separate x into its integer and fractional parts. In other words, these functions separate x such that x = f + i where 0 < |f| < 1 and i equal or less than |x|.

These functions return the signed fractional part of x and store the integer part in the value pointed to by iptr.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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.
```

13.11.3 remainder()

Computes the remainder of x / y.

```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainder(long double x, long double y);
```

Parameter

`x`

The dividend.

`y`

The divisor.

Remarks

These functions return the remainder, `r`, where `r = x - n y`, `r` is greater or equal to 0 and less than y, and y is non-zero.

The behavior of `remainder()` is independent of the rounding mode.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of remainder() usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double var1 = 2.0;
double var2 = 4.0;
printf("remainder(%f,%f) = %f\n", var1, var2, remainder(var1,
var2));
return 0;
}
Output:
remainder(2.000000, 4.000000) = 2.000000.
```

13.11.4 remquo()

Computes the remainder.

```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

Parameter

x

The dividend.

y

The divisor.

quo

Pointer to an integer in which to store the quotient.

Remarks

The argument quo points to an integer whose sign is the sign of x/y and whose magnitude is congruent mod 2n 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.

These functions return the remainder of x / y.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.12 Manipulation

Change and retrieve the properties of floating-point values.

13.12.1 copysign()

Copies the sign of a number.

```
#include <math.h>
double copysign (double x, double y);
```

Parameter

x

Magnitude.

y

Sign.

Remarks

This function produces a value with the magnitude of x and the sign of y. It regards the sign of zero to be positive. This function returns a signed NaN value if x is NaN.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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", e, f, copysign(e,f));
return 0;
}
Output:
Copysign(10.000000, -3.000000) = -10.000000.
```

13.12.2 nan()

Converts a character string to not-a-number.

```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Parameter

tagp

A pointer to a character string.

Remarks

A quiet NAN is returned, if available.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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", e, f, copysign(e,f));

return 0;

}

Output:

Copysign(10.000000, -3.000000) = -10.000000.
```

13.12.3 isgreater(), isgreaterequal(), isless(), islessequal(), islessgreater(), isunordered()

Compare floating point values, including `NaN` (Not a Number).

```
#include <math.h>
int isgreater(x, y);
int isgreaterequal(x, y);
int isless(x, y);
int islessequal(x, y);
int islessgreater(x, y);
int isunordered(x, y);
```

Parameter

`x`

A value of type `float`, `double`, or `long double`.

`y`

A value of type `float`, `double`, or `long double`.

Remarks

These macros compare two floating point numbers and return a boolean value. Unlike their expression counterparts, these macros accept the NaN value without raising a floating-point exception.

NaN is not in the range of floating point values from negative infinity to positive infinity, so it cannot be put in any order when compared to other values. In other words, A NaN value is not greater than, less than, or equal to any other floating point value.

Table 13-1. Counterparts

Macro	Equivalent to Expression
isgreater(x, y)	x > y
isgreaterequal(x, y)	x >= y
isless(x, y)	x < y
islessequal(x, y)	x <= y
islessgreater(x, y)	x < y || x > y
isunordered(x, y)	isnan(x) || isnan(y)

13.12.4 nextafter()

Returns the next representable value.

```
#include <math.h>

double nextafter(double x, double y);

float nextafterf(float x, float y);

long double nextafterl(long double x, long double y);
```

Parameter

`x`

Current representable value.

`y`

Direction to compute the next representable value.

Remarks

These functions compute the next representable value, after x in the direction of y. Thus, if y is less than x, `nextafter()` returns the largest representable floating-point number that is less than x.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of nextafter() Usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double x = 7.0;
double y;
y = x + 1.0;
printf("nextafter(%f, %f) = %f\n", x, y, nextafter(x,y));
return 0;
}
Output:
nextafter(7.000000, 8.000000) = 7.000004
```

13.12.5 nexttoward()

Returns the next representable value.

```
#include <math.h>
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Parameter

x

Current representable value.

y

Direction to compute the next representable value.

Remarks

These functions perform identically to their nextafter counterparts but has these differences:

- The y argument is always of type `long double`.
- If x equals y, these functions return y converted to x's type

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

13.13 Maximum and Minimum

Compute maximum and minimum values.

13.13.1 fdim()

Computes the positive difference of two numbers.

```
#include <math.h>

double fdim(double x, double y);

float fdimf(float x, float y);

long double fdiml(long double x, long double y);
```

Parameter

`x`

The minuend.

`y`

The subtrahend.

Remarks

If x is greater or equal to y, the functions returns `x - y`. If x is less than y, the functions set `errno` to `ERANGE` and fpclassify(fdim(x, x)) does not return `FP_NORMAL`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fdim() usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double k = 12;
double l = 4;
printf("|(%f - %f)| = %f\n", k, l, fdim(k,l));
return 0;
}
Output:
| ( 12.000000 - 4.0000000 ) | = 8.0000000
```

13.13.2 fmax()

Return the maximum of two values.

```
#include <math.h>
double fmax(double x, double y);
double fmaxf(float x, float y );
double fmaxl(long double x, long double y);
```

Parameter

x

First argument.

y

Second argument.

Remarks

These functions return x if x is greater or equal to y. Otherwise, they return y.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: 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",m,n,fmax(m,n));
return 0;
}
Output:
fmax(4.000000, 6.000000) = 6.000000
```

13.13.3 fmin()

Return the minimum of two values.

```
#include <math.h>
double fmin(double x, double y);
double fminf(float x, float y );
double fminl(long double x, long double y);
```

Parameter

x

First argument.

y

Second argument.

Remarks

These functions return x if x is less than or equal to y. Otherwise, they return y.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fmin() usage

```
#include <math.h>
```

```
#include <stdio.h>

int main(void)

{

double m = 4;

double n = 6;

printf("fmin(%f, %f)=%f.\n",m,n,fmin(m,n));

return 0;

}

Output:

fmin(4.000000, 6.000000) = 4.000000.
```

13.14 Multiply-Addition

An optimized combination of multiplication and addition.

13.14.1 fma()

Computes a multiplication and addition.

```
#include <math.h>

double fma(double x, double y, double z);

float fmaf(float x, float y, float z);

long double fmal(long double x, long double y, long double z);
```

Parameter

x

The first multiplication value.

y

The second multiplication value.

z

The addition value.

Remarks

These functions compute and return (x * y) + z. These functions may be more accurate than using the compiler's regular multiplication (*) and addition (+) operators.

The functions compute this value to virtually infinite precision and apply rounding once to the result type according to the rounding mode specified by the `FLT_ROUNDS`.

If the result cannot be properly expressed in the result type, the functions set `errno` to `EDOM` and `fpclassify(fdim(x, x))` does not return `FP_NORMAL`.

This facility may not be available on configurations of the EWL that run on platforms that do not have floating-point math capabilities.

Listing: Example of fma() usage

```
#include <math.h>
#include <stdio.h>
int main(void)
{
double k = 12;
double l = 4;
printf("|(%f - %f)| = %f\n", k, l, fma(k,l));
return 0;
}
Output:
| ( 12.000000 - 4.0000000 ) | = 8.0000000
```

Chapter 14
setjmp.h

Provides a means of saving and restoring a processor state.

The `setjmp.h` functions are typically used for programming error and low-level interrupt handling. The `setjmp()` function saves the current processor state in its `jmp_buf` argument. The `jmp_buf` type holds the processor program counter, stack pointer, and registers. The `longjmp()` function restores the processor to the state recorded in a variable of type `jmp_buf`. In other words, `longjmp()` returns program execution to a point in the program where `setjmp()` was called. Because the `jmp_buf` variable can be global, the `setjmp()` and `longjmp()` calls do not have to be in the same function body.

Variables assigned to registers through compiler optimization may be corrupted during execution between `setjmp()` and `longjmp()` calls. Avoid this situation by declaring affected variables as volatile.

14.1 Functions in setjmp.h

The setjmp.h header file provides following functions.

14.1.1 longjmp()

Restores processor state saved by `setjmp()`.

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

Parameter

`env`

A processor state initialized by a call to `setjmp()`

`val`

A non-zero value that `setjmp()` will return.

Remarks

This function restores the calling environment (in other words, it returns program execution) to the state saved by the last call to `setjmp()` that used the env variable. Program execution continues from the call to `setjmp()`, which returns the argument val.

EWL redefines `longjmp()` for AltiVec support. To avoid an undefined result, make sure that both the "to" compilation unit (the source file that calls `setjmp()`) and "from" compilation unit (the source file that calls `longjmp()`) have AltiVec code generation enabled.

After a program invokes `longjmp()`, execution continues as if the corresponding call to the `setjmp()` function had just returned the value specified by val. The `longjmp()` function cannot cause the `setjmp()` function to return the value 0. If val is 0, the `setjmp()` function returns 1.

The env variable must be initialized by a previous call to `setjmp()` before being used by `longjmp()` to avoid undefined results in program execution.This facility may not be available on some configurations of the EWL.

14.1.2 setjmp()

Restores processor state saved by `setjmp()`.

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

Parameter

`env`

The current processor state.

`val`

A non-zero value that `setjmp()` will return.

Remarks

The `setjmp()` function saves the calling environment in the env argument. The argument must be initialized by setjmp() before being passed as an argument to `longjmp()`.

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 facility may not be available on some configurations of the EWL.

Listing: Example of setjmp() and longjmp() usage

```
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>
/* Let main() and doerr() both have
* access to 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)
break;
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)
{
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
```

Chapter 15
signal.h

This header file declares data types, macros, and functions sending and receiving software interrupts.

Signals are invoked, or raised, using the raise() function. When a signal is raised its associated function is executed.

In the EWL implementation a signal can only be invoked through the `raise()` function and, in the case of the `SIGABRT` `abort()` function. 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 calls the `exit()` function. The ISO/IEC standard specifies that the `SIG` prefix used by the `signal.h` macros is reserved for future use. You should avoid using the prefix to prevent conflicts with future specifications of the Standard Library.

The type `sig_atomic_t` can be accessed as an incorruptible, atomic entity during an asynchronous interrupt. The number of signals is defined by __signal_max, defined in `signal.h`.

CAUTION

Using non-reentrant functions from within a signal handler is not recommended in any system that can throw signals in hardware. Signals are interrupts, and can be invoked at any point during a program's execution. Also, even functions designed to be re-entrant can fail if you re-enter them from a signal handler.

15.1 Functions in signal.h

The signal.h header file provides following functions.

15.1.1 raise()

Raises a signal.

```
#include <signal.h>

int raise(int sig);
```

Parameter

`sig`

A signal to raise.

Remarks

The `raise()` function calls the signal handling function associated with signal sig. The function returns a zero if the signal invocation is successful, it returns a nonzero value if it is unsuccessful. This facility may not be available on some configurations of the EWL.

15.1.2 signal()

Associates a signal handler with a signal.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Parameter

`sig`

A signal number.

`func`

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.

The sig argument is the signal number associated with the signal handling function. The signals defined in signal.h are listed in the table below.

Table 15-1. Signal Description

Macro	Signal
SIGABRT	Abort signal. Defined as a positive integer value. This signal is called by the abort() function.
SIGBREAK	Terminates the calling program.
SIGFPE	Floating point exception signal. Defined as a positive integer value.
SIGILL	Illegal instruction signal. Defined as a positive integer value.
SIGINT	Interactive user interrupt signal. Defined as a positive integer value.
SIGSEGV	Segment violation signal. Defined as a positive integer value.
SIGTERM	Terminal signal. Defined as a positive integer value. When raised this signal terminates the calling program by calling the exit() function.

The func argument is the signal handling function. This function is either programmer supplied or one of the pre-defined signal handlers listed in the table below.

Table 15-2. Signal-handling Functions

Function	Action
SIG_IGN	Nothing. It is used as a function argument in signal() to designate that a signal be ignored.
SIG_DFL	Aborts the program by calling abort().
SIG_ERR	Returned by signal() when it cannot honor a request passed to it.

When it is raised, a signal handler's execution is preceded by the invocation of signal(sig, SIG_DFL). This call to signal() 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.

This function 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 facility may not be available on some configurations of the EWL.

Listing: 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
CodeWarrior Implementation of the C Standard Library 153
5
userhandler!
Press return.
6
7
8
9
```

Chapter 16
stdarg.h

Facilities for defining functions that accept a variable number of arguments.

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 uses the `va_list` type and the macros `va_start()`, `va_arg ()`, and `va_end()`. The function uses a variable of type `va_list` to hold the list of function arguments. The macro `va_start()` initializes the `va_list` variable. Invoke this macro before accessing the function's arguments. The macro `va_arg()` retrieves each of the arguments in the argument list. Finally, use `va_end` to allow a normal return from the function.

16.1 Functions in stdarg.h

The stdarg.h header file provides following functions.

16.1.1 va_arg

Macro to return an argument value.

```
#include <stdarg.h>

type va_arg(va_list ap, type);
```

Parameter

`ap`

A variable list

`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.

16.1.2 va_copy

Copies and initializes a variable argument list.

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Parameter

`dest`

The destination for the copied variable argument list.

`src`

The argument list to copy.

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.

16.1.3 va_end

Prepare to exit the function normally.

```
#include <stdarg.h>

void va_end(va_list ap);
```

Parameter

`ap`

A variable argument list

Remarks

The `va_end` macro cleans the stack to allow a proper function return. Invoke this macro at the end of a void function or before the function's `return` statement.

16.1.4 va_start

Initialize the variable-length argument list.

```
#include <stdarg.h>

void va_start(va_list ap, lastparm);
```

Parameter

`ap`

A variable list

`lastparm`

The last named parameter.

Remarks

The `va_start()` macro initializes and assigns the argument list to ap. The lastparm parameter is the last named parameter before the ellipsis (. . .) in the function prototype.

Listing: Example of va_start Usage

```
#include <stdarg.h>

#include <stdio.h>

void multisum(int *dest, ...);

void multisum(int *dest, ...)

{

va_list ap;
```

```
int n;

int 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. */

}

int main(void)

{

int all;

all = 0;

multisum(&all, 13, 1, 18, 3, 0);

printf("%d\n", all);

return 0;

}

Output:

35
```

Chapter 17
stdbool.h

Facilities for boolean integral values.

This header file defines some convenient macros for manipulating boolean values. The table below lists these macros.

Table 17-1. Definitions in stdbool.h

Macro	Action
`bool`	Define variables of boolean type. Expands to the built-in boolean type, `Bool`.
`true`	Represent a logical true value. Expands to 1.
`false`	Represent a logical false value. Expands to 0.
`bool_true_false_are_defined`	Check for availability of `stdbool.h`facilities. When defined, expands to 1.

The facilities in this header file are only available when the compiler is configured to compile source code for C99 (ISO/IEC 9899:1999). Refer to the Build Tools Reference for information on compiling C99 source code.

Chapter 18
stddef.h

Commonly used macros and types that are used throughout the Embedded Warrior Library.

The table below lists the facilities that this header file provides.

Table 18-1. Facilities in stddef.h

Macro	Description
`NULL`	Expands to a value that represents the null pointer constant.
`offsetof(structure, member)`	Computes a value of type size_t that is the offset, in bytes, of a member from the base of its structure. If the member is a bit field the result is undefined.
`ptrdiff_t`	A data type that the compiler uses to hold the result of subtracting one pointer's value from another.
`size_t`	The unsigned type returned by the `sizeof()` operator.
`wchar_t`	A data type that is large enough to hold all character representations in the wide character set.

Chapter 19
stdint.h

Defines integer types that are suitable for specific uses and defines macros to describe their properties and manipulate literal values.

19.1 Integer Types

Defines integer types with specific properties.

The `stdint.h` header file defines several types of integer based on variations of the compiler's built-in integer types (`unsigned int`, `int`, `long int`, and so on). Unlike the general properties of the compiler's integer types, these type definitions have specific properties. The type names describe the integer's properties. Type names beginning with `int` represent types for signed integers in two's-complement form. Type names beginning with `uint` are unsigned integer types.

These integer types are:

- Exact-width integers - These types occupy the exact number of bits specified by their type names. The names of these types take the form `intN_t` and `uintN_t`, where N represents the exact number of bits in values of these types. The names of these types are `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, and `uint64_t`.
- Minimum-width integers - These types occupy at least the number of bits specified by their type names. For each of these integer types, the library guarantees that there is no integer type that has a smaller size. The names of these types take the form `int_leastN_t` and `uint_least_N_t`. The `N` specifies that the integer type has N or more bits and that no smaller integer type holds this number of bits.

  The names of these types are `int_least8_t`, `uint_least8_t`, `int_least16_t`, `uint_least16_t`, `int_least32_t`, `uint_least32_t`, `int_least64_t`, and `uint_least64_t`.

- Fastest integers - These integer types represent the optimal size for the target processor. These types are closest to the native numeric types that the target processor uses. The names of these types take the form `int_fastN_t` and `uint_fast_N_t`. The `N` specifies that the integer type has N or more bits and that no smaller integer type holds this number of bits.

  The names of these types are `int_fast8_t`, `uint_fast8_t`, `int_fast16_t`, `uint_fast16_t`, `int_fast32_t`, `uint_fast32_t`, `int_fast64_t`, and `uint_fast64_t`.

- Pointer-width integers - These types are large enough to hold a pointer value. Converting a `void` pointer value to this integer type then converting this integer value to a pointer again will result in the original pointer value.

  The names of these types are `intptr_t`, `uintptr_t`.

- Greatest-width integers - These integer types are large enough to contain any integer value. Converting any integer value to this integer type then converting this integer value to the original integer type will result in the original integer value. Values of type `intmax_t` can contain any signed integer value. Values of type `uintmax_t` can contain any unsigned integer.

19.2 Integer Limits

Defines macros that describe the numeric properties of the types defined in `stdint.h` and other files in the standard library.

Macro names beginning with `INT` specify properties for signed integer types and names beginning with `UINT` specify properties for unsigned integers. A macro name ending with `MIN` specifies the minimum value that its corresponding signed integer type may represent. An unsigned integer type's minimum value is always 0. A macro name ending with `MAX` defines the maximum value that its corresponding integer value can represent.

These limit values are:

- Exact-width integer limits - These macros describe the limits of the exact-width integer types defined in file `stddint.h`. The names of these macros are `INTn_MIN`, `INTn_MAX`, and `UINTn_MAX`, where n represents the number of bits in the corresponding exactwidth integer type.
- Minimum-width integer limits - These macros describe the limits of the minimum-width integer types defined in file `stddint.h`. The names of these macros are `INT_LEASTn_MIN`, `INT_LEASTn_MAX`, and `UINT_LEASTn_MAX`, where n represents the number of bits in the corresponding minimum-width integer type.

- Fastest integers - These macros describe the limits of the fastest-width integer types defined in file stddint.h. The names of these macros are INT_FASTn_MIN, INT_FASTn_MAX, and UINT_FASTn_MAX, where n represents the number of bits in the corresponding fastest-width integer type.
- Pointer-width integer limits - These macros describe the limits of the pointer-width integer types defined in file stddint.h. The names of these macros are INTPTR_MIN, INTPTR_MAX, and UINTPTR_MAX.
- Greatest-width integer limits - These macros describe the limits of the greatest-width integer types defined in file stddint.h. The names of these macros are INTMAX_MIN, INTMAX_MAX, and UINTMAX_MAX.
- Pointer difference limits - These macros describe the limits of the ptrdiff_t type defined in file stddef.h. The names of these macros are PTRDIFF_MIN, PTRDIFF_MAX.
- Atomic signal value limits - These macros describe the limits of the sig_atomic_t type defined in file signal.h. The names of these macros are SIG_ATOMIC_MIN, SIG_ATOMIC_MAX.
- Operator sizeof() value limits - This macro describe the limit of the size_t type defined in file stddef.h. The name of this macros is SIZE_MAX.
- Wide character value limits - These macros describe the limits of the wchar_t type defined in file stddef.h. The names of these macros are WCHAR_MIN, WCHAR_MAX.
- Wide character integer value limits - These macros describe the limits of the wint_t type defined in file wchar.h.The names of these macros are WINT_MIN, WINT_MAX.

19.3 Integer Constant Types

Defines macros for specifying integer literal values.

These macros are defined in file stdint.h. They accept a numeric literal value and convert the value to generate a numeric constant for the types defined in file stdint.h. These macros follow the same naming convention of these types. Use macros beginning with INT to specify signed integer constants and macros beginning with UINT to specify unsigned integer constants.

These macros are:

- Exact-width, minimum-width, and fastest-width integer constants - These macros take the form INTn_C(and UINTn_C(, where n represents the exact or minimum width of the integer type and x represents a numeric literal value.
- Maximum-width integer constants - These macros take the form INTMAX_C(and UINTMAX_C(, where x represents a numeric literal value.
  Listing: Example of integer constant usage

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
int main(void)
{
uint32_t i = UINT32_C(371932);
printf("%" PRId32 " %" PRIdMAX "\n", i, INTMAX_C(INTMAX_MAX));
return 0;
}
```

Chapter 20
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.

20.1 Streams

A stream is a logical abstraction that isolates input and output operations from the physical characteristics of terminals and structured storage devices.

Streams map a program's data and the data that is actually stored on the external devices. Two forms of mapping are supported, for text streams and for binary streams. Streams also provide buffering, which is a memory management technique that reads and writes large blocks of data. 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.

If a standard input/output stream 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.

20.1.1 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 newline 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 EWL, 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" .

20.1.2 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.

20.1.3 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.

20.1.4 Wide Character and Byte Character Stream Orientation

A stream is without orientation when that stream has been associated with a file but before an operation occurs on the stream.

There are two types of stream orientation for input and output:

- wide character (`wchar_t`) orientation
- byte (`char`) orientation

Once any operation is performed on that stream, that stream is first assigned its orientation by that operation. The stream's orientation 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. For example, a byte-oriented input/output function does not have an effect on a wide-oriented stream.

The predefined console streams, `stdin`, `stdout`, and `stderr` have no orientation at program startup.

20.1.5 Unicode, Wide Characters, and Multibyte Encoding

The Unicode character set known as UCS-2 (Universal Character Set containing 2 bytes) is a fixed width encoding scheme that uses 16 bits per character.

The Unicode character set 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 EWL as wide characters of type `wchar_t` and can be manipulated with the wide character functions defined in the library.

To reduce the size of a file that contains wide character text, the library offers functions to read and write wide characters using multibyte encoding. Instead of storing wide characters in a file as a sequence of wide characters, multibyte encoding takes advantage of each wide character's bit patterns to store it in one or more sequential bytes.

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. The library uses the `mbstate_t` type to record a shift state. With nonmodal 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 EWL, 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 0x00 to 0x7F the encoding is direct and only a single byte is used.

20.2 File Operations

Facilities for deleting and renaming files, and managing temporary files.

20.2.1 remove()

Deletes a file.

```
#include <stdio.h>
int remove(const char *filename);
```

Parameter

`filename`

A pointer to a character string containing the name of a file.

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.

Listing: 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;
}
```

20.2.2 rename()

Changes the name of a file.

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Parameter

`old`

The old file name.

`new`

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.

Listing: Example of rename() usage

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
char oldname[50];
char newname[50];
// get the current filename from the user
printf("Please enter the current file name.\n");
gets(oldname);
// get the new filename from the user
printf("Please enter the new file name.\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 file name.

boots.txt

Please enter the new file name.

sandals.txt
```

20.2.3 tmpfile()

Opens a temporary file.

```
#include <stdio.h>
FILE *tmpfile(void);
```

Remarks

The `tmpfile()` function creates and opens a binary file for output that is automatically removed when it is closed or when the program terminates.

The `tmpfile()` function 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 facility may not be available on configurations of the EWL that run on platforms without file systems.

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.

Listing: 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.
fclose(f);
return 0;
}
```

20.2.4 tmpnam()

Creates a unique temporary file name.

```
#include <stdio.h>
char *tmpnam(char *s);
```

Parameter

`s`

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 that `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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: 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 then delete the file.
fclose(f);
remove(tempname);
return 0;
}
```

20.3 File Access

Facilities for opening and closing files and managing file buffers.

20.3.1 fclose()

Close an open file.

```
#include <stdio.h>

int fclose(FILE *stream);
```

Parameter

`stream`

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 and real-time operating systems this function may only be applied to the `stdin`, `stdout`, and `stderr` files.

`fclose()` returns a zero if it is successful and returns an `EOF` if it fails to close a file.

Listing: 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
```

20.3.2 fdopen()

Converts a file descriptor to a stream.

```
#include <stdio.h>
FILE *fdopen(int fildes, char *mode);
FILE *_fdopen(int fildes, char *mode);
```

Parameter

`fildes`

A file descriptor, obtained from `fileno()`.

`mode`

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()`.

If it is successful, `fdopen()` returns a stream. If it encounters an error, `fdopen()` returns `NULL`.

Listing: 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;

}
```

20.3.3 fflush()

Empties a stream's buffer to the storage device.

```
#include <stdio.h>

int fflush(FILE *stream);
```

Parameter

`stream`

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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

The function `fflush()` returns `EOF` if a write error occurs, otherwise it returns zero.

Listing: 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
```

20.3.4 fopen()

Opens a file as a stream.

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Parameter

`filename`

A pointer to a character string containing the name of the file to open.

`mode`

A pointer to a character string specifying the operations be performed.

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 20.1 describes the values for 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-offile.

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

All write modes, including update mode (`"w+"`) and write/read update mode (`"wb+"`), delete data in a file when an existing file is opened.

The table below describes the values for mode.

Table 20-1. Modes for fopen()

Mode	Description
"r"	Specifies read mode. Opens an existing text file for reading only.
"w"	Specifies write mode. Creates a new text file for writing, or opens then truncates an existing file. Writing starts at the beginning of the file.
"a"	Specifies append mode. Creates a new text file for writing, or opens then truncates an existing file. Writing starts at the end-offile position.
"r+"	Specifies update mode. Opens an existing text file for reading and writing.
"w+"	Specifies update mode. Creates a new text file for reading and writing, or opens then truncates an existing file for reading and writing. The file position is at the beginning of the file.
"a+"	Specifies update mode. Creates a new text file for reading and writing, or opens then truncates an existing file for reading and writing. The file position is at the end of file.
"rb"	Specifies binary read mode. Opens an existing file for binary reading only.
"wb"	Specifies binary write mode. Creates a new file for binary writing, or opens then truncates an existing file for binary writing. Writing starts at the beginning of the file.
"ab"	Specifies binary append mode. Creates a new file for binary writing, or opens then truncates an existing file for binary writing. Writing starts at the end-of-file position.
"r+b"or "rb+"	Specifies binary update mode. Opens an existing file for reading and writing binary data.
"w+b"or "wb+"	Specifies binary update mode. Creates a new file for reading and writing binary data, or opens then truncates an existing file for reading and writing. The file position is at the beginning of the file.
"a+b"or "ab+"	Specifies binary update mode. Creates a new file for reading and writing binary data, or opens then truncates an existing file for reading and writing. The file position is at the end of the file.

`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.

Listing: 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("count.txt", "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("count.txt", "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 count.txt:
0 1 2 3 4 5 6 7 8 9 10
11
12
13
14
15
16
17
18
19
```

20.3.5 freopen()

Re-directs a stream to another file.

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE   *stream);
```

Parameter

`filename`

A pointer to a character string containing the name of the file to open.

`mode`

A pointer to a character string specifying the operations be performed.

`stream`

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`).

Listing: 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;
}
```

20.3.6 setbuf()

Changes a stream's buffer.

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Parameter

`buf`

A pointer to a new buffer.

`stream`

A pointer to a file stream.

Remarks

The `setbuf()` function allows the programmer to set the buffer 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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: Example of setbuf() usage

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
FILE *f;
char name[80];
// Get a file name 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
```

20.3.7 setvbuf()

Change the buffering scheme for a stream.

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Parameter

`stream`

A pointer to a `FILE` stream.

`buf`

A buffer for input and output

`mode`

A buffering mode.

`size`

The size of the buffer.

Remarks

The `setvbuf()` function allows the manipulation of the buffering scheme as well as the size of the buffer that the stream uses. Call this function after opening the stream but before reading from or writing to it.

The buf argument is a pointer to a character array. The size argument specifies 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 library creates its own buffer of size bytes. The mode argument specifies the buffering scheme to use:

- IOFBF: flush the buffer when it is full.
- IOLBF: flush the buffer when reading or writing an end-of-line character or when the buffer is full.
- IONBF: write or read directly to the file with no buffering.

This function returns zero if it is successful and returns a nonzero value if it fails.This facility may not be available on some configurations of the EWL.

Listing: Example of setvbuf() Usage

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
FILE *f;
char name[80];
printf("Enter the name of the file to write to.\n");
fgets(name, 80, stdin);
if ((f = fopen(name, "w")) == NULL) {
printf("Can't open file %s.\n", name);
exit(1);
}
setvbuf(f, NULL, _IOLBF, BUFSIZ);
fprintf(f, "This file is now\n");
fprintf(f, "line buffered.\n");
fclose(f);
return 0;
}
```

20.4 Formatted Input/Output

Facilities for reading and writing textual representations of binary data.

20.4.1 Reading Formatted Input

Describes conversion specifiers used for reading text.

Functions that read formatted input, `fscanf()` and its related functions, accept an argument that specifies the format of the input text that the function should expect. This argument is a pointer to a character string containing normal text, whitespace (space, tab, newline characters), and conversion specifications. The normal text specifies literal characters that must be matched in the input stream. A whitespace character tells the function to skip whitespace characters until a non-whitespace character is encountered. A conversion specification tells the function which characters in the input stream should be converted to binary values and stored.

The conversion specifications must have matching arguments in the order they appear in the formatting argument. The arguments matching the conversion specification arguments must be pointers to objects of the relevant types.

A conversion specification describes the format of a character representation and how it should be converted to a binary value. A conversion specification contains these characters, in order from left to right:

- a percent sign (%)
- optional maximum field width specification or assignment suppression
- optional argument size or type specification
- the type of conversion to perform

A conversion specification's characters must not be separated by white space. Doubling the percent sign (%%) results in the output of a single %.

After the initial percent sign (%), a conversion specification contains an optional maximum width, specified in decimal digits. This width specifies the maximum number of characters to read for the conversion.

Instead of specifying width, use the optional assignment suppression character (*) to read an item without assigning it to an argument. A conversion specification with an assignment suppression must not have a corresponding argument.

The next part of a conversion specification is optional. It specifies additional type or size information about the argument in which the conversion's result will be stored. The following table lists these optional items.

The last character of a conversion specification denotes the type of conversion to perform and the type of the argument in which the value of the conversion will be stored.

The conversion specifier `%[` begins a scanset. A scanset specifies which characters to accept. A scanset ends with a `]` character. The characters read from input that match a scanset are stored in the character string pointed to by the scanset's corresponding argument.

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 nul character is appended to the end of the stored character array.

For example, the conversion specifier %[abcdef] specifies that the scanset is `abcdef` and any of the characters "a" through "f" are to be accepted and stored. When a character outside this set is encountered, the function will stop reading and storing the scanset conversion and continue with the rest of the conversion specifiers.

For example, assuming the declaration:

```
char str[20];
```

the execution of

```
sscanf("acdfxbe", "%[abcdef]", str);
```

will store acdf in `str`. The text "xbe" 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 define an exclusionary scanset. This kind of scanset specifies that any character outside a scanset will be accepted.

For example, the statement

```
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 because it will be interpreted as introducing the members of an exclusionary scanset. For example, %[^abc] defines the exclusionary scanset "abc" but %[a^bc] defines the scanset "abc^" . %[^a^bc] and %[^^abc] both define a scanset equivalent to "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]` specifies the scanset "]abc" and `%[^]abc]` defines the exclusionary scanset "^]abc" . In any other position, the] will be interpreted as terminating the scanset. For example, `%[ab]cd]` specifies a scanset "ab" followed by the character `c` and `d`.

EWL adds an extension to the scanset interpretation rules specified in the ISO/IEC standards. EWL interprets such a use of - in a scanlist as defining a range of characters. For example, the conversion specification

```
%[a-z]
```

is the same as the

```
%[abcdefghijklmnopqrstuvwxyz]
```

To include the - character in the scanset, it must be either listed first (possibly after an initial ^) or last. Examples: %[-abc], %[abc-], %[^-abc], or %[^abc-]. 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 the result is undefined.

Table 20-2. Specifying Additional Size or Type Information About an Argument

Character	Description
h	The following d, i, o, u, x, X or n conversion will be stored in an object of type `short int` or `unsigned short int`.
hh	The following d, i, o, u, x, X or n conversion will be stored in an object of type `char` or `unsigned char`.
l	When used with integer conversion specifier, the l flag indicates that the argument points to an object of type `long int` or `unsigned long int`. When used with floating point conversion specifier, the l flag that the argument points to an object of type `double`. When used with a c or s conversion specifier, the l flag indicates that the corresponding argument is a pointer to an object of type `wchar_t`.
ll	The following integer conversion specifier has an argument that points to an object of type `long long` or `unsigned long long`.
L	The following floating-point conversion specifier has an argument that points to an object of type `long double`.
v	The argument points to an object containing an AltiVec vector of type `bool char`, `signed char`, or `unsigned char` when followed by c, d, i, o, u, x, or X. A vector of type `float`, when followed by f.
vh or hv	The argument points to an object containing an AltiVec vector of type `short`, `short bool`, `bool short`, or `pixel` when followed by c, d, i, o, u, x, or X.
vl or lv	The argument points to an object containing an AltiVec vector of type `int`, `unsigned in`, or `bool int` when followed by c, d, i, o, u, x, or X.

The table below specifies the types of conversion.

Table 20-3. Specifying the Type of Conversion

Character	Conversion
c	A character. Whitespace characters are not skipped.
d	A signed decimal number.
i	A signed decimal, octal, or hexadecimal number. The character representation can be prefixed by a plus (+) or minus (-) sign. Octal numbers begin with the zero (0) character. Hexadecimal numbers begin with 0x or 0X.
e, E, f, F, g, or G	A floating point number in scientific notation.
n	Stores the number of characters read by the function so far. Its corresponding argument must be a pointer to an `int`.

Table continues on the next page...

Table 20-3. Specifying the Type of Conversion (continued)

Character	Conversion
o	An unsigned octal.
p	A memory address. The input text must be the same as the format used by the p output format.
s	A character string. The function terminates the string with a nul-character when it encounters a whitespace character or the maximum number of characters has been read.
#s	A character string. The corresponding argument is a pointer to Pascal character string. A Pascal string is a length byte followed by the number of characters specified by the length byte. This conversion type is an extension to the ISO/IEC standards.
u	An unsigned decimal.
x or X	An unsigned hexadecimal.
[scanset]	A scanset specification.

20.4.2 Formatting Text for Output

Describes conversion specifiers used for formatting text for output.

The `printf()` function and its related functions accept a formatting argument. This argument specifies how to format the arguments that follow it. The formatting argument points to a character string containing normal text and conversion specifications. The formatting functions send normal text directly to output. The functions replace conversion specifications with formatted text based on matching arguments passed to the functions. Conversion specifications must have matching arguments in the same order in which they occur in the formatting string.

A conversion specification describes type of character representation its associated argument is to be converted to. A conversion specification contains these characters, in order from left to right:

- a percent sign (%)
- optional flags to modify the argument's format
- optional minimum field width specification
- optional precision specification
- optional argument size or type specification
- the type of conversion to perform

A conversion specification's characters must not be separated by white space. Doubling the percent sign (%%) results in the output of a single %.

After the initial percent sign, optional flags modify the formatting of the argument; the argument can be left- or right-justified, and numerical values can be padded with zeroes or output in alternate forms. More than one flag character may appear in a conversion specification.The following table 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 value that EWL allows is 509.

The optional precision width is a period character (.) followed by decimal digit string. The default precision is 6 digits after the decimal point. For floating point values, the precision width specifies the number of digits to print after the decimal point. For integer values, the precision width works the same as, 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.

An optional conversion modifier specifies additional information about the corresponding argument's type or size. These characters appear before the conversion specifier. The table below describes these conversion modifiers.

The terminating characters, the conversion type, specify the conversion to apply to the matching argument. The table below describes the conversion types.

Table 20-4. Flags for Modifying Conversion Specifies

Character	Description
-	The conversion will be left-justified. By default, arguments are right-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, this flag has no effect. For s conversion types, the matching argument is considered to be a pointer to a Pascal string and is output as a character string. For o conversion types, this flag prefixes the conversion with a 0. For x conversion types, the conversion is

Table continues on the next page...

Table 20-4. Flags for Modifying Conversion Specifies (continued)

Character	Description
	prefixed with a 0x. For e, E, f, g, and G conversions, this flag forces a decimal point in the output. For g and G conversions, 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

The table below describes these conversion modifiers.

Table 20-5. Specifying the Size or Type of an Argument

Character	Description
h	The following d, i, o, u, x, X or n conversion will be stored in an object of type `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	An AltiVecvector of type `bool char`, `signed char`, or `unsigned char` when followed by c, d, i, o, u, x, or X. A vector of type `float`, when followed by f.
vh or hv	An AltiVecvector of type `short`, `short bool`, `bool short`, or `pixel` when followed by c, d, i, o, u, x, or X.
vl or lv	An AltiVecvector of type `int`, `unsigned in`, or `bool int`when followed by c, d, i, o, u, x, or X.

The table below lists the conversion types.

Table 20-6. Specifying the Type of Conversion

Character	Conversion
c	A character.

Table continues on the next page...

Table 20-6. Specifying the Type of Conversion (continued)

Character	Conversion
d or i	A signed decimal number.
`e or E`	The floating point argument (of type `float`or `double`) is output in scientific notation. This specifier places one digit before the decimal point and appends a representation of a base-10 exponent. If the precision width is 0, no decimal point is output. The exponent value 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.
`f or F`	The corresponding floating point argument (of type `float` or `double`) is printed in decimal notation. If the precision width is 0, the decimal point is not printed. For the f conversion specifier, an argument of type `double` that contains infinity produces `inf`. A `double` argument representing the not-a-number value produces `nan`. The F conversion specifier uses uppercase letters, producing `INF` and `NAN` instead.
`g or 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.
`n`	Stores the number of items output by the function so far. Its corresponding argument must be a pointer to an `int`.
`o`	An unsigned octal.
`p`	A pointer. The argument is output using the X conversion type format.
`s`	A character string. The corresponding argument must be a pointer to a nul terminated character string. The nul character is not output.
`#s`	A character string. The corresponding argument is a pointer to Pascal character string. A Pascal string is a length byte followed by the number of characters specified by the length byte. This conversion type is an extension to the ISO/IEC standards.
`u`	An unsigned hexadecimal. The lowercase x uses lowercase letters (abcdef) while uppercase X uses uppercase letters (ABCDEF).
`x` or `X`	An unsigned hexadecimal.

20.4.3 fprintf()

Formats then writes text to a stream.

```
#include <stdio.h>

int fprintf(FILE *stream,

const char *format, ...);
```

Parameter

`stream`

A pointer to a file stream.

`format`

A pointer to a character string containing format information.

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.

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. To flush a stream's buffer, use `fflush()` or one of the file positioning operations (`fseek()`, `fsetpos()`, or `rewind()`).

This facility may not be available on configurations of the EWL that run on platforms without file systems.

The function returns the number of arguments written or a negative number if an error occurs.

Listing: Example of fprintf() 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
```

20.4.4 fscanf()

Read formatted text from a stream.

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Parameter

`stream`

A pointer to a file stream.

`format`

A pointer to a character string containing format information.

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.

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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: 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.
```

20.4.5 printf()

Output formatted text.

```
#include <stdio.h>
int printf(const char *format, ...);
```

Parameter

`format`

A pointer to a character string containing format information.

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()` returns the number of arguments that were successfully output or returns a negative value if it fails.

Listing: Example of printf() usage

```
#include <stdio.h>
int main(void)
{
int i = 25;
char c = 'M';
short int d = 'm';
static char s[] = "woodworking!";
static char pas[] = "\pwoodworking 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:
woodworking! printf() demonstration:
The last line contained 37 characters
Pascal string output: woodworking again!
25 19 000019 31
25
M 77 0000000077
m 109 0000000109
$49.95
1038.11
1038.110
1038.1101
1038.11005
```

Listing: 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 output 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
```

20.4.6 scanf()

Converts formatted text from `stdin` to binary data.

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Parameter

`format`

The format string.

Remarks

The `scanf()` function reads text and converts the text read to programmer specified types.

`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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: 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: 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
```

20.4.7 sscanf()

Converts formatted text in a character string to binary data.

```
#include <stdio.h>
int sscanf(char *s, const char *format, ...);
```

Parameter

`s`

A pointer to the character string from which to convert text to binary data.

`format`

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.

`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 facility may not be available on configurations of the EWL that run on platformswithout file systems.

Listing: Example of sscanf() Usage

```
#include <stdio.h>
int main(void)
{
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
```

20.4.8 snprintf()

Formats a character string array.

```
#include <stdio.h>
int snprintf(char * s, size_t n, const char * format, ...);
```

Parameter

`s`

A pointer to the character string from which to store the formatted text.

`n`

Maximum number of characters to store in s.

`format`

A pointer to a 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.

`snprintf()` returns the number of characters that would have been assigned to s, had n been sufficiently large, not including the nul character or a negative value if an encoding error occurred. Thus, the nul-terminated output will have been completely written if and only if the returned value is nonnegative and less than n.

This facility may not be available on configurations of the EWL that run on platformswithout file systems.

Listing: 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
```

20.4.9 sprintf()

Formats a character string array.

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Parameter

s

A pointer to the character string to write to.

format

A pointer to 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. `sprintf()` returns the number of characters assigned to `s`, not including the nul character.

Be careful when using this function. It can be a source for serious buffer overflow bugs. Unlike `snprintf()`, the programmer cannot specify a limit on the number of characters to store in s.

This facility may not be available on configurations of the EWL that run on platformswithout file systems.

Listing: Example of sprintf() Usage

```
#include <stdio.h>

int main(void)

{

int i = 1;

char s[] = "woodworking";

char dest[50];

sprintf(dest, "%s is number %d!", s, i);

puts(dest);

return 0;

}

Output:

woodworking is number 1!
```

20.4.10 vfprintf()

Writes formatted output to a stream.

```
#include <stdarg.h>

#include <stdio.h>

int vfprintf(FILE *stream, const char *format, va_list arg);
```

Parameter

`stream`

A pointer to a file stream.

`format`

A format string.

`arg`

An 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.

`vfprintf()` returns the number of characters written or EOF if it failed.

This facility may not be available on configurations of the EWL that run on platformswithout file systems.

Listing: 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
```

20.4.11 vfscanf()

Converts formatted text from a stream.

```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE *stream, const char *format, va_list arg);
```

Parameter

`stream`

A pointer to a file stream.

`format`

A format string.

`arg`

An 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.

`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 facility may not be available on configurations of the EWL that run on platformswithout file systems.

Listing: Example of vfprintf() 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 filef
printf(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.
```

20.4.12 vprintf()

Writes formatted output to stdout.

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Parameter

`format`

A pointer to a format string.

`arg`

An 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.

`vprintf()` returns the number of characters written or a negative value if it failed.

This facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: 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 *
```

20.4.13 vscanf()

Converts formatted text read from stdin into binary data.

```
#include <stdio.h>

int vscanf(const char *format, va_list arg);
```

Parameter

`format`

A pointer to a format string.

`arg`

An argument list.

Remarks

The `scanf()` function works identically to the `scanf()` function. Instead of the variable list of arguments that can be passed to `scanf()`, `vscanf()` accepts its arguments in an array of type `va_list` processed by the `va_start()` macro from the `stdarg.h` header file.

`vscanf()` returns the number items converted successfully or `EOF` if it failed before it could process the first conversion.

This facility may not be available on configurations of the EWL that run on platforms without file systems.

20.4.14 vsnprintf()

Formats a character string.

```
#include <stdarg.h>

#include <stdio.h>

int vsnprintf(char * s, size_t n, const char * format, va_list arg);
```

Parameter

`s`

A pointer to a character string in which to store the formatted text.

`n`

Maximum number of characters to store in s.

`format`

A pointer to a format string.

`arg`

An 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.

`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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: 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
```

20.4.15 vsprintf()

Writes formatted output to a string.

```
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

Parameter

s

A pointer to a character string in which to store the formatted text.

`format`

A pointer to a format string.

`arg`

An 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.

Be careful when using this function. It can be a source for serious buffer overflow bugs. Unlike `vsnprintf()`, the programmer cannot specify a limit on the number of characters to store in s.

`vsprintf()` returns the number of characters written to `s` not counting the terminating null character. Otherwise, `EOF` on failure.

This facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: Example Of vsprintf() Usage

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

20.4.16 vsscanf()

Reads formatted text from a character string.

```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * s, const char * format, va_list arg);
```

Parameter

`s`

A pointer to a character string from which to read formatted text.

`format`

A format string.

`arg`

An 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, w hich must have been initialized by the va_start() macro from the stdarg.h h eader file. The vfscanf() does not invoke the va_end macro.

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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: Example Of vsscanf() Usage

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

20.5 Character Input/Output

Facilities for reading and writing character data.

20.5.1 fgetc()

Read the next character from a stream.

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Parameter

`stream`

A pointer to a file stream.

Remarks

The `fgetc()` function reads the next character from `stream` and advances its file position indicator.

`fgetc()` returns the character as an unsigned char converted to an `int` . If the end-offile 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 facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

Listing: 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
```

20.5.2 fgets()

Reads a character string from a stream.

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Parameter

`s`

A pointer to an array that will contain the string.

`n`

The maximum number of characters to read.

`stream`

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.

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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

`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.

20.5.3 fputc()

Writes a character to a stream.

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Parameter

`c`

The character to write.

`stream`

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()`).

This facility may not be available on configurations of the EWL that run on platforms without file systems.

`fputc()` returns the character written if it is successful, and returns `EOF` if it fails.

Listing: 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
```

20.5.4 fputs()

Writes a character array to a stream.

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Parameter

`s`

A pointer to the character string to output.

`stream`

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()`).

This facility may not be available on configurations of the EWL that run on platforms without file systems.

`fputs()` returns a zero if successful, and returns a nonzero value when it fails.

Listing: 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
```

20.5.5 getc()

Reads the next character from a stream.

```
#include <stdio.h>

int getc(FILE *stream);
```

Parameter

`stream`

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.

Listing: 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
```

20.5.6 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.

Listing: Example of getchar() usage

```
#include <stdio.h>

int main(void)
```

```
{
// We use int, not char, because EOF value is outside
// of char's range.
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!
```

20.5.7 gets()

Read a character array from `stdin`.

```
#include <stdio.h>
char *gets(char *s);
```

Parameter

`s`

The string being written 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-offile is reached.

This function reads and ignores the newline character (`'\n'`). The newline character is not stored `s`. The function terminates the character string with a null character.

This function can be the cause of buffer overflow bugs because it does not limit the number of characters that it stores in s. use the function `fgets()` instead, which allows the programmer to specify the length of the character string where input will be stored.

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.

20.5.8 putc()

Write a character to a stream.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Parameter

`c`

The character to write to a file.

`stream`

A pointer to a 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.

Listing: 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
```

20.5.9 putchar()

Write a character to `stdout`.

```
#include <stdio.h>
 int putchar(int c);
```

Parameter

`c`

The character to write to the `stdout` file.

Remarks

The `putchar()` function outputs `c` to `stdout` and advances the stream's file position indicator.

The `putc()` works identically to the `fputc()` function, except that it is written as a macro.

This function returns the character written when successful and returns `EOF` when it fails.

Listing: Example of putchar() Usage

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
static char test[] = "flying fish and quail eggs\n";
int i;
/* Output the test character array. */
for (i = 0; test[i] > 0; i++)
{
if (putchar(test[i]) == EOF)
break;
}
return 0;
}
Output:
flying fish and quail eggs
```

20.5.10 puts()

Write a character string to `stdout`.

```
#include <stdio.h>
int puts(const char *s);
```

Parameter

`s`

A pointer to a nul-terminated character string.

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.

Listing: 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
```

20.5.11 ungetc()

Places a character back into a stream.

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Parameter

c

The character to return to a stream.

`stream`

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 facility may not be available on configurations of the EWL that run on platforms without file systems.

Listing: 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("double.txt", "w+")) == NULL) {
printf("Can't open.\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..
```

20.6 Binary Input/Output

Facilities for reading and writing raw data.

20.6.1 fread()

Reads binary data from a stream.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size,
size_t nmemb, FILE *stream);
```

Parameter

`ptr`

A pointer to the array in which the data will be stored.

`size`

The size of an array element, in characters.

`nmemb`

The maximum number of elements to read.

`stream`

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.

This facility may not be available on configurations of the EWL that run on platforms without file systems. `fread()` returns the number of items read successfully.

Listing: 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
```

20.6.2 fwrite()

Writes binary data to a stream.

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
size_t nmemb, FILE *stream);
```

Parameter

`ptr`

A pointer to the array to write to the stream.

`size`

The size of an array element, in characters.

`nmemb`

The maximum number of elements to write.

`stream`

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()`).

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system. `fwrite()` returns the number of items successfully written to `stream`.

20.7 File Positioning

Facilities for getting and setting the position in the file for the next read or write operation.

20.7.1 fgetpos()

Gets a stream's current file position indicator value.

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Parameter

`stream`

A pointer to a file stream.

`pos`

A pointer to a file position.

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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

`fgetpos()` returns zero when successful and returns a nonzero value when it fails.

Listing: 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
```

20.7.2 fseek()

Move the file position indicator.

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Parameter

`stream`

A pointer to a file stream.

`offset`

The number of characters to move.

`whence`

The starting position to move from.

Remarks

The `fseek()` function moves the file position indicator to allow random access to a file.

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` sets the file position indicator to offset bytes from the beginning of the file. In this case offset must be equal or greater than zero.
- `SEEK_CUR` sets the file position indicator to offset bytes from its current position. The offset argument can be a negative or positive value.
- `SEEK_END` moves the file position indicator to 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.

The function has limited use when used with MS-DOS text files opened in text mode because of carriage return/line feed translations. The seek operations may be incorrect near the end of the file due to end-of-file translations.

The only 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`.

This facility may not be available on configurations of the EWL that run on platforms without file systems.

`fseek()` returns zero if it is successful and returns a nonzero value if it fails.

Listing: 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);
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
```

20.7.3 fsetpos()

Set the file position indicator.

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Parameter

`stream`

A pointer to a stream.

`pos`

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.

This facility may not be available on configurations of the EWL that run on platforms without file systems.

`fsetpos()` returns zero if it is successful and returns a nonzero value if it fails.

20.7.4 ftell()

Returns the value of the file position indicator.

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Parameter

`stream`

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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

`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.

20.7.5 rewind()

Resets the file position indicator to the beginning of the file.

```
#include <stdio.h>
void rewind(FILE *stream);
```

Parameter

`stream`

A pointer for 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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

Listing: Example of rewind() Usage

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
FILE *f;
char filename[80], buf[80];
// get a file name from the user
printf("Enter a file name 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 file name to read.
itwerks
Reading first line twice.
Once: flying fish and quail eggs
Twice: flying fish and quail eggs
```

20.8 File Error Handling

Facilities for checking for and reporting error conditions for files.

20.8.1 clearerr()

Clears a stream's end-of-file and error status.

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Parameter

`stream`

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.

Listing: 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.
```

20.8.2 feof()

Checks the end-of-file status of a stream.

```
#include <stdio.h>
int feof(FILE *stream);
```

Parameter

`stream`

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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

`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.

Listing: 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.
```

20.8.3 ferror()

Check the error status of a stream.

```
#include <stdio.h>
int ferror (FILE *stream);
```

Parameter

`stream`

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.

This facility may have limited capability on configurations of the EWL that run on platforms that do not have console input/output or a file system.

`ferror()` returns a nonzero value if `stream`'s error status is on, and returns zero if `stream`'s error status is off.

Listing: 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.
```

20.8.4 perror()

Output an error message to `stderr`.

```
#include <stdio.h>
void perror(const char *s);
```

Parameter

`s`

Error message to print.

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`.

Listing: 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
```

Input and Output for Wide Characters and Multibyte Characters

Facilities for reading and writing wide and multibyte characters.

20.9 Input and Output for Wide Characters and Multibyte Characters

Facilities for reading and writing wide and multibyte characters.

20.9.1 fwide()

Determines the orientation of a stream.

```
#include <stdio.h>
int fwide(FILE *stream, int orientation);
```

Parameter

`stream`

A pointer to a stream.

`orientation`

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 has no orientation, stream is made to be wide-oriented. If the value of orientation is less than zero and stream has no orientation, stream is made to be byte-oriented. Otherwise, if value of `orientation` is zero then 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.

Listing: 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
```

20.9.2 _wfopen()

Opens a file with a wide character file name as a stream.

```
#include <stdio.h>
FILE *_wfopen(const wchar_t *wfilename, const wchar_t  *wmode);
```

Parameter

`wfilename`

A pointer to a wide-character file name.

`wmode`

A pointer to a wide-character-encoded opening mode.

Remarks

The `wfopen()` function is a wide character implementation of `fopen()`.

20.9.3 _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);
```

Parameter

`wfilename`

A pointer to a wide-character file name.

`wmode`

A pointer to a wide-character opening mode.

`stream`

A pointer to a file stream.

Remarks

The `wfreopen()` function is a wide character implementation of `freopen()`.

20.9.4 _wremove()

Deletes a file.

```
#include <stdio.h>
int remove(const wchar_t *wfilename);
```

Parameter

`wfilename`

A pointer to a wide-character file name.

Remarks

The `fremove()` function is a wide character variation of `remove()`.

20.9.5 _wrename()

Changes the name of a file.

```
#include <stdio.h>

int rename(const char *old, const wchar_t *new);
```

Parameter

`old`

A pointer to a wide-character string containing the old file name.

`new`

A pointer to a wide-character string containing the new file name.

Remarks

The `wrename()` function implements a wide character variation of `rename()`.

20.9.6 _wtmpnam()

Creates a unique temporary file name using wide characters.

```
#include <stdio.h>

wchar_t *_wtmpnam(wchar_t *ws);
```

Parameter

`ws`

A pointer to a 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()`.

Chapter 21
stdlib.h

The `stdlib.h` header file provides groups of functions for string conversion, pseudorandom number generation, memory management, environment communication, searching and sorting, multibyte character conversion, and integer arithmetic.

21.1 Numeric Conversion

Converting numeric values to and from textual representations.

21.1.1 atof()

Converts a character string to a numeric value of type double.

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

Parameter

`nptr`

A pointer to the string to convert.

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`.

Listing: 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
```

21.1.2 atoi()

Converts a character string to a value of type int.

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

Parameter

`nptr`

A pointer to the string to convert.

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.

21.1.3 atol()

Converts a character string to a value of type long.

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

Parameter

nptr

A pointer to 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.

21.1.4 atoll()

Converts a character string to a value of type longlong.

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

Parameter

nptr

A pointer to 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

```
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.

atoll() returns an integer value of type long long.

This function is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.1.5 strtod()

Converts a character array to a floating point value of type double.

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

Parameter

nptr

A pointer to a null-terminated character string to convert.

endptr

A pointer to a character pointer or a null pointer.

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 (examples: 103.578, 1.03578e+02, or 0x1.9efef9p+6).

If the endptr argument is not a null pointer, then strtod() assigns 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`.

Listing: 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+6777";
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|
```

21.1.6 strtof()

Character array conversion to floating point value of type `float`.

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

Parameter

`nptr`

A pointer to a nul-terminated character string to convert.

`endptr`

A pointer to a position in nptr that 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 (examples: `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`.

21.1.7 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);
```

Parameter

`nptr`

A pointer to a nul-terminated character string to convert.

`endptr`

A pointer to the position in nptr that could not be converted.

`base`

A numberic 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`.

Listing: 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[] = "4733777";

static char sb[] = "0x10*****";

static char sc[] = "66E00M???";

static char sd[] = "Q0N50Mabcd";

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|
```

21.1.8 strtoll()

Character array conversion to integer value of type `long long int`.

```
#include <stdlib.h>

unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Parameter

`nptr`

A pointer to a nul-terminated character string to convert.

`endptr`

A pointer to the position in nptr that could not be converted.

`base`

A numberic 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.

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 is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.1.9 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);
```

Parameter

`nptr`

A pointer to a nul-terminated character string to convert.

`endptr`

A pointer to the position in nptr that could not be converted.

`base`

A numberic 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.

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" l` ocale, 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 is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.2 Pseudo-Random Number Generation

Generating and managing pseudo-random numbers.

21.2.1 rand()

Generates 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 pseudorandom 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.

Listing: 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");
}
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
```

21.2.2 srand()

Sets the seed for the pseudo-random number generator.

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

Parameter

seed

An initial 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.

21.3 Memory Management

Reserving and releasing heap memory.

21.3.1 _calloc()

Allocates space for a group of objects.

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

Parameter

`nmemb`

number of elements to allocate

`elemsize`

size of an element

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.

Listing: Example of _calloc() usage

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
```

```
static char s[] = "woodworking compilers";
char *sptr1, *sptr2, *sptr3;
/* Allocate the memory three different ways... */
/* One: allocate a 30-character block of */
/* uninitialized memory. */
sptr1 = (char *) _malloc(30);
strcpy(sptr1, s);
printf("Address of sptr1: %p\n", sptr1);
/* Two: allocate 20 characters of unitialized memory. */
sptr2 = (char *) _malloc(20);
printf("sptr2 before reallocation: %p\n", sptr2);
strcpy(sptr2, s);
/* Re-allocate 10 extra characters */
/* (for a total of 30 characters). */
/* 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
woodworking compilers
woodworking compilers
woodworking compilers
```

21.3.2 _free()

Releases previously allocated memory to heap.

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

Parameter

`ptr`

a pointer to allocated memory

Remarks

The `_free()` function releases a previously allocated memory block. 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. A pointer must only be deallocated once.

21.3.3 _malloc()

Allocates a block of heap memory.

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

Parameter

`size`

The number of contiguous characters to allocate.

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.

_malloc() returns a pointer to the first byte of the allocated block if it is successful and return a null pointer if it fails.

21.3.4 _realloc()

Changes the size of an allocated block of heap memory.

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

Parameter

ptr

A pointer to an allocated block of memory.

size

The new size to allocate.

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().

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.

Listing: Example of _realloc() usage

```
#include <stdlib.h>

int main(void

{

int* vec;

int* newvec

int count;

count = 3;

vec = _calloc(count, sizeof(int));

if (vec == NULL) {

printf("Could not allocate.\n");

exit(1);

}

vec[0] = 32;

vec[1] = 39;

vec[2] = 41;

/* Assign _realloc()'s result to newvec to preserve

vec's value if the call to _realloc() fails. */

count = 5;

newvec = _realloc(vec, count * sizeof(int));

if (newvec != NULL)

{

vec = newvec;

vec[3] = 58;

vec[4] = 82;

}

return 0;

}
```

21.3.5 vec_calloc()

Clears and allocates memory on a 16 byte alignment.

```
#include <stdlib.h>

void *vec_calloc(size_t nmemb, size_t size);
```

Parameter

`nmemb`

number of elements to allocate

`elemsize`

size of an element

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 is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.3.6 vec_free()

Frees memory allocated by `vec_malloc, vec_calloc` and `vec_realloc`

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

Parameter

`ptr`

a pointer to 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.

21.3.7 vec_malloc()

Allocates memory on a 16 byte alignment.

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

Parameter

`size`

The size, in characters, 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 is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.3.8 vec_realloc()

Reallocates memory on a 16 byte alignment.

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

Parameter

`ptr`

A pointer to an allocated block of memory.

`size`

The new size to allocate.

Remarks

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 is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.4 Environment Communication

Interacting with the operating system.

21.4.1 abort()

Abnormallly terminates program.

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

Remarks

The `abort()` function raises the `SIGABRT` signal and quits the program to returnto the operating system.

The `abort()` function will not terminate the program if a programmer-installed signal handler uses `longjmp()` instead of returning normally.

Listing: 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.
```

21.4.2 atexit()

Install a function to be executed at a program's exit.

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

Parameter

`func`

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.

Listing: Example of atexit() Usage

```
#include <stdlib.h>
#include <stdio.h>
/* Prototypes */
void prompt(void);
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 prompt(void)
{
int c;
printf("Press Return.");
c = getchar();
}
void first(void)
{
printf("First exit function.\n");
prompt();
}
void second(void)
{
printf("Second exit function.\n");
prompt();
}
void third(void)
{
printf("Third exit function.\n");
prompt();
}
Output:
Third exit function.
Press Return.
Second exit function.
Press Return.
First exit function.
Press Return.
```

21.4.3 _Exit()

Terminates program normally.

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

Parameter

`status`

The exit error value.

Remarks

This function does not return to the caller. Instead, it passes a status value to the host operating system, like the function `exit()`.

The effects on open stream buffers is implementation defined.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

21.4.4 exit()

Terminates a program normally.

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

Parameter

`status`

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 returns to the operating system with the value in `status`.

Listing: 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;
}
```

21.4.5 getenv()

Accesses the environment list.

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

Parameter

`name`

A pointer to a character string containing the variable to search for.

Remarks

`getenv()` returns a pointer to the value of the environment value, encoded as a character string. It returns NULL on failure.

Listing: 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;

}

Output:

path = c:\Program Files\Freescale\codewarrior;c:\WINNT\system32
```

21.4.6 _putenv()

Enters an item into the environment list.

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

Parameter

`name`

A pointer to the item to add to the list.

Remarks

The function `putenv()` returns `NULL` on success or minus one on failure to enter the environmental variable.

This function is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.4.7 system()

Environment list assignment.

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

Parameter

`string`

A pointer to a character string containing a command for the host operating system.

Remarks

The `system()` function returns zero if successful or minus one on failure.

21.5 Searching and Sorting

Searching and sorting array elements.

21.5.1 bsearch()

Uses the binary search algorithm to make an efficient search for an item in a sorted array.

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

Parameter

`key`

Search criteria.

`base`

The array to be searched.

`num`

The number of elements in the array.

`size`

The size of each element in the array.

`compare`

A pointer to a comparison function.

Remarks

The `key` argument points to the item you want to search for.

The `base` argument points to the first element of the array to search. This array must already be sorted in ascending order, based on the comparison requirements of the function pointed to by the `compare` argument.

The `compare` argument is a pointer to a programmer-supplied function that `bsearch()` calls to compare two elements of the array. This comparison function takes two element pointers as arguments. If the element that the first argument points is equal to the element that the second argument points to, the comparison function must return `0`. If the first argument is less than the second, the comparison function must return a negative number. If the first argument is greater than the second argument, the function must return a positive number.

`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`).

Listing: 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
/* Telephone directory record */
typedef struct direntry{
```

```
char lname[15]; /* Key field, see comp(). */
char fname[15];
char phone[15];
} direntry;
int comp(const direntry *, const direntry *);
direntry *look(char *);
direntry directory[MAXDIR];
int reccount = 0;
int main(void)
{
direntry *next = directory;
direntry *ptr = NULL;
char lookstr[15];
printf("Telephone directory program.\n");
printf("Enter blank last name when done.\n");
do {
printf("\nLast name: ");
gets(next->lname);
if (strlen(next->lname) == 0)
break;
printf("First name: ");
gets(next->fname);
printf("Phone number: ");
gets(next->phone);
++reccount;
} while (reccount < MAXDIR));
printf("Thank you. Now sorting. . .\n");
/* Sort the array using qsort(). */
qsort(directory, reccount, sizeof(direntry), comp);
printf("Enter last name to search for,\n");
printf("blank to quit.\n");
printf("\nLast name: ");
gets(lookstr);
while (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(direntry), 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:
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.
```

21.5.2 qsort()

Sorts an array.

```
#include <stdlib.h>
void qsort(void *base,size_t nmemb, size_t size,
int (*compare) (const void *, const void *))
```

Parameter

`base`

A pointer to the array to be sorted.

`nmemb`

The number of elements.

`size`

The size of a single element, expressed as a number of characters.

`compare`

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()`.

- 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.

21.6 Integer Arithmetic

Absolute value and division operations for integer values.

21.6.1 abs()

Computes the absolute value of an integer.

```
#include <stdlib.h>

int abs(int i);
```

Parameter

`i`

Value from which to compute the absolute value.

Remarks

`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.

Listing: Example of abs()

```
#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.
```

21.6.2 div()

Computes the integer quotient and remainder.

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

Parameter

`numer`

The numerator.

`denom`

The denominator.

Remarks

The `div_t` type is defined in `div_t.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.

Listing: 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
```

21.6.3 labs()

Computes long integer absolute value.

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

Parameter

j

The value to be computed.

Remarks

This function returns the absolute value of its argument as a value of type `long int`.

21.6.4 ldiv()

Computes the long integer quotient and remainder.

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

Parameter

`numer`

The numerator.

`denom`

The denominator.

Remarks

The `ldiv_t` type is defined in `div_t.h` as typedef struct { long int quot, rem; } ldiv_t;

`ldiv()` divides denom into numer and returns the quotient and remainder as a `ldiv_t` type.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

21.6.5 llabs()

Computes long long integer absolute value.

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

Parameter

`j`

Value from which to compute the absolute value.

Remarks

`llabs()` 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 is not specified in the ISO/IEC standards. It is an extension of the standard library.

21.6.6 lldiv()

Computes the long long integer quotient and remainder.

```
#include <stdlib.h>

lldiv_t lldiv(long long numer, long long denom);
```

Parameter

`numer`

The numerator.

`denom`

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 a `lldiv_t` type.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries

21.7 Wide-Character and Multibyte Character Conversion

Converting between wide and multibyte characters and character strings.

21.7.1 mblen()

Computes the length of an encoded multibyte character string, encoded as defined by the LC_CTYPE category of the current locale.

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Parameter

s

A pointer to the multibyte character string to measure.

n

The maximum size of the multibyte character string.

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 does or does 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.

21.7.2 mbtowc()

Translates a multibyte character to a wchar_t type.

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Parameter

pwc

The wide-character destination.

s

The string to convert.

n

The number of wide characters in pwc.

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.

21.7.3 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);
```

Parameter

s

A pointer to a multibyte character string.

wchar

A wide character to convert.

Remarks

The EWL 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.

21.7.4 mbstowcs()

Converts a multibyte character array encoded as defined by the LC_CTYPE category of the current locale to a wchar_t array.

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Parameter

pwcs

A pointer to the wide-character destination of the conversion.

s

A pointer to the multibyte character string to convert.

n

The number of wide characters in pwc.

Remarks

This function 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 it encounters an invalidly encoded character, 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.

21.7.5 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);
```

Parameter

`s`

A pointer to a multibyte character string destination for the conversion.

`pwcs`

A pointer to the wide-character string to convert.

`n`

The maximum number of wide characters in pwcs to convert.

Remarks

The EWL 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 nul 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.

Chapter 22
string.h

This header file declares functions for comparing, copying, concatenating, and searching character strings and memory.

The functions in this header file follow a naming convention that describes how the function varies its operation and on what kind of data it operates on. Each function begins with one of these prefixes:

- str - Operates on nul-terminated character strings.
- strn - Accepts a parameter that specifies the maximum length of the string to operate on.
- stri - Ignores letter case. These functions are non-standard.
- mem - Operates on a char-based block of memory.

22.1 Copying Characters

Copying characters between memory areas and null-terminated strings.

22.1.1 memcpy()

Copies a contiguous memory block.

```
#include <string.h>

void *memcpy(void *dest, const void *source, size_t n);
```

Parameter

`dest`

A pointer to an area of memory to copy to.

`source`

A pointer to an area of memory to copy.

`n`

The maximum number of characters to copy from source to dest.

Remarks

This function copies the first n characters from the item pointed to by source to the item pointed to by dest. The function has undefined behavior if the areas pointed to by dest and source overlap. Use the `memmove()` function to reliably copy overlapping memory blocks.

The function returns the value of dest. This facility may not be available on some configurations of the EWL.

22.1.2 memmove()

Copy an overlapping contiguous memory block.

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

Parameter

`dest`

A pointer to an area of memory to copy to.

`source`

A pointer to an area of memory to copy.

`n`

The maximum number of characters to copy from source to destination.

Remarks

This function copies the first n characters of the item pointed to by source to the item pointed to by dest. Unlike `memcpy()`, the `memmove()` function safely copies overlapping memory blocks.

This function returns the value of `dest`.

This facility may not be available on some configurations of the EWL.

22.1.3 strcpy()

Copies one character array to another.

```
#include <string.h>

char *strcpy(char *dest, const char *source);
```

Parameter

`dest`

A pointer to a memory area to copy the character string to.

`source`

A pointer to the character string to copy.

Remarks

This function 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 will be null-terminated.

This function has undefined behavior if the arrays pointed to by dest and source overlap. This function can be the cause of buffer overflow bugs in your program if the memory area pointed to by dest is not large enough to contain source. Consider using `strncpy()` instead.

The function returns the value of dest.

This facility may not be available on some configurations of the EWL.

Listing: Example of strcpy() usage

```
#include <string.h>

#include <stdio.h>

int main(void)

{

char s[] = "woodworking";

/* String d must be large enough to contain string s. */
```

```
char d[30] = "";
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=woodworking
d=
s=woodworking
d=woodworking
```

22.1.4 strncpy()

Copies a specified number of characters.

```
#include <string.h>
char *strcpy(char *dest, const char *source, size_t n);
```

Parameter

`dest`

A pointer to a memory to copy the character string to.

`source`

A pointer to the character string to copy to dest.

`n`

The maximum number of characters 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. This function has undefined behavior if dest and source overlap.

If a null character ('\0') is reached in source before n characters have been copied, the function continues padding dest with null characters until n characters have been copied to dest.

The function does not terminate dest with a null character if n characters are copied fromsource before reaching a null character.

The function returns the value of dest.

This facility may not be available on some configurations of the EWL.

Listing: Example of strncpy() usage

```
#include <string.h>
#include <stdio.h>
#define MAX 50
int main(void)
{
char d[MAX];
char s[] = "123456789ABCDEFG";
strncpy(d, s, 9);
/* Terminate d explicitly in case strncpy() did not. */
d[MAX] = '\0';
puts(d);
return 0;
}
Output:
123456789
```

22.2 Concatenating Characters

Appending characters to null-terminated strings.

22.2.1 strcat()

Concatenates two character arrays.

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

Parameter

`dest`

The destination string.

`source`

The source string to append to dest.

Remarks

This 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. The function terminates the resulting character array with a null character.

The function returns the value of dest.

This facility may not be available on some configurations of the EWL.

Listing: Example of strcat() usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
/* Specify a destination string that has enough room to append
s2. */
char s1[100] = "The quick brown fox ";
char s2[] = "jumped over the lazy dog.";
strcat(s1, s2);
puts(s1);
return 0;
}
Output:
The quick brown fox jumped over the lazy dog.
```

22.2.2 strncat()

Appends up to a maximum of a specified number of characters to a character array.

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

Parameter

`dest`

A pointer to a character to append to.

`source`

A pointer to the character string to append to dest.

`n`

The maximum number of characters to append from source.

Remarks

This function appends up to a maximum of n characters from the character array pointed to by source and an extra null character ('\0'). 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 the function encounters a null character in source before n characters have been appended, the function appends a null character to dest and stops.

The function returns the value of dest.

This facility may not be available on some configurations of the EWL.

Listing: Example of strncat() Usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char s1[100] = "abcdefghijklmnopqrstuv";
char s2[] = "wxyz0123456789";
strncat(s1, s2, 4);
puts(s1);
```

```
return 0;
}
Output:
abcdefghijklmnopqrstuvwxyz
```

22.3 Comparing Characters

Comparing characters in memory areas and null-terminated strings.

22.3.1 memcmp()

Compare two blocks of memory.

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

Parameter

`s1`

The memory to compare.

`s2`

The comparison memory.

`n`

The maximum length to compare.

Remarks

This function compares the first n characters of s1 to s2, one character at a time.

The function returns a zero if all n characters pointed to by s1 and s2 are equal. The function returns a negative value if the first non-matching character pointed to by s1 is less than the character pointed to by s2. The function returns a positive value if the first non-matching character pointed to by s1 is greater than the character pointed to by s2.

This facility may not be available on some configurations of the EWL.

22.3.2 strcmp()

Compares two character strings.

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

Parameter

`s1`

The string to compare.

`s2`

The comparison string.

Remarks

This function compares the character array pointed to by s1 to the character array pointed to by s2. The function starts at the beginning of each and stops the comparison when it reaches a null character in s1 or s2. Both strings must be null terminated character arrays. The function returns:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2
- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2

This facility may not be available on some configurations of the EWL.

Listing: Example of strcmp() usage

```
#include <string.h>
#include <stdio.h>
#define MAX 20
int main (void)
{
char s1[] = "butter";
char s2[] = "olive oil";
```

```
char dest[MAX];
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
```

22.3.3 strcoll()

Compare two character arrays according to locale.

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

Parameter

`s1`

The string to compare.

`s2`

The comparison string.

Remarks

The ISO/IEC standards specify that this function compares two character arrays based on the `LC_COLLATE` component of the current locale.

However, the EWL implementation of `strcoll()` ignores the current locale. Instead, it compares two character arrays using `strcmp()`. It is included in the string library to conform to the ISO/IEC C Standard Library specification.

The function returns the same values that `strcmp()` returns.

This facility may not be available on some configurations of the EWL.

22.3.4 strncmp()

Compares up to a maximum number of characters.

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

Parameter

`s1`

A pointer to the character string to compare.

`s2`

A pointer to the comparison string.

`n`

The maximum number of characters to compare.

Remarks

This 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 nullterminated character arrays.

The function stops if it reaches a null character before `n` characters have been compared. The function returns:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2
- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2

This facility may not be available on some configurations of the EWL.

Listing: Example of strncmp() usage

```
#include <string.h>
#include <stdio.h>
int main(void)
```

```
{
char s1[] = "12345anchor";
char 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
```

22.3.5 strxfrm()

Transforms a locale-specific character array.

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

Parameter

`dest`

A pointer to a memory area to copy the transformed string to.

`source`

A pointer to the character string to transform.

`n`

The maximum number of characters in source to transform.

Remarks

This 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 transformation creates a string in dest so that applying the `strcmp()` function to two transformed strings 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 EWL implementation of this function ignores the current locale. Instead, this function 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 ISO/IEC C Standard Library specification.

The function returns the length of `dest` after it has received `source`.

This facility may not be available on some configurations of the EWL.

22.4 Searching Characters

Searching for characters in memory areas and null-terminated strings.

22.4.1 memchr()

Searches for an occurrence of a character.

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
```

Parameter

`s`

The memory to search

`c`

The character to search for.

`n`

The maximum length to search.

Remarks

This function looks for the first occurrence of c in the first n characters of the memory area pointed to by s.

The function returns a pointer to the found character, or a null pointer (`NULL`) if c cannot be found.

This facility may not be available on some configurations of the EWL.

Listing: 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! */
char s1[ARRAYSIZE] = "laugh* giggle 231!";
char s2[ARRAYSIZE] = "grunt sigh# snort!";
char dest[ARRAYSIZE];
char *strptr;
int len1, len2, lendest;
/* Clear destination string using memset(). */
memset(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(s1, s2, len1) > 0)
memcpy(dest, s1, len1);
else
memcpy(dest, 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(s1, '*', len1);
memmove(strptr, 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!
```

22.4.2 strchr()

Searches for an occurrence of a character in a character string.

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

Parameter

`s`

The string to search.

`c`

The character to search for.

Remarks

This function searches for the first occurrence of the character c in the character array pointed to by s. The function stops when it finds c or when it reaches the null character. The s argument must point to a null-terminated character array.

The function returns a pointer to the successfully located character. If it fails it returns a null pointer (`NULL`).

This facility may not be available on some configurations of the EWL.

Listing: Example of strchr() usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char s[] = "tree * tomato eggplant garlic";
```

```
char *strptr;
strptr = strchr(s, '*');
if (strptr != NULL)
puts(strptr);
return 0;
}
Output:
* tomato eggplant garlic
```

22.4.3 strcspn()

Find the first character of one string that is in another string.

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

Parameter

`s1`

The string to search.

`s2`

A string containing a list of characters to search for.

Remarks

This function finds the first occurrence in the string pointed to by `s1` of any character in the string pointed to by `s2`. These strings must be null-terminated. The function starts examining characters at the beginning of `s1` and continues searching until a character in `s1` matches a character in `s2`.

The function returns the index of the first character in `s1` that matches a character in `s2`. The function considers the null characters that terminate the strings.

This facility may not be available on some configurations of the EWL.

Listing: Example of strcspn() usage

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

```
int main(void)
{
char s1[] = "chocolate *cinnamon* 2 ginger";
char s2[] = "1234*";
size_t i;
printf(" s1 = %s\n s2 = %s\n", s1, s2);
i = strcspn(s1, s2);
printf("Index returned by strcspn() is %d.\n", i);
printf("Indexed character is %c.\n", s1[i]);
return 0;
}
Output:
s1 = chocolate *cinnamon* 2 ginger
s2 = 1234*
Index returned by strcspn() is 10.
Indexed character is *.
```

22.4.4 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);
```

Parameter

s1

A pointer to the string to search.

s2

A pointer to a list of characters to search for.

Remarks

This 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.

The function 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 facility may not be available on some configurations of the EWL.

Listing: Example of strpbrk usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char s1[] = "orange banana pineapple *plum";
char s2[] = "*%#$";
puts(strpbrk(s1, s2));
return 0;
}
Output:
*plum
```

22.4.5 strrchr()

Searches a string for the last occurrence of a character.

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

Parameter

s

The string to search.

c

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.

The function returns a pointer to the character found or returns a null pointer (NULL) if it fails.

This facility may not be available on some configurations of the EWL.

Listing: Example of strrchr() Usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
const char *lastptr;
char s[] = "Carly Chuck";
lastptr = strrchr(s, 'C');
if (lastptr != NULL)
puts(lastptr);
return 0;
}
Output:
Chuck
```

22.4.6 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);
```

Parameter

s1

A pointer to a character string to search.

s2

A pointer to a list of characters to search for.

Remarks

This function finds the first character in the character string s1 that is not in the string pointed to by 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.

The function returns the index of the first character in s1 that does not match a character in s2.

This facility may not be available on some configurations of the EWL.

Listing: Example of strspn() Usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char s1[] = "create *build* construct";
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
```

22.4.7 strstr()

Searches for a character array within another.

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

Parameter

s1

A pointer to the string to search in.

s2

A pointer to the string to search for.

Remarks

This 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 character arrays.

The function returns a pointer to the first occurrence of s2 in s1 and returns a null pointer (NULL) if s2 cannot be found.

This facility may not be available on some configurations of the EWL.

Listing: Example of strstr() Usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char s1[] = "tomato carrot onion";
char s2[] = "on";
const char* found;
found = strstr(s1, s2);
if (found != NULL)
puts(found);
return 0;
}
Output:
onion
```

22.4.8 strtok()

Extract tokens within a character array.

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

Parameter

`str`

A pointer to a character string to separate into tokens.

`sep`

A pointer to a character string containing separator characters.

Remarks

This function divides a character array pointed to by str into separate tokens. The sep argument points to a character array containing one or more separator characters. The tokens in str are extracted by successive calls to `strtok()`.

The function works by a sequence of calls. 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 separator string can be found, the function returns a null pointer (`NULL`). If no characters from the separator string are found, the function returns a pointer to the original string. Otherwise the function returns a pointer to the beginning of this first token.

Make subsequent calls to `strtok()` 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()`. The function modifies the character array pointed to by str.

This facility may not be available on some configurations of the EWL.

Listing: Example of strtok() usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
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
```

22.5 memset()

Sets 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);
```

Parameter

`dest`

A pointer to an area of memory to set.

`c`

The character to store in dest.

`n`

The number of characters to set.

Remarks

This function assigns `c` to the first `n` characters of the item pointed to by `dest`.

The function returns the value of `dest`.

This facility may not be available on some configurations of the EWL.

22.6 strerror()

Translate an error number into an error message.

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

Parameter

`errnum`

The error number to translate.

Remarks

This function returns a pointer to a null-terminated character array that contains an error message that corresponds to the error code in errnum. The character strings must not be modified by the program because it may be overwritten by a subsequent calls to the `strerror()` function.

Typically the value in `errnum` will come from the global variable `errno`, but `strerror()` will provide a message translation for any value of type `int`.

This facility may not be available on some configurations of the EWL.

Listing: 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
```

22.7 strlen()

Computes the length of a character array.

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

Parameter

`s`

A pointer to the string to evaluate.

Remarks

This 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.

This function's behavior is undefined if the character string pointed to by s is not null terminated.

The function returns the number of characters in a character array not including the terminating null character.

This facility may not be available on some configurations of the EWL.

Listing: Example of strlen() Usage

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char s[] = "antidisestablishmentarianism";
printf("The length of %s is %ld.\n", s, strlen(s));
return 0;
}
Output:
The length of antidisestablishmentarianism is 28.
```

Chapter 23
time.h

This header file provides functions for reading the system clock and for manipulating date and time values.

23.1 Functions in time.h

This topic lists the functions available in time.h header file.

23.1.1 time_t, clock_t, tm

Data types for manipulating date and time values.

```
#include <time.h>
typedef clock_t /* ... */ ;
typedef time_t /* ... */ ;
struct tm {
int tm_sec;
int tm_min;
int tm_hour;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

Parameter

`tm_sec`

Seconds, from 0 to 59.

`tm_min`

Minutes, from 0 to 59.

`tm_hour`

Hours, from 0 to 23.

`tm_mday`

Day of the month, from 1 to 31.

`tm_mon`

Month of the year, from 0 to 11. January is month 0.

`tm_year`

Year, beginning at 1900.

`tm_wday`

Day of the week, from 0 to 6. Sunday is day 0.

`tm_yday`

Day of the year, from 0 to 365. January 1 is day 0.

`tm_isdst`

Daylight savings time. Positive if daylight savings time is in effect, zero if it is not, and negative if such information is not available.

Remarks

The `clock_t` type is a numeric, system-dependent type returned by the `clock()` function.

The `time_t` type is a system-dependent type used to represent a calendar date and time as seconds elapsed since a fixed date. A value of type `time_t` represents the number of UTC seconds since 1970 January 1.

The type's range and precision are defined in the ISO/IEC C standard as implementation defined. The EWL implementation uses an `unsigned long int` for `time_t`. Note that this type cannot represent dates or times that exceed the size of the maximum value of `time_t` (`ULONG_MAX`). Similarly, since `time_t` is unsigned, negative values are also out of range.

23.2 Date and Time Manipulation

Retrieving, constructing, and operating on time and date values.

23.2.1 clock()

A program relative invocation of the system time.

```
#include <time.h>
clock_t clock(void);
```

Remarks

Use this function 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 function returns a value of type `clock_t` representing the approximation of time since the system was started. The function does not return an error value if an error occurs.

This facility may not be available on some configurations of the EWL.

Listing: Example of clock() usage

```
#include <time.h>
#include <stdio.h>
#define MAXLOOP 100000
int main()
{
clock_t start;
clock_t end;
double secs = 0;
int i;
start = clock();
end = clock();
for (i = 0; i < MAXLOOP; ++i)
{
; /* Do nothing. */
```

```
}
secs = (double)(end - start) / (double)CLOCKS_PER_SEC;
printf("Elapsed seconds = %f \n", secs);
return 0;
}
Output:
Elapsed seconds = 0.033333
```

23.2.2 difftime()

Computes the difference between two time values.

```
#include <time.h>
double difftime(time_t t1, time_t t2);
```

Parameter

`t1`

A time value.

`t2`

A time value.

Remarks

This function returns the difference of `t1` minus `t2`, expressed in seconds.

This facility may not be available on some configurations of the EWL.

23.2.3 mktime()

Converts a structure of type tm to a value of type `time_t`.

```
#include <time.h>
time_t mktime(struct tm *ts);
```

Parameter

`ts`

A pointer to a time structure to convert.

Remarks

This function converts ts to a value of type `time_t`. The function returns this converted value.

The function also adjusts the fields in ts 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 `ts->tm_min` is 65, then the function will set `ts->tm_min` to 5 and increment `ts->tm_hour` by 1.

The function also corrects the values in `ts->tm_wday` and `ts->tm_yday`.

This facility may not be available on some configurations of the EWL.

23.2.4 time()

Returns the current system calendar time.

```
#include <time.h>
time_t time(time_t *t);
```

Parameter

`t`

A pointer to time value or `NULL`.

Remarks

This function returns the computer system's calendar time. If t is not a null pointer, the calendar time is also assigned to the item it points to.

The function returns the system current calendar time.

This facility may not be available on some configurations of the EWL.

Listing: 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
```

23.2.5 tzname()

Contains the abbreviated names of the time zones for local standard time and daylight standard time.

```
#include <time.h>
extern char *tzname[2];
```

Remarks

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

23.2.6 tzset()

Synchronizes the library's internal data with the operating system's time zone information.

```
#include <time.h>
void tzset(void);
```

Remarks

This function reads the value of the TZ environment variable to initialize Embedded Warrior Library's internal time zone information.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

23.3 Date and Time Conversion

Converting date and time values to and from character strings.

23.3.1 asctime()

Convert a `tm` structure to a character array.

```
#include <time.h>
char *asctime(const struct tm *t);
```

Parameter

`t`

A pointer to a `tm` structure describing the time and date to convert.

Remarks

This function converts a `tm` structure, pointed to by `t`, 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`".

The function returns a null-terminated character array pointer containing the textual representation of the date and time described by `t`.

This facility may not be available on some configurations of the EWL.

Listing: 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
```

23.3.2 ctime()

Convert a `time_t` type to a character array.

```
#include <time.h>
char *ctime(const time_t *timer);
```

Parameter

`timer`

A pointer to a value of type `time_t` to convert.

Remarks

This function converts a `time_t` type to a character array with the same format as generated by `asctime()`.

The function returns a null-terminated character array pointer containing the converted `time_t` value. This facility may not be available on some configurations of the EWL.

Listing: 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
```

23.3.3 gmtime()

Converts a `time_t` value to Coordinated Universal Time (UTC).

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

Parameter

`time`

A pointer to a time value.

Remarks

This function converts the calendar time pointed to by `time` into a broken-down time, expressed as UTC. The function function returns a pointer to that object.

This facility may not be available on some configurations of the EWL.

Listing: 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
```

23.3.4 localtime()

Converts a value of type `time_t` to a structure of type `tm`.

```
#include <time.h>

struct tm *localtime(const time_t *time);
```

Parameter

`time`

A pointer to a time value to convert.

Remarks

This function function converts a value of type `time_t`, pointed to by timer, and converts it to a structure of type `tm`. The function returns this pointer. The pointer is `static`; it is overwritten each time `localtime()` is called.

This facility may not be available on some configurations of the EWL.

23.3.5 strftime()

Formats a `tm` structure.

```
#include <time.h>
size_t strftime(char *s, size_t max, const char *format, const struct tm *ts);
```

Parameter

`b`

A pointer to a character string in which to store the formatted text.

`max`

The maximum number of characters that may be stored at `s`.

`format`

A pointer to a character string that describes how to format the text.

ts

A pointer to time structure to convert to text.

Remarks

This function converts ts to a null-terminated character array, s, using the format specified by format. The number of characters stored in the formatted string will not exceed the number specified by max.

The format argument points to a character array containing normal text and conversion specifications similar to the format string used by the snprintf() function. Conversion specifiers for date and time values 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.

A conversion specifier that has a E prefix specifies that the resulting text should use the locale's alternate textual representation. The O prefix specifies the locale's alternate numeric symbols. 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.

The strftime() function returns the total number of characters that were stored in s if the total number of characters, including the null character, is less than the value of max. If the formatted string cannot fit in s, strftime() returns 0.

The table below lists the conversion specifies recognized by strftime().

Table 23-1. Conversion Specifiers for Time and Date Value

Conversion Specifier	Generates
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	Equivalent to " %A %B %d %T %Y".
C	The year divided by 100 and truncated to an integer, as a two-digit decimal number from 00 to 99.
d	Day of the month as a 2-digit decimal number from 01 to 31.
D	Equivalent to " %m/%d/%y".
e	The day of the month as a decimal number. Single digit values are preceded by a space character.
F	Equivalent to " %Y-%m-%d".
g	The last 2 digits of the week-based year (as described by the ISO/IEC 8601 standard).

Table continues on the next page...

Table 23-1. Conversion Specifiers for Time and Date Value (continued)

Conversion Specifier	Generates
G	The week-based year (as described by the ISO/IEC 8601 standard).
h	Equivalent to " %b".
H	The hour of the 24-hour clock as a 2-digit decimal number from 00 to 23.
I	The hour of the 12-hour clock as a 2-digit decimal number from 01 to 12.
j	The day of the year as a 3-digit decimal number from 001 to 366.
m	The month as a 2-digit decimal numberfrom 01 to 12.
M	The minute as a 2-digit decimal numberfrom 00 to 59.
n	Newline character.
p	Locale's representation of the AM or PM designation for a 12-hour clock.
r	Locale's 12-hour clock time.
R	Equivalent to "%H:%M".
S	The second as a 2-digit decimal number from 00 to 60.
t	Horizontal tab.
T	Equivalent to "%H:%M:%S".
u	The weekday as a single-digit decimal number from 1 to 7.
U	The week number of the year as a 2-digit decimal number from 00 to 53. Sunday is considered the first day of the week.
w	The weekday as a single-digit decimal number from 0 to 6. Sunday is the first day of the week.
W	The week of the year as a 2-digit decimal number from 00 to 51. Monday is the first day of the week.
x	Equivalent to " %A %B %d %Y"
X	Equivalent " %T"
y	The last two digits of the year as a decimal number.
Y	The year as a 4-digit number.
z	The time zone offset from UTC or the empty string if the time zone is unknown.
Z	The locale's time zone name, its abbreviation, or the empty string if the time zone is unknown.

This facility may not be available on some configurations of the EWL.

Listing: Example of strftime() Usage

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#define MAXTS 1000
int main(void)
{
```

```
time_t lclTime;
struct tm *now;
char ts[MAXTS];
size_t result;
lclTime = time(NULL);
now = localtime(&lclTime);
result = strftime(ts, MAXTS, "Today's abbreviated name is %a",
now);
if (result == 0)
exit(1);
puts(ts);
result = strftime(ts, MAXTS, "Today's full name is %A", now);
if (result == 0)
exit(1);
puts(ts);
return 0;
}
Output:
Today's abbreviated name is Wed
Today's full name is Wednesday
```

Chapter 24
tgmath.h

Type-generic macros for invoking functions declared in `math.h` and `complex.h`.

The `math.h` and `complex.h` header files declare many functions in variants to accept arguments and return values in each of the floating-point types, `double`, `float`, and `long double`. A function typically uses the `double` type. A function of the same name with a suffix of "f" uses the `float` type and a function with a suffix of "l" uses the `long double` type. For example, `math.h` declares the `cos()`, `cosf()`, and `cosl()` functions. These functions all behave similarly but accept and return values of type `double`, `float`, and `long double`, respectively.

The macros defined in the `tgmath.h` header file choose and invoke a corresponding function in `math.h` or `complex.h` based on the types of the arguments passed to the macro.

- The header file follows these rules, in this order, to choose a function to invoke:
- if any of the macro's arguments has a complex type, use the complex variant of the macro's corresponding function
- if any of the macro's arguments has a type of `long double` then use a `long double` variant
- if any of the macro arguments has type `double` or an integer type then use a `double` variant
- otherwise, use the `float` variant

The table below table lists the macros in `tgmath.h` and the corresponding function variants in `math.h` and `complex.h`.

Table 24-1. Macros in tgmath.h

Macro	Math.h Variant	Complex.h Variant
acos()	acos	cacos
asin()	asin	casin
atan()	atan	catan
acosh()	acosh	cacosh

Table continues on the next page...

Table 24-1. Macros in tgmath.h (continued)

Macro	Math.h Variant	Complex.h Variant
asinh()	asinh	casinh
atanh()	atanh	catanh
cos()	cos	ccos
sin()	sin	csin
tan()	tan	ctan
cosh()	cosh	ccosh
sinh()	sinh	csinh
tanh()	tanh	ctanh
exp()	exp	cexp
pow()	pow	cpow
sqrt()	sqrt	csqrt
fabs()	fabs	cabs
log()	log	clog
atan2()	atan2	
cbrt()	cbrt	
ceil()	ceil	
copysign()	copysign	
erf()	erf	
erfc()	erfc	
exp2()	exp2	
expm1()	expml	
fdim()	fdim	
floor()	floor	
fma()	fma	
fmax()	fmax	
fmin()	fmin	
fmod()	fmod	
frexp()	frexp	
hypot()	hypot	
ilogb()	ilogb	
ldexp()	ldexp	
lgamma()	lgamma	
llrint()	llrint	
llround()	llround	
log10()	log10	
log1p()	log1p	
log2()	log2	
logb()	logb	
lround()	lround	

Table continues on the next page...

Table 24-1. Macros in tgmath.h (continued)

Macro	Math.h Variant	Complex.h Variant
lrint()	lrint	
nearbyint()	nearbyint	
nextafter()	nextafter	
nexttoward()	nexttoward	
remainder()	remainder	
remquo()	remquo	
rint()	rint	
round()	round	
scalbn()	scalbn	
scalbln()	scalbln	
tgamma()	tgamma	
trunc()	trunc	
carg()		carg
cimag()		cimag
conj()		conj
cproj()		cproj
creal()		creal

Chapter 25
wchar.h

This header file declares functions and data types for manipulating wide characters and multibyte strings.

25.1 Wide-Character Formatted Input and Output

Input and output functions for formatting wide character strings.

```
  #include <wchar.h>
  int fwprintf(FILE * file, const wchar_t * format, ...);
  int fwscanf(FILE * file, const wchar_t * format, ...);
  int swprintf(wchar_t * S, size_t N, const wchar_t * format, ...);
  int vfwprintf(FILE * file, const wchar_t * format_str, va_list arg);
  int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list
arg);
  int vwprintf(const wchar_t * format, va_list arg);
  int wprintf(const wchar_t * format, ...);
  int swscanf(const wchar_t * s, const wchar_t * format, ...);
  int vfwscanf(FILE * file, const wchar_t * format_str, va_list arg);
  int vswscanf(const wchar_t * s,
  const wchar_t * format, va_list arg);
  int vwscanf(const wchar_t * format, va_list arg);
  int wscanf(const wchar_t * format, ...);
```

These functions operate identically to their counterpart functions in `stdio.h`. But instead of operating on character strings, these functions operate on wide character strings.

The table below matches these wide character functions to their equivalent char-based functions in `stdio.h`.

Table 25-1. Wide Character Formatting Functions

Function	Wide Character Equivalent
fwprintf()	fprintf()
swprintf()	sprintf()
vfwprintf()	vfprintf()
vswprintf()	vsprintf()

Table continues on the next page...

Table 25-1. Wide Character Formatting Functions (continued)

Function	Wide Character Equivalent
vwprintf()	vprintf()
wprintf()	printf()
fwscanf()	fscanf()
swscanf()	sscanf()
vfwscanf()	vfscanf()
vswscanf()	vsscanf()
vscanf()	vscanf()
wscanf()	scanf()

25.2 Wide-Character Input and Output

Character input and output functions for wide characters.

```
include <wchar.h>
wchar_t fgetwc(FILE * file);
wchar_t *fgetws(wchar_t * s, int n, FILE * file);
wchar_t getwc(FILE * file);
wchar_t getwchar(void);
wchar_t fputwc(wchar_t c, FILE * file);
int fputws(const wchar_t * s, FILE * file);
wchar_t putwc(wchar_t c, FILE * file);
wchar_t putwchar(wchar_t c);
```

These functions operate identically to their counterpart functions in `stdio.h`. But instead of operating on character strings, these functions operate on wide character strings.

The table below matches these wide character functions to their equivalent `char`-based functions in `stdio.h`.

Table 25-2. Wide Character Input/Output Functions

Function	Wide Character Equivalent
fgetwc()	fgetc()
fgetws()	fgets()
fputwc()	fputc()
fputws()	fputs()
getwc()	getc()
getwchar()	getchar()
putwc()	putc()
putwchar()	putchar()

25.3 Wide-Character Utilities

Manipulating and converting wide-wharacter strings.

25.3.1 Wide-Character Numerical Conversion

Convert among numerical types and wide-character strings.

```
  #include <wchar.h>
  double wcstod(wchar_t * str, char ** end);
  float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
  long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
  long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
  long long int wcstoll(const wchar_t * restrict nptr, wchar_t   ** restrict endptr,  int
base);
  unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int
base);
  unsigned long long int wcstoull( const wchar_t * restrict nptr, wchar_t ** restrict
endptr, int base);
```

These functions operate identically to their counterpart functions in `stdlib.h`. But instead of operating on character strings, these functions operate on wide character strings.

The table below matches these wide character functions to their equivalent `char`-based functions in `stdlib.h`.\

Table 25-3. Wide Character Numerical Conversion Functions

Function	Wide Character Equivalent
wcstod()	strtod()
wcstof()	strtof()
wcstol()	strtol()
wcstoll()	strtoll()
wcstoul()	strtoul()
wcstoull()	strtoull()

25.3.2 Wide-Character String Manipulation

Manipulate wide character strings.

```
#include <wchar.h>
wchar_t * wcscat(wchar_t * dst, const wchar_t * src);
wchar_t * wcschr(const wchar_t * str, const wchar_t chr);
int wcscmp(const wchar_t * str1, const wchar_t * str2);
int wcscoll(const wchar_t *str1, const wchar_t * str2);
wchar_t * wcscpy(wchar_t * dst, const wchar_t * src);
size_t wcscspn(const wchar_t * str, const wchar_t * set);
size_t wcslen(const wchar_t * str);
wchar_t * wcsncat(wchar_t * dst,
const wchar_t * src, size_t n);
int wcsncmp(const wchar_t * str1,
const wchar_t * str2, size_t n);
wchar_t * wcsncpy(wchar_t * dst,
const wchar_t * src, size_t n);
wchar_t * wcspbrk(const wchar_t * str, const wchar_t * set);
wchar_t * wcsrchr(const wchar_t * str, wchar_t chr);
size_t wcsspn(const wchar_t * str, const wchar_t * set);
wchar_t * wcsstr(const wchar_t * str, const wchar_t * pat);
wchar_t * wcstok(wchar_t * str, const wchar_t * set, wchar_t ** ptr););
size_t wcsxfrm(wchar_t * str1, const wchar_t * str2, size_t n);
void * wmemchr(const void * src, int val, size_t n);
int wmemcmp(const void * src1, const void * src2, size_t n);
void * wmemcpy(void * dst, const void * src, size_t n);
void * wmemmove(void * dst, const void * src, size_t n);
void * wmemset(void * dst, int val, size_t n);
```

These functions operate identically to their counterpart functions in `string.h`. But instead of operating on character strings, these functions operate on wide character strings.

The table below matches these wide character functions to their equivalent `char`-based functions in `string.h`.

Table 25-4. Wide Character String Functions

Function	Wide Character Equivalent
`wcscat()`	`strcat()`
`wcschr()`	`strchr()`
`wcscmp()`	`strcmp()`
`wcscoll()`	`strcoll()`
`wcscpy()`	`strcpy()`
`wcscspn()`	`strcspn()`
`wcslen()`	`strlen()`
`wcsncat()`	`strncat()`
`wcsncmp( )`	`strncmp()`
`wcsncpy()`	`strncpy()`
`wcspbrk()`	`strpbrk()`
`wcsrchr()`	`strrchr()`
`wcsspn()`	`strspn()`
`wcsstr()`	`strstr()`
`wcstok()`	`strtok()`

Table continues on the next page...

Table 25-4. Wide Character String Functions (continued)

Function	Wide Character Equivalent
wcsxfrm()	strxfrm()
wmemcmp()	memcmp()
wmemchr()	memchr()
wmemcpy()	memcpy()
wmemmove()	memmove()
wmemset()	memset()

25.4 Wide-Character Date and Time Manipulation

Convert time and date to wide character strings.

```
  #include <wchar.h>
  size_t wcsftime(wchar_t * str, size_t max_size, const wchar_t * format_str, const struct
tm * timeptr);
  wchar_t * wctime(const time_t * timer);
```

These functions operate identically to their counterpart functions in `stdio.h`. But instead of operating on character strings, these functions operate on wide character strings.

The table below matches these wide character functions to their equivalent `char`-based functions in `stdio.h`.

Table 25-5. Wide Character Date and Time Functions

Function	Wide Character Equivalent
wcsftime()	csftime()
wctime()	ctime()

25.5 Wide-Character Conversion

Converting between wide and multibyte characters and character strings.

25.5.1 btowc()

Converts a character to a wide character.

```
#include <wchar.h>

wint_t btowc(int c);
```

Parameter

`c`

The character to be converted.

Remarks

This returns the wide character representation of the argument. The function returns `WEOF` 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.

25.5.2 mbrlen()

Computes the length of a multibyte-encoded character string.

```
#include <stdlib.h>

int mbrlen(const char *s, size_t n, mbstate_t * ps);
```

Parameter

`s`

A pointer to a multibyte-encoded character string.

`n`

The maximum size.

`ps`

The current state of translation between multibyte and wide character. Ignored if the encoding scheme is non-modal.

Remarks

This function returns the length of the multibyte character pointed to by s. It examines a maximum of n characters. This function operates similarly to `mblen()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

EWL supports the "C" locale with UTF-8 encoding only. It returns the value of

```
mbrtowc(NULL, s, n, ps)
```

25.5.3 mbrtowc()

Converts a multibyte-encoded character to a wide character.

```
#include <wchar.h>
int mbrtowc(wchar_t *pwc,
const char *s, size_t n, mbstate_t * ps);
```

Parameter

`pwc`

A pointer to a wide character in which to store the result.

`s`

A pointer to the multibyte string to convert.

`n`

The maximum number of bytes at s with which to form a multibyte character.

`ps`

The current state of conversion. Ignored if the encoding scheme is non-modal.

Remarks

This function translates a multibyte character to a `wchar_t` character according to the encoding specified in the `LC_CTYPE` component of the current locale. This function operates identically to `mbtowcs()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the multibyte encoding scheme is non-modal.

If s is a null pointer, this call is equivalent to

mbrtowc(NULL, "", 1, ps);

In other words, the function treats s as an empty multibyte string and ignores the pwc and n arguments. Such a call also has the effect of resetting the conversion state.

If s is not a null pointer, the 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 the number of bytes that make up a complete, valid character is

less than or equal to n and pwc is not a null pointer, the function converts the multibyte character, pointed to by s to a character of type `wchar_t` using the encoding scheme specified in the `LC_CTYPE` component of the current locale. It stores this wide character at the location pointed to by pwc.

The function returns one of these values:

- Zero if s points to a nul character.
- The number of bytes in the complete multibyte character that was converted to a wide character if s points to a complete, valid multibyte character of n or fewer bytes. The wide character is stored in the location referred to by pwc if it is not `NULL`.
- -2 if the next n bytes pointed to by s constitute an incomplete but potentially valid multibyte character. No wide character is stored at the location referred to by pwc.
- -1 if the next n bytes pointed to by s do not constitute a complete, valid multibyte character. The function stores `EILSEQ` in errno. No wide character is stored at the location referred to by pwc.

25.5.4 mbsinit()

Returns the state of a multibyte conversion state.

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

Parameter

`ps`

A pointer to a multibyte conversion state.

Remarks

This function returns true if ps is a null pointer or it points to a conversion state that is at the first position of a multibyte-encoded character. The function returns false otherwise.

25.5.5 mbsrtowcs()

Converts a multibyte character string to a wide character string.

```
#include <stddlib.h>

size_t mbsrtowcs(wchar_t *pwcs, const char **s, size_t n, mbstate_t * ps);
```

Parameter

`pwcs`

A pointer to a wide character string in which to store the converted string.

`s`

A pointer to a pointer to the multibyte string to convert.

`n`

The maximum number of wide characters to store at pwcs.

`ps`

The current state of conversion. Ignored if the encoding scheme is non-modal.

Remarks

This function operates identically to 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. Also, this function returns extra information about the conversion in the s argument.

The EWL implementation of `mbsrtowcs()` converts a sequence of multibyte characters encoded according to the scheme specified in the `LC_CTYPE` component of the current locale, indirectly pointed to by s. If pwcs is not a null pointer, the function stores not more than n of the corresponding Unicode characters. The function stops if it reaches a null character or an invalid multibyte character and stores a null wide character to terminate the wide character string at pwcs.

If conversion stops because of a terminating null character, the function stores a null pointer in the object pointed to by s. Otherwise, the function stores a pointer to the address just beyond the last multibyte character converted, if any.

The function returns the number of multibyte characters successfully converts, not including the terminating null character. If the function encounters an invalidly encoded multibyte character, it stores the value of `EILSEQ` in `errno` and returns the value (size_t) (-1).

25.5.6 wcrtomb()

Converts a wide character to a multibyte character.

```
#include <wchar.h>

int wcrtomb(char *s, wchar_t wchar, mbstate_t * ps);
```

Parameter

`s`

A pointer to a multibyte character buffer of at least `MB_CUR_MAX` characters long.

`wchar`

A wide character to convert.

`ps`

A pointer to a multibyte conversion state. Ignored if the encoding scheme is non-modal.

Remarks

This function ranslates a `wchar_t` type to a multibyte character according to the encoding scheme specified in the `LC_CTYPE` component of the current locale. Thisfunction operates identically to `wctomb()` except that it has an additional parameter of type `mbstate_t*`, which is ignored if the encoding scheme is non-modal.

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, the function determines the length of the UTF-8 multibyte encoding that corresponds to the wide character wchar and stores that sequence in the multibyte string pointed to by s. At most `MB_CUR_MAX` bytes are stored. The function returns the number of bytes stored in the string `s`.

25.5.7 wctob()

Converts a wide character to a character.

```
#include <wchar.h>

int wctob(wint_t wc);
```

Parameter

`wc`

The wide character to be converted.

Remarks

The function `wctob()` returns the single byte representation of the argument wc as an unsigned character value. If wc does not contain a wide character that can be translated to a regular character, this function returns `EOF`.

The function returns its result as a value of type `int` to accommodate values of type `unsigned char` and the value `EOF`.

Chapter 26
wctype.h

Macros for testing the kind of wide-character and for converting alphabetic wide characters to uppercase or lowercase.

26.1 Macros in wctype.h

The wctype.h header file has following macros.

26.1.1 iswalnum(), iswalpha(), iswblank(), iswcntrl(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit()

Tests for membership in subsets of the wide-character set.

```
#include <wctype.h>
int iswalnum(wint_t c);
int iswalpha(wint_t c);
int iswblank(wint_t c);
int iswcntrl(wint_t c);
int iswdigit(wint_t c);
int iswgraph(wint_t c);
int iswlower(wint_t c);
int iswprint(wint_t c);
int iswpunct(wint_t c);
int iswspace(wint_t c);
int iswupper(wint_t c);
int iswxdigit(wint_t c);
```

Parameter

`c`

A wide-character value to test.

Remarks

These functions provide the same facilities as their counterparts in the `ctype.h` header file. However, these functions test wide characters and the `EOF` value.

The c argument is of type `wint_t` so that the `EOF` value, which is outside the range of the `wchar_t` type, may also be tested.

The table below lists these functions and their counterparts in `ctype.h`.

Table 26-1. Wide Character Testing Functions

Function	ctype.h Equivalent
`iswalnum(c)`	`isalnum(c)`
`iswalpha(c)`	`isalpha(c)`
`iswblank(c)`	`isblank(c)`
`iswcntrl(c)`	`iscntrl(c)`
`iswdigit(c)`	`isdigit(c)`
`iswgraph(c)`	`isgraph(c)`
`iswlower(c)`	`islower(c)`
`iswprint(c)`	`isprint(c)`
`iswpunct(c)`	`ispunct(c)`
`iswspace(c)`	`isspace(c)`
`iswupper(c)`	`isupper(c)`
`iswxdigit(c)`	`isxdigit(c)`

26.1.2 towlower(), towupper()

Converts alphabetic wide characters to lowercase or uppercase.

```
#include <ctype.h>
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);
```

Parameter

`c`

A wide-character value to convert.

Remarks

The `towlower()` function converts an uppercase alphabetic character to its equivalent lowercase character. It returns all other characters unchanged. The `towupper()` function converts a lowercase alphabetic character to its uppercase equivalent. It returns all other characters unchanged.

26.1.3 wctrans()

Constructs a property value for character remapping.

```
#include <wchar.h>
wctrans_t wctrans(const char *name);
```

Parameter

`name`

A pointer to a character string containing a description of the remapping.

Remarks

Constructs a value that represents a mapping between wide characters. The value of name can be either `"toupper"` or `"tolower"`.

This facility may not be available on some configurations of the EWL.

26.1.4 towctrans()

Maps a wide-character value to another wide-character value.

```
#include <wchar.h>
wint_t towctrans(wint_t c, wctrans_t value);
```

Parameter

`c`

The character to remap.

`value`

A value returned by `wctrans()`.

Remarks

Maps the first argument to an upper or lower value as specified by `value`. Returns the remapped character.

This facility may not be available on some configurations of the EWL.

Chapter 27
EWL Extra Library

Useful, non-standard facilities.

The EWL Extras Library is a companion library to EWL. It contains useful functions, macros, and types that are not specified in the ISO/IEC Standards. This library also has facilities for UNIX and POSIX.1 compatibility.

27.1 Multithreading

The Embedded Warrior Library (EWL) has non-standard features for multithreaded systems.

27.1.1 Introduction to Multithreading

Most modern operating systems are said to be multithreaded because they allow the creation of additional threads within a process beyond the one that begins the execution of the process.

In a multithreaded operating system, a process may consist of more than one thread, all of them executing simultaneously from the user's point of view.

Of course, unless there is more than one processor, the threads are not really executing simultaneously. Instead, the operating system gives the impression of simultaneity by multiplexing among the threads. The operating system determines which thread gets control of the processor at any particular time. There are two models for operating a multithreaded process:

- 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 allows the next thread to gain control. In this model, the execution of a thread can only be interrupted at points explicitly under the thread's control.
- In the preemptive model, the operating system switches between the threads at the operating system's discretion. These switches occur at arbitrary and unpredictable times and points in the code being executed. The thread usually does not need to be aware of when these switches occur.

In the rest of this section we will assume a preemptive model of multithreading.

We use the term thread to refer to the smallest amount of processor context state necessary to encapsulate a computation. Practically speaking, a thread typically consists of a register set, a stack, a reference to the executable code's address space, and a references to the data address space. Some parts of the data space are private to the thread while other parts may be shared with other threads in the same process. 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. Variables that belong to the storage class `static` outside of functions 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 multithreading 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 uses the thread-local storage class.

A preemptive thread switch may occur between any two machine instructions, which might not coincide with a boundary between two source code statements. A thread switch can also occur part-way through the evaluation of a source code expression. One important consequence of this possibility 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, called a race condition, makes debugging and validation difficult.

Multithreading requires mechanisms to protect against race conditions. Various methods exist for protecting segments of code from being executed by two or more threads at the same time. A program that is suitably protected against errors in the presence of multithreading is said to be thread-safe.

27.1.2 Definitions

There are no standards for implementing multithreading. In particular, neither the C99 Standard nor the POSIX Standard makes reference to threads. For the EWL library, we define thread-safety as:

An EWL Library function will be said to be "thread-safe" if two or more simultaneously executing threads in a single process can all call the function without danger of mutual interference.

For most functions in the library, the meaning of thread safety is clear. Some functions, such as `rand()` or `strtok()` are allowed to maintain internal state variables and would appear, by definition, to be not thread-safe.

The following EWL library functions have special precautions to make them thread-safe. The remaining EWL functions are intrinsically thread-safe.

- `asctime()`
- `atexit()`
- `_calloc()`
- `ctime()`
- `exit()`
- `fgetc()`
- `fgetpos()`
- `fgets()`
- `fgetwc()`
- `fgetws()`
- `fopen()`
- `fprintf()`
- `fputc()`
- `fputs()`
- `fputwc()`
- `fputws()`
- `fread()`
- `_free()`
- `fscanf()`
- `fwscanf()`
- `gmtime()`
- `putchar()`
- `_realloc()`
- `srand()`
- `vfprintf()`
- `vwprintf()`
- `wscanf()`

27.1.3 Reentrant Functions

All functions in EWL are thread-safe by defining the EWL_THREADSAFE macro as 1 when compiling EWL itself. When the EWL_THREADSAFE macro is 0, many of the EWL functions lose their thread safe attributes. It may be useful to leave the EWL_THREADSAFE macro as 0 even on a multithreaded system to improve execution 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. The GNU Compiler Collection (GCC) and other library vendors provide an assortment of helper functions, all with a r suffix, to indicate they are naturally thread safe. Following this convention, the EWL offers variations of some standard functions that are reentrant. The table below lists reentrant function in standard header files.

Table 27-1. Reentrant Functions in Standard Headers

Header File	Reentrant Function
stdlib.h	rand_r()
string.h	strerror_r()
time.h	asctime_r(), ctime_r(), gmtime_r(), localtime_r()

27.2 extras_io.h

Extra file and input/output facilities.

27.2.1 chsize()

Changes a file's size.

```
#include <extras/extras_io.h>

int chsize(int handle, long size);
```

Parameter

handle

The handle of the file being changed.

size

The size to change to.

Remarks

If a file is truncated all data beyond the new end of file is lost. If a file is extended, the function fills the extended area of the file with null characters (`'\0'`). This function returns zero on success and -1 if a failure occurs.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.2.2 filelength()

Retrieves the file length based on a file handle.

```
#include <extras/extras_io.h>

int filelength(int fileno);
```

Parameter

`fileno`

A file handle.

Remarks

If successful, this function returns a file's length. If it fails, the function returns -1. This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.2.3 tell()

Returns the current offset for a file.

```
#include <extras/extras_io.h>

long int tell(int fildes);
```

Parameter

`fildes`

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 facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

Listing: Example of tell() usage

```
#include <stdio.h>
#include <extras/extras_io.h>
int main(void)
{
int fd;
long int pos;
char hello[] = "je me souviens";
fd = open("mytest", O_RDWR | O_CREAT | O_TRUNC);
write(fd, hello, sizeof(hello));
pos = tell(fd);
if (pos < 0)
exit(1);
printf("%ld.", pos);
close(fd);
return 0;
}
Output:
14
```

27.3 extras_malloc.h

Extra memory management facilities.

27.3.1 heapmin()

Releases the heap memory back to the system.

```
#include <extras/extras_malloc.h>

int heapmin(void);
```

Remarks

This function releases your program's heap memory to the system. After calling this function, a program's behavior is undefined if it refers to memory allocated with heap allocation functions like `_malloc()` and `_realloc()`. The function returns zero if successful. If it fails, the function returns -1 and sets `errno`.

This facility is not specified in the ISO/IEC standards.

It is an EWL extension of the standard libraries.

27.4 extras_stdlib.h

Extra facilities for numeric conversion and pseudo-random number generation.

27.4.1 gcvt()

Converts a floating point value to a null terminated character string.

```
#include <extras/extras_stdlib.h>

char *gcvt(double value, int digits, char *buffer);
```

Parameter

`value`

A floating point value to convert.

`digits`

The number of significant digits to convert.

`buffer`

The string to hold the converted floating point value.

Remarks

The character string stored at buffer includes the decimal point and sign of value. This function returns a pointer to the buffer argument.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.4.2 itoa()

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

```
#include <extras/extras_stdlib.h>

char * itoa(int val, char *str, int radix);
```

Parameter

`val`

An integer value to convert.

`str`

A pointer to the string to store the converted value.

`radix`

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. The function returns a pointer to the converted string.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.4.3 ltoa()

Converts an long integer value to a null-terminated character string.

```
#include <extras/extras_stdlib.h>

char * ltoa(long int val, char *str, int radix);
```

Parameter

`val`

The long integer value to convert.

`str`

A pointer to an array in which to store the converted value.

`radix`

The numeric base to convert the number to.

Remarks

The radix is the base of the number in a range of 2 to 36. The function returns a pointer to the converted string, `str`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.4.4 rand_r()

Reentrant function to generate a pseudo-random integer value.

```
#include <extras/extras_stdlib.h>

int rand_r(unsigned int *context);
```

Parameter

`context`

The seed value.

Remarks

The `rand_r()` function provides the same service as `rand()`, 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 facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.4.5 ultoa()

Converts an unsigned long value to a null terminated character string.

```
#include <extras/extras_stdlib.h>

char * ultoa(unsigned long val, char *str, int radix);
```

Parameter

`val`

The integer value to convert

`str`

A pointer to the string to store the converted value.

`radix`

The numeric base to convert the number to.

Remarks

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

The function returns a pointer to the converted string, `str`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5 extras_string.h

Extra character string manipulation.

27.5.1 strcmpi()

String comparison that ignores letter case.

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

Parameter

`s1`

A pointer to a null-terminated character string.

`s2`

A pointer to a null-terminated character string.

Remarks

This function operates identically to `stricmp()`.

This facility may not be available on some configurations of the EWL.

27.5.2 strdup()

Creates a duplicate string in heap memory.

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

Parameter

`str`

A pointer to a null-terminated character string to copy.

Remarks

This function returns a pointer to a newly-allocated, duplicated string or `NULL` if unsuccessful. This duplicated string may be released from heap memory with the `_free()` function.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.3 strerror_r()

Translates an error number into an error message in a thread-safe manner.

```
#include <extras/extras_string.h>
int strerror_r(int errnum, char *str, size_t bufflen);
```

Parameter

`errnum`

The error number to translate.

`str`

A pointer to a memory area to copy the error message string to.

`bufflen`

The size of the storage buffer pointed to by str.

Remarks

This function provides the same service as `strerror()` but is reentrant because the caller to `strerror_r()` provides the storage, str, for the error message string.

The function returns zero if it succeeds. If it fails, the function returns a non-zero value.

This facility may not be available on some configurations of the EWL.

27.5.4 stricmp()

String comparison that ignores letter case.

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

Parameter

`s1`

A pointer to a null-terminated character string.

`s2`

A pointer to a null-terminated character string.

Remarks

This function returns one of these values:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2
- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2 This facility may not be available on some configurations of the EWL.

27.5.5 stricoll()

Locale-aware collating string comparison that ignores letter case.

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

Parameter

`s1`

A pointer to a null-terminated character string.

`s2`

A pointer to a null-terminated character string.

Remarks

This function compares each character at s1 and s2 using the collating sequence specified by the `LC_COLLATE` component of the current locale. This function returns one of these values:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2

- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.6 strlwr()

Converts a character string to lowercase.

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

Parameter

`str`

A pointer to a null-terminated character string.

Remarks

This function converts all uppercase alphabetic characters at str to their lowercase counterparts. The function does not modify any other characters. The function returns a pointer to `str`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.7 strncasecmp()

Character string comparison with length specified and ignored letter case.

```
#include <extras/extras_string.h>
int strncasecmp(const char *s1, const char *s2, size_t max);
```

Parameter

`str1`

A pointer to a character string.

`str2`

A pointer to a character string.

`max`

The maximum number of characters to compare.

Remarks

This function compares the characters at s1 and s2, ignoring letter case. It stops when it reaches the null character or when it has compared max characters.

This function returns one of these values:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2
- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.8 strncmpi()

Character string comparison with length specified and ignored letter case.

```
#include <extras/extras_string.h>

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

Parameter

`str1`

A pointer to a character string.

`str2`

A pointer to a character string.

`max`

The maximum number of characters to compare.

Remarks

This function operates identically to the `strncasecmp()` function.

This facility may not be available on some configurations of the EWL.

27.5.9 strncoll()

Locale-aware collating string comparison with limited length.

```
#include <extras/extras_string.h>

int strncoll(const char *s1, const char *s2, size_t max);
```

Parameter

`str1`

A pointer to a character string.

`str2`

A pointer to a character string.

`max`

The maximum number of characters to compare.

Remarks

The function performs the comparison according to the collating sequence specified by the `LC_COLLATE` component of the current locale. The function stops when it has reached a null character in one of the strings or when it has compared max characters.

This function returns one of these values:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2
- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.10 strnicmp()

Character string comparison with length specified and ignored letter case.

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

Parameter

`str1`

A pointer to a character string.

`str2`

A pointer to a character string.

`max`

The maximum number of characters to compare.

Remarks

This function operates identically to the `strncasecmp()` function.

This facility may not be available on some configurations of the EWL.

27.5.11 strnicoll()

Locale-aware collating string comparison that ignores letter case and limits length.

```
#include <extras/extras_string.h>
int stricoll(const char *s1, const char *s2, size_t max);
```

Parameter

`s1`

A pointer to a null-terminated character string.

`s2`

A pointer to a null-terminated character string.

`max`

The maximum number of characters to compare.

Remarks

This function compares each character at s1 and s2 using the collating sequence specified by the LC_COLLATE component of the current locale. It ignores the case of alphabetic characters. The function stops when it reaches a null character or when it has compared max characters.

This function returns one of these values:

- zero if all characters in s1 are identical to and appear in the same order as the characters in s2
- a negative value if the numeric value of first non-matching character in s1 is less than its counterpart in s2
- a positive value if the numeric value of first non-matching character in s1 is greater than its counterpart in s2

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.12 strnset()

Fills a character string with a character, limiting length.

```
#include <extras/extras_string.h>
char * strnset(char *str, int c, size_t max);
```

Parameter

`str`

A pointer to a character string.

`c`

The character to fill with.

`max`

The maximum number of characters to fill.

Remarks

This function stores c in the string pointed to by str. The function stops filling the string when it reaches a null character (which it leaves intact) or when it has filled max characters. The function always ensures that the character string is null-terminated.

The function returns `str`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.13 strrev()

Reverses a null-terminated string.

```
#include <extras/extras_string.h>
char * strrev(char *str);
```

Parameter

`str`

A pointer to the null-terminated string to reverse.

Remarks

This function reverses the null-terminated string at str and returns a pointer to it.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.14 strset()

Fills a character string with a character.

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

Parameter

`str`

A pointer to a character string.

`c`

The character to fill with.

Remarks

This function stores c in the string pointed to by str. The function always ensures that the character string is null-terminated.

The function returns `str`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.15 strspnp()

Returns pointer to first character in a string that is not in another.

```
#include <extras/extras_string.h>

char * strspnp(char *s1, const char *s2);
```

Parameter

`s1`

A pointer to a null-terminated character string to search.

`s2`

A pointer to a null-terminated character string containing characters to search for.

Remarks

This function determines the first position in the string at s1 that does not have a character in s2.

The function returns a pointer to a position in s1 or `NULL`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.16 strupr()

Converts a character string to uppercase.

```
#include <extras/extras_string.h>

char * strupr(char *str);
```

Parameter

`str`

A pointer to a null-terminated character string.

Remarks

This function converts all lowercase alphabetic characters at str to their uppercase counterparts. The function does not modify any other characters. The function returns a pointer to `str`.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.5.17 strtok_r()

Thread-safe extraction of tokens within a character array.

```
#include <extras/extras_string.h>

char *strtok_r(char *str, const char *sep, char** tmp);
```

Parameter

`str`

A pointer to a character string to separate into tokens.

`sep`

A pointer to a character string containing separator characters.

`tmp`

A pointer to a character pointer.

Remarks

This function performs a thread-safe operation of its counterpart, `strtok()`. Unlike `strtok()`, this function takes an extra argument, tmp, which the function uses to store its progress.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

Listing: Example of strtok_r() usage

```
#include <extras/extras_string.h>
#include <stdio.h>
int main(void)
{
char s1[] = "(a+b)*(c+d)";
char s2[] = "(e*f)+(g*h)";
char *token;
char *separator = "()+*";
char *tmp1;
char *tmp2;
/* First calls to strtok_r(). */
token = strtok_r(s1, separator, &tmp1);
puts(token);
token = strtok_r(s2, separator, &tmp2);
puts(token);
/* Subsequent calls to strtok_r(). */
token = strtok_r(NULL, separator, &tmp1);
puts(token);
token = strtok_r(NULL, separator, &tmp2);
puts(token);
return 0;
}
Output:
a
e
b
f
```

27.6 extras_time.h

Extra date and time facilities.

27.6.1 asctime_r()

Thread-safe conversion of a `tm` structure to a character array.

```
#include <extras/extras_time.h>
char * asctime_r(const struct tm * t, char * s);
```

Parameter

`t`

A pointer to a `tm` structure describing the time and date to convert.

`s`

A pointer to a character string array in which to store the result.

Remarks

This function provides a reentrant version of the `asctime()` function. Unlike `asctime()`, this function requires that the caller provides storage for the string in which to store the textual representation of the date described by t. The size of this character array must be at least 26 characters.

The `asctime_r()` function always returns the value of s.

This function may require extra library support.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.6.2 ctime_r()

Thread-safe conversion of a value of type `time_t` to a null-terminated character array.

```
#include <extras/extras_time.h>
char* ctime_r(const time_t * timer, char* s);
```

Parameter

`timer`

A pointer to a value of type `time_t` to convert.

`s`

A pointer to a character string array in which to store the result.

Remarks

This function provides the same service as `ctime()`. Unlike `ctime()`, this function requires that the caller provides the storage for string s and the size of the storage must be at least 26 characters long.

The function always returns the value of `s`.

This function may require extra library support.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.6.3 gmtime_r()

Thread-safe conversion of a `time_t` value to Coordinated Universal Time (UTC).

```
#include <extras/extras_time.h>
struct tm * gmtime_r(const time_t *time, struct tm * st);
```

Parameter

`time`

A pointer to a time value.

`st`

A pointer to a time structure.

Remarks

This function provides the same service as `gmtime()`. Unlike `gmtime()`, this function requires that the caller provides the storage for the `tm` structure. The `gmtime_r()` function always returns the value of `st`.

This function may require extra library support.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.6.4 localtime_r()

Thread-safe conversion of a value of type `time_t` to a structure of type tm.

```
#include <extras/extras_time.h>
struct tm * localtime_r(const time_t * time, struct tm * ts);
```

Parameter

`time`

A pointer to a time value to convert.

`ts`

A pointer to a time structure in which to store the converted time information.

Remarks

This function provides the same service as `localtime()`, but is reentrant. Unlike `localtime()` this function requires that the caller provides the storage for the `tm` structure.

The function always returns the value of `ts`.

This function may require extra library support.

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

This facility may not be available on some configurations of the EWL.

27.6.5 strdate()

Stores a textual representation of the current date in a character string.

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

Parameter

`str`

A pointer to a character string in which to store the current date.

Remarks

This function stores a null-terminated character string of the date in the buffer pointed to by str. The format of this string is `mm/dd/yy`. The buffer must be at least 9 characters long.

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

This facility is not specified in the ISO/IEC standards. It is an EWL extension of the standard libraries.

27.7 extras_wchar.h

Defines non-standard facilities for wide characters.

```
#include <extras/extras_wchar.h>
wchar_t *itow(int n, wchar_t *str, int radix);
double watof(const wchar_t *str);
wchar_t *wcsdup(const wchar_t *str);
int wcsicmp(const wchar_t *s1, const wchar_t *s2);
int wcsicoll(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcslwr(wchar_t *str);
int wcsncoll(const wchar_t *s1, const wchar_t *s2, size_t max);
int wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t n);
int wcsnicoll(const wchar_t *s1, const wchar_t *s2, size_t max);
wchar_t *wcsnset(wchar_t *str, wchar_t wc, size_t n);
wchar_t *wcsrev(wchar_t *str);
wchar_t *wcsset(wchar_t *str, wchar_t wc);
wchar_t *wcsspnp(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsupr(wchar_t *str);
wchar_t *wstrrev(wchar_t * str);
int wtoi(const wchar_t *a);
```

These non-standard, wide-character functions operate identically to regular-sized character functions in the rest of the standard library.

The table below matches these wide character functions to equivalent char-based functions.

Table 27-2. Wide-character Functions Equivalent to Char-based Functions

Function	Wide Character Equivalent	Header File
itow()	itoa()	extras_wchar.h
watof()	atof()	stdlib.h
wcsdup()	strdup()	extras_wchar.h
wcsicmp()	stricmp()	extras_wchar.h
wcsicoll()	stricoll()	extras_wchar.h
wcslwr()	strlwr()	extras_wchar.h
wcsncoll()	strncoll()	extras_wchar.h

Table continues on the next page...

Table 27-2. Wide-character Functions Equivalent to Char-based Functions (continued)

Function	Wide Character Equivalent	Header File
wcsnicmp()	strnicmp()	extras_wchar.h
wcsnicoll()	strnicoll()	extras_wchar.h
wcsnset()	strnset()	extras_wchar.h
wcsrev()	strrev()	extras_wchar.h
wcsset()	strset()	extras_wchar.h
wcsspnp()	strspnp()	extras_wchar.h
wcsupr()	strupr()	extras_wchar.h
wstrrev()	strrrev()	extras_wchar.h
wtoi()	atoi()	stdlib.h

27.8 stat.h

Extra date and time facilities.

File manipulation facilities for UNIX compatibility.

27.8.1 Data Types in stat.h

The `stat.h` header file defines several data types. The table below lists the types used to describe a file's properties.

Table 27-3. Data Types in stat.h

Data Type	Represents
dev_t	Device type.
gid_t	Group ID.
ino_t	Serial number.
mode_t	File attributes.
nlink_t	Number of links to a file.
off_t	File size, in bytes.
uid_t	User ID.

The `stat` structure contains information about a file. The table below lists the members in this structure.

Table 27-4. Structure Members in stat.h

Structure	Data Type	Contains
st_mode	mode_t	File attributes.
st_ino	ino_t	File serial number.
st_dev	dev_t	ID of device on which this file is stored.
st_nlink	nlink_t	Number of links to the file.
st_uid	uid_t	The file owner's user ID.
st_gid	gid_t	The file owner's group ID.
st_size	off_t	The file's size, in bytes.
st_atime	time_t	Time of last access to the file.
st_mtime	time_t	Time of the last modification to the file.
st_ctime	time_t	Time that the file was created.
st_blksize	long	Optimal block size.
st_blocks	long	Number of blocks allocated to the file.

The table below describes the file attributes that the facilities in `stat.h` recognize.

Table 27-5. File Modes

Mode	File Property
S_IFMT	File type.
S_IFIFO	FIFO queue.
S_IFCHR	Character special.
S_IFDIR	Directory.
S_IFBLK	Blocking stream.
S_IFREG	Regular file.
S_IFLNK	Symbolic link.
S_IFSOCK	Socket.
S_IRGRP	Read permission, file group class.
S_IROTH	Read permission, file other class.
S_IRUSR	Read permission, file owner class.
S_IRWXG	Permission, file group class.
S_IRWXO	Permission, file group class.
S_IRWXU	Permission, file group class.
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	Execute permission, file group class
S_IXOTH	Execute permission, file other class.
S_IXUSR	Execute permission, file owner class.

27.8.2 mkdir()

Makes a directory.

```
#include <extras/sys/stat.h>
int mkdir(const char *path, int mode);
int _mkdir(const char *path);
```

Parameter

`path`

The path name, including the new directory name.

`mode`

The open mode. Ignored.

Remarks

These functions create the new folder specified by path. It ignores the argument mode.

If successful, these functions returns `zero`. If they encounter an error, these functions returns -1 and set `errno`.

This facility may not be available on some configurations of the EWL.

Listing: Example for mkdir() Usage

```
#include <stdio.h>
#include <extras/sys/stat.h>
int main(void)
{
if (mkdir("Asok", 0) == 0)
printf("Directory 'Asok' is created.");
return 0;
}
```

27.8.3 rmdir()

Removes a directory.

```
#include <extras/sys/stat.h>
int rmdir(const char *path);
```

Parameter

`path`

The path name, including the directory name to be removed.

Remarks

This function removes a folder specified by path.

If successful, this function returns `zero`. If it encounters an error, this function returns -1 and set `errno`.

This facility may not be available on some configurations of the EWL.

Listing: Example for rmdir() Usage

```
#include <stdio.h>
#include <extras/sys/stat.h>
int main(void)
{
if (rmdir("Asok", 0) == 0)
printf("Directory 'Asok' was removed.");
return 0;
}
```

Index

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWEWLCREF
Revision 10.x, 02/2014