cbase

A C Foundation Library
Version 1.4
25 Apr 2025

Mark A. Lindner

Table of Contents

1 Overview of cbase

Cbase is a library that aims to simplify systems software development under UNIX. The library consists of several groups of functions and macros that simplify such common tasks as memory allocation, string parsing, subprocess execution, filesystem traversal, and so on.

In addition, the library includes efficient implementations of some common data structures, such as linked lists, queues, and hash tables, as well as other less common data structures. Many of these types of tasks involve tricky pointer arithmetic and memory management and are inherently error-prone. Moving this common logic into a library frees the developer from having to recode and debug this functionality each time it is needed.

Finally, the library provides a simple, high-level interface to several UNIX IPC mechanisms, including semaphores, shared memory, and Berkeley sockets.

The following chapters describe all of the datatypes, constants, macros, and functions in the library in detail. Function and datatype indices are provided at the end of the manual.

1.1 Using cbase

The *cbase* library exists in two configurations (the multi-threaded configuration and the non-threaded configuration) and two forms (a static library and a shared library), for a total of four versions. The single-threaded versions are named '`libcbase.a`' and '`libcbase.so`', and the multi-threaded versions are named '`libcbase_mt.a`' and '`libcbase_mt.so`'.

Therefore, to link with the single-threaded version of the *cbase* library, issue a command such as:

```
gcc file1.o file2.o -o myprogram -lcbase
```

And similarly, to link with the multi-threaded version:

```
gcc file1.o file2.o -o myprogram -lcbase_mt
```

If your program uses the networking functions, you may need to link with additional libraries. No additional libraries are required for GNU/Linux. For Solaris, the *socket* and *nsl* libraries are required, e.g.:

```
gcc file1.o file2.o -o myprogram -lcbase -lsocket -lnsl
```

It is extremely important that multi-threaded programs *only* be linked with the multi-threaded version of the library, and that single-threaded programs *only* be linked with the non-threaded version. Other combinations will produce undefined behavior in your programs.

On some systems, if both shared and static versions of a given library are present (as they are with cbase), the linker selects the shared library by default, producing a dynamically linked executable. When a program is linked in this way, all library dependencies must be specified at link time, so that the linker can generate all of the necessary library stubs in the executable. Since the cbase library contains references to functions that are typically defined in various system libraries, those libraries must be explicitly linked into your program. This list varies by operating system, but typically includes *crypt* (the crypto library), *rt* (the POSIX real-time library), and *dl* (the dynamic runtime linker library).

It is possible to force the linker to link against static libraries, producing a statically linked executable. The appropriate switches vary across linkers and operating systems. If you're using **gcc**, you can use the **-static** switch to accomplish this. For example:

```
gcc -static file1.o file2.o -o myprogram -lcbase
```

You can simplify the compiling and linking process, particularly for static linking, by using the **pkg-config** utility (version 0.20 or newer) to produce the appropriate preprocessor and linker flags for cbase. Make sure the environment variable '`PKG_CONFIG_PATH`' is defined to include the absolute path to the '`lib/pkgconfig`' directory beneath the cbase installation directory. Then, you can compile a program that uses cbase as follows:

```
gcc -static file1.c file2.c -o myprogram \
  `pkg-config --cflags --libs --static libcbase`
```

Substitute '`libcbase_mt`' for '`libcbase`' above when linking with the multithreaded version of the library. Omit the '`-static`' and '`--static`' switches if dynamic linking is desired. The backquoted expression will evaluate to a list of all necessary preprocessor and linker flags, including flags for any additional system libraries that are required.

All of the definitions in the library can be made available in your source code by including the master header file '`cbase/cbase.h`'.

1.2 API Conventions

This section describes the naming and calling conventions for constants, macros, datatypes, and functions in the *cbase* library.

All functions and macros begin with the prefix '`C_`'.

All constants begin with the prefix '`C_`', with the following exceptions: `TRUE`, `FALSE`, `NUL`, and `CRLF`.

All datatypes begin with the prefix '`c_`' and end with the suffix '`_t`', with the following exception: *uint_t*.

Unless their fields are expressly documented, all datatypes which begin with the prefix '`c_`' should be considered *opaque*. This means that pointers to these datatypes serve only as "handles" which are passed to and returned by the functions which operate on those datatypes. The caller should never directly modify these datatypes or the fields of the data structures that they represent. The internal layout of these data structures may change in future versions of the library; manipulating them only through API calls will ensure that your code will continue to compile and function properly.

Most functions which return a pointer will by convention return `NULL` on failure.

Functions which return a boolean (*c_bool_t*) or an integer will by convention return `TRUE` or a nonzero value to indicate success and `FALSE` or 0 to indicate failure. Note that this is different from the typical UNIX convention of returning `0` on success and a nonzero value (typically `-1`) on failure. Since `TRUE` is defined as a nonzero value and `FALSE` is defined as `0`, the following forms are equivalent:

```
if(C_system_cdhome() == FALSE)
  puts("Failed.");

if(! C_system_cdhome())
  puts("Failed.");
```

1.3 Reentrancy and Threading

With some exceptions (as documented), the functions in the *cbase* library are reentrant. This means that they can be safely accessed from concurrent threads without the need for explicit synchronization.

However, calls to functions which manipulate data structures (such as linked lists, hash tables, and string buffers) must be synchronized by the caller in such a way that only one thread is modifying the data structure at any given time. For example, an insert into a linked list by one thread with a concurrent insert (or some other operation) on the same linked list by another thread might result in corruption of the data structure.

In most cases, it is necessary to use reader/writer locks to ensure that a reader (a thread that is accessing but not modifying the data structure) does not access the data structure while another thread is modifying it.

1.4 Portability Notes

This version of the library works with recent versions of GNU/Linux and MacOS. It may work on other UNIX-based or UNIX-like platforms (e.g, Solaris, IRIX), but it has not been tested on such platforms for many years, on account of the general obsolescence of those platforms.

The real-time scheduler is based on the POSIX.4 real-time signal facility. At the time this code was written, this facility was only available on Solaris; it was broken on GNU/Linux and was not available at all on OS X. On the latter platforms, the event loop in the multithreaded version of the library is therefore implemented using `nanosleep()` instead.

2 Types, Macros and Constants

This chapter describes convenience types, macros, and constants. These are defined in the header '`cbase/defs.h`'.

2.1 Basic Types

The type *c_bool_t* represents a boolean value that (by convention) can take on the values `TRUE` or `FALSE`; these are macros defined to be `(1)` and `(0)`, respectively.

The type *c_byte_t* represents an unsigned 8-bit value (0 - 255).

The type *uint_t* represents an unsigned integer.

2.2 Convenience Macros

The following convenience macros are provided:

`C_max` (*a*, *b*) [Macro]
`C_min` (*a*, *b*) [Macro]
`C_sgn` (*a*) [Macro]

The first two macros correspond to the mathematical *max* and *min* functions. `C_max()` returns *a* if *a* `>` *b* and *b* otherwise. `C_min()` returns *a* if *a* `<` *b* and *b* otherwise. The third macro corresponds to the mathematical *sgn* function. It returns `-1` if *a* `<` 0, `1` if *a* `>` 0, or `0` if *a* == 0. Note that these macros may evaluate their arguments more than once, so they should not be applied to expressions that have side-effects (e.g., "`b++`").

`C_bit_set` (*i*, *b*) [Macro]
`C_bit_clear` (*i*, *b*) [Macro]
`C_bit_isset` (*i*, *b*) [Macro]

These macros set, clear, and test the *b*'th bit of *i* (which is presumably an integer) using bitwise operators. Each evaluates its arguments only once.

`C_offsetof` (*type*, *element*) [Macro]

This macro returns the offset, in bytes, of the element named *element* in the aggregate type *type* (which is presumably a *struct*). It works similarly to the X11 macro `XtOffsetOf()`.

`C_lengthof` (*array*) [Macro]

This macro returns the length of (the number of elements in) the array *array*, which is assumed to be a stack-allocated array. If *array* is a pointer, the results are undefined.

2.3 Miscellaneous Constants

The header file also defines the following constants:

`NUL` The **NUL** character (ASCII value 0); not to be confused with `NULL`, the *NULL* pointer, a constant that is defined in '`stdio.h`'.

`CRLF` The string `"\r\n"`, or carriage return and line feed.

3 System Functions

This chapter describes functions involving system facilities such as I/O, subprocess execution, error handling, and memory management. They are divided into several groups; the functions in a group share a common name prefix for that group; e.g., all filesystem-related functions have names that begin with '`C_file_`'. All of the constants, macros, and functions described in this chapter are defined in the header '`cbase/system.h`'.

The following sections describe each group in detail.

3.1 Byte Order Conversion Functions

The following functions convert various numeric types between host and network byte order.

`uint16_t C_byteord_htons` (*uint16_t* `val`) [Function]
`uint16_t C_byteord_ntohs` (*uint16_t* `val`) [Function]

These functions convert an unsigned 16-bit "short" integer *val* to and from network byte order, respectively. The functions return the converted value.

`uint32_t C_byteord_htonl` (*uint32_t* `val`) [Function]
`uint32_t C_byteord_ntohl` (*uint32_t* `val`) [Function]

These functions convert an unsigned 32-bit "long" integer *val* to and from network byte order, respectively. The functions return the converted value.

`uint64_t C_byteord_htonll` (*uint64_t* `val`) [Function]
`uint64_t C_byteord_ntohll` (*uint64_t* `val`) [Function]

These functions convert an unsigned 64-bit "long long" integer *val* to and from network byte order, respectively. The functions return the converted value.

`float C_byteord_htonf` (*float* `val`) [Function]
`float C_byteord_ntohf` (*float* `val`) [Function]

These functions convert an 32-bit floating point value *val* to and from network byte order, respectively. The functions return the converted value.

`double C_byteord_htond` (*double* `val`) [Function]
`double C_byteord_ntohd` (*double* `val`) [Function]

These functions convert an 64-bit double-precision floating point value *val* to and from network byte order, respectively. The functions return the converted value.

3.2 Debugging and Tracing Functions

The following functions are provided to aid in the debugging and tracing of code.

`void C_debug_printf` (*const char* *`*format`, ...) [Function]

This function is similar to `printf()`, and is intended for use in generating debug output. The function is actually implemented as a macro which evaluates to a call to an internal library function if the `DEBUG` macro is defined; otherwise, it is defined as a no-op, which essentially prevents the debug call from being compiled into the calling code.

Debug messages are written to the debugging stream (`stderr` by default). The stream is explicitly flushed after the message is written.

If tracing is enabled, the message will be preceded by the source file name and line number of the `C_debug_printf()` call. In the multi-threaded version of the library, the message will be preceded by the calling thread's ID as well.

`void C_debug_set_trace` (*c_bool_t* ***flag***) [Function]
`void C_debug_set_stream` (*FILE ** ***stream***) [Function]

These functions alter the behavior of the `C_debug_printf()` function described above.

`C_debug_set_trace()` enables or disables tracing based on the value of *flag*. If tracing is enabled, the filename and line number of the `C_debug_printf()` call will be prepended to each line of debug output.

`C_debug_set_stream()` sets the output stream for debug messages to *stream*; the default stream is `stderr`.

`void C_debug_set_termattr` (*c_bool_t* ***flag***) [Function]

This function enables or disables the use of ANSI color and text style terminal attributes for debug messages. This feature is enabled by default, and causes all debug messages (when written to a tty) to be printed in a bold font, and assertion failure messages in particular to be printed in red.

`C_assert` (***expr***) [Macro]

This macro evaluates an assertion; it is provided as a replacement for the more rudimentary `assert()` C library function. The macro works as follows.

If the expression *expr* evaluates to zero (`0`), the assertion fails. A message that indicates the failure and contains the text of the expression itself is written to the debugging stream in the same manner as with `C_debug_printf()`, and then the process is aborted via a call to the `abort()` C library function. For all other (non-zero) values, the macro behaves as a no-op.

3.3 Dynamic Linker Functions

The following functions provide a means to dynamically load and unload object files at runtime and to obtain pointers to symbols (including variables and functions) defined in those files. These functions are based on the `dlopen()`, `dlsym()`, and `dlclose()` library functions.

The type *c_dlobject_t* represents a loadable object.

`c_dlobject_t * C_dlobject_create` (*const char ****path***) [Function]
`c_bool_t C_dlobject_destroy` (*c_dlobject_t ****obj***) [Function]

These functions create and destroy loadable objects. `C_dlobject_create()` creates a new loadable object for the object file specifed by *path*. The object will be created in a non-loaded state. The function returns a pointer to the new loadable object structure on success, or `NULL` on failure.

`C_dlobject_destroy()` destroys the loadable object *obj*, if it is not currently loaded. It returns `TRUE` on success and `FALSE` on failure.

`c_bool_t C_dlobject_load` (*c_dlobject_t ****obj***, *c_bool_t* ***lazy***) [Function]
`c_bool_t C_dlobject_unload` (*c_dlobject_t ****obj***) [Function]

These functions load and unload the loadable object *obj*. `C_dlobject_load()` loads the object *obj* into memory. The flag *lazy* specifies whether symbols will be resolved as they are accessed (lazy relocation), or all at once when the object is loaded. The function will fail if *obj* is already loaded.

`C_dlobject_unload()` unloads the loadable object *obj*. The function will fail if *obj* is not currently loaded.

The functions return `TRUE` on success and `FALSE` on failure. In the event of a load/unload failure, a linker-specific error is stored in *obj* and may be accessed via `C_dlobject_error()`, which is described below.

`void * C_dlobject_lookup` (*c_dlobject_t* ****obj***, *const char* ****symbol***) [Function]

This function looks up the symbol named *symbol* in the loadable object *obj*. It returns a pointer to the symbol on success, or `NULL` on failure. In the event of a lookup failure, a linker-specific error is stored in *obj* and may be accessed via `C_dlobject_error()`, which is described below.

`c_bool_t C_dlobject_isloaded` (*c_dlobject_t* ****obj***) [Function]

This function (which is implemented as a macro) returns `TRUE` if the loadable object *obj* is currently loaded, and `FALSE` otherwise.

`const char * C_dlobject_error` (*c_dlobject_t* ****obj***) [Function]

In the case of a failed call to one of the dynamic linker library functions, an error message is stored in *obj*; this function (which is implemented as a macro) returns that message.

`const char * C_dlobject_path` (*c_dlobject_t* ****obj***) [Function]

This function (which is implemented as a macro) returns the file path of the loadable object *obj*.

3.4 Error Handling Functions

The following functions are provided to simplify the reporting of user- and system-level error messages to the console.

`void C_error_init` (*const char* ****progname***) [Function]

A call to this function initializes the error handling routines. The argument *progname* is the name of the currently executing program, which can be obtained from the argument list as `argv[0]`. The function internally stores a copy of this pointer.

`void C_error_printf` (*const char* ****format***, *...*) [Function]

This function receives arguments in the same manner as the `printf()` library function. It writes the program name to standard error, then passes its arguments to the `vfprintf()` library function for formatted output to standard error. It then flushes the standard error stream.

`void C_error_usage` (*const char* ****usage***) [Function]

This function prints the command-line usage information message *usage* to standard error. It then flushes the standard error stream.

`void C_error_syserr` (*void*) [Function]

This function is a higher-level interface to the `strerror()` library function. It obtains the error code from the latest system call executed, formats it as a string, and writes it to standard error. All other functionality is identical to `C_error_printf()` above.

`const char * C_error_string` (*void*) [Function]

This function returns a textual error message for the error code from the last-executed cbase library routine.

`int C_error_get_errno` (*void*) [Function]

This function returns the error code from the last-executed cbase library routine. A return value of `0` by convention denotes that the last call executed successfully (no error); note however, that most library routines do not modify the error code at all if they complete successfully. In the multi-threaded version of the library, this function returns a thread-specific error value. In the single-threaded version, it simply returns the value of `c_errno`.

Calling this routine is equivalent to evaluating `c_errno`, which is defined as a macro that evaluates to a call to `C_error_get_errno()`.

Therefore in both single-threaded and multi-threaded code, it is safe to call `C_error_get_errno()` or to evaluate `c_errno` as if it were a global variable; either approach will correctly return the error code for the current thread or process. Note that since `c_errno` evaluates to a function call, it cannot be used as an lvalue; that is, it is not possible to assign values to it.

`void C_error_set_errno` (*int* `err`) [Function]

This function sets the error code for the currently executing cbase library routine to *err*. This function is provided for use by cbase extension libraries and is not intended for use by user code.

In the multi-threaded version of the library, `C_error_set_errno` is defined as a function that sets a thread-specific error value. Otherwise, it is defined as a function that simply assigns *err* to the global *int* variable `c_errno`.

3.5 Exception Handling Functions

The cbase library provides a rudimentary form of exception handling built on top of the `setjmp()` and `longjmp()` library functions. This facility can be used to simplify error handling and avoid the use of `goto` statements to skip over logic once an error is encountered.

Exceptions are `int` values. For improved readability, a program can define its exceptions as a set of integer constants or as an enumeration.

`C_try {` ... `}` [Macro]
`C_catch (`*`variable`*`) {` ... `}` [Macro]
`C_throw (`*`exception`*`)` [Macro]

These macros implement the customary 'try', 'catch', and 'throw' exception handling statements.

`C_try` introduces a *try* block, which is a new scope that must be enclosed in braces.

The *try* block must be followed by a *catch* block, which is also a new scope that must be enclosed in braces.

`C_catch` takes a single argument, *variable*, which is the name of a variable. Within the *catch* block, an `int` variable with that name will be assigned to the exception value that was thrown.

`C_throw` throws the exception *exception*, which must be a non-zero integer. (If *exception* is 0, the behavior is undefined.) The stack is unwound to the topmost frame of the *try* block, all remaining logic in the *try* block is skipped, and control jumps to the beginning of the *catch* block.

Attempting to *throw* or *catch* when there is no enclosing *try* block is a programming error, and results in an assertion.

The library maintains a stack of exception contexts, so *try*/*catch* blocks can be nested to arbitrary depth.

`C_throws` (*exception*, ...) [Macro]

This macro may be used to annotate a function with the list of exceptions that it can potentially throw. This macro has no behavior; it is simply used as a documentation aid. For example:

```
extern int some_function(int arg) C_throws(FILE_ERROR, IO_ERROR);
```

The following code snippet illustrates the use of the exception handling macros. Note that in this example, the `open_file()` and `read_int_from_file()` functions, which are defined elsewhere, can potentially throw exceptions. Note also that in the case of an exception, the file still must be closed in the *catch* block to avoid a resource leak.

```
const int FILE_ERROR = 1;
const int IO_ERROR = 2;
const int BAD_VALUE_ERROR = 3;

extern FILE *open_file(const char *path) C_throws(FILE_ERROR);
extern int read_int_from_file(FILE *fp) C_throws(IO_ERROR);
extern void close_file(FILE *fp);

C_try
{
  FILE *fp;
  int x;

  fp = open_file("./data.txt");

  x = read_int_from_file(fp);
  if (x <= 0)
    C_throw(BAD_VALUE_ERROR);

  close_file();
}
C_catch(exc)
{
  close_file();

  switch (exc)
  {
    case FILE_ERROR:
      printf("File error occurred.\n");
      break;
    case IO_ERROR:
      printf("I/O error occurred.\n");
      break;
    case BAD_VALUE_ERROR:
      printf("Invalid value read from file.\n");
   }
}
```

3.6 Process Control Functions

The following functions provide subprocess control, including higher-level interfaces to system calls such as `execv()`, and piping.

`int C_exec_run` (*char* ****argv**, *int* **fdin**, *int* **fdout**, *c_bool_t* **waitf**) [Function]
`int C_exec_run_cwd` (*char* ****argv**, *int* **fdin**, *int* **fdout**, *c_bool_t* **waitf**, *const char* ***cwd**) [Function]

`C_exec_run()` is a higher-level interface to the `execv()` system call. It executes the command specified by *argv* as a subprocess, connecting its standard input stream to the *fdin* file descriptor (or to '`/dev/null`' if *fdin* is negative) and its standard output and standard error streams to the *fdout* file descriptor (or to '`/dev/null`' if *fdout* is negative). If *waitf* is `TRUE`, the function additionally performs a `waitpid()` system call to wait for the subprocess to finish executing.

`C_exec_run_cwd()` is identical to `C_exec_run()`, except that the additional argument *cwd* specifies a new working directory for the spawned subprocess. If this path does not exist or is not readable, the working directory of the subprocess will remain unchanged.

If *waitf* is `TRUE`, the functions return the exit value from the subprocess. Otherwise, they return `0` on success or `-1` on failure.

`int C_exec_va_run` (*int* **fdin**, *int* **fdout**, *c_bool_t* **waitf**, *... /* , NULL */*) [Function]
`int C_exec_va_run_cwd` (*int* **fdin**, *int* **fdout**, *c_bool_t* **waitf**, *const char* ***cwd**, *... /* , NULL */*) [Function]

These are variable argument list versions of `C_exec_run()` and `C_exec_run_cwd()`. The command name and arguments are passed as a `NULL`-terminated list of *char ** arguments rather than as a string vector. The other arguments have the same meaning as in `C_exec_run()` and `C_exec_run_cwd()`, and the functionality is identical.

`int C_exec_pipefrom` (*char* ****argv**, *int* ***fd**) [Function]
`int C_exec_pipefrom_cwd` (*char* ****argv**, *int* ***fd**, *const char* ***cwd**) [Function]

`C_exec_pipefrom()` executes the command specified by *argv* as a subprocess, redirecting its standard input stream to '`/dev/null`', and connecting its standard output and standard error streams to a new file descriptor whose value is stored at *fd*. It returns immediately after forking the subprocess. Subsequent reads from the file descriptor at *fd* will effect a piping of output from the subprocess into the caller.

`C_exec_pipefrom_cwd()` is identical to `C_exec_pipefrom()`, except that the additional argument *cwd* specifies a new working directory for the spawned subprocess. If this path does not exist or is not readable, the working directory of the subprocess will remain unchanged.

The functions return `0` on success, or `-1` on failure.

The following code illustrates the use of `C_exec_pipefrom()` to process the output from an execution of the `ls` program.

```
int fd, r;
char buf[100], **args;
FILE *fp;

args = C_string_va_makevec(NULL, "/bin/ls", "-al",
                           "/usr/local/bin", NULL);
r = C_exec_pipefrom(args, &fd);
if(r != -1)
{
  fp = fdopen(fd, "r");
  while(C_io_gets(fp, buf, sizeof(buf), '\n') != EOF)
    printf("Received: %s\n", buf);
  fclose(fp);
}

C_free_vec(args);
```

`int C_exec_pipeto` (*char* **`argv`, *int* *`fd`) [Function]
`int C_exec_pipeto_cwd` (*char* **`argv`, *int* *`fd`, *const char* *`cwd`) [Function]

`C_exec_pipeto()` executes the command specified by *argv* as a subprocess, redirecting its standard output and standard error streams to '`/dev/null`', and connecting its standard input stream to a new file descriptor whose value is stored at *fd*. It returns immediately after forking the subprocess. Subsequent writes to the file descriptor at *fd* will effect a piping of input into the subprocess from the caller.

`C_exec_pipeto_cwd()` is identical to `C_exec_pipeto()`, except that the additional argument *cwd* specifies a new working directory for the spawned subprocess. If this path does not exist or is not readable, the working directory of the subprocess will remain unchanged.

The functions return `0` on success, or `-1` on failure.

`int C_exec_va_call` (*const char* *`arg`, *... /* , NULL */*) [Function]

This function is intended for use as a replacement for the unsafe `system()` library function. It executes in a subprocess the command specified by the `NULL`-terminated variable argument list, waits for the subprocess to finish, and returns the exit value returned by the subprocess. No stream redirection takes place: the subprocess reads from and writes to the parent's standard I/O streams. This function is implemented using a call to `C_exec_run()`, which calls the `execvp()` system call.

On success, the function returns the exit status from the subprocess. On failure, it returns `-1`.

`int C_exec_wait` (*pid_t* **`pid`**) [Function]

This function is an interface to the `waitpid()` system call. It waits for the process with process ID of *pid* to complete, and returns its exit status.

3.7 Filesystem Functions

The following functions provide for the manipulation of files and directories in the UNIX filesystem.

`c_bool_t C_file_readdir` (*const char **`path`, *c_dirlist_t **`dir`, *int* `flags`) [Function]

This function reads the names of the files in the directory specified by *path* into the directory list pointed to by *dir*. The type *c_dirlist_t* represents a directory file list. Specific directory reading options are specified in the *flags* argument, which is a bitwise OR of the following macros:

`C_FILE_SKIPDOT`
: Specifies that the '`.`' entry (referring to the current directory) not be included in the list.

`C_FILE_SKIP2DOT`
: Specifies that the '`..`' entry (referring to the parent directory) not be included in the list.

`C_FILE_SKIPHIDDEN`
: Specifies that hidden files (those beginning with '`.`') not be included in the list.

`C_FILE_ADDSLASH`
: Specifies that a slash ('`/`') be appended to directory names.

`C_FILE_SKIPDIRS`
: Specifies that directories not be included in the list.

`C_FILE_SKIPFILES`
: Specifies that ordinary files not be included in the list.

`C_FILE_SEPARATE`
: Specifies that directory names and file names be separated into two separate lists.

`C_FILE_SORT`
: Specifies that the resulting list(s) of names be sorted alphabetically.

The *c_dirlist_t* type is a structure that contains the following members:

`char **files`	A vector of file names (or all names, if `C_FILE_SEPARATE` was not specified).
`char **dirs`	A vector of directory names (or `NULL`, if `C_FILE_SEPARATE` was not specified).
`uint_t nfiles`	The number of regular file names read.
`uint_t ndirs`	The number of directory names read.

The function ignores all files which are not regular files or directories, including symbolic links.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *path* is `NULL` or invalid).

`c_bool_t C_file_traverse` (*const char *`path`, c_bool_t* (**`examine`*) (*const char *file, const struct stat *fst, uint_t depth, void *hook*), *void *`hook`*) [Function]

This is a general purpose directory tree traversal function. It begins descending a directory tree rooted at *path*, recursing on subdirectories. For each directory or regular file encountered, it calls the function *examine*() with its name, its *struct stat* information, and the file's depth from the original *path*. If the *examine*() function returns `TRUE`, the traversal continues. Otherwise, if it returns `FALSE`, `C_file_traverse()` sets the working directory back to *path* and returns immediately.

The pointer *hook* may be used to pass arbitrary context data during the traversal. This pointer will be passed to the *examine*() function on each invocation.

The *examine*() function should not change the current working directory, as this will confuse the traversal routine. If changing the working directory is unavoidable, the function should save the working directory on entry and restore it before returning.

The function returns `TRUE` if the tree traversal completed successfully. It returns `FALSE` if any of the arguments were invalid, if it could not set the working directory to *path*, or if *examine*() returned `FALSE` during the traversal.

The following code fragment prints an outline-style list of all of the files and subdirectories beginning at the current working directory.

```
c_bool_t examine(const char *file, const struct stat *fst,
               uint_t depth, void *hook)
{
  int i;

  /* indent and print filename */
  for(i = depth * 2; (i--); putchar(' '));

  puts(file);
  return(TRUE);
}

void outline(void)
{
  C_file_traverse(".", examine, NULL);
}
```

`const char * C_file_getcwd` (*void*) [Function]

This function is a higher-level interface to the `getcwd()` system call. It calls `getcwd()` with an initial path buffer, and if the buffer is too small to hold the entire path, it resizes the buffer and tries again, *ad infinitum*. The function returns a dynamically allocated string containing the current path on success, or `NULL` on failure. The returned string must eventually be freed by the caller.

`c_bool_t C_file_issymlink` (*const char* ***path***) [Function]
`c_bool_t C_file_isdir` (*const char* ***path***) [Function]
`c_bool_t C_file_isfile` (*const char* ***path***) [Function]
`c_bool_t C_file_ispipe` (*const char* ***path***) [Function]

These functions return `TRUE` if the file specified by *path* exists and is of the specified type (symbolic link, directory, regular file, or pipe, respectively), and `FALSE` otherwise.

`c_bool_t C_file_mkdirs` (*const char* ***path***, *mode_t* ***mode***) [Function]

This function creates all intermediate directories specified in *path*. Directories will be created with the permissions specified by *mode*. The function returns `TRUE` on success or `FALSE` on failure.

`void * C_file_load` (*const char* ***path***, *size_t* ***len***) [Function]

This function loads the entire file specified by *path* into memory. The size of the file (that is, the number of bytes read) is stored at *len*.

On success, the function returns a pointer to the dynamically allocated buffer containing the data. The buffer will contain the entire contents of the file, plus a single trailing `NUL` byte; this allows the data to be interpreted as a string if the source was a text file.

On failure (for example, if *path* or *len* is `NULL`, or if the file does not exist or cannot be read) the function returns `NULL`.

3.8 Mandatory File Locking Functions

The following functions implement mandatory file locks. Reader/writer locks and simple filesystem-based semaphores can be implemented using these functions. These functions lock and unlock entire files; finer granularity locks may be implemented using the `fcntl()` system call.

`c_bool_t C_file_lock` (*FILE* ***fp***, *int* ***type***) [Function]

This function locks an open file *fp*. The type of lock is specified by *type* and must be one of `C_FILE_READ_LOCK` (for a read lock) or `C_FILE_WRITE_LOCK` (for a write lock); these constants are defined in '`cbase/util.h`'. If the file referred to by *fp* is currently locked by another thread or process, this function blocks until that lock is released.

The function returns `TRUE` if the lock operation succeeds, and `FALSE` if it fails.

`c_bool_t C_file_trylock` (*FILE* ***fp***, *int* ***type***) [Function]

This function is similar to `C_file_lock()` above, except that if the lock cannot be obtained, the function returns immediately with a value of `FALSE`.

The function returns `TRUE` if the lock operation succeeds, and `FALSE` if it fails.

`c_bool_t C_file_unlock` (*FILE* ***fp***, *int* ***type***) [Function]

This function removes the lock on the file *fp*. The type of lock is specified by *type*, and must match the *type* value that was used for the `C_file_lock()` or `C_file_trylock()` call that established this lock.

The function returns `TRUE` if the unlock operation succeeds, and `FALSE` if it fails.

3.9 I/O Functions

The following functions simplify reading from and writing to files and obtaining input from the console.

`int C_io_getchar` (*uint_t* **`delay`**) [Function]

This function is meant to be used as a replacement for `getchar()` in interactive contexts. It accepts a single character of input from the user by disabling all buffering and character processing on the controlling terminal. Any character can be entered, including those that have special meaning to the shell. The argument *delay* specifies how many seconds to wait for a character to be entered.

If the delay expires before a character is typed, the function returns `EOF`. Otherwise, it returns the ASCII value of the character. The value of *delay* may be 0 to effect a poll of the keyboard.

`int C_io_gets` (*FILE **`fp`, *char **`buf`, *size_t* `bufsz`, *char* `termin`) [Function]

This function is meant to be used as a replacement for the `fgets()` library function. It reads data from the stream *fp* into the buffer *buf* until *bufsz* - 1 bytes have been read or the terminator character *termin* is encountered. Once either condition has occurred, the input buffer is terminated with a `NUL` character. The terminator character is discarded.

The function returns the number of bytes actually read, or `EOF` if no more characters are available. A return value of `0` means that the terminator character was the first character encountered; it does not signify an end-of-file condition.

If *fp* or *buf* is `NULL`, the function returns `EOF` immediately.

`int C_io_getpasswd` (*const char **`prompt`, *char **`buf`, *size_t* `bufsz`) [Function]

This function is a general-purpose password input routine. It writes the prompt string *prompt* to standard output, turns off echoing on the controlling terminal, reads data from standard input into the buffer *buf* in the same manner as `C_io_gets()` above, and then turns echoing back on. The input buffer is terminated with a `NUL` character. The newline terminator character is discarded.

`char * C_io_getline` (*FILE **`fp`, *char* `termin`, *int **`len`) [Function]

This function is a dynamic line input routine. It reads a line of unlimited length from the stream *fp*, stopping once the terminator character *termin* is encountered. It allocates memory as needed to store the data being read from the stream.

Once the terminator character is encountered, it is discarded and the string is `NUL`-terminated. The number of characters read (not including the terminator) is stored at *len* if it is non-`NULL`.

On success, the function returns a pointer to the dynamically allocated buffer, which is exactly large enough to hold the `NUL`-terminated string. On failure (for example, on end-of-file), it returns `NULL`.

`char * C_io_getline_buf` (*FILE **`fp`, *char* `termin`, *c_buffer_t **`buf`) [Function]

This function is similar to `C_io_getline()` above, except that the buffer specified by *buf* is used to store the data. This buffer will be resized as necessary to accommodate the data read. The number of characters read (not including the terminator) is stored in the `datalen` field of the buffer *buf*, and may be obtained by using the macro `C_buffer_datalen()`.

On success, the function returns the pointer to the data in *buf*, as returned by the macro `C_buffer_data()`. On failure (for example, on end-of-file), it returns `NULL`.

`C_getchar` () [Macro]

This is a convenience macro. It evaluates to an expression involving a call to `C_io_getchar()`, passing `C_IO_GETCHAR_DELAY` as an argument.

`C_gets` (*buf*, *bufsz*) [Macro]

This is a convenience macro. It evaluates to an expression involving a call to `C_io_gets()`, passing `stdin` as the stream and '`\n`' (newline) as the terminator character. It may be used as a direct replacement for the `gets()` library function.

`C_getline` (*fp*, *len*) [Macro]

This is a convenience macro. It evaluates to an expression involving a call to `C_io_getline()`, passing '`\n`' (newline) as the terminator character.

`int C_io_fprintf` (*FILE ** ***stream***, *const char **__*format*__, ...) [Function]

This is a threadsafe wrapper for the `fprintf()` library function. It locks *stream* via a call to `flockfile()` before proceeding to call `fprintf()`, and then unlocks the stream after that function completes. This ensures that only one thread at a time can write to the stream.

The function returns the value returned by `fprintf()`. In the single-threaded version of the library, this function simply invokes `fprintf()`.

`int C_printf` (*const char **__*format*__, ...) [Function]

This function (which is implemented as a macro) calls `C_io_fprintf()`, described above, passing `stdout` as the stream.

3.10 Logging Functions

The following functions provide a simple, minimal API for writing log messages to the console and/or a log file.

`void C_log_set_console` (*c_bool_t* ***flag***) [Function]

This function enables or disables the writing of log messages to the console (that is, the standard error stream) based on the value of *flag*. By default, log messages are written to the console.

`void C_log_set_stream` (*FILE **__*stream*__) [Function]

This function specifies the *stream* for log messages. If *stream* is `NULL`, logging to a stream will be disabled.

`void C_log_set_termattr` (*c_bool_t* ***flag***) [Function]

This function enables or disables the use of ANSI color and text style terminal attributes for log messages. This feature is enabled by default, and causes all log messages (when written to a tty) to be printed in a bold font, and to be color coded according to severity.

`void C_log_info` (*const char *`fmt`, ...*) [Function]
`void C_log_warning` (*const char *`fmt`, ...*) [Function]
`void C_log_error` (*const char *`fmt`, ...*) [Function]

These `printf()`-style functions write informational, warning, and error messages, respectively, to the console and/or the logging stream.

Each message will be prefixed by the current date and time.

Messages written to the console (the standard error stream) will be written in a bold font and color-coded to reflect the severity of the error, assuming that the stream is a tty and that terminal attributes are enabled.

3.11 Memory Management Functions

The following functions and macros simplify memory management in C. These routines provide more convenient interfaces to the standard `malloc()` family of library functions.

`void * C_mem_manage` (*void *`p`, size_t `elemsz`, c_bool_t `clearf`*) [Function]

This is a general-purpose memory management function. It is a higher-level interface to the `realloc()` library function. The argument *p* is a pointer to the memory to be managed in the case of a reallocation request, or `NULL` for an initial allocation request. The new or initial size of the memory, in bytes, is specified by *elemsz*. If *clearf* is `TRUE`, the newly allocated memory is zeroed; if this is a reallocation request that increases the size of a memory segment, the existing data is preserved and only the additional memory is zeroed.

If the memory allocation fails, the allocation error handler (if one has been installed) will be called. If the error handler returns `TRUE`, another attempt will be made to allocate the requested memory; otherwise, it will be assumed that the allocation request cannot be satisfied.

The function returns a pointer to the newly allocated memory on success, or `NULL` on failure.

Some of the memory management macros described below evaluate to calls to this function.

`void C_mem_set_errorfunc` (*c_bool_t (*`func`)(void)*) [Function]

This function allows the user to specify a memory allocation request failure handler. *func* will be called each time `C_mem_manage()` fails to allocate memory. This function may return `TRUE` to indicate that another attempt should be made to allocate the memory, or `FALSE` to indicate that no additional memory can be made available.

If *func* is `NULL`, any currently-installed handler is removed. See `C_mem_manage()` for a description of the conditions under which *func* will be called.

`void C_mem_set_alloc_hook` (*void (*`func`)(const void *`p_old`, const void *`p_new`, size_t `len`)*) [Function]

This function sets the memory allocation hook function to *func*. The allocation hook will be called each time memory is allocated, reallocated, or freed using the memory functions described in this section, or by any of the functions in this library which perform dynamic memory allocation. The allocation hook function may be uninstalled by passing `NULL` for *func*. By default, there is no allocation hook installed.

The meaning of the arguments passed to the allocation hook functions depends on the type of memory operation that was performed, as described below. The hook is always called *after* the memory operation has been performed.

- When a memory segment is being initially allocated, *p_old* will be `NULL`, *p_new* will be a pointer to the new segment, and *len* will be the length of the new segment.
- When an existing memory segment is resized (reallocated), *p_old* will be the pointer to the original segment, *p_new* will be a pointer to the resized segment, and *len* will be the new length of the segment.
- When a memory segment is being freed, *p_old* will be a pointer to the segment that was freed, *p_new* will be `NULL`, and *len* will be 0.

Never dereference the pointer *p_old*, as it no longer points to allocated memory.

`void C_mem_default_alloc_hook` (*const void* ***`p_old`**, *const void* ***`p_new`**, *size_t* **`len`**) [Function]

This function is a default memory allocation hook function which writes an informational log message for each memory allocation using `C_log_info()`.

`void * C_mem_free` (*void* ***`p`**) [Function]

This function is simply an interface to the `free()` library function. If *p* is not `NULL`, it is passed to `free()`. The function always returns `NULL`.

`void C_mem_freevec` (*char* ****`v`**) [Function]

This function frees a `NULL`-terminated character string vector *v*, by first dereferencing and freeing each character array in turn, and then freeing the character pointer array itself.

`uint_t C_mem_va_free` (*uint_t* **`n`**, *...*) [Function]

This function is a variable-argument interface to the `free()` library function. The argument *n* specifies how many pointers (which must be of type *void **) are being passed. Each subsequent non-`NULL` argument is passed to `free()`.

The function returns the number of non-`NULL` arguments that were processed.

`size_t C_mem_defrag` (*void* ***`p`** *size_t* **`elemsz`**, *size_t* **`len`**, *c_bool_t* (***`isempty`**)(*void *elem*)) [Function]

This is a general-purpose memory defragmentation routine. It interprets the memory at *p* as an array of *len* elements, each *elemsz* bytes in size. *isempty* is a pointer to a function which, when passed a pointer to one of the elements in the array, determines if it is “used” or “empty” and returns `FALSE` or `TRUE`, respectively.

The function iterates through the elements in the array, testing each element using *isempty*(), and moving contiguous blocks of “used” elements toward the beginning of the array until all “free” space has been coalesced at the end. The order of the elements within the array is preserved.

The function returns the number of “used” elements in the array. This return value can be used in a subsequent call to one of the memory allocation routines to reduce the size of the defragmented array to allow the unused space to be reclaimed by the system.

`C_malloc` (*n*, *type*) [Macro]
`C_calloc` (*n*, *type*) [Macro]

These macros are convenient replacements for the `malloc()` and `calloc()` library functions. They take as arguments the number of elements to allocate *n*, and the element's *type* (such as *int* or *struct foobar*). They return a properly cast pointer to the memory. Hence, the call:

```
C_malloc(5, char *)
```

would have a return value of type *char ***, that is, a pointer to 5 contiguous character pointers.

`C_calloc()` is similar, except that it additionally zeroes the newly allocated memory. Both macros evaluate to expressions involving calls to `C_mem_manage()`.

`C_realloc` (*p*, *n*, *type*) [Macro]

This macro is a convenient replacement for the `realloc()` library function. It takes as arguments a pointer *p* (of any type), the number of elements to allocate *n* (which is presumed to be of type *size_t*), and the element's *type* (such as *int* or *struct foobar*). It reallocates the memory beginning at *p* to the new size, returning a properly cast pointer to the resized memory. The macro evaluates to an expression involving a call to `C_mem_manage()`.

`C_zero` (*p*, *type*) [Macro]
`C_zeroa` (*p*, *n*, *type*) [Macro]

These macros zero the specified memory. `C_zero()` zeroes the element of type *type* at location *p*. `C_zeroa()` zeroes the *n*-element array of type *type* at location *p*.

`C_free` (*p*) [Macro]
`C_free_vec` (*v*) [Macro]
`C_va_free` (*n*, ...) [Macro]

These are additional convenience macros for freeing memory. `C_free()` frees the memory at *p*, which can be a pointer of any type. The macro is for use in place of calls to the `free()` library function. It evaluates to an expression involving a call to `C_mem_free()`.

`C_free_vec()` and `C_va_free()` are identical to `C_mem_free_vec()` and `C_mem_va_free()`, respectively.

`C_new` (*type*) [Macro]
`C_newa` (*n*, *type*) [Macro]
`C_newb` (*n*) [Macro]
`C_newstr` (*n*) [Macro]

These are additional convenience macros for allocating memory. `C_new()` allocates space for one element of the specified type. `C_newa()` allocates space for an array of *n* elements of the specified type. (These are reminiscent of `new` and `new[]` in C++.) `C_newb()` allocates space for an *n*-byte segment. `C_newstr()` allocates space for a character array of length *n* plus a trailing NUL. The variable *n* is assumed to be of type *size_t*.

These macros all expand to expressions involving the `C_calloc()` macro, hence the returned memory will always be zeroed. All of these macros return properly-cast pointers to the newly allocated memory, so an explicit cast is not necessary. For example:

```
char *s = C_newstr(45);
struct foobar *item = C_new(struct foobar);
int *scores = C_newa(20, int);
```

3.12 Memory Pool Functions

The following functions provide a simple memory pool mechanism. A *memory pool* is a block of memory allocated on the heap. The application may allocate smaller segments of memory from this pool. When these segments are no longer needed, they are all released at once by deallocating the pool.

The type *c_mempool_t* represents a memory pool.

`c_mempool_t * C_mempool_create` (*size_t* **`size`**) [Function]
`void C_mempool_destroy` (*c_mempool_t* ***`pool`**) [Function]

These functions create and destroy memory pools.

`C_mempool_create()` creates a new memory pool of the given *size*, returning a pointer to the newly created pool on success, or `NULL` on failure (for example, if *size* is less than 1, or if the memory allocation failed). The memory in the pool is always initially zeroed.

`C_mempool_destroy()` destroys the memory pool *pool*. All segments allocated from the pool should no longer be accessed, as they refer to memory that has been released.

`void * C_mempool_alloc` (*c_mempool_t* ***`pool`**, *size_t* **`size`**) [Function]

This function allocates a memory segment *size* bytes in length from the memory pool *pool*. It returns a pointer to the segment on success, or `NULL` on failure (for example, if there is not enough unused memory in the pool to satisfy the request). The returned pointer will always be word-aligned, and the size of the segment returned will be rounded up to the next multiple of the word size.

`size_t C_mempool_avail` (*c_mempool_t* ***`pool`**) [Function]

This function returns the number of bytes available to be allocated from the memory pool *pool*.

`C_palloc1` (*pool*, *type*) [Macro]
`C_palloc` (*pool*, *n*, *type*) [Macro]
`C_pallocstr` (*pool*, *n*) [Macro]

These are convenience macros for allocating memory from a memory *pool*. `C_palloc1()` allocates space for one element of the given *type*. `C_palloc()` allocates space for an array of *n* elements of the given *type*. `C_pallocstr()` allocates space for a string of length *n* - 1. The variable *n* is assumed to be of type *size_t*.

All of these macros return properly-cast pointers to the newly allocated memory, so an explicit cast is not necessary.

3.13 Memory Mapped Files

The following functions provide a high-level interface to memory mapped files. A *memory mapped file* is a disk file that has been mapped into the calling process's address space; the contents of the file may be manipulated by modifying memory directly. This mechanism is especially useful for very large files, as the operating system takes care of the details of paging portions of the file in and out of memory as needed, and of keeping the contents of the disk file in sync with the data that is in memory.

The type *c_memfile_t* represents a memory mapped file.

`c_memfile_t * C_memfile_open` (*const char* ***file***, *c_bool_t* ***readonly***) [Function]

This function opens the specified *file* and maps it into memory. If *readonly* is `TRUE`, the file will be mapped as read-only; otherwise both reading and writing will be allowed.

The function returns a pointer to the new *c_memfile_t* structure on success, or `NULL` on failure. The function `C_memfile_base()` may be used to obtain a pointer to the beginning of the file in memory.

`c_bool_t C_memfile_close` (*c_memfile_t* ***mf***) [Function]

This function unmaps the memory mapped file *mf* and closes the file. A sync operation is performed just before the file is unmapped to ensure that any pending updates are persisted to the disk file.

The function returns `TRUE` on success, or `FALSE` on failure.

`c_bool_t C_memfile_sync` (*c_memfile_t* ***mf***, *c_bool_t* ***async***) [Function]

The operating system periodically performs a sync operation to keep the contents of the disk file up to date with the memory-resident image of the file. This function forces such an update to take place immediately on the memory mapped file *mf*. If *async* is `TRUE`, the update is performed asynchronously, and the function returns immediately. Otherwise, it is performed synchronously and the function returns when the update is complete.

The function returns `TRUE` on success, or `FALSE` on failure.

`c_bool_t C_memfile_resize` (*c_memfile_t* ***mf***, *off_t* ***length***) [Function]

This function resizes the memory mapped file *mf* to the new size *length*. The new length must be at least 0. If *length* is smaller than the current length of the file, the file will be truncated and the excess data will be lost. If *length* is larger than the current length of the file, the file will grow to the new size, and the extra bytes will be zeroed.

The function returns `TRUE` on success, or `FALSE` on failure.

`void * C_memfile_base` (*c_memfile_t* ***mf***) [Function]

This function returns a pointer to the beginning of the memory occupied by the memory mapped file *mf*. It is implemented as a macro.

`void * C_memfile_pointer` (*c_memfile_t* ***mf***, *off_t* ***offset***) [Function]

This function returns a pointer to the given offset *offset* in the memory mapped file *mf*. It is implemented as a macro.

`off_t C_memfile_length` (*c_memfile_t* ***mf***) [Function]

This function returns the current length of the memory mapped file *mf*. It is implemented as a macro.

3.14 System Information Functions

The following functions provide various information about the system and about users and groups.

`c_bool_t C_system_ingroup` (*const char* ***login***, *const char* ***group***) [Function]

This function determines if the user whose login name is *login* belongs to the group named *group*. It can be used for authentication purposes.

The function returns `TRUE` if the user is a member of the named group, or `FALSE` if not or upon failure (for example, if either argument is `NULL` or an empty string).

`c_sysinfo_t * C_system_getinfo` (*void*) [Function]

This function obtains various types of information about the system, the current process and user, the terminal line, and the system time. The function stores the information in a static data structure, to which it returns a pointer. The *c_sysinfo_t* structure has the following members:

`char *login`	The current user's login name.
`char *fullname`	The current user's full name.
`char *homedir`	The current user's home directory.
`char *shell`	The current user's login shell.
`uid_t uid`	The process's real user ID.
`gid_t gid`	The process's real group ID.
`uid_t euid`	The process's effective user ID.
`gid_t egid`	The process's effective group ID.
`char *hostname`	The hostname.
`char *osname`	The operating system name.
`char *osver`	The OS version.
`char *osrel`	The OS release.
`char *arch`	The system's architecture type.
`pid_t pid`	The process ID.
`pid_t ppid`	The parent process ID.
`time_t stime`	The system time at which this function was first called.
`char *term`	The terminal line for the current process.

The first time the function is called, it fills the static data structure with values retrieved from the system. Subsequent calls immediately return the pointer to this structure.

`uid_t C_system_get_uid` (*void*) [Function]
`gid_t C_system_get_gid` (*void*) [Function]
`pid_t C_system_get_pid` (*void*) [Function]
`char * C_system_get_login` (*void*) [Function]
`char * C_system_get_fullname` (*void*) [Function]
`char * C_system_get_homedir` (*void*) [Function]
`char * C_system_get_hostname` (*void*) [Function]

`char * C_system_get_term` (*void*) [Function]

These are convenience functions that retrieve some of the more interesting values from the static *c_sysinfo_t* structure.

`c_bool_t C_system_cdhome` (*void*) [Function]

This function attempts to set the current working directory to the current user's home directory. It obtains the home directory path via a call to `C_system_get_homedir()`.

The function returns `TRUE` on success, or `FALSE` on failure.

`c_bool_t C_system_passwd_generate` (*const char *`passwd`*, *char *`buf`*, *size_t* `bufsz`) [Function]

This function generates an encrypted password using the standard UNIX `crypt()` function. The string *passwd* is encrypted using a randomly generated salt, and up to *bufsz* - 1 bytes of the resulting ciphertext (the first two bytes of which are the salt) are written to the buffer *buf*, which is unconditionally `NUL`-terminated.

The function returns `TRUE` on success. On failure (for example, if *passwd* or *buf* is `NULL`, or if *bufsz* is 0) the function returns `FALSE`.

`c_bool_t C_system_passwd_validate` (*const char *`plaintext`*, *const char *`ciphertext`*) [Function]

This function validates a plaintext password against the encrypted form of the password using the standard UNIX `crypt()` routine. The string *plaintext* is encrypted using the first two bytes of *ciphertext* as the salt. If the resulting ciphertext matches *ciphertext*, the function returns `TRUE`. If the ciphertext strings do not match, or upon failure (for example, if either of the arguments is `NULL`) it returns `FALSE`.

4 Utility Functions

This chapter describes various utility functions. They are divided into several groups; the functions in a group share a common name prefix for that group; e.g., all string-related functions have names that begin with '`C_string_`'. All of the constants, macros, and functions described in this chapter are defined in the header '`cbase/util.h`'.

The following sections describe each group in detail.

4.1 Bitstring Functions

The following functions provide for the manipulation of *bit strings* of arbitrary length. Bit strings may be used to efficiently store groups of related boolean (on/off) flags; each 8-bit byte can represent 8 such flags.

The type *c_bitstring_t* represents a bit string.

`c_bitstring_t * C_bitstring_create` (*uint_t* ***nbits***) [Function]
`c_bool_t C_bitstring_destroy` (*c_bitstring_t* *****bs***) [Function]

These functions create and destroy bit strings. `C_bitstring_create` creates a new bit string *nbits* in length. The function returns a pointer to the newly created bit string on success. On failure, it returns `NULL` (for example, if *nbits* `<` 1).

`C_bitstring_destroy()` destroys the bit string *bs*, freeing all memory associated with the bit string. It returns `TRUE` on success, or `FALSE` on failure (for example, if *bs* is `NULL`.)

`c_bool_t C_bitstring_set` (*c_bitstring_t* *****bs***, *uint_t* ***bit***) [Function]
`c_bool_t C_bitstring_clear` (*c_bitstring_t* *****bs***, *uint_t* ***bit***) [Function]

These functions set and clear the bit at offset *bit* in the bit string *bs*. `C_bitstring_set()` sets the specified *bit*, and `C_bitstring_clear()` clears it.

The functions return `TRUE` on success, or `FALSE` on failure (for example, if *bs* is `NULL` or *bit* is out of range).

`c_bool_t C_bitstring_set_range` (*c_bitstring_t* *****bs***, *uint_t* ***sbit***, *uint_t* ***ebit***) [Function]
`c_bool_t C_bitstring_clear_range` (*c_bitstring_t* *****bs***, *uint_t* ***sbit***, *uint_t* ***ebit***) [Function]

These functions set or clear a range of bits in the bit string *bs*. `C_bitstring_set_range()` sets all of the bits from offsets *sbit* to *ebit*, inclusive. `C_bitstring_clear_range()` clears all of the bits from offsets *sbit* to *ebit*, inclusive.

The functions return `TRUE` on success, or `FALSE` on failure (for example, if *bs* is `NULL`, if *sbit* or *ebit* is out of range, or if *ebit* `<` *sbit*).

`c_bool_t C_bitstring_set_all` (*c_bitstring_t* *****bs***) [Function]
`c_bool_t C_bitstring_clear_all` (*c_bitstring_t* *****bs***) [Function]

These functions set or clear all of the bits in the bit string *bs*. `C_bitstring_set_all()` sets all bits in the bit string, and `C_bitstring_clear_all()` clears all bits.

The functions return `TRUE` on success, or `FALSE` on failure (for example, if *bs* is `NULL`).

`c_bool_t C_bitstring_isset` (*c_bitstring_t* ***bs**, *uint_t* **bit**) [Function]
`c_bool_t C_bitstring_isclear` (*c_bitstring_t* ***bs**, *uint_t* **bit**) [Function]

These functions test bits in the bit string *bs*. `C_bitstring_isset()` determines if the bit at offset *bit* is set, returning `TRUE` if so and `FALSE` otherwise. `C_bitstring_isclear()` determines if the bit at offset *bit* is cleared, returning `TRUE` if so and `FALSE` otherwise.

The functions return `FALSE` upon failure (for example, if *bs* is `NULL` or if *bit* is out of range).

`c_bool_t C_bitstring_compare` (*c_bitstring_t* ***bs1**, *c_bitstring_t* ***bs2**) [Function]

This function compares the bit string *bs1* to the bit string *bs2*. It returns `TRUE` if all of the bits that are set in *bs1* are also set in *bs2*, and `FALSE` otherwise.

The function returns `FALSE` on failure (for example, if *bs1* or *bs2* is `NULL`).

`uint_t C_bitstring_size` (*c_bitstring_t* ***bs**) [Function]

This function, which is implemented as a macro, returns the size (in bits) of the bit string *bs*.

4.2 Data Buffer Functions

The following functions and macros are provided for the manipulation of dynamically allocated data buffers. The type *c_buffer_t* represents a data buffer. It contains the following members:

`char *buf`	A pointer to the buffer.
`size_t bufsz`	The size of the buffer.
`size_t datalen`	The amount of data in the buffer.
`void *hook`	A hook for associating arbitrary data.

The *c_buffer_t* type can be used as a convenient means of passing sized buffers. The `datalen` member can be used to indicate how much significant data is in a buffer. Arbitrary data can also be associated with a buffer via the `hook` pointer.

`c_buffer_t * C_buffer_create` (*size_t* **bufsz**) [Function]
`void C_buffer_destroy` (*c_buffer_t* ***buf**) [Function]

These functions create and destroy arbitrarily-sized data buffers. `C_buffer_create()` creates a new data buffer *bufsz* bytes in length; the buffer's memory is initially zeroed. The function returns a pointer to the newly allocated buffer.

`C_buffer_destroy()` destroys the buffer *buf*, freeing both the buffer memory and the *c_buffer_t* data structure itself.

`c_buffer_t * C_buffer_resize` (*c_buffer_t* ***buf**, *size_t* **newsz**) [Function]

This function resizes the buffer *buf* to a new size of *newsz*. bytes. The semantics are similar to that of `C_realloc()`: if the new size is larger than the previous size, the additional bytes are zeroed.

The function returns *buf* on success. On failure (for example, if *newsz* is 0), it returns `NULL`.

`void C_buffer_clear` (*c_buffer_t* `*buf`) [Function]

This function zeroes the buffer *buf*.

`C_buffer_data` (*buf*) [Macro]
`C_buffer_size` (*buf*) [Macro]
`C_buffer_datalen` (*buf*) [Macro]
`C_buffer_hook` (*buf*) [Macro]

These macros access the corresponding fields in the buffer structure *buf*.

4.3 Hexadecimal Encoding Functions

The following functions encode data to and decode data from a 7-bit ASCII hexadecimal representation. Each byte of data is encoded as a two-character long representation of the hexadecimal value of the byte; e.g., the value 0x3F will be encoded as the characters '`3F`'. Similarly, an ASCII string consisting of hexadecimal values encoded in such a representation can be converted to a corresponding array of bytes. Arbitrary data can be encoded or decoded using these routines.

`C_hex_isdigit` (*c*) [Macro]

This macro evaluates to `TRUE` if *c* is a valid hexadecimal character (0 - 9, A - F, or a - f), or `FALSE` otherwise.

`char C_hex_tonibble` (*int* `v`) [Function]
`int C_hex_fromnibble` (*char* `c`) [Function]

These routines convert nibbles (values in the range 0 to 15) to and from an ASCII hexadecimal representation.

`C_hex_tonibble()` converts the value *v* to a corresponding hexadecimal character: 0 - 9 or A - F. If the value of *v* is outside the acceptable range for a nibble, the function returns `NUL`.

`C_hex_fromnibble()` converts the hexadecimal character *c* to a corresponding integer value between 0 and 15. If *c* is not a hexadecimal digit character (as determined by the application of `C_hex_isdigit()`), the function returns `-1`.

`c_bool_t C_hex_tobyte` (*char* `*s`, *int* `v`) [Function]
`int C_hex_frombyte` (*char * const* `s`) [Function]

These routines convert bytes (values in the range 0 to 255) to and from an ASCII hexadecimal representation.

`C_hex_tobyte()` stores the two-character ASCII representation of the byte *v* at *s*. If the value of *v* is outside the acceptable range for a byte, the function returns `FALSE`; otherwise it returns `TRUE`.

`C_hex_frombyte()` returns the numeric value of the byte whose ASCII representation is stored at *s*. If the two characters at *s* are not hexadecimal characters, the function returns `-1`.

`c_bool_t C_hex_encode` (*char * const* `data`, *size_t* `len`, *char ** `s`) [Function]
`c_bool_t C_hex_decode` (*const char* `*s`, *size_t* `len`, *char* `*data`) [Function]

These routines convert arbitrary data to and from an ASCII hexadecimal representation.

`C_hex_encode()` encodes *len* bytes beginning at *data* into the corresponding hexadecimal representation. The encoding is stored beginning at *s*, which must point to a buffer of at least *len* * 2 bytes.

`C_hex_decode()` decodes *len* bytes of hexadecimal representation beginning at *s* The decoded data is stored beginning at *data*, which must point to a buffer of at least *len* / 2 bytes. Note that *len* must be a multiple of 2.

The functions return `TRUE` on success, or `FALSE` if passed invalid arguments or data.

4.4 Random Number Functions

The following functions are provided for the generation of random numbers.

`void C_random_seed` (*void*) [Function]

In the single-threaded version of the library, this function seeds the random number generator (via a call to `srand()`) with the sum of the current value of the system clock and the current process ID. In the multi-threaded version of the library, this function is a no-op since the sequence is seeded automatically, as described below.

`uint_t C_random` (*uint_t* **`n`**) [Function]

This function returns a random unsigned integer in the range [0, *n*-1] (via a call to `rand()`). In the multi-threaded version of the library, this function maintains a different random number sequence for each calling thread; the first call to this function by a thread seeds the random number sequence for that thread with the sum of the current value of the system clock and the thread ID of the calling thread.

4.5 String Manipulation and Parsing Functions

The following functions parse and manipulate character arrays (strings).

`char * C_string_clean` (*char* ***`s`**, *char* **`fillc`**) [Function]

This function replaces all non-printable characters in the string *s* with the fill character *fillc*.

The function returns *s* on success, or `NULL` on failure (for example, if *s* is `NULL` or *fillc* is `NUL`).

`char * C_string_tolower` (*char* ***`s`**) [Function]

This function converts all uppercase characters in the string *s* to lowercase. It returns *s*.

`char * C_string_toupper` (*char* ***`s`**) [Function]

This function converts all lowercase characters in the string *s* to uppercase. It returns *s*.

`c_bool_t C_string_isnumeric` (*const char* ***`s`**) [Function]

This function determines if the string *s* is composed strictly of numeric characters (0 - 9).

The function returns `TRUE` if *s* is numeric and `FALSE` otherwise.

`c_bool_t C_string_startswith` (*const char *`s`*, *const char *`prefix`*) [Function]

This function determines if the string *s* begins with the string *prefix*. It returns `TRUE` if *prefix* is a prefix of *s* and `FALSE` otherwise.

`c_bool_t C_string_endswith` (*const char *`s`*, *const char *`suffix`*) [Function]

This function determines if the string *s* ends with the string *suffix*. It returns `TRUE` if *suffix* is a suffix of *s* and `FALSE` otherwise.

`char * C_string_trim` (*char *`s`*) [Function]

This function trims whitespace from both ends of the string *s*. That is, all whitespace up to the first non-whitespace character in the string and all whitespace after the last non-whitespace character in the string is discarded. The remaining text is moved so that it is flush with the beginning of the string.

The function returns *s*.

`char ** C_string_split` (*char *`s`*, *const char *`sep`*, *size_t *`len`*) [Function]

This function is a higher-level interface to the `strtok_r()` library function. It splits the string *s* into a series of "words," breaking words on any combination of characters from the string *sep*, as does the `strtok_r()` function.

On success, the number of words parsed is stored at *len* if it is non-`NULL`, and the words are returned as a `NULL`-terminated string vector, which is dynamically allocated and must eventually be freed by the caller. On failure, `NULL` is returned.

`char * C_string_dup` (*const char *`s`*) [Function]

This function is identical to the `strdup()` library function. It duplicates the string *s* in memory and returns a pointer to the new copy. It is provided because `strdup()` is not specified by the POSIX.1 standard, and thus may not be available on all systems. This function should be used in place of `strdup()` as it allocates memory using `C_newstr()` rather than by calling `malloc()` directly.

`char * C_string_dup1` (*const char *`s`*, *char `c`*) [Function]

This function duplicates the string *s* in memory and appends the character *c* to the end of the duplicated string. It returns a pointer to the new string on success, or `NULL` on failure.

`char * C_string_chop` (*char *`s`*, *const char *`termin`*) [Function]

This function searches the string *s* for the first occurrence of any character in *termin* and replaces it with `NUL`. If no terminator character is found, the string is left unmodified. The function may be used to "chop" unneeded text from the end of a string (such as a CR+LF pair at the end of a line read from a socket or DOS file).

The function returns *s*.

`char * C_string_rchop` (*char *`s`*, *const char *`termin`*) [Function]

This function searches the string *s* for the last occurrence of any character in *termin* and replaces it with `NUL`. If no terminator character is found, the string is left unmodified. The function may be used to "chop" unneeded text from the end of a string (such as a CR+LF pair at the end of a line read from a socket or DOS file).

The function returns *s*.

`const char * C_string_tokenize` (*const char* ***s***, *const char* ***delim***, *const char* ****ctx***, *size_t* ***len***) [Function]

This function is a non-destructive string tokenizer; it differs from `strtok_r()` in that the source string is not modified. The function searches for the next token in *s* that does not consist of any characters in *delim*. The argument *ctx* is used to store the tokenizer context. If *s* is `NULL`, tokenization resumes from the last character analyzed; otherwise, it starts at the beginning of *s*.

The function stores the length of the token at *len* and returns a pointer to the beginning of the token. If no token is found, or on error (for example, if *delim*, *ctx*, or *len* is `NULL`) the function returns `NULL`.

The following code snippet extracts words from a sentence:

```
const char *text = "How now, brown cow?";
const char *ctx, *word;
uint_t len;

for(word = C_string_tokenize(text, " .,?!", &ctx, &len);
    word;
    word = C_string_tokenize(NULL, " .,?!", &ctx, &len))
{
  printf("Word found: %.*s\n", len, word);
}
```

`c_bool_t C_string_copy` (*char* ***buf***, *size_t* **bufsz**, *const char* ***s***) [Function]

This function performs a safe string copy. At most *bufsz* - 1 characters from *s* are copied to *buf*; the string *buf* is unconditionally `NUL`-terminated.

The function returns `TRUE` if the entire string *s* was copied to *buf*; otherwise it returns `FALSE`.

`c_bool_t C_string_va_copy` (*char* ***buf***, *size_t* **bufsz**, *... /* , NULL */*) [Function]

This is a variable argument list version of the `C_string_copy()` function. It copies the first string into *buf*, and then concatenates all of the remaining strings (if any) onto the text already in *buf*. At most *bufsz* - 1 total bytes are copied to *buf*; the string *buf* is unconditionally `NUL`-terminated.

The function returns `TRUE` if all of the strings were completely copied to *buf*; otherwise it returns `FALSE`.

`c_bool_t C_string_concat` (*char* ***buf***, *size_t* **bufsz**, *const char* ***s***) [Function]

This function performs a safe string concatenation. It concatenates as much of the string *s* onto the string *buf* as is possible without overflowing *buf*. The resulting string will be at most *bufsz* - 1 characters in length, and will be unconditionally `NUL`-terminated.

The function returns `TRUE` if the entire string *s* was concatenated onto *buf*; otherwise it returns `FALSE`.

`c_bool_t C_string_va_concat` (*char* ***buf**, *size_t* **bufsz**, *... /* , NULL */*) [Function]

This is a variable argument list version of the `C_string_concat()` function. It concatenates each string onto the text that is already in *buf*, without overflowing *buf*. The resulting string will be at most *bufsz* - 1 characters in length, and will be unconditionally `NUL`-terminated.

The function returns `TRUE` if all of the strings were completely copied to *buf*; otherwise it returns `FALSE`.

`char ** C_string_sortvec` (*char* ****v**, *size_t* **len**) [Function]

This function quicksorts the string vector *v*. It performs a call to the `qsort()` library function to accomplish the alphabetical sorting. The argument *len* specifies the number of elements in *v*.

The function returns *v* on success, or `NULL` on failure.

`char * C_string_va_make` (*const char* ***first**, *... /* , NULL */*) [Function]

This function constructs a new string which is a concatenation of all of the strings in the NULL-terminated list of *char ** arguments. The new string is dynamically allocated and must eventually be freed by the caller.

If *first* is `NULL`, the function returns `NULL`.

`char ** C_string_va_makevec` (*size_t* ***len**, *... /* , NULL */*) [Function]

This function accepts a pointer to an integer, *len*, and a `NULL`-terminated list of *char ** arguments. It creates a string vector from its arguments, storing the number of strings in the vector at *len* if it is non-`NULL`.

The function returns the newly created vector on success, or `NULL` on failure.

`char ** C_string_valist2vec` (*const char* ***first**, *va_list* **vp**, *size_t* ***slen**) [Function]

This function converts a variable-argument list pointer *vp* into a character string vector. The size of the vector is stored at *slen* if it is non-`NULL`. The function returns the newly created vector on success, or `NULL` on failure.

`uint_t C_string_hash` (*const char* ***s**, *uint_t* **modulo**) [Function]

This function hashes the string *s* to an unsigned integer in the range [0, *modulo* - 1]. The particular algorithm used is one that combines bitwise operators and addition on the characters in the string. It was devised by Joe I. Williams. The function returns the hash value on success, or `0` on failure (for example, if *s* is `NULL` or empty, or if *modulo* is 0).

`int C_string_compare` (*const void* ***s1**, *const void* ***s2**) [Function]

This function is a wrapper for the `strcmp()` library function. It typecasts *s1* and *s2* to *char ** and passes them to `strcmp()`, returning that function's return value.

This function is passed to the `qsort()` library function by routines which use that function to sort strings. It is provided here for that purpose.

`int C_string_compare_len` (*const char* ***s1**, *size_t* **len1**, *const char* ***s2**, *size_t* **len2**) [Function]

This function compares two substrings, *s1* and *s2*, returning a negative value if *s1* is less than *s2*, a positive value if *s1* is greater than *s2*, and 0 if the two substrings are equal. At most *len1* and *len2* characters of *s1* and *s2*, respectively, are compared.

4.6 String Buffer Functions

The following functions and macros are provided for the manipulation of dynamically allocated string buffers. A *string buffer* is much like a data buffer, except that it also includes a write pointer; this write pointer is advanced each time text is written to the buffer.

The string stored in a string buffer is always `NUL`-terminated. Each of the functions below writes as much of the requested text as possible to the string buffer, but will never attempt to write past the end of the buffer.

The type *c_strbuffer_t* represents a string buffer.

`c_strbuffer_t * C_strbuffer_create` (*size_t* **bufsz**) [Function]
`c_bool_t C_strbuffer_destroy` (*c_strbuffer_t* ***sb**) [Function]

These functions create and destroy string buffers, respectively. `C_strbuffer_create()` creates a new, empty string buffer that is *bufsz* bytes in size. It returns a pointer to the newly created string buffer on success, or `NULL` on failure (for example, if *bufsz* is less than 1).

`C_strbuffer_destroy()` destroys the string buffer *sb*, deallocating all memory associated with the buffer. It returns `TRUE` on success, or `FALSE` on failure (for example, if *sb* is `NULL`).

`c_bool_t C_strbuffer_clear` (*c_strbuffer_t* ***sb**) [Function]

This function clears the string buffer *sb*. The buffer is reset so that it contains an empty string. The function returns `TRUE` on success and `FALSE` on failure (for example, if *sb* is `NULL`).

`c_bool_t C_strbuffer_strcpy` (*c_strbuffer_t* ***sb**, *const char* ***s**) [Function]

This function copies the string *s* into the string buffer *sb*, replacing any text currently in the buffer. The function returns `TRUE` on success. If the buffer could not accommodate the entire string, or upon failure (if *sb* or *s* is `NULL`) the function returns `FALSE`.

`c_bool_t C_strbuffer_strcat` (*c_strbuffer_t* ***sb**, *const char* ***s**) [Function]

This function concatenates the string *s* onto the text in the string buffer *sb*. The function returns `TRUE` on success. If the buffer could not accommodate the entire string, or upon failure (if *sb* or *s* is `NULL`) the function returns `FALSE`.

`c_bool_t C_strbuffer_sprintf` (*c_strbuffer_t* ***sb**, *const char* ***format**, ...) [Function]

This function appends a formatted string onto the text in the string buffer *sb*; the behavior is identical to that of the `sprintf()` library function. The function returns `TRUE` if the entire formatted string was successfully written. If the buffer could not accommodate the entire string, or upon failure (for example, if *sb* or *format* is `NULL`) the function returns `FALSE`.

`c_bool_t C_strbuffer_putc` (*c_strbuffer_t* ***sb***, *char* **c**) [Function]

This function appends the single character *c* onto the text in the string buffer *sb*. The function returns `TRUE` if the character was successfully written, or `FALSE` if the buffer is already full or upon failure (for example, if *sb* is `NULL` or *c* is `NUL`).

`size_t C_strbuffer_size` (*c_strbuffer_t* ***sb***) [Function]

This function (which is implemented as a macro) returns the size of the string buffer *sb*.

`size_t C_strbuffer_strlen` (*c_strbuffer_t* ***sb***) [Function]

This function returns the length of the string in the string buffer *sb*.

`const char * C_strbuffer_string` (*c_strbuffer_t* ***sb***) [Function]

This function (which is implemented as a macro) returns a pointer to the beginning of the string in the string buffer *sb*. This string is guaranteed to be `NUL`-terminated.

4.7 Time Functions

The following functions parse and format time values.

`c_bool_t C_time_format` (*time_t* **t**, *char* ***buf***, *size_t* **bufsz**, *const char* ***format***) [Function]

This function is a higher-level interface to the `strftime()` library function. It formats the given time *t* (or the current time, if *t* is 0) according to the format string *format*. Up to *bufsz* bytes of the formatted text are written to the string *buf*.

The function returns `TRUE` on success or `FALSE` on failure.

`time_t C_time_parse` (*const char* ***s***, *const char* ***format***) [Function]

This function is a higher-level interface to the `strptime()` library function. It parses a string representation *s* of a date/time that is expected to be in the specified *format*. It returns the parsed time as a *time_t* value on success, or `(time_t)0` on failure.

4.8 CPU Timer Functions

These functions provide CPU benchmarking through the use of timers that can be started and stopped. The type *c_timer_t* represents a CPU timer.

`c_timer_t * C_timer_create` (*void*) [Function]

This function creates a new timer. It returns a pointer to the new timer.

`void C_timer_destroy` (*c_timer_t* ***timer***) [Function]

This function destroys the specified *timer*.

`void C_timer_start` (*c_timer_t* ***timer***) [Function]

This function resets and starts the specified *timer*. To start a timer without resetting it, use the `C_timer_resume()` function. Calling this function on a timer that is already running has no effect.

`void C_timer_stop` (*c_timer_t* ***timer***) [Function]

This function stops the specified *timer*. A stopped timer can be resumed with the `C_timer_resume()` function. Calling this function on a timer that is not running has no effect.

`void C_timer_resume` (*c_timer_t* ***timer***) [Function]

This function resumes the specified *timer*. Calling this function on a timer that is already running has no effect.

`void C_timer_reset` (*c_timer_t* ***timer***) [Function]

This function resets the specified *timer*.

`float C_timer_user` (*c_timer_t* ***timer***) [Function]

This function (which is implemented as a macro) returns the amount of user time (CPU time spent inside user space by the current thread or process) accumulated by the specified *timer* in hundredths of seconds. It is not meaningful to call this function with a timer that is not stopped.

`float C_timer_system` (*c_timer_t* ***timer***) [Function]

This function (which is implemented as a macro) returns the amount of system time (CPU time spent inside kernel space by the current thread or process) accumulated by the specified *timer* in hundredths of seconds. It is not meaningful to call this function with a timer that is not stopped.

`long C_timer_elapsed` (*c_timer_t* ***timer***) [Function]

This function (which is implemented as a macro) returns the amount of elapsed (real) time accumulated by the specified *timer*, in milliseconds. It is not meaningful to call this function with a timer that is not stopped.

`time_t C_timer_created` (*c_timer_t* ***timer***) [Function]

This function returns the system time at which the specified *timer* was created. It is implemented as a macro.

`c_bool_t C_timer_isrunning` (*c_timer_t* ***timer***) [Function]

This function returns `TRUE` if *timer* is currently running, and `FALSE` otherwise. It is implemented as a macro.

4.9 String Vector Functions

The following functions operate on string vectors. The *c_vector_t* type is an encapsulation type for `NULL`-terminated character string vectors (arrays of pointers to character arrays). Vectors are commonly used in the UNIX system to pass lists of strings; the `argv` vector passed into a program's `main()` function is probably the most well-known example. Some system calls, such as the `execv()` family of functions, also accept vector arguments.

`c_vector_t * C_vector_start` (*uint_t* ***resize_rate***) [Function]

This function creates a new vector. The argument *resize_rate* is a value that specifies the rate at which memory for the vector backbone will be allocated. A value of 40, for example, specifies that memory is allocated for 40 *char ** pointers at a time. The smaller the value, the more frequently memory will be reallocated (thus decreasing performance), but the more efficient memory use will be.

On success, the function returns a pointer to the new *c_vector_t* structure, which can be used as a "handle" in subsequent calls to the vector manipulation functions. On failure (for example, if *resize_rate* is 0), it returns `NULL`.

`c_bool_t C_vector_store` (*c_vector_t* `*v`, *const char* `*s`) [Function]

This function "stores" a new string *s* in the vector *v*. If the vector is full, it is automatically resized. Note that the string *s* is not copied into the vector; only the pointer is stored in the vector backbone. Therefore it is the responsibility of the caller to ensure that the memory that *s* occupies is not volatile.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *v* or *s* is `NULL`).

`c_bool_t C_vector_contains` (*c_vector_t* `*v`, *const char* `*s`) [Function]

This function searches for the string *s* in the vector *v*. It returns `TRUE` if the string is found and `FALSE` otherwise.

`char ** C_vector_end` (*c_vector_t* `*v`, *size_t* `*len`) [Function]

This function signifies that no more strings will be stored in the vector *v*. The vector is `NULL`-terminated, the length of the vector is stored at *len* if it is not `NULL`, and all memory associated with the vector other than the backbone itself, including the *c_vector_t* structure, is deallocated.

The function returns the vector as a pointer to an array of character pointers on success, or `NULL` upon failure (for example, if *v* is `NULL`).

`void C_vector_abort` (*c_vector_t* `*v`) [Function]

This function aborts construction of the vector *v*, freeing all memory associated with the vector, including the *c_vector_t* structure and the vector backbone.

`C_vector_free` (*v*) [Macro]

This is a convenience macro. It is identical to `C_free_vec()`.

5 Data Structure Functions

Since the ultimate purpose of every program is to manipulate data, data structures are an integral part of any piece of software. The following functions provide implementations of various data structures, including linked lists, hash tables, and others. All of the constants, macros, and functions described in this chapter are defined in the header '`cbase/data.h`'.

5.1 Basic Data Types

The types *c_link_t* and *c_tag_t* are used extensively by the linked-list and hashtable functions, but are documented for their possible use in other contexts.

5.1.1 Links

The type *c_link_t* is a structure that contains the following members:

`void *data`	Pointer to link data.
`c_link_t *prev`	Pointer to previous link.
`c_link_t *next`	Pointer to next link.

The type *c_link_t* can be used in any context that requires the chaining of data elements. New links can be created via calls to `C_new()`.

`C_link_data` (*link*) [Macro]
`C_link_prev` (*link*) [Macro]
`C_link_next` (*link*) [Macro]

These three macros access the corresponding fields in the link structure *link*. The first macro returns a *void ** pointer and the other two return pointers to *c_link_t* structures.

5.1.2 IDs

The type *c_id_t* represents a non-negative integer ID. It is defined as an *unsigned long long*, a 64-bit unsigned integer.

5.1.3 Data Elements

The type *c_datum_t* represents a data element with an ID. It is a structure that contains the following members:

`c_id_t key`	The key.
`void * data`	Pointer to data.

The type *c_datum_t* can be used in any context that requires data elements to be tagged with a numeric key. New data elements can be created via calls to `C_new()`.

`C_datum_key` (*`datum`*) [Macro]
`C_datum_value` (*`datum`*) [Macro]

These two macros access the corresponding fields in the datum structure *datum*. The first macro returns a *c_id_t* value and the second returns a *void ** pointer.

5.1.4 Tags

The type *c_tag_t* is a structure that contains the following members:

`char *key`	Pointer to key string.
`void *data`	Pointer to data.

This type can be used in any context that requires the labelling of a data element with a string. New tags can be created via calls to `C_new()`.

`C_tag_key` (*tag*) [Macro]
`C_tag_data` (*tag*) [Macro]

These two macros access the corresponding fields in the tag structure *tag*. The first macro returns a *void ** pointer and the second returns a *char ** pointer.

5.2 B-Trees

The following functions operate on b-trees. A *b-tree* is a special type of balanced tree in which each node has at most *n* data elements and *n* + 1 children, for some even value of *n*. The value of *n* is the *order* of the b-tree, and is specified at the time the b-tree is created. B-trees are balanced and sorted, to provide for very efficient lookup, even in the case of a b-tree that contains tens of thousands of elements.

The type *c_btree_t* represents a b-tree.

`c_btree_t * C_btree_create` (*uint_t* ***order***) [Function]
`void C_btree_destroy` (*c_btree_t* ****tree***) [Function]

These functions create and destroy b-trees. `C_btree_create()` creates a new, empty b-tree of order *order* (which must be at least 2) and returns a pointer to the new tree on success, or `NULL` on failure.

`C_btree_destroy()` frees all memory associated with the b-tree *tree*. If a destructor has been specified for the b-tree, all user data is destroyed as well using that destructor.

`c_bool_t C_btree_set_destructor` (*c_btree_t* ****tree***, *void* (****destructor***)(*void **)) [Function]

This function sets the destructor for the b-tree *tree*. The function *destructor* will be called for each element that is deleted from the b-tree as a result of a call to `C_btree_delete()` or `C_btree_destroy()`; a pointer to the data element being destroyed will be passed to the destructor. A value of `NULL` may be passed to remove a previously installed destructor.

There is no default destructor; if no destructor is set for the b-tree, user data will not be automatically freed.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *tree* is `NULL`).

`c_bool_t C_btree_store` (*c_btree_t* ****tree***, *c_id_t* ***key***, *const void* ****data***) [Function]
`void * C_btree_restore` (*c_btree_t* ****tree***, *c_id_t* ***key***) [Function]

These functions store data in and restore data from the b-tree *tree*.

`C_btree_store()` stores a new datum with the specified *key* and data value *data* in the b-tree. The function returns `TRUE` on success or `FALSE` on failure (for example, if *tree* or *data* is `NULL`, or if the b-tree already contains a datum with the specified *key*).

`C_btree_restore()` returns a pointer to the data value of the datum whose key is *key* in the b-tree. If an element with the specified *key* does not exist in the tree, or upon failure (for example, if *key* is `0` or *tree* is `NULL`), the function returns `NULL`.

`c_bool_t C_btree_delete` (*c_btree_t* ***tree***, *c_id_t* ***key***) [Function]

This function deletes the data element with the specified *key* from the b-tree *tree*. It returns `TRUE` on success or `FALSE` on failure (for example, if *tree* is `NULL`, if *key* is `0`, or if an element with the specified *key* does not exist in the b-tree).

`c_bool_t C_btree_iterate` (*c_btree_t* ***tree***, *c_bool_t* (***consumer***)(*void *elem, void *hook*), *void* ***hook***) [Function]

This function is a generic iterator for b-trees. It performs a breadth-first traversal of the b-tree *tree*. For each node in the tree, it calls the user-supplied function *consumer*(), passing to it a pointer to the element, and the pointer *hook* (which can be used to pass around state information). The *consumer*() function is expected to return `TRUE` as long as traversal should continue; if it returns `FALSE`, or when the entire tree has been traversed, this function exits, returning `FALSE` in the former case, or `TRUE` in the latter.

`uint_t C_btree_order` (*c_btree_t* ***tree***) [Function]

This function returns the order of the b-tree *tree*. It is implemented as a macro.

5.3 Linked Lists

The following functions operate on linked lists, which are constructed from *c_link_t* structures. A linked list is a chain of elements in which each element has a link to the element before it and the element after it. These functions do not store actual data; they merely organize pointers to data into a linked list structure. It is up to the caller to allocate and manage memory for the data.

A *link pointer* is associated with each linked list. The link pointer is like a bookmark in the list—it points to one of the links in the list, and can be moved around within the list. The link pointer simplifies the task of *list traversal*. This link pointer is stored inside the linked list data structure itself.

In some cases, data corruption may result if two threads simultaneously call functions that adjust the link pointer in a list, or if a function that is traversing a list calls another function that, as a side effect, moves the link pointer of that list. In these cases, use the reentrant forms of the functions below, which end in an '`_r`' suffix. Rather than using the link pointer within the list itself, these functions use a link pointer supplied by the caller.

Linked lists are useful in a very wide variety of contexts—too many, indeed, to list here.

The type *c_linklist_t* represents a linked list.

`c_linklist_t * C_linklist_create` (*void*) [Function]
`void C_linklist_destroy` (*c_linklist_t* *`l`) [Function]

These functions create and destroy linked lists. `C_linklist_create()` allocates memory for a new, empty linked list and returns a pointer to the new list on success, or `NULL` on failure.

`C_linklist_destroy()` frees all memory associated with the linked list *l*. If a destructor has been specified for the list, all user data is destroyed as well using that destructor.

`c_bool_t C_linklist_set_destructor` (*c_linklist_t* *`l`, *void* (*`destructor`)(*void* *)) [Function]

This function sets the destructor for the linked list *l*. The function *destructor* will be called for each element that is deleted from the linked list as a result of a call to `C_linklist_delete()`, `C_linklist_delete_r()`, or `C_linklist_destroy()`; a pointer to the data element being destroyed will be passed to the destructor. A value of `NULL` may be passed to remove a previously installed destructor.

There is no default destructor; if no destructor is set for the linked list, user data will not be automatically freed.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *l* is `NULL`).

`c_bool_t C_linklist_store` (*c_linklist_t* *`l`, *const void* *`data`) [Function]
`void * C_linklist_restore` (*c_linklist_t* *`l`) [Function]

These functions store data in and restore data from the linked list *l*.

`C_linklist_store()` creates a new link which points to *data* and stores it in the linked list *l* at the current position of the list's link pointer (see `C_linklist_move()`). Note that the data at *data* is not duplicated; only the pointer is copied into the linked list. The new link is always inserted before the link that the link pointer points to. If the link pointer is set to `NULL`, then the new link is inserted after the tail of the list, and hence becomes the new tail. The function returns `TRUE` on success or `FALSE` on failure (for example, if *l* or *data* is `NULL`).

`C_linklist_restore()` returns a pointer to the data element of the link at the current position of *l*'s link pointer. On failure (for example, if *l* is `NULL`, or if the link pointer is pointing past the end of the list) the function returns `NULL`.

`c_bool_t C_linklist_prepend` (*c_linklist_t* *`l`, *const void* *`data`) [Function]
`c_bool_t C_linklist_append` (*c_linklist_t* *`l`, *const void* *`data`) [Function]

These functions work similarly to `C_linklist_store()`, described above, except that they store the new element at the beginning or the end of the list, respectively. The list's link pointer is not modified as a result of either of these calls.

`c_bool_t C_linklist_store_r` (*c_linklist_t* *`l`, *const void* *`data`, *c_link_t* **`p`) [Function]
`void * C_linklist_restore_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]

These are reentrant versions of the functions `C_linklist_store()` and `C_linklist_restore()` described above. They both accept an additional argument, *p*, which is the address of the link pointer to be used.

`c_bool_t C_linklist_delete` (*c_linklist_t* *`l`) [Function]

This function deletes the link at the linked list *l*'s link pointer. It returns `TRUE` on success or `FALSE` on failure (for example, if *l* is `NULL` or if the link pointer points past the end of the list). The link pointer is adjusted to point to the link following the link that was deleted (or to `NULL` if the tail was deleted).

`c_bool_t C_linklist_delete_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]

This is the reentrant version of the function `C_linklist_delete()` described above. It accepts an additional argument, *p*, which is the address of the link pointer to be used.

`c_bool_t C_linklist_search` (*c_linklist_t* *`l`, *const void* *`data`) [Function]

This function searches the linked list *l* from beginning to end for a link whose data member is *data*. (Comparison is done by comparing the data pointers only.) If the item is found, the link pointer is left pointing to the matched link and the function returns `TRUE`. Otherwise, it returns `FALSE`.

`c_bool_t C_linklist_search_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]

This is the reentrant version of the function `C_linklist_search()` described above. It accepts an additional argument, *p*, which is the address of the link pointer to be used.

`c_bool_t C_linklist_move` (*c_linklist_t* *`l`, *int* `where`) [Function]
`c_bool_t C_linklist_move_head` (*c_linklist_t* *`l`) [Function]
`c_bool_t C_linklist_move_tail` (*c_linklist_t* *`l`) [Function]
`c_bool_t C_linklist_move_next` (*c_linklist_t* *`l`) [Function]
`c_bool_t C_linklist_move_prev` (*c_linklist_t* *`l`) [Function]
`c_bool_t C_linklist_move_end` (*c_linklist_t* *`l`) [Function]

These functions move the linked list *l*'s link pointer. The link pointer affects where data will be stored in the list and whence data will be restored from the list.

`C_linklist_move()` moves *l*'s link pointer to the location specified by *where*, which can have one of the following values: `C_LINKLIST_HEAD`, `C_LINKLIST_TAIL`, `C_LINKLIST_NEXT`, `C_LINKLIST_PREV`, `C_LINKLIST_END`. These values move the link pointer to the head, tail, next link, previous link, or end of the list, respectively. Calls to move to the head or to the tail always succeed, returning `TRUE` (unless *l* is `NULL`). Calls to move to the next or previous link return `TRUE` if a link exists in the specified direction, or `FALSE` if such movement would move the pointer off either end of the list (or if *l* is `NULL`).

The link pointer may become `NULL` as a result of this call; this signifies that it is pointing just past the end of the linked list, and therefore subsequent calls to `C_linklist_restore()` will return `NULL`. This is useful for traversing lists.

The five convenience functions `C_linklist_move_head()`, `C_linklist_move_tail()`, `C_linklist_move_next()`, `C_linklist_move_prev()`, and `C_linklist_move_end()` automatically pass the appropriate values for *where* to `C_linklist_move()`. They are implemented as macros.

The following example illustrates a `for` loop that iterates through a linked list of strings:

```
c_linklist_t *list;
void *data;

for(C_linklist_move_head(list);
    (data = C_linklist_restore(list)) != NULL;
    C_linklist_move_next(list))
  printf("Data: %s\n", (char *)data);
```

`c_bool_t C_linklist_move_r` (*c_linklist_t* *`l`, *int* `where`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_move_head_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_move_tail_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_move_next_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_move_prev_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_move_end_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]

These are the reentrant versions of the `C_linklist_move()` family of functions described above. They each accept an additional argument, *p*, which is the address of the link pointer to be used.

`c_bool_t C_linklist_ishead` (*c_linklist_t* *`l`) [Function]
`c_bool_t C_linklist_istail` (*c_linklist_t* *`l`) [Function]
`c_bool_t C_linklist_isend` (*c_linklist_t* *`l`) [Function]

These functions (which are implemented as macros) test the linked list *l*'s link pointer to determine if it is at the head, tail, or end of the list. They return `TRUE` if so and `FALSE` otherwise.

`c_bool_t C_linklist_ishead_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_istail_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]
`c_bool_t C_linklist_isend_r` (*c_linklist_t* *`l`, *c_link_t* **`p`) [Function]

These are the reentrant versions of the functions `C_linklist_ishead()`, `C_linklist_istail()`, and `C_linklist_isend()`, described above. They each accept an additional argument, *p*, which is the address of the link pointer to be used.

`c_link_t * C_linklist_head` (*c_linklist_t* *`l`) [Function]
`c_link_t * C_linklist_tail` (*c_linklist_t* *`l`) [Function]

These functions (which are implemented as macros) return the head and tail link, respectively, of the linked list *l*. A return value of `NULL` indicates that the list is empty.

`size_t C_linklist_length` (*c_linklist_t* *`l`) [Function]

This function (which is implemented as a macro) returns the length of the linked list *l*, that is, the number of links in the list.

5.4 Queues

A queue is a FIFO (first in, first out) data structure with a beginning and an end. Items are enqueued onto the end of the list and dequeued from the beginning. These functions do not store actual data; they merely organize pointers to data into a linked list structure. It is up to the caller to allocate and manage memory for the data.

Queues are useful in producer/consumer contexts, where data items must be processed in the order that they are received, and where new items may potentially arrive more quickly than they can be processed.

The type *c_queue_t* represents a queue.

`c_queue_t * C_queue_create` (*void*) [Function]
`void C_queue_destroy` (*c_queue_t *q*) [Function]

These functions create and destroy queues. `C_queue_create()` allocates memory for a new, empty queue and returns a pointer to the new queue on success, or `NULL` on failure.

`C_queue_destroy()` frees all memory associated with the queue *q*. If a destructor has been specified for the queue, all user data is destroyed as well using that destructor.

`c_bool_t C_queue_set_destructor` (*c_queue_t *q, void (*destructor)(void *)*) [Function]

This function sets the destructor for the queue *q*. The function *destructor* will be called for each element that is deleted from the queue as a result of a call to `C_queue_destroy()`; a pointer to the data element being destroyed will be passed to the destructor. A value of `NULL` may be passed to remove a previously installed destructor.

There is no default destructor; if no destructor is set for the queue, user data will not be automatically freed.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *q* is `NULL`).

`c_bool_t C_queue_enqueue` (*c_queue_t *q, const void *data*) [Function]
`void * C_queue_dequeue` (*c_queue_t *q*) [Function]

These functions enqueue data onto and dequeue data from the queue *q*. `C_queue_enqueue()` enqueues *data* as a new item at the end of the queue. It returns `TRUE` on success, or `FALSE` on failure (for example, if *q* or *data* is `NULL`).

`C_queue_dequeue()` dequeues an item from the beginning of the queue, returning a pointer to the dequeued data on success, or `NULL` on failure (for example, if *q* is `NULL` or empty).

`size_t C_queue_length` (*c_queue_t *q*) [Function]

This function (which is implemented as a macro) returns the length of the queue *q*, that is, the number of items in the queue.

5.5 Stacks

The following functions operate on stacks. A stack is a LIFO (last in, first out) data structure with a top and a bottom. Items are pushed onto or popped off the top of the stack. These functions do not store actual data; they merely organize pointers to data into a stack structure. It is up to the caller to allocate and manage memory for the data.

Stacks are useful in a variety of contexts, most typically in parsers and interpreters.

The type *c_stack_t* represents a stack.

`c_stack_t * C_stack_create` (*void*) [Function]
`void C_stack_destroy` (*c_stack_t* *`s`) [Function]

These functions create and destroy stacks. `C_stack_create()` allocates memory for a new, empty stack and returns a pointer to the new stack on success, or `NULL` on failure.

`C_stack_destroy()` frees all memory associated with the stack *s*. If a destructor has been specified for the stack, all user data is destroyed as well using that destructor.

`c_bool_t C_stack_set_destructor` (*c_stack_t* *`s`, *void* (*`destructor`)(*void* *)) [Function]

This function sets the destructor for the stack *s*. The function *destructor* will be called for each element that is deleted from the stack as a result of a call to `C_stack_destroy()`; a pointer to the data element being destroyed will be passed to the destructor. A value of `NULL` may be passed to remove a previously installed destructor.

There is no default destructor; if no destructor is set for the stack, user data will not be automatically freed.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *s* is `NULL`).

`c_bool_t C_stack_push` (*c_stack_t* *`s`, *const void* *`data`) [Function]
`void * C_stack_pop` (*c_stack_t* *`s`) [Function]

These functions push data onto and pop data off the stack *s*. `C_stack_push()` pushes *data* onto the top of the stack. It returns `TRUE` on success, or `FALSE` on failure (for example, if *s* or *data* is `NULL`).

`C_stack_pop()` pops the topmost item off the stack. It returns the popped data on success, or `NULL` on failure (for example, if *s* is `NULL` or empty).

`void * C_stack_peek` (*c_stack_t* *`s`) [Function]

This function returns the topmost item on the stack *s*, without removing the item from the stack as does `C_stack_pop()`. A return value of `NULL` indicates that the stack is empty.

`size_t C_stack_depth` (*c_stack_t* *`s`) [Function]

This function (which is implemented as a macro) returns the depth of the stack *s*, that is, the number of items on the stack.

5.6 Hashtables

The following functions operate on hashtables. A hashtable is implemented as an array of *n* linked lists of tags, where *n* is the number of buckets in the hashtable. These functions do not store actual data; they merely organize pointers to data into a hashtable structure. It is up to the user to allocate and manage memory for the data.

Hashtables are typically used as data dictionaries; a data item is tagged with a unique string and then stored in the hashtable. This unique string, or *key*, is subsequently used to retrieve that data item.

Items are stored in a hashtable as follows. First, a hashing algorithm is used to convert the item's key to an integer between 0 and *n* - 1, where *n* is the number of buckets in the hashtable. The item is then placed in the appropriate bucket. Each bucket is implemented as a linked list of items.

An item lookup consists of hashing the desired key to an integer, selecting the appropriate bucket, and then doing a linear search through the items in that bucket, comparing keys until the desired item is found.

Item lookup in a hashtable is faster than in a linked list or other similar data structure. The maximum number of comparisons required to find an item in a hashtable is equal to the number of items in the longest linked list in the hashtable. Increasing the number of buckets will decrease the number of collisions in the hashtable; that is, the likelihood that any two keys will hash to the same value (and fall into the same bucket) will be reduced. This will result in less items in each bucket and hence shorter linked lists to search through.

The type *c_hashtable_t* represents a hashtable.

`c_hashtable_t * C_hashtable_create` (*uint_t* **`buckets`**) [Function]
`void C_hashtable_destroy` (*c_hashtable_t* ***`h`**) [Function]

These functions create and destroy hashtables, respectively.

`C_hashtable_create()` creates a new, empty hashtable with the specified number of *buckets*. It returns a pointer to the new hashtable structure on success, or `NULL` on failure (for example, if *buckets* is 0). The larger the value for *buckets*, the more efficient a table lookup will be; values less than 10 are generally not useful.

`C_hashtable_destroy()` frees all memory associated with the hashtable *h*. If a destructor has been specified for the hash table, all user data is destroyed as well using that destructor.

`c_bool_t C_hashtable_set_destructor` (*c_hashtable_t* ***`h`**, *void* (***`destructor`**)(*void* *)) [Function]

This function sets the destructor for the hashtable *h*. The function *destructor* will be called for each element that is deleted from the hashtable as a result of a call to `C_hashtable_delete()` or `C_hashtable_destroy()`; a pointer to the data element being destroyed will be passed to the destructor. A value of `NULL` may be passed to remove a previously installed destructor.

There is no default destructor; if no destructor is set for the hashtable, user data will not be automatically freed.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *h* is `NULL`).

`c_bool_t C_hashtable_store` (*c_hashtable_t* ***`h`**, *const char* ***`key`**, *const void* ***`data`**) [Function]
`void * C_hashtable_restore` (*c_hashtable_t* ***`h`**, *const char* ***`key`**) [Function]

These functions store data in and restore data from the hashtable *h*.

`C_hashtable_store()` creates a new tag with the given *key* and *data*, hashes *key*, and uses the hash value as an index into the array of linked lists in the hashtable. It then inserts the tag at the head of the appropriate linked list. Note that this function does not actually copy data into the hashtable; it only stores the pointer *data* in the appropriate linked list in the hashtable. The function returns `TRUE` on success, or `FALSE` on failure (for example, if *h* or *key* is `NULL`).

`C_hashtable_restore()` searches the hashtable for a tag whose key matches *key*. It does this by hashing *key*, selecting the appropriate linked list in the hashtable, and then doing a linear search in that linked list to find the desired element. The function returns a pointer to the matching tag's data field if a match is found, or `NULL` if no match was found or if *h* or *key* is `NULL`.

`c_bool_t C_hashtable_delete` (*c_hashtable_t* *`h`, *const char* *`key`) [Function]

This function searches for a tag whose key matches *key* in the same manner as `C_hashtable_restore()`, and removes the matching tag from the table.

The function returns `TRUE` on success, or `FALSE` if no match was found or if *h* or *key* is `NULL`.

`char ** C_hashtable_keys` (*c_hashtable_t* *`h`, *size_t* *`len`) [Function]

This function returns all of the keys in the hashtable *h* as a string vector. If *len* is not `NULL`, the length of the vector is stored at *len*. If *h* is `NULL`, the function returns `NULL`. The returned vector is dynamically allocated and must eventually be freed by the caller.

`c_bool_t C_hashtable_set_hashfunc` (*uint_t* (*`func`)(*const char *s, uint_t modulo*)) [Function]

This function allows the user to specify an alternate hashing function *func*. The default hashing function is `C_string_hash()`. *func*() must accept a string and a modulo value and return an unsigned integer in the range [0, `modulo` - 1]. This function stores the pointer *func*() in a static area within the library for use in subsequent calls to the hashtable functions. The function returns `TRUE` on success, or `FALSE` if *func*() is `NULL`.

`size_t C_hashtable_size` (*c_hashtable_t* *`h`) [Function]

This function (which is implemented as a macro) returns the size of the hashtable *h*; that is, the number of elements stored in the table.

5.7 Dynamic Arrays

The following functions manipulate dynamic arrays. A dynamic array is like a regular array: it is a contiguous segment of memory that stores a series of equally-sized elements. Unlike with regular arrays, the user does not have control over where in the array an element will be stored. A new element is stored in the first free slot available, and the index of this slot is returned to the caller. This index is used to restore the element from the array. Memory management for dynamic arrays is done automatically; as an array begins to fill up, it is resized to accommodate more elements.

As elements are deleted from the array, the array does not shrink because to periodically defragment an array while maintaining the relative order of elements within it is inefficient. Instead, a deleted element is marked as an empty slot in the array, and is filled by a subsequent store operation. If an array becomes heavily fragmented, it may be defragmented via a call to `C_darray_defragment()`.

Dynamic arrays have the interesting property that they can be quickly written to and read from a file. Since no pointers are used, the data in the array is easily relocatable.

Dynamic arrays are intended for use in applications which generate a database that grows steadily in size over time—a database in which deletions are much less frequent than inserts.

The type *c_darray_t* represents a dynamic array.

`c_darray_t * C_darray_create (`*uint_t* **`resize_rate`**, *size_t* **`elemsz`**`)` [Function]
`void C_darray_destroy (`*c_darray_t* *****`a`**`)` [Function]

These functions create and destroy dynamic arrays.

`C_darray_create()` creates a new dynamic array for elements of size *elemsz* bytes. The argument *resize_rate* specifies the rate at which the array will be resized. Specifically, when space in the dynamic array is exhausted, it will be resized to make space for *resize_rate* * 8 more elements. Larger values for *resize_rate* will increase performance (since memory reallocation will be less frequent), but will decrease the efficiency with which memory is used. The function returns a pointer to the new dynamic array on success, or `NULL` on failure (for example, if *elemsz* is 0 or if *resize_rate* is less than 1 or greater than `C_DARRAY_MAX_RESIZE`.

`C_darray_destroy()` frees all memory associated with the dynamic array *a*. This includes memory occupied by data stored in the array.

`void * C_darray_store (`*c_darray_t* *****`a`**, *const void* *****`data`**, *uint_t* *****`index`**`)` [Function]
`void * C_darray_restore (`*c_darray_t* *****`a`**, *uint_t* **`index`**`)` [Function]

These functions store data in and restore data from the dynamic array *a*.

`C_darray_store()` copies the element pointed to by *data* (which is assumed to be of the correct size for this dynamic array) into the array *a*, resizing the array if necessary. The caller will have to typecast *data* to *void ** before passing it to this function. The index into the array at which the element was stored is stored at *index* if it is not `NULL`. The function returns a pointer to the beginning of the element within the array on success, or `NULL` on failure (for example, if *a* or *data* is `NULL`). The caller will have to typecast this return value back to the correct type before accessing its contents. If the caller does not need to modify the element immediately after it is stored, it may choose to ignore the return value and only remember the value at *index* for future retrievals.

`C_darray_restore()` locates the element at the index *index* within the array *a*, returning a pointer to the beginning of the element on success, or `NULL` on failure (for example, if the specified index points to an “empty” element, if *index* is out of range, or if *a* is `NULL`).

`c_bool_t C_darray_delete (`*c_darray_t* *****`a`**, *uint_t* **`index`**`)` [Function]

This function marks as “empty” the element whose index is *index* in the array *a*, effectively deleting it from the array. Subsequent calls to `C_darray_restore()` using this index will return `NULL` until the free slot is refilled.

`c_darray_t * C_darray_load (`*const char* *****`path`**`)` [Function]
`c_bool_t C_darray_save (`*c_darray_t* *****`a`**, *const char* *****`path`**`)` [Function]

These functions read dynamic arrays from and write dynamic arrays to the file specified by *path*.

`C_darray_load()` reads a dynamic array from a file, and returns a pointer to the loaded array on success, or `NULL` if the load failed (for example, if *path* does not exist or is not readable).

`C_darray_save()` writes the dynamic array *a* to a file, returning `TRUE` on success, or `FALSE` on failure (for example, if *a* is `NULL` or *path* could not be opened for writing). If the write fails, a partially-written file may exist as a result of this call.

The format of the file is binary; it consists of an image of the *c_darray_t* structure, followed by a free-list bitstream, followed by a contiguous block of elements. A dynamic array file should not be modified directly.

`c_darray_t * C_darray_defragment` (*c_darray_t* *`a`) [Function]

This function defragments the dynamic array *a* while preserving the relative order of the elements within the array. This is done by creating a new dynamic array, copying all non-deleted elements from *a* to the new array, and then destroying the old array. The function returns a pointer to the new array on success. On failure, `NULL` is returned (for example, if *a* is `NULL`). If there are no deleted elements in *a*, the function returns *a* unmodified. Note that defragmentation of the array results in a shifting of elements within it; therefore references to particular elements via *void ** pointers or index offsets may become invalidated.

`c_bool_t C_darray_iterate` (*c_darray_t* *`a`, *c_bool_t* (*`iter`)(*void *elem, uint_t index, void *hook*), *uint_t* `index`, *void* *`hook`) [Function]

This function is a generic iterator for dynamic arrays. For each non-deleted element in the array *a* beginning at the index *index*, it calls the user-supplied function *iter*(), passing to it a pointer to the beginning of the element, the `index` of the element within the dynamic array, and the pointer *hook* (which can be used to pass around state information). The *iter*() function is expected to return `TRUE` as long as traversal should continue; if it returns `FALSE`, or when the entire array has been traversed, this function exits, returning `FALSE` in the former case, or `TRUE` in the latter.

`size_t C_darray_size` (*c_darray_t* *`a`) [Function]

This function (which is implemented as a macro) returns the size of the dynamic array *a*, that is, the number of non-deleted elements in the array.

5.8 Dynamic Strings

Dynamic strings inherit many of the semantics of random-access files. A dynamic string is a segment of memory which can be written to and read from as if it were a normal ASCII file. The string is resized automatically when data is written past the end of the string or when the string is truncated to a specific length. A dynamic string has a seek pointer that can be moved back and forth, much like the seek pointer in a file. Like dynamic arrays, dynamic strings can be written to and read from disk.

The type *c_dstring_t* represents a dynamic string.

`c_dstring_t * C_dstring_create` (*uint_t* `blocksz`) [Function]
`char * C_dstring_destroy` (*c_dstring_t* *`d`) [Function]

These functions create and destroy dynamic strings.

`C_dstring_create()` creates a new dynamic string with the specified block size *blocksz*. The block size determines the rate at which memory will be reallocated as the string grows and shrinks. The larger the block size, the less frequently memory reallocation will take place, but the more inefficient memory use will be. The function returns a pointer to the new dynamic string on success, or `NULL` on failure (for example, if *blocksz* is less than `C_DSTRING_MIN_BLOCKSZ`). Specifically, memory will be allocated and deallocated in increments of *blocksz* bytes. Therefore, if frequent writes of long strings are anticipated, *blocksz* should be sufficiently large, and if infrequent writes of long strings or frequent writes of short strings are anticipated, *blocksz* should be smaller.

`C_dstring_destroy()` deallocates all memory associated with the dynamic string *d*, not including the string itself. This string is `NUL` terminated and the function returns a pointer to it. On failure, `NULL` is returned (for example, if *d* is `NULL`).

`c_bool_t` **`C_dstring_putc`** (*c_dstring_t* ***`d`**, *char* **`c`**) [Function]
`c_bool_t` **`C_dstring_puts`** (*c_dstring_t* ***`d`**, *const char* ***`s`**) [Function]
`c_bool_t` **`C_dstring_puts_len`** (*c_dstring_t* ***`d`**, *const char* ***`s`**, *size_t* **`len`**) [Function]

These functions write data to the dynamic string *d* at the current location of the string's seek pointer. `C_dstring_putc()` writes the single character *c* (which cannot be `NUL`), `C_dstring_puts()` writes the string *s*, and `C_dstring_puts_len()` writes exactly *len* characters from the string *s*. All three functions update the seek pointer to point immediately past the last character written. If the data to be written would go past the end of the string, the string is automatically resized to accommodate it.

The functions return `TRUE` on success, or `FALSE` on failure (for example, if *d* or *s* is `NULL`, if *len* is less than 1, or if *c* is `NUL`).

`char` **`C_dstring_getc`** (*c_dstring_t* ***`d`**) [Function]
`char *` **`C_dstring_gets`** (*c_dstring_t* ***`d`**, *char* ***`s`**, *size_t* **`len`**, *char* **`termin`**) [Function]

These functions read data from the dynamic string *d* starting at the current location of the seek pointer.

`C_dstring_getc()` reads a single character and moves the seek pointer forward one character. The function returns the character read on success, or `NUL` if the seek pointer is already at the end of the string or if *d* is `NULL`.

`C_dstring_gets()` reads characters into *s* until *len* - 1 characters have been read or the *termin* character is encountered, whichever occurs first. The buffer *s* is unconditionally `NUL` terminated; if the terminator character is encountered, it is discarded and replaced by `NUL` in *s*. The seek pointer is updated to point immediately past the last character read. The function returns *s* on success, or `NULL` on failure (for example, if the seek pointer is already at the end of the string, if *d* is `NULL`, or if *len* is 0).

`c_bool_t` **`C_dstring_seek`** (*c_dstring_t ****`d`**, *unsigned long* **`where`**, *int* **`whence`**) [Function]
`c_bool_t` **`C_dstring_ungetc`** (*c_dstring_t ****`d`**) [Function]
`c_bool_t` **`C_dstring_rewind`** (*c_dstring_t ****`d`**) [Function]
`c_bool_t` **`C_dstring_append`** (*c_dstring_t ****`d`**) [Function]

`C_dstring_seek()` moves the seek pointer for the dynamic string *d*. The argument *where* specifies the relative or absolute number of characters by which the pointer should be moved. The argument *whence* specifies how the *where* argument should be interpreted. A value of `C_DSTRING_SEEK_REL` signifies that *where* is a relative offset from the seek pointer's current position (and may be positive or negative). A value of `C_DSTRING_SEEK_ABS` specifies that *where* is an absolute offset from the beginning of the dynamic string (and must be positive). A value of `C_DSTRING_SEEK_END` specifies that *where* is an absolute offset from the end of the dynamic string toward the beginning (and must be positive as well). The function returns `TRUE` on success, or `FALSE` on failure (for example, if the new seek position would be out of range, or if *d* is `NULL`, or if the value of *whence* is invalid).

`C_dstring_ungetc()` moves the seek pointer back one character, `C_dstring_rewind()` moves it to the beginning of the dynamic string, and `C_dstring_append()` moves it to the end. These functions are implemented as macros, and their return values are the same as those for the `C_dstring_seek()` function.

`c_bool_t` **`C_dstring_trunc`** (*c_dstring_t ****`d`**, *off_t* **`length`**) [Function]

This function truncates the dynamic string *d* to a length of *size* characters. The new size must be less than or equal to the current size; a dynamic string cannot be lengthened using this function. The function also moves the seek pointer to the end of the dynamic string after the truncation is performed.

The function returns `TRUE` on success, or `FALSE` on failure (for example, if *d* is `NULL` or if *size* is greater than the current length of the dynamic string).

`c_dstring_t *` **`C_dstring_load`** (*const char ****`path`**, *uint_t* **`blocksz`**) [Function]
`c_bool_t` **`C_dstring_save`** (*c_dstring_t ****`d`**, *const char ****`path`**) [Function]

These functions read dynamic strings from and write dynamic strings to the file specified by *path*.

`C_dstring_load()` reads a dynamic string into memory. The meaning of *blocksz* is the same as in `C_dstring_create()`; it specifies the resize rate for the new string. On success, the function returns a pointer to the loaded dynamic string. On failure, it returns `NULL` (for example, if *blocksz* is greater than `C_DSTRING_MAX_BLOCKSZ`, if *d* is `NULL`, or if *path* does not exist or could not read).

`C_dstring_save()` writes the dynamic string *d* to a file. It returns `TRUE` on success, or `FALSE` on failure (for example, if *d* is `NULL` or if *path* could not be written). If the save fails, a partially written file may exist as a result of this call.

The format of a dynamic string file is plain ASCII. It is simply a file containing the string. It is safe to modify dynamic string files directly. This implies that plain ASCII files can be created with a text editor and then read in as dynamic strings.

`off_t C_dstring_length` (*c_dstring_t *d*) [Function]

This function returns the length (in characters) of the dynamic string *d*. It is implemented as a macro.

6 Real-Time Scheduler Functions

This chapter describes an implementation of a real-time scheduler whose functionality is very similiar to that of the UNIX *cron* daemon.

The scheduler allows events to be scheduled in real time. As with *cron*, events can be scheduled for specific dates and times. An event can be scheduled to fire only once or repetitively depending on a date and time specification.

Date and time specifications are made in the same manner as in *crontab* files. A specification consists of five fields separated by whitespace or colons (:). These fields specify integer patterns for matching minute, hour, day of month, month of year, and day of week, in that order. Each of the patterns may be either an asterisk (*), which denotes a wildcard that matches all acceptable values, or a list of elements separated by commas (,), where each element is either a single integer value or a range of values denoted by a pair of integers separated by a dash (-).

Values for minute must range from 0-59, for hour from 0-23, for day of month from 1-31, for month of year from 1-12, and for day of week from 0-6 with 0 denoting Sunday.

Following are some example specifications and their meanings:

```
0 0 5,15 * 1-5
```

Midnight on the 5th and 15th of every month, but never on a Saturday or Sunday.

```
15 3 * * 0-4,6
```

3:15 am every day except Fridays.

```
0 * * 1 *
```

Every hour, on the hour, in January.

When a scheduled event fires, the user-supplied handler function is invoked; it receives a pointer to the event structure as an argument. This callback mechanism allows arbitrary code to be executed at specific dates and times.

All of the functions described in this chapter are defined in the header '`cbase/sched.h`'.

6.1 Scheduler Control Functions

The following functions control the event scheduler. Due to idiosyncrasies inherent in UNIX signal handling, the scheduler can only be used on a per-process basis, and hence these functions are not threadsafe. When used in a multithreaded application, calls to these functions should be protected by a mutex lock.

`c_bool_t C_sched_init` (*void*) [Function]
`c_bool_t C_sched_shutdown` (*void*) [Function]

These functions initialize and shut down the real-time scheduler.

`C_sched_init()` initializes the real-time scheduler. The current disposition of the real-time signal `SIGRTMIN` is saved in static storage within the library, and is then reassigned to call the scheduler's internal event handler. The function returns `TRUE` on success, or `FALSE` on failure (for example, if the scheduler has already been initialized).

`C_sched_shutdown()` shuts down the real-time scheduler. The scheduler is deactivated, and the disposition of the real-time signal `SIGRTMIN` is restored.

All events being managed by this scheduler are then deactivated via calls to `C_sched_event_deactivate()`. The function returns `TRUE` on success, or `FALSE` on failure (for example, if the scheduler was not initialized).

`void C_sched_poll` (*void*) [Function]

With the single-threaded version of the library, a program must periodically poll the scheduler to allow events to be fired at their scheduled times. This function is provided for that purpose. It enumerates all of the registered events and fires any that are due at the time of the call. Since the scheduling granularity is one minute, this function must be called exactly once per minute to ensure proper scheduler behavior.

In the multi-threaded version of the library, the scheduler runs in a dedicated thread, hence this function is a no-op and should not be used.

6.2 Event Scheduling Functions

The following functions manipulate scheduler events, which are represented by the type *c_schedevt_t*.

`c_schedevt_t * C_sched_event_create` (*const char *`timespec`, c_bool_t `once`, void *`hook`, void (*`handler`)(c_schedevt_t *, time_t), void (*`destructor`)(c_schedevt_t *), uint_t `id`*) [Function]

`c_bool_t C_sched_event_destroy` (*c_schedevt_t *`e`*) [Function]

These functions create and destroy scheduler events. `C_sched_event_create()` creates a new scheduler event. The date/time specification string is passed as *timespec*. The flag *once* specifies whether the event should fire once or multiple times; if *once* is `TRUE`, the scheduler will deactivate the event via a call to `C_sched_event_deactivate()` immediately after the first time it fires. The pointer *hook* may be used to attach arbitrary user data to the event structure; this data may be retrieved using the function `C_sched_event_data()`, described below. The argument *handler* is a pointer to a handler function that will be invoked when the event fires; the event structure itself, and the time at which the event fired (as a *time_t* value) will be passed to the handler upon its invocation. The argument *destructor* is a pointer to an optional destructor function which should be called when this event is deactivated via a call to `C_sched_event_deactivate()`. Either or both of *handler* and *destructor* may be `NULL`, but a `NULL` value for *handler* is not useful. The parameter *id* is a numeric ID to assign to the event.

The function returns a pointer to the newly created event structure on success, or `NULL` on failure (for example, if *timespec* is an invalid specification string).

`C_sched_event_destroy()` destroys the scheduler event *e*. All memory associated with the event (not including any user-supplied data) is deallocated. The function returns `TRUE` on success, or `FALSE` on failure (for example, if *e* is `NULL`).

`c_bool_t C_sched_event_activate` (*c_schedevt_t *`e`*) [Function]

`c_bool_t C_sched_event_deactivate` (*c_schedevt_t *`e`*) [Function]

These functions activate and deactive the scheduler event *e*.

`C_sched_event_activate()` activates the event *e* by adding it to the scheduler's event list. The function returns `TRUE` on success, or `FALSE` on failure (for example, if *e* is `NULL`, or is already in the scheduler's event list).

`C_sched_event_deactivate()` deactivates the event *e* by removing it from the scheduler's event list. If a destructor function was specified for this event when it was created via `C_sched_event_create()`, that function is invoked with *e* as an argument. The function returns `TRUE` on success, or `FALSE` on failure (for example, if *e* is `NULL`, or is not in the scheduler's event list).

`c_schedevt_t * C_sched_event_find` (*uint_t* ***id***) [Function]

This function searches for a scheduler event with ID *id* in the scheduler's event list. It returns a pointer to the matching event structure on success, or `NULL` on failure.

`void * C_sched_event_data` (*c_schedevt_t *****e***) [Function]

This function (which is implemented as a macro) returns the user-data for the scheduler event *e*.

`uint_t C_sched_event_id` (*c_schedevt_t *****e***) [Function]

This function (which is implemented as a macro) returns the ID for the scheduler event *e*.

7 IPC Functions

"IPC" refers to *Inter-Process Communications*, a set of mechanisms provided by UNIX to facilitate collaboration between processes. Shared memory, semaphores, signals, and pseudoterminals are some of the most commonly used IPC mechanisms.

All of the constants, macros, and functions described in this chapter are defined in the header '`cbase/ipc.h`'.

7.1 File Descriptor Passing Functions

The following functions provide a means of passing file descriptors between unrelated processes. File descriptors (and other types of descriptors, depending on the system) may be passed over stream pipes. A "stream pipe" refers to a STREAMS-based full duplex pipe or, more commonly, a UNIX-domain socket.

`c_bool_t C_fd_send` (*int* **`sd`**, *int* **`fd`**) [Function]

This function sends the file descriptor *fd* over the stream pipe *sd*. It returns `TRUE` on success or `FALSE` on failure.

`c_bool_t C_fd_recv` (*int* **`sd`**, *int ****`fd`**) [Function]

This function receives a file descriptor over the stream pipe *sd*, storing the received descriptor at *fd*. It returns `TRUE` on success or `FALSE` on failure.

7.2 Semaphore Functions

The following functions provide a simple API to POSIX semaphores, which are counting semaphores. A *counting semaphore* is a simple synchronization mechanism that can be used to coordinate the actions of multiple processes or threads. A process or thread *waits* for a semaphore, acquires the semaphore, performs some task, and then *posts* the semaphore, thereby releasing it.

The initial value of the semaphore, which is a positive integer, specifies how many instances of a resource are being guarded. For example, if the initial value is 2, then at most two threads or processes can lock the semaphore at a time. A semaphore created with an initial value of 1 is called a *binary semaphore* and is essentially the same as a mutex—it can be used to guard a single instance of a resource or to protect critical sections of code.

The type *c_sem_t* represents a counting semaphore.

`c_sem_t * C_sem_create` (*const char ****`name`**, *mode_t* **`mode`**, *uint_t* **`value`**) [Function]

`void C_sem_destroy` (*c_sem_t ****`sem`**) [Function]

These functions create and destroy semaphores. `C_sem_create()` initializes a new semaphore with the specified symbolic *name* and initial value of *value*. The POSIX standard specifies that *name* must begin with a slash character ('`/`') and may contain no other slash characters. For best portability, the length of *name* should not exceed 14 characters. The initial *value* must be an integer between 1 and `C_SEM_MAX_VALUE`.

If the underlying POSIX semaphore object with the specified *name* did not already exist, it is created. This new object is created with world, group, and owner access permission as specified by the *mode* parameter.

The function returns a pointer to the new semaphore on success, or `NULL` on failure.

`C_sem_destroy()` destroys the semaphore *sem*. If no other processes are using this semaphore, the underlying POSIX semaphore object is destroyed.

`c_bool_t C_sem_wait` (*c_sem_t* ***sem***) [Function]
`c_bool_t C_sem_trywait` (*c_sem_t* ***sem***) [Function]

These functions wait on the semaphore *sem*. `C_sem_wait()` waits for the semaphore, blocking the calling process or thread until the semaphore is acquired. `C_sem_trywait()` attempts to acquire the semaphore, returning immediately if it cannot be locked.

Both functions return `TRUE` if the semaphore was successfully acquired and `FALSE` otherwise.

`c_bool_t C_sem_post` (*c_sem_t* ***sem***) [Function]

This function posts the semaphore *sem*. It returns `TRUE` if the semaphore was successfully released and `FALSE` otherwise.

`const char * C_sem_name` (*c_sem_t* ***sem***) [Function]

This function (which is implemented as a macro) returns the symbolic name of the semaphore *sem*.

`int C_sem_value` (*c_sem_t* ***sem***) [Function]

This function returns the "current" value of the semaphore *sem*. The returned value may or may not be the actual semaphore value at the time that the function returns; it is only guaranteed to have been current at some point during the call.

On success, the function returns the current value of the semaphore. On failure, it returns -1.

`uint_t C_sem_initial_value` (*c_sem_t* ***sem***) [Function]

This function (which is implemented as a macro) returns the initial value with which the semaphore *sem* was created.

7.3 Shared Memory Functions

The following functions provide a simple API to POSIX shared memory objects. A shared memory segment is simply an arbitrary block of memory that is mapped into more than one process's address space; data written to the shared memory by one process is immediately visible to all other processes that have mapped that segment. It is the fastest and most flexible form of IPC.

The type *c_shmem_t* represents a shared memory segment.

`c_shmem_t * C_shmem_create` (*const char* ***name***, *size_t* **`size`**, *mode_t* **`mode`**) [Function]

This function creates a new shared memory segment with the specified symbolic *name* and size in bytes, *size*. The POSIX standard specifies that *name* must begin with a slash character ('`/`') and may contain no other slash characters. For best portability, the length of *name* should not exceed 14 characters. The size of the segment will be rounded up to the nearest system page size as reported by `sysconf()`.

If the underlying POSIX shared memory object with the specified *name* did not already exist, it is created, and the memory in the segment is zeroed. The new object is created with world, group, and owner access permission as specified by the *mode* parameter.

The function returns a pointer to the new *c_shmem_t* structure on success, or `NULL` on failure. The function `C_shmem_base()` may be used to obtain a pointer to the shared memory block itself.

`void C_shmem_destroy` (*c_shmem_t* ***mem***) [Function]

This function destroys the shared memory segment *mem*. The memory is unmapped from the calling process's address space. If no other processes have this segment mapped, the underlying POSIX shared memory object is destroyed. Finally, the *mem* structure itself is deallocated.

`void * C_shmem_base` (*c_shmem_t* ***mem***) [Function]

This function (which is implemented as a macro) returns a pointer to the base of the shared memory segment *mem*.

`size_t C_shmem_size` (*c_shmem_t* ***mem***) [Function]

This function (which is implemented as a macro) returns the size, in bytes, of the shared memory segment *mem*.

`c_bool_t C_shmem_resize` (*c_shmem_t* ***mem***, *size_t* ***size***) [Function]

This function resizes the shared memory segment *mem* to a new size of *size*. The requested size will be rounded up to the nearest system page size as reported by `sysconf()`.

The function returns `TRUE` on success or `FALSE` on failure.

`const char * C_shmem_name` (*c_shmem_t* ***mem***) [Function]

This function (which is implemented as a macro) returns the symbolic name of the shared memory segment *mem*.

7.4 Signal Handling Functions

The following function converts signal IDs to names.

`const char * C_signal_name` (*int* ***sig***) [Function]

This function returns a string representation of the signal specified by *sig*, or `NULL` if *sig* does not refer to a known signal.

7.5 Terminal Functions

The following functions operate on terminals and pseudoterminals. Interactive programs such as mail readers and editors run on terminals, but sometimes it is useful to control an interactive program with another program. An IPC mechanism known as a *pseudoterminal* can be used for this purpose. A pseudoterminal appears to be a normal terminal to the slave process, and as a FIFO pipe to the master process. The master process feeds input to and receives input from the slave process through the pipe, and the slave process reads from and writes to its terminal as it would normally.

These functions are not reentrant.

7.5.1 Terminal Control Functions

The following functions operate on UNIX file descriptors which are assumed to refer to terminal (tty) lines.

`c_bool_t C_tty_raw` (*int* `fd`) [Function]

This function puts the terminal associated with the file descriptor *fd* into *raw mode*. Once in this mode, the terminal driver does not do any buffering or processing of input or output. The current terminal settings for *fd* are stored in a static buffer inside the library, and may be restored via a call to `C_tty_unraw()`.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_ENOTTY`	*fd* does not refer to a terminal.
`C_ETCATTR`	The call to `tcgetattr()` or `tcsetattr()` failed.

`c_bool_t C_tty_unraw` (*int* `fd`) [Function]

This function undoes the changes to the attributes of the terminal associated with the file descriptor *fd* that resulted from the latest call to `C_tty_raw()`. The terminal attributes are restored to what they were just before the call to `C_tty_raw()`.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_ENOTTY`	*fd* does not refer to a terminal.
`C_ETCATTR`	The call to `tcsetattr()` failed.
`C_EINVAL`	There was no previous call to `C_tty_raw()`.

`c_bool_t C_tty_store` (*int* `fd`) [Function]

This function stores the current attributes of the terminal associated with the file descriptor *fd* in a static buffer inside the library. These attributes can later be restored via a call to `C_tty_restore()`. The function is useful when a terminal's attributes must be modified from their original values temporarily by a program, or when the attributes of one terminal must be copied to another terminal (or to a pseudoterminal).

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_ENOTTY`	*fd* does not refer to a terminal.
`C_ETCATTR`	The call to `tcgetattr()` failed.

`c_bool_t C_tty_restore` (*int* `fd`) [Function]

This function restores the terminal attributes saved with the latest call to `C_tty_store()`, applying them to the terminal associated with the file descriptor *fd*.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_ENOTTY`	*fd* does not refer to a terminal.
`C_ETCATTR`	The call to `tcgetattr()` failed.
`C_EINVAL`	There was no previous call to `C_tty_store()`.

`c_bool_t C_tty_sane` (*int* **`fd`**) [Function]

This function sets the attributes on the terminal associated with the file descriptor *fd* to sane (default) values. Specifically, the *input* flag word is set to `(ICRNL | IXON | IXOFF)`, the *output* flag word is set to `(OPOST)`, the *control* flag word is set to `(CREAD | HUPCL)`, and the *functions* flag word is set to `(ECHO | ECHOE | ECHOK | ICANON | ISIG | IEXTEN)`. The control characters are set to the following ASCII values: `VEOF` = 4, `VEOL` = 28, `VERASE` = 8, `VINTR` = 21, `VKILL` = 3, `VQUIT` = 255, `VSUSP` = 255, `VSTART` = 17, `VSTOP` = 19.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_ENOTTY`	*fd* does not refer to a terminal.
`C_ETCATTR`	The call to `tcsetattr()` failed.

`c_bool_t C_tty_getsize` (*uint_t* **`*columns`**, *uint_t* **`*rows`**) [Function]

This function determines the size of the terminal, in characters. If successful, it stores the number of columns and rows (lines) at *columns* and *rows*, respectively, and returns `TRUE`. On failure, it leaves the values at *columns* and *rows* unmodified and returns `FALSE`.

7.5.2 Pseudoterminal Control Functions

The following functions provide an interface to System V or BSD pseudoterminals. A pseudoterminal consists of a master and a slave; the child process reads from and writes to the slave device as if it were a “real” terminal, and the parent process reads from and writes to the child through the master device.

The type *c_pty_t* represents a pseudoterminal device pair.

`c_pty_t * C_pty_create` (*void*) [Function]

This function allocates a pseudoterminal from the system. On success, it returns a pointer to the new *c_pty_t* structure. The functions `C_pty_master_fd()` and `C_pty_slave_fd()`, described below, can be used to obtain the file descriptors for the corresponding devices. On failure, the function returns `NULL` and sets `c_errno` to one of the following values:

`C_EGETPTY`	The call to `grantpt()`, `unlockpt()`, or `ptsname()` (System V) or `openpty()` (BSD) failed.
`C_EOPEN`	The call to `open()` failed.
`C_EIOCTL`	The call to `ioctl()` failed.

`c_bool_t C_pty_destroy` (*c_pty_t* **`*pty`**) [Function]

This function closes the master and slave devices for the pseudoterminal *pty* and deallocates the data structure at *pty*. The function returns `TRUE` on success, or `FALSE` on failure (for example, if *pty* is `NULL`).

`int C_pty_master_fd` (*c_pty_t* **`*pty`**) [Function]

`int C_pty_slave_fd` (*c_pty_t* **`*pty`**) [Function]

These functions (which are implemented as macros) return the file descriptors for the master and slave device, respectively, of the pseudoterminal *pty*.

`const char * C_pty_slave_name` (*c_pty_t* *****pty***) [Function]

This function (which is implemented as a macro) returns the path of the device file for the slave device associated with the pseudoterminal *pty*.

8 Networking Functions

This chapter describes a high-level, abstracted interface to the Berkeley socket IPC mechanism. A *socket* is similar to a pipe, but the two endpoints of a socket may exist on different hosts. This means that sockets can be used to transfer data between two networked machines, whether they are both on the same local area network, or connected to the Internet from opposite sides of the globe.

Sockets are generally employed as a communications medium in client/server systems. Generally, a server process creates a *master* or *listening* socket, binds the socket to a specific port number, and listens for connections on that socket. Clients that know the IP address or DNS name of the host and the port number on which the server is listening can connect to that server. Once a connection is established, data can be easily exchanged between the server and the client.

In the traditional model, a server typically forks a subprocess to handle each incoming connection; otherwise the server would only be able to service one client at a time. The subprocess communicates with the client and exits when the connection is closed. Meanwhile, the main server process continues to listen for new connections.

Since all of the networking functions in this library are reentrant, they can be used to write a multithreaded server, in which one thread is tasked with listening for new connections and spawning (or assigning) a worker thread for each incoming connection.

As anyone who has written network code in UNIX knows, the socket functions provide a very low-level and cumbersome networking API. The complexity of the API is inherent in its flexibility, but in general only a subset of the available functions and flags are used. Furthermore, networking code is very similar across many servers. This library greatly simplifies the development of networked applications by hiding most of this complexity.

All of the constants, macros, and functions described in this chapter are defined in the header '`cbase/net.h`'.

Most of the functions described below return boolean or integer values. On failure, they return `FALSE` or `-1`, respectively, and set the global variable `c_errno` to reflect the type of error. The error codes are defined in the header file '`cbase/cerrno.h`'. Some of the error codes indicate that a system call or socket library function failed; in this case, the `errno` variable can be examined to get the system-defined error code.

Note that in the threaded version of the library, `c_errno` is defined as a macro that returns a thread-specific error value.

8.1 Network Information Functions

The following functions can be used to obtain information about network services (namely, those listed in the '`/etc/services`' file), and to resolve IP addresses into DNS names.

`in_port_t C_net_get_svcport` (*const char* ***name**, *uint_t* ***type**) [Function]

This function looks up the port number for the service named *name*. The value at *type* specifies which type of service to search for. It can be one of `C_NET_TCP`, `C_NET_UDP`, or `C_NET_UNKNOWN`. If the value at *type* is `C_NET_UNKNOWN`, it is modified to reflect the actual type of the service found; `C_NET_OTHER` is stored at *type* if the service is neither TCP- nor UDP-based.

On success, the function returns the port number of the named service. On failure, it returns `-1` and sets `c_errno` to one of the following values:

`C_EINVAL`	*name* or *type* is `NULL`, or the value at *type* is invalid.
`C_ESVCINFO`	The call to `getservbyname_r()` failed.

`c_bool_t C_net_get_svcname` (*in_port_t* **`port`**, *uint_t* *****`type`**, *char* *****`buf`**, *size_t* **`bufsz`**) [Function]

This function is the inverse of `C_net_get_svcport()`. It attempts to find a service on the specified *port* of the specified *type*, and writes up to *bufsz* - 1 bytes of the service's name at *buf*. The buffer is unconditionally `NUL`-terminated. If the value at *type* is `C_NET_UNKNOWN`, it is modified to reflect the actual type of the service: `C_NET_OTHER` is stored at *type* if the service is neither TCP- nor UDP-based.

The function returns `TRUE` if the service was found. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*type* or *buf* is `NULL`, *bufsz* is 0, or the value at *type* is invalid.
`C_ESVCINFO`	The call to `getservbyport_r()` failed.

`c_bool_t C_net_resolve` (*const char* *****`ipaddr`**, *char* *****`buf`**, *size_t* **`bufsz`**) [Function]

This function attempts to resolve the dot-separated IP address at *ipaddr* into a valid DNS name. Up to *bufsz* - 1 bytes of the resolved name are written at *buf*. The buffer is unconditionally `NUL`-terminated.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*ipaddr* is `NULL` or is an empty string, or *buf* is `NULL`.
`C_EADDRINFO`	The call to `inet_addr()` or `gethostbyaddr_r()` failed.

`c_bool_t C_net_resolve_local` (*char* *****`addr`**, *char* *****`ipaddr`**, *size_t* **`bufsz`**, *in_addr_t* *****`ip`**) [Function]

This function obtains the address of the local host in one or more formats. If *addr* is not `NULL`, up to *bufsz* - 1 bytes of the local host's canonical DNS name are written at *addr*, and the buffer is unconditionally `NUL`-terminated. If *ipaddr* is not `NULL`, up to *bufsz* - 1 bytes of the host's address in the form of a dot-separated IP address are written at *ipaddr*, and the buffer is unconditionally `NUL`-terminated. Finally, if *ip* is not `NULL`, the local host's packed IP address is stored at *ip*.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*addr*, *ipaddr*, and *ip* are all `NULL`, or *bufsz* is 0.
`C_EADDRINFO`	The call to `gethostbyname_r()` failed.

8.2 Socket Control Functions

The following routines provide control functions, such as connecting and disconnecting sockets, setting socket options, and obtaining socket addresses.

The type *c_socket_t* represents a socket.

`c_socket_t * C_socket_create` (*int* **`type`**) [Function]
`c_bool_t C_socket_destroy` (*c_socket_t* **`*s`**) [Function]

These two functions create and destroy sockets.

`C_socket_create()` creates a new socket of the specified *type*. The value of *type* may be either `C_NET_TCP` (for reliable, connection-based stream sockets) or `C_NET_UDP` (for unreliable, connectionless or connection-based datagram sockets). The function returns the newly created *c_socket_t* structure on success. On failure, it returns `NULL` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`, or the value of *type* is invalid.
`C_ESOCKET`	The call to `socket()` failed.

`C_socket_destroy()` shuts down and closes the socket *s*, freeing all memory associated with the socket, including the *c_socket_t* structure. The function will only destroy the socket if it is created but not connected or listening, or if it has been shut down via a call to `C_socket_shutdown()`. It returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`.
`C_EBADSTATE`	The socket is not in a created or shut down state.

`c_bool_t C_socket_create_s` (*c_socket_t* **`*s`**, *int* **`type`**) [Function]
`c_bool_t C_socket_destroy_s` (*c_socket_t* **`*s`**) [Function]

These functions are static variants of `C_socket_create()` and `C_socket_destroy()`, respectively. They do not perform any allocation or deallocation of *c_socket_t* structures, but rather operate on pointers to preallocated structures.

`C_socket_create_s()` initializes the socket structure at *s* as a new socket. It returns `TRUE` on success and `FALSE` on failure.

`C_socket_destroy_s()` disposes the socket at *s*. It returns `TRUE` on success and `FALSE` on failure.

`c_bool_t C_socket_listen` (*c_socket_t* **`*s`**, *in_port_t* **`port`**) [Function]

This function binds the socket *s* to a local address and, if the socket is a TCP socket, initiates listening on the specified TCP *port*. It is typically used by a server process to prepare for incoming connection requests which are subsequently accepted using `C_socket_accept()`. The function may also be used with a multicast UDP socket to notify the operating system that the socket should only receive multicast datagrams that are destined for the specified *port*. The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`.
`C_EBADSTATE`	The socket is not in a created state.
`C_EADDRINFO`	The call to `gethostbyname_r()` failed.
`C_EBIND`	The call to `bind()` failed.
`C_ELISTEN`	The call to `listen()` failed.

`c_socket_t * C_socket_accept` (*c_socket_t* **`*s`**) [Function]

This function accepts a pending connection request on the socket *s*, returning a new socket which may be used to communicate with the client process. If *s* is in a non-blocking state and no connection is pending, the function returns immediately with a value of `NULL`; otherwise, it blocks until a connection request arrives. This new socket is created as a blocking socket and will be connected to the client.

The function returns the newly created socket on success. On failure, it returns `NULL` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* or *ns* is `NULL`.
`C_EBADSTATE`	The socket is not in a listening state.
`C_EBLOCKED`	The socket is marked as non-blocking and no connection is currently pending.
`C_EACCEPT`	The call to `accept()` failed.

`c_bool_t C_socket_accept_s` (*c_socket_t* ***s**, *c_socket_t* ***ms**) [Function]

This function is a static variant of `C_socket_accept()`. It does not perform any allocation or deallocation of *c_socket_t* structures, but rather operates on pointers to preallocated structures.

`C_socket_accept_s()` accepts a pending connection request on the socket *ms* and initializes the socket structure at *s* as a socket for that connection. The function returns `TRUE` on success and `FALSE` on failure.

`c_bool_t C_socket_connect` (*c_socket_t* ***s**, *const char* ***host**, *in_port_t* **port**) [Function]

This function connects the socket *s* to a port on a remote host. The argument *host* is the address of the remote host; it may be either a dot-separated IP address or a valid DNS name. The argument *port* specifies which TCP or UDP port to connect to on the remote host. This function is normally used by a client process that wishes to connect to a server.

TCP sockets must be connected before they can be used to transfer data, while UDP sockets may be used in either a connected or unconnected state. Connecting a UDP socket binds it to a specific address, which means that the destination address need not be specified for each datagram sent. Connecting a UDP socket also allows higher-level I/O routines in this library to be used with the socket.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* or *host* is `NULL`, or *host* is an empty string.
`C_EBADSTATE`	The socket is not in a created state.
`C_EADDRINFO`	The call to `gethostbyaddr_r()` or `gethostbyname_r()` failed, most likely because *host* is not a valid host address.
`C_ENOCONN`	The connection was refused.
`C_ECONNECT`	The call to `connect()` failed.
`C_ETIMEOUT`	The connect operation timed out.

`c_bool_t C_socket_shutdown` (*c_socket_t* ***s**, *uint_t* **how**) [Function]

This function shuts down reading, writing, or reading and writing on the socket *s*. The socket must be in a connected state in order to be shut down, and it cannot be destroyed until it is in a shut down state. The argument *how* specifies how the socket is to be shut down: `C_NET_SHUTRD` for reading, `C_NET_SHUTWR` for writing, or `C_NET_SHUTALL` for both reading and writing. If a socket is shut down for writing only, then the process at the remote end of the socket will receive an `EOF` if it attempts to read from it. Conversely, if it is shut down for reading, the process at the local end will receive an `EOF` if it attempts to read from it. This function may be called repeatedly on a connected socket.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`, or the value of *how* is invalid.
`C_EBADSTATE`	The socket is not in a connected or partially shut down state.

`c_bool_t C_socket_get_peeraddr` (*c_socket_t* ***s**, *char* ***buf**, *size_t* **bufsz**) [Function]

This function obtains the address of the peer of the socket *s*, that is, the name of the host on the remote end of the connection. At most *bufsz* - 1 bytes of the host name are written to *buf*, and the buffer is unconditionally `NUL`-terminated. The address may either be a DNS name, such as “ftp.uu.net” or, if the address could not be resolved, a dot-separated IP address, such as “132.32.5.1”.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* or *buf* is `NULL`, or *bufsz* is 0.
`C_EBADSTATE`	The socket is not in a connected state.
`C_EADDRINFO`	The address could not be determined (the call to `gethostbyaddr_r()` failed).

`in_addr_t C_socket_get_ipaddr` (*c_socket_t* ***s**) [Function]
`in_addr_t C_socket_get_peeripaddr` (*c_socket_t* ***s**) [Function]

These functions obtain the packed IP address of each end of the socket *s*. `C_socket_get_ipaddr()` returns the IP address of the local end of the socket, and `C_socket_get_peeripaddr()` returns the IP address of the remote end of the socket. These values will only be meaningful if the socket is in a connected state.

These functions are implemented as macros.

`c_bool_t C_socket_fopen` (*c_socket_t* ***s**, *int* **buffering**) [Function]

This function opens a stream for the socket *s* that can be used with the *stdio* library functions. The pointer to this stream may be obtained with the `C_socket_get_fp()` macro. The argument *buffering* specifies what type of buffering will be performed on the new stream: `C_NET_BUFFERING_NONE` for no buffering, `C_NET_BUFFERING_LINE` for line buffering, or `C_NET_BUFFERING_FULL` for full buffering.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`, or the value of *buffering* is invalid.
`C_EBADSTATE`	The socket is not in a connected state.
`C_EBADTYPE`	*s* is not a TCP socket.
`C_EFDOPEN`	The call to `fdopen()` failed.

`c_bool_t C_socket_fclose` (*c_socket_t* ***s**) [Function]

This function closes the *stdio* stream associated with the socket *s*, if one has been opened via a call to `C_socket_fopen()`.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`.

`c_socket_t * C_socket_reopen` (*int* **sd**) [Function]

This function creates a socket structure for the socket whose descriptor is *sd*.

On success, the function returns the newly created *c_socket_t* structure. On failure, it returns `NULL` and sets `c_errno` to one of the following values:

`C_EINVAL`	*sd* is negative.
`C_EFCNTL`	The call to `fcntl()` failed.
`C_EBADTYPE`	The socket is neither a TCP nor UDP socket.
`C_ESOCKINFO`	*sd* does not refer to a socket, or the call to `getpeername()` failed.

`c_bool_t C_socket_reopen_s` (*c_socket_t* ***s**, *int* **sd**) [Function]

This function is a static variant of `C_socket_reopen()`. It does not perform any allocation or deallocation of *c_socket_t* structures, but rather operates on pointers to preallocated structures.

`C_socket_reopen_s()` initializes the socket structure at *s* as the socket whose descriptor is *sd*. The function returns `TRUE` on success and `FALSE` on failure.

`c_bool_t C_socket_set_option` (*c_socket_t* ***s**, *uint_t* **option**, *c_bool_t* **flag**, *uint_t* **value**) [Function]

This function sets the option *option* on the socket *s*. Valid options are as follows:

`C_NET_OPT_BLOCK`
: Changes the blocking state on the socket. If a socket is in an unblocked state, any I/O or control operation on the socket that would cause the process or thread to block returns immediately with an error code of `C_EBLOCKED`. Blocking is turned on if *flag* is `TRUE` and turned off if *flag* is `FALSE`.

`C_NET_OPT_LINGER`
: Changes the linger mode on the socket. By default, linger mode is off; when a socket is closed and there is still data in the socket send buffer, the operating system will attempt to deliver this data before destroying the socket, but the close operation will return immediately. If linger mode is turned on (*flag* is `TRUE`), the process will be blocked for up to *value* seconds when it closes a socket while the operating system attempts to deliver any pending data; any data still not delivered after this interval (which may be 0 seconds) will be discarded. The default setting is off.

`C_NET_OPT_REUSEADDR`
: While there are various uses for this option, the most common use is to allow a server to bind to its port even if there are existing clients connected to that port. The option is turned on if *flag* is `TRUE` and turned off if *flag* is `FALSE`. The default setting is off. The function `C_socket_listen()` automatically turns on this option.

`C_NET_OPT_OOBINLINE`
: Specifies whether out-of-band data should be sent inline on the socket. The option is turned on if *flag* is `TRUE` and turned off if *flag* is `FALSE`. The default setting is off.

`C_NET_OPT_KEEPALIVE`
: This option provides a means to detect if the host at the remote end of the socket is still alive by sending a periodic TCP *probe*. If the option is turned on, the operating system will terminate the connection if the remote host does not respond to the probes. The option is turned on if *flag* is `TRUE` and turned off if *flag* is `FALSE`. The default setting is off.

`C_NET_OPT_RECVBUF`
: Changes the size of the socket receive buffer. The argument *value* specifies the new size of the receive buffer, in bytes, and must be greater than 0. The argument *flag* is ignored.

`C_NET_OPT_SENDBUF`
: Changes the size of the socket send buffer. The argument *value* specifies the new size of the send buffer, in bytes, and must be greater than 0. The argument *flag* is ignored.

This function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL` or the value of *option* is invalid.
`C_EFCNTL`	The call to `fcntl()` failed.
`C_EBADSTATE`	An attempt was made to change the blocking state on a socket that is shut down.
`C_ESOCKINFO`	The call to `setsockopt()` failed.

`c_bool_t C_socket_get_option` (*c_socket_t* ***`s`**, *uint_t* **`option`**, *c_bool_t* ***`flag`**, *uint_t* ***`value`**) [Function]

This function gets the current settings for the option *option* on the socket *s*. The arguments *flag* and *value* are used to store the settings for the given option; which of these arguments is used depends on the type of option. See `C_socket_set_option()` above for a description of the available options and their settings.

This function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s*, *flag*, or *value* is `NULL`, or the value of *option* is invalid.
`C_ESOCKINFO`	The call to `getsockopt()` failed.

`c_bool_t C_socket_block` (*c_socket_t* ***`s`**) [Function]
`c_bool_t C_socket_unblock` (*c_socket_t* ***`s`**) [Function]
`c_bool_t C_socket_isblocked` (*c_socket_t* ***`s`**) [Function]

These convenience functions change and test the blocking state on the socket *s*; they are implemented as macros which evaluate to the appropriate calls to `C_socket_set_option()` and `C_socket_get_option()`.

`C_socket_block()` marks the socket *s* as blocking, and `C_socket_unblock()` marks the socket *s* as non-blocking. `C_socket_isblocked()` returns `TRUE` if the socket is in blocking mode and `FALSE` if it is non-blocking.

`int C_socket_get_fd` (*c_socket_t* *__s__) [Function]
`FILE * C_socket_get_fp` (*c_socket_t* *__s__) [Function]

These functions (which are implemented as macros) return the socket descriptor and file stream pointer, respectively, for the socket *s*. `C_socket_get_fp()` returns the stream pointer, if one has been created via `C_socket_fopen()`; otherwise it returns `NULL`.

`int C_socket_get_type` (*c_socket_t* *__s__) [Function]

This function returns the type of the socket *s*, either `C_NET_TCP` or `C_NET_UDP`. It is implemented as a macro.

`void C_socket_set_timeout` (*c_socket_t* *__s__, *int* `sec`) [Function]
`int C_socket_get_timeout` (*c_socket_t* *__s__) [Function]

These functions set and get the I/O timeout for the socket *s*. They are implemented as macros.

`C_socket_set_timeout()` sets the timeout to *sec* seconds. The socket receive and send functions described below all return an error if the corresponding I/O operation times out after the given number of seconds.

`void C_socket_set_conn_timeout` (*c_socket_t* *__s__, *int* `sec`) [Function]
`int C_socket_get_conn_timeout` (*c_socket_t* *__s__) [Function]

These functions set and get the connection timeout for the socket *s*. They are implemented as macros.

`C_socket_set_conn_timeout()` sets the timeout to *sec* seconds. The `C_socket_connect()` function will return an error if a connection cannot be established within the specified number of seconds. The default, system-imposed timeout of roughly 75 seconds is an upper bound on this timeout; therefore passing values greater than 75 will not lengthen the timeout. Timeout values of 0 or less are interpreted as an infinite timeout.

`void C_socket_set_userdata` (*c_socket_t* *__s__, *void* *`data`) [Function]
`void * C_socket_get_userdata` (*socket_t* *__s__) [Function]

These functions set and get the “user data” field of the socket *s*. This field is simply a pointer which can be used to attach arbitrary data to a socket. The functions are implemented as macros.

8.3 Socket Multicast Functions

These functions provide UDP multicast functionality. Multicast addresses range from 224.0.0.0 through 239.255.255.255. A UDP datagram sent to a multicast address is delivered to all hosts on the network which have joined the multicast group specified by that address. Multicasting is described in detail in chapter 19 of *UNIX Network Programming Volume 1* by W. Richard Stevens.

`c_bool_t C_socket_mcast_join` (*c_socket_t* *__s__, *const char* *`addr`) [Function]
`c_bool_t C_socket_mcast_leave` (*c_socket_t* *__s__, *const char* *`addr`) [Function]

These functions provide a means for joining and leaving a multicast group. `C_socket_mcast_join()` assigns the UDP socket *s* to the multicast group specified by the address *addr*, which may either be a valid DNS name or a dot-separated IP address. `C_socket_mcast_leave()` removes the UDP socket *s* from the multicast group specified by the address *addr*.

These functions return `TRUE` on success. On failure, they return `FALSE` and set `c_errno` to one of the following values:

`C_EINVAL`	*s* or *addr* is `NULL`, or *addr* is an empty string.
`C_EBADTYPE`	*s* is not a UDP socket.
`C_EADDRINFO`	The call to `gethostbyaddr_r()` or `gethostbyname_r()` failed, most likely because *addr* is not a valid network address.
`C_ESOCKINFO`	The call to `setsockopt()` failed.

`c_bool_t C_socket_mcast_set_ttl` (*c_socket_t* ***s**, *c_byte_t* ***ttl***) [Function]

This function sets the time-to-live value on the socket *s* to *ttl*. It returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`.
`C_EBADTYPE`	*s* is not a UDP socket.
`C_ESOCKINFO`	The call to `setsockopt()` failed.

`c_bool_t C_socket_mcast_set_loop` (*c_socket_t* ***s**, *c_bool_t* ***loop***) [Function]

This function enables or disables the loopback function for the UDP socket *s*. If loopback is enabled (*loop* is `TRUE`), then any multicast datagrams that are sent out over this socket from the local host will also be delivered back to the host. If loopback is disabled, they are not. The loopback feature is enabled by default.

The function returns `TRUE` on success. On failure, it returns `FALSE` and sets `c_errno` to one of the following values:

`C_EINVAL`	*s* is `NULL`.
`C_EBADTYPE`	*s* is not a UDP socket.
`C_ESOCKINFO`	The call to `setsockopt()` failed.

8.4 Socket I/O Functions

The following functions are high-level routines for reading data from and writing data to TCP and UDP sockets.

`int C_socket_recv` (*c_socket_t* ***s**, *char* ***buf***, *size_t* ***bufsz***, *c_bool_t* ***oobf***) [Function]

`int C_socket_send` (*c_socket_t* ***s**, *const char* ***buf***, *size_t* ***bufsz***, *c_bool_t* ***oobf***) [Function]

These functions read data from and write data to the socket *s*. They may be used with TCP sockets or with connected UDP sockets. If *s* is a TCP socket and *oobf* is `TRUE`, the data is read or written *out-of-band*.

`C_socket_recv()` reads up to *bufsz* bytes of data from the socket *s* into *buf*. If *s* is a UDP socket, it attempts to receive the data as a single datagram. If *s* is a TCP socket, then the function continually loops, reading as much data as it can on each iteration, until either the buffer has been filled, or no more data is available for reading on the socket. If the socket is marked as blocking, the function will block waiting for more data to arrive, and will then continue to read the data.

`C_socket_send()` writes *bufsz* bytes of data starting at *buf* to the socket *s*. If *s* is a UDP socket, it attempts to send the buffer as a single datagram. If *s* is a TCP socket, then the function continually loops, writing as much data as it can on each iteration, until either the entire buffer has been written, or the socket is unable to accept any more data. If the socket is marked as blocking, the function will block waiting for it to drain, and will then continue writing the data.

If an error or timeout occurs after *n* bytes of data have been read or written, these functions return -*n*. If all of the data is read or written successfully, the functions return the number of bytes read or written (normally equal to *bufsz*). The functions return `0` if an error occurs before any data has been read or written. On timeout or failure, `c_errno` is set to one of the following values:

`C_EINVAL`	*s* or *buf* is `NULL`, or *bufsz* is 0.
`C_EBADTYPE`	*s* is neither a TCP nor a UDP socket.
`C_EBADSTATE`	The socket is not in a connected state.
`C_ELOSTCONN`	The connection was lost during data transfer.
`C_EBLOCKED`	The socket is marked as non-blocking and the requested transfer would block the process.
`C_ESEND`	The call to `send()` or `sendto()` failed.
`C_ERECV`	The call to `recv()` or `recvfrom()` failed.
`C_EMSG2BIG`	*s* is a UDP socket, and *bufsz* is too many bytes to send as one datagram.

`int` **`C_socket_sendto`** (*c_socket_t **`s`, *const char **`buf`, *size_t* `bufsz`, *const char **`addr`, *in_port_t* `port`) [Function]

`int` **`C_socket_recvfrom`** (*c_socket_t **`s`, *char **`buf`, *size_t* `bufsz`, *char **`addr`, *size_t* `addrsz`) [Function]

These functions send and receive datagrams over the unconnected UDP socket *s*.

`C_socket_sendto()` sends *bufsz* bytes beginning at *buf* to the specified *port* on the remote host named *addr*. `C_socket_recvfrom()` receives *bufsz* bytes from a remote host and writes them to *buf*, storing up to *addrsz* - 1 bytes of the remote host's address at *addr* and unconditionally `NUL`-terminates the buffer. This address is either a DNS name or, if the address could not be resolved, a dot separated IP address.

These functions return the number of bytes written or read upon success, *0* if the socket is marked as blocking and the operation would block, or `-1` upon failure. On block or failure, `c_errno` is set to one of the following values:

`C_EINVAL`	*s* or *addr* is `NULL`, or (for `C_socket_sendo()`) *addr* is an empty string.
`C_EBADTYPE`	*s* is not a UDP socket.
`C_EBADSTATE`	The socket is not in a created state.
`C_EADDRINFO`	The remote source or destination address could not be determined.
`C_ELOSTCONN`	The connection was lost.
`C_EBLOCKED`	The socket is marked as non-blocking and the requested operation would block.
`C_ESENDTO`	The call to `sendto()` failed.
`C_ERECVFROM`	The call to `recvfrom()` failed.

`C_EMSG2BIG`	*bufsz* is too many bytes to send or receive as one datagram.

`int C_socket_sendreply` (*c_socket_t* ***s**, *const char* ***buf**, *size_t* **bufsz**) [Function]

`int C_socket_recvreply` (*c_socket_t* ***s**, *char* ***buf**, *size_t* **bufsz**) [Function]

These functions are similar to `C_socket_sendto()` and `C_socket_recvfrom()` above, except that they reuse the remote address currently set for the UDP socket *s*. Specifically, `C_socket_sendreply()` sends a buffer of data to the address from which the last datagram was received on *s*, and `C_socket_recvreply()` receives a buffer of data from the address to which the last datagram was sent on *s*. These functions are intended for use on unconnected UDP sockets.

On success, the functions return the number of bytes sent or received. On failure, they return `-1` and set `c_errno` to one of the following values:

`C_EINVAL`	*buf* or *s* is `NULL` or *bufsz* is 0.
`C_EBADTYPE`	*s* is not a UDP socket.
`C_EBADSTATE`	The socket is not in a created state.
`C_ELOSTCONN`	The connection was lost.
`C_EBLOCKED`	The socket is marked as non-blocking and the requested operation would block.
`C_ESENDTO`	The call to `sendto()` failed.
`C_ERECVFROM`	The call to `recvfrom()` failed.
`C_EMSG2BIG`	*bufsz* is too many bytes to send or receive as one datagram.

`int C_socket_sendline` (*c_socket_t* ***s**, *const char* ***buf**) [Function]

`int C_socket_recvline` (*c_socket_t* ***s**, *char* ***buf**, *size_t* **bufsz**) [Function]

These functions read and write "lines" to and from the TCP socket *s*. The socket must be in blocking mode for use with these functions.

`C_net_socket_sendline()` writes the data at *buf*, followed by a CR+LF pair, to the socket *s*. The function returns when all of the data has been written or a timeout occurs.

`C_net_socket_recvline()` reads data into *buf*. It continues reading until *bufsz* - 1 bytes have been read, or a CR+LF pair has been encountered in the input, whichever occurs first. The CR+LF pair, if present, is discarded, and the buffer is unconditionally `NUL`-terminated. The function returns when all of the data has been read, or a timeout occurs.

If an error or timeout occurs after *n* bytes of data have been read or written, these functions return -*n*. If all of the data was written successfully, the functions return the number of bytes read or written. The functions return `0` if an error occurs before any data has been read or written. On timeout or failure, `c_errno` is set to one of the following values:

`C_EINVAL`	*s* or *buf* is `NULL`, or *bufsz* is 0.
`C_EBADTYPE`	*s* is not a TCP socket.
`C_EBADSTATE`	The socket is not in a connected state, or it is in non-blocking mode.
`C_ELOSTCONN`	The connection was lost during data transfer.
`C_ETIMEOUT`	A timeout occurred while waiting to read or write data.

`C_ESEND`	The call to `send()` or `sendto()` failed.
`C_ERECV`	The call to `recv()` or `recvfrom()` failed.

`int C_socket_writeline` (*c_socket_t* ***`s`**, *const char* ***`buf`**, *const char* ***`termin`**, *uint_t* **`slen`**, *uint_t* **`snum`**) [Function]

`int C_socket_readline` (*c_socket_t* ***`s`**, *char* ***`buf`**, *size_t* **`bufsz`**, *char* **`termin`**, *uint_t* **`slen`**, *uint_t* **`snum`**) [Function]

`int C_socket_rl` (*c_socket_t* ***`s`**, *char* ***`buf`**, *size_t* **`bufsz`**, *char* **`termin`**) [Function]

`int C_socket_wl` (*c_socket_t* ***`s`**, *const char* ***`buf`**, *const char* ***`termin`**) [Function]

These interfaces are deprecated, and are emulated by macros for backward compatibility. They evaluate to calls to the `C_socket_sendline()` and `C_socket_recvline()` functions, described above, ignoring the *termin*, *slen*, and *snum* arguments.

9 Library Information Functions

The following functions provide runtime information about the cbase library itself. All of the functions described in this chapter are defined in the header '`cbase/version.h`'.

`const char * C_library_version` (*void*) [Function]

This function returns a string containing the version number of the library. This version number corresponds to the version of the package.

`const char * C_library_info` (*void*) [Function]

This function returns a string containing information about the library, including the package name and version, the author, bug report email address, and copyright.

`const char ** C_library_options` (*void*) [Function]

This function returns a `NULL`-terminated array of strings which represent the set of options that are enabled in the library. Currently only the following option is defined: '`threaded`' (if the multi-threaded version of the library is in use).

Appendix A References

The following books proved to be indispensable during the implementation of this library.

- Gallmeister, Bill O. *POSIX.4: Programming for the Real World.*
- Lewine, Donald. *POSIX Programmer's Guide.*
- Nichols, Bradford, et. al. *Pthreads Programming.*
- Stevens, W. Richard. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI.* 2nd ed.
- Stevens, W. Richard. *UNIX Network Programming, Volume 2: Interprocess Communications.* 2nd ed.

Appendix B License

The cbase library is distributed under the terms of the LGPL. The complete text of the license appears below.

d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

   b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

   c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

   d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

   e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

   For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights,

from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

Function Index

Type Index

Symbol Index