FAT File System

Dynamic C 8.51 introduced a FAT (File Allocation Table) file system. The small footprint of this well-defined industry-standard file system makes it ideal for embedded systems. The Dynamic C implementation of FAT has a directory structure that can be accessed with either Unix or DOS style paths. The standard directory structure allows monitoring, logging, Web browsing, and FTP updates of files.

FAT module version 1.02 supports SPI-based serial flash devices. FAT versions 2.01 and 2.05 also support SPI-based serial flash devices and require Dynamic C 9.01 or later. FAT version 2.05 introduces support for NAND flash devices. FAT version 2.10 extends µC/OS-II compatibility to make the FAT API reentrant from multiple tasks. FAT version 2.13 adds support for SD cards and requires Dynamic C 10.21 or later. In all versions of the FAT, a battery-backed write-back cache reduces wear on the flash device and a round-robin cluster assignment helps spread the wear over its surface.

Please be sure check the Rabbit website for software patches and updates to Dynamic C, the FAT filesytem, and for your specific hardware:

`www.rabbit.com/support/downloads/`

The FAT library can be used in either *blocking* or *non-blocking* mode and supports both FAT12 and FAT16. (See Section A.3.1 for more information on these FAT types.)

Let's define some terms before continuing.

- A *device* is a single physical hardware item such as a hard drive, a serial flash or a NAND flash. E.g., one serial flash is a single *device*. The device, in turn, can host one to four *partitions*.
- A *partition* is a range of logical sectors on a device. A real-world example of a partition is what is commonly known as the C drive on a PC.
- A *driver* is the software interface that handles the hardware-specific aspects of any communication to or from the device.
- *Blocking* is a term that describes a function's behavior in regards to completion of the requested task. A blocking function will not return until it has completely finished its task. In contrast, a *non-blocking* function will return to its calling function before the task is finished if it is waiting for something. A non-blocking function can return a code that indicates it is not finished and should be called again. Used in conjunction with cooperative multitasking, non-blocking functions allow other processes to proceed while waiting for hardware resources to finish or become available.

Operations performed by the Dynamic C FAT implementation are:

- Formatting and partitioning of devices
- Formatting partitions
- File operations: create, open, close, delete, seek, read and write
- Directory[1] operations: create, read and delete
- Labels: create and delete

1. Overview of Document

This document describes the Dynamic C FAT file system. We take a look at a sample program in Section 2. Two additional sample programs, one for use with the blocking mode of the FAT and the other for use with the non-blocking mode are described in Section 3. Then Section 4 gives detailed descriptions of the various FAT file system functions (formatting, opening, reading, writing, and closing). Short, focused examples accompany each description. Section 5 provides a complete function reference section for the application programming interface (API). There is some general information about FAT file systems and also some web links for further study in Appendix A. And lastly, there are instructions for custom configurations in Appendix B.

> **Note:** All error codes returned from the Dynamic C FAT file system are defined in `LIB/.../FILESYSTEM/ERRNO.LIB`.

2. Running Your First FAT Sample Program

To run FAT samples, you need a Rabbit-based board with a supported flash type, such as the SPI-based serial flash device available on the RCM3300 or the RCM3700. FAT versions 2.01 and 2.05 require Dynamic C 9.01 or later. FAT version 2.05 extends the list of supported flash types to include NAND flash devices, such as those on the RCM3360 and 3370. FAT version 2.13 requires Dynamic C 10.21 or later and adds support for SD cards, available on the RCM4300 and 4310.

The board must be powered up and connected to a serial port on your PC through the programming cable to download a sample program.

In this section we look at `fat_create.c`, which demonstrate the basic use of the FAT file system. If you are using a serial or NAND flash device that has not been formatted or a removable device that was not formatted using Dynamic C, you must run `Samples\FileSystem\Fmt_Device.c` before you can run any other sample FAT program. The program, `Fmt_Device.c`, creates the default configuration of one partition that takes up the entire device.

If you are using an SD card, run `Fmt_Device.c` to remove the factory FAT32 partition and create a FAT16 partition. Be aware that although multiple partitions are possible on removable cards, most PC's will not support cards formatted in this fashion.

1. We use the terms *directory* and *subdirectory* somewhat interchangeably. The exception is the root directory—it is never called a subdirectory. Any directory below the root directory may be referred to as a directory or a subdirectory.

If you are using a removable NAND flash (XD cards), running `Fmt_Device.c` causes the device to no longer be usable without the Rabbit-based board or the Rabbit USB Reader for XD cards. Insert the NAND flash device into a USB-based flash card reader and format it to regain this usability. Note that this will only work if you have ***not*** defined the macro `NFLASH_CANERASEBADBLOCKS`. Defining this macro in a running application destroys proprietary information on the first block of the device, making it difficult to regain the usability of the NAND device when used without the Rabbit-based board.

If you are using FAT version 2.01 or later, you do not have to run `Fmt_Device.c` if you initialize the FAT file system with a call to `fat_AutoMount()` instead of `fat_Init()`. The function `fat_AutoMount()` can optionally format the device if it is unformatted; however, `fat_AutoMount()` will not erase and overwrite a factory-formatted removable device such as an SD card. If you are using an SD card, run `Fmt_Device.c` or erase the first three pages with the appropriate flash utitity (`sdflash_inspect.c` or `nflash_inspect.c`).

After the device has been formatted, open `Samples\FileSystem\fat_create.c`. Compile and run the program by pressing function key F9.

In a nutshell, `fat_create.c` initializes FAT, then creates a file, writes “Hello world!” to it, and then closes the file. The file is re-opened and the file is read, which displays “Hello world!” in the Dynamic C Stdio window. Understanding this sample will make writing your own FAT application easier.

The sample program has been broken into two functional parts for the purpose of discussion. The first part deals with getting the file system up and running. The second part is a description of writing and reading files.

2.1 Bringing Up the File System

We will look at the first part of the code as a whole, and then explain some of its details.

File Name: `Samples\FileSystem\fat_create.c`

```
#define FAT_BLOCK                                   // use blocking mode
#use "fat.lib"                                      // of FAT library

FATfile my_file;                                    // get file handle
char buf[128];                                      // 128 byte buffer for read/write of file

int main(){
   int i;
   int rc;                                          // Check return codes from FAT API
   long prealloc;                                   // Used if the file needs to be created.
   fat_part *first_part;                            // Use the first mounted FAT partition.

   rc = fat_AutoMount( FDDF_USE_DEFAULT );

   first_part = NULL;
   for(i=0;i < num_fat_devices * FAT_MAX_PARTITIONS; ++i)
   {                                                // Find the first mounted partition
      if ((first_part = fat_part_mounted[i]) != NULL) {
         break;                                     // Found mounted partition, so use it
      }
   }

   if (first_part == NULL) {                        // Check if mounted partition was found
      rc = (rc < 0) ? rc : -ENOPART;                // None found, set rc to a FAT error code
   } else{
      printf("fat_AutoMount() succeeded with return code %d.\n", rc);
      rc = 0;                                       // Found partition; ignore error, if any
   }

   if (rc < 0){                                     // negative values indicate error
      if (rc == -EUNFORMAT)
         printf("Device not Formatted, Please run Fmt_Device.c\n");
      else
         printf("fat_AutoMount() failed with return code %d.\n", rc);
      exit(1);
   }    // OK, file system exists and is ready to access. Let's create a file.
```

The first two statements:

```
#define FAT_BLOCK
#use "fat.lib"
```

cause the FAT library to be used in blocking mode.

FAT version 2.01 introduces a configuration library that chooses initialization settings based on the board type.The statement `#use "fat.lib"` brings in this configuration library, which in turn brings in the appropriate device driver library. The following table lists the device drivers that are available in the different FAT versions.

Table 1.

FAT Version	Device Driver
1.02, 2.01	`sflash_fat.lib`
2.05	`sflash_fat.lib` `nflash_fat.lib`

2.13	`sflash_fat.lib`
	`nflash_fat.lib`
	`SD_fat.lib`

Defining the macro `_DRIVER_CUSTOM` notifies `fat_config.lib` that a custom driver or hardware configuration is being used. For more information on how this works, see Appendix A.

Next some static variables are declared: a file structure to be used as a handle to the file that will be created and a buffer that will be used for reading and writing the file.

Now we are in `main()`. First there are some variable declarations: the integer `rc` is for the code returned by the FAT API functions. This code should always be checked, and *must* be checked if the non-blocking mode of the FAT is used. The descriptions for each function list possible return codes.

The variable `prealloc` stores the number of bytes to reserve on the device for use by a specific file. These clusters are attached to the file and are not available for use by any other files. This has some advantages and disadvantages. The obvious disadvantage is that it uses up space on the device. Some advantages are that having space reserved means that a log file, for instance, could have a portion of the drive set aside for its use only. Another advantage is that if you are transferring a known amount of information to a file, pre-allocation not only sets aside the space so you know you will not get half way through and run out, but it also makes the writing process a little faster as the allocation of clusters has already been dealt with so there is no need to spend time getting another cluster.

This feature should be used with care as pre-allocated clusters do not show up on directory listings until data is actually written to them, even though they have locked up space on the device. The only way to get unused pre-allocated clusters back is to delete the file to which they are attached, or use the `fat_truncate()` or `fat_split()` functions to trim or split the file. In the case of `fat_Split()`, the pre-allocated space is not freed, but rather attached to the new file created in the split.

Lastly, a pointer to a partition structure is declared with the statement:

```
fat_part *first_part;
```

This pointer will be used as a handle to an active partition. (The `fat_part` structure and other data structures needed by the FAT file system are discussed in `fat_AutoMount()`.) The partition pointer will be passed to API functions, such as `fat_open()`.

Now a call is made to `fat_AutoMount()`. This function was introduced in FAT version 2.01 as a replacement for `fat_Init()`. Whereas `fat_Init()` can do all the things necessary to ready the first partition on the first device for use, it is limited to that. The function `fat_AutoMount()` is more flexible because it uses data from the configuration file `fat_config.lib` to identify FAT partitions and to optionally ready them for use, depending on the flags parameter that is passed to it. The flags parameter is described in the function description for `fat_AutoMount()`.

For this sample program, we are interested in the first usable FAT partition. The `for` loop after the call to `fat_AutoMount()` finds the partition, if one is available.

The `for` loop allows us to check every possible partition by using `num_fat_devices`, which is the number of configured devices, and then multiplying the configured devices by the maximum number of allowable partitions on a device, which is four. The `for` loop also makes use of `fat_part_mounted`, an array of pointers to partition structures that is populated by the `fat_autoMount()` call.

2.2 Using the File System

The rest of `fat_create.c` demonstrates how to use the file system once it is up and running.

File Name: `Samples\FileSystem\fat_create.c`

```
   prealloc = 0;

   rc = fat_Open( first_part, "HELLO.TXT", FAT_FILE, FAT_CREATE,
                  &my_file, &prealloc );

   if (rc < 0) {
      printf("fat_Open() failed with return code %d\n", rc);
      exit(1);
   }
   rc = fat_Write( &my_file, "Hello, world!\r\n", 15 );

   if (rc < 0) {
      printf("fat_Write() failed with return code %d\n", rc);
      exit(1);
   }
   rc = fat_Close(&my_file);
   if (rc < 0) {
      printf("fat_Close() failed with return code %d\n", rc);
   }

   rc = fat_Open( first_part, "HELLO.TXT",FAT_FILE, 0, &my_file,
                  NULL);

   if (rc < 0) {
      printf("fat_Open() (for read) failed, return code %d\n", rc);
      exit(1);
   }
   rc = fat_Read( &my_file, buf, sizeof(buf));
   if (rc < 0) {
      printf("fat_Read() failed with return code %d\n", rc);
   }
   else {
      printf("Read %d bytes:\n", rc);
      printf("%*.*s", rc, rc, buf);  // Print a string which is not NULL terminated
      printf("\n");
   }
   fat_UnmountDevice( first_part->dev );
   printf("All OK.\n");
   return 0;
}
```

The call to `fat_Open()` creates a file in the root directory and names it `HELLO.TXT`. A file must be opened before you can write or read it.

```
rc = fat_Open(first_part, "HELLO.TXT", FAT_FILE, FAT_CREATE,
              &my_file, &prealloc);
```

The parameters are as follows:

- `first_part` points to the partition structure initialized by `fat_AutoMount()`.
- `"HELLO.TXT"` is the file name, and is always an absolute path name relative to the root directory. All paths in Dynamic C must specify the full directory path explicitly.
- `FAT_FILE` identifies the type of object, in this case a file. Use `FAT_DIR` to open a directory.
- `FAT_CREATE` creates the file if it does not exist. If the file does exist, it will be opened, and the position pointer will be set to the start of the file. If you write to the file without moving the position pointer, you will overwrite existing data.

  Use `FAT_OPEN` instead of `FAT_CREATE` if the file or directory should already exist. If the file does not exist, you will get an `-ENOENT` error.

  Use `FAT_MUST_CREATE` if you know the file does not exist. This is a fail-safe way to avoid opening and overwriting an existing file since an `-EEXIST` error is returned if you attempt to create a file that already exists.
- `&my_file` is a file handle that points to an available file structure. It will be used for this file until the file is closed.
- `&prealloc` points to the number of bytes to allocate for the file. You do not want to pre-allocate any more than the minimum number of bytes necessary for storage, and so `prealloc` was set to 0. You could also use `NULL` instead of `prealloc` and `prealloc = 0`.

Next, the sample program writes the data "Hello, world!\r\n" to the file.

```
fat_Write( &my_file, "Hello, world!\r\n", 15 );
```

The parameters are as follows:

- `&my_file` is a pointer to the file handle opened by `fat_Open()`.
- `"Hello, world!\r\n"` is the data written to the file. Note that `\r\n` (carriage return, line feed) appears at the end of the string in the call. This is essentially a FAT (or really, DOS) convention for text files. It is good practice to use the standard line-end conventions. (If you just use `\n`, the file will read just fine on Unix systems, but some DOS-based programs may have difficulties.)
- `15` is the number of characters to write. Be sure to select this number with care since a value that is too small will result in your data being truncated, and a value that is too large will append any data that already exists beyond your new data.

The file is closed to release the file handle to allow it to be used to identify a different file.

```
rc = fat_Close( &my_file );
```

The parameter `&my_file` is a handle to the file to be closed. Remember to check for any return code from `fat_Close()` since an error return code may indicate the loss of data.

The file must be opened for any further work, even though `&my_file` may still reference the desired file. The file must be open to be active, so we call `fat_Open()` again. Now the file can be read.

```
rc = fat_Read( &my_file, buf, sizeof(buf));
```

The function `fat_Read()` returns the number of characters actually read. The parameters are as follows:

- `&my_file` is a handle to the file to be read.
- `buf` is a buffer for reading/writing the file that was defined at the beginning of the program.
- `sizeof(buf)` is the number of bytes to be read into `buf`. It does not have to be the full size of the buffer

Characters are read beginning at the current position of the file. (The file position can be changed with the `fat_Seek()` function.) If the file contains fewer than `sizeof(buf)` characters from the current position to the end-of-file marker (EOF), the transfer will stop at the EOF. If the file position is already at EOF, 0 is returned. The maximum number of characters read is 32767 bytes per call.

The file can now be closed. Call `fat_UnmountDevice()` [1] rather than simply calling `fat_Close()` to ensure that any data stored in cache will be written to the device. With a write-back cache, writes are delayed until either:

- all cache buffers are full and a new FAT read request requires a “dirty” cache buffer to be written out before the read can take place, or
- cache buffers for a partition or a device are being flushed due to an unmount call or explicit flush call.

Calling `fat_UnmountDevice()` will close all open files and unmount all mounted FAT partitions. This is the safest way to shut down a device. The parameter `first_part->dev` is a handle to the device to be unmounted.

```
fat_UnmountDevice( first_part->dev );
```

> **Note:** A removable device must be unmounted in order to flush its data before removal. Failure to unmount any partition on a device that has been written to could corrupt the file system. With the RCM43xx modules, there is a usage LED that is turned on when the SD card is mounted, and turned off when the SD card is unmounted.

1. Call fat_UnmountPartition() when using a FAT module prior to version 2.06.

3. More Sample Programs

This section studies blocking sample `FAT_SHELL.C` and non-blocking sample `FAT_NB_Costate.c` More sample programs are in the Dynamic C folder `Samples\FileSystem\FAT`. For example, there is `udppages.c`, an application that shows how to combine HTTP, FTP and zserver functionality to create web content than can be updated via FTP.

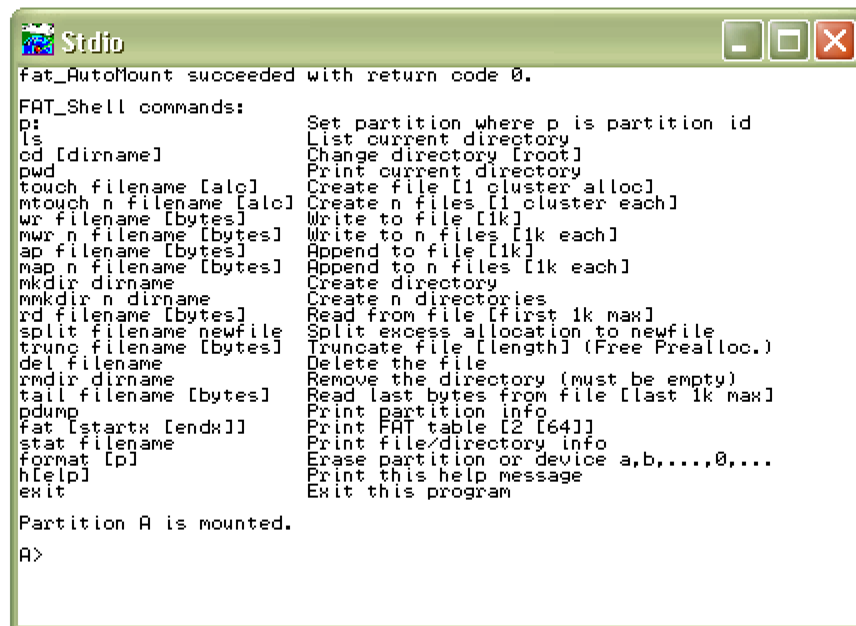
As described in Section 2, you will need a target board or module with a supported flash device, powered up and connected to a serial port on your PC through the programming cable.

3.1 Blocking Sample

The sample program `Samples\FileSystem\FAT_SHELL.C` allows you to use the FAT library by entering DOS-like or Unix-like commands. To run this sample, open Dynamic C, then open `FAT_SHELL.C`. Compile and run `FAT_SHELL.C` by pressing F9. If the flash device has not been formatted and partitioned, `FAT_SHELL.C` will format and partition the flash device, and then you will be prompted to run `FAT_SHELL.C` again (just press F9 when prompted). A display similar to the one shown in Figure 1 will open in the Dynamic C Stdio window.

Optional parameters are denoted by the square braces [and] following the command name. The [alc] after "touch" and "mtouch" indicates an optional allocation amount in bytes. The square braces in the description indicate the default value that will be used if the optional parameter is not given.

Figure 1. List of Shell Commands



You can type "h" and press enter at any time to display the FAT shell commands.

In the following examples the commands that you enter are shown in boldface type. The response from the shell program is shown in regular typeface.

```
> ls
Listing '' (dir length 16384)
   hello.txt rhsvdA len=15       clust=2
>
```

This shows the HELLO.TXT file that was created using the FAT_CREATE.C sample program. The file length is 15 bytes. Cluster 2 has been allocated for this file. The "ls" command will display up to the first six clusters allocated to a file.

The flag, rhsvdA, displays the file or directory attributes, with upper case indicating that the attribute is turned on and lower case indicating that the attribute is turned off. In this example, the archive bit is turned on and all other attributes are turned off.

These are the six attributes:

- **r** - read-only
- **h** - hidden file
- **s** - system
- **v** - volume label
- **d** - directory
- **a** - archive

To create a directory named DIR1, do the following:

```
> mkdir dir1
Directory '/dir1' created with 1024 bytes
>
```

This shows that DIR1 was created, and is 1024 bytes (size may vary by flash type).

Now, select DIR1:

```
> cd dir1
PWD = '/dir1'
>
```

Add a new file called RABBIT.TXT:

```
> touch rabbit.txt
File '/dir1/rabbit.txt' created with 1024 bytes
>
```

Note that the file name was appended to the current directory. Now we can write to RABBIT.TXT. The shell program has predetermined characters to write, and does not allow you to enter your own data.

```
> wr rabbit.txt
File '/dir1/rabbit.txt' written with 1024 bytes out of 1024
>
```

To see what was written, use the “rd” command.

```
> rd rabbit.txt

rabbit.txt 1024 The quick brown fox jumps over the lazy dog
rabbit.txt 1024 The quick brown fox jumps over the lazy dog
.
.
rab

Read 1024 bytes out of 1024
>
```

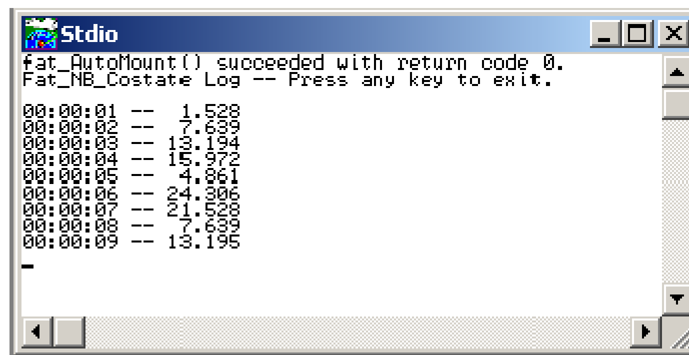
3.2 Non-Blocking Sample

To use the FAT file system in non-blocking mode, do not include the statement `#define FAT_BLOCK` in your application. The program interface to the library is the same as the blocking version, with the exception of the return code `-EBUSY` from many of the API functions.

The sample program `Fat_NB_Costate.c` in the `Samples\FileSystem` folder is an example of a non-blocking application. To view the code in its entirety, open it in Dynamic C. The following discussion will not examine every line of code, but will focus on what shows the non-blocking nature of the FAT library and how the application takes advantage of it.

Run Fat_`NB_Costate.c` and after 10 seconds the Stdio window will show something similar to the following:

Figure 2. Screen Shot of Fat_NB_Costate.c Running



Each line is an entry into a file that is stored in the FAT file system. The file is appended once every second and read and displayed once every ten seconds. In addition to the file system use and the screen output, if you are using an RCM3300, RCM3700 or PowerCore FLEX development board, the application blinks the LED on your board.

The code preceding `main()` brings in the required library and declares the file structure. And, as expected, there is no `#define` for the macro `FAT_BLOCK`. At the start of `main()` some system variable are created and initialized. This is followed by the code to bring up the FAT file system, which is similar to what we examined in Section 2.1 when looking at `fat_create.c`, with two essential differences. One, since we have initialized the FAT to be in non-blocking and we are making some calls to FAT functions that must return before we can continue, we must wait for the return.

A `while` loop accomplishes our goal of blocking on the function call until it returns something other than busy.

```
while ((rc = fat_Open( first_part, name, FAT_FILE, FAT_MUST_CREATE,
   &file, &alloc)) == -EBUSY);
```

The second difference from our earlier sample is the statement right before `fat_Open()`:

```
file.state = 0;
```

This is required before opening a file when using non-blocking mode in order to indicate that the file is not in use. Only do this once. After you have opened the file, do not alter the contents of the file structure.

If `fat_Open()` succeeds we can go into the non-blocking section of the program: three costatements inside an endless `while` loop. The benefit of using the non-blocking mode of the FAT file system is realized when using costatements, an extension of Dynamic C that implements cooperative multitasking. Instead of waiting while a function finishes its execution, the application can accomplish other tasks.

3.2.1 Costatement that Writes a File

The first costate is named `putdata`. It waits for one second and then creates a string to timestamp the entry of a randomly generated number that is then appended to a file.

```
while (1){
   costate putdata always_on
   {
      waitfor (DelaySec(1));        // Wait for one second to elapse
```

Note that the `always_on` keyword is used. This is required when using a named costatement to force it to execute every time it is encountered in the execution thread (unless it is made inactive by a call to `CoPause()`).

It is easy to suspend execution within a costate by using the `waitfor` keyword. The costate will relinquish control if the argument to `waitfor` (in this case a call to `DelaySec()`) evaluates to `FALSE`. The next time the execution thread reaches `putdata`, `waitfor` will be called again. This will go on until `DelaySec()` returns `TRUE`, i.e., when one second has elapsed from the time `DelaySec()` was first called from within `waitfor`.

After the one second delay, the string to write to the file is placed in a buffer and a looping variable and position pointer are initialized.

```
sprintf(obuf, "%02d:%02d:%02d -- %6.3f \n", h, m, s, (25.0 * rand()));
ocount = 0;
optr = obuf;
```

Before the buffer contents can be written to a file in the FAT file system, we must ensure that no collisions occur since there is another `costate` that will attempt to read the file every ten seconds. A file can not be read from and written to at the same time. In the following code the `waitfor` keyword is used with the global variable `filestate` (defined at the top of the application) to implement a locking mechanism. As soon as the file becomes available for `putdata`, it is marked unavailable for `showdata`.

```
waitfor (filestate == 0);         // Wait until file is available
filestate = 1;                    // Show file is being updated
```

The next block of code appends the latest entry into the file that was opened at the start of the application.

```
   while (ocount < REC_LEN){     // Loop until entire record is written
      waitfor((rc = fat_Write(&file, optr, REC_LEN - ocount))!= -EBUSY);
      if (rc < 0){
         printf("fat_Write: rc = %d\n",rc);
         while ((rc = fat_UnmountDevice(first_part->dev)) == -EBUSY);
         return rc;
      }
      optr += rc;                 // Move output pointer
      ocount += rc;               // Add number of characters written
   }
   filestate = 0;                 // Show file is idle
}
```

Again, `waitfor` is used to voluntarily relinquish control, this time while waiting for the write function to complete. If an error occurs during the write operation the device is unmounted and the application exits. Otherwise the loop counter and the buffer position pointer are advanced by the number of bytes actually written. Since this can be less than the requested number of bytes, it is best to check in a loop such as the `while` loop shown in `putdata`.

The last action taken by `putdata` is to reset `filestate`, indicating that the open file is available.

3.2.2 Costatement that Reads and Displays a File

The costatement named `showdata` waits for ten seconds. Then it waits for the open file to be available, and when it is, immediately marks it as unavailable.

```
costate showdata always_on{
   waitfor (DelaySec(10));
   waitfor (filestate == 0);
   filestate = 2;
```

The next statement modifies the internal file position pointer. The first time this costate runs, `readto` is zero, meaning the position pointer is at the first byte of the file. The variable `readto` is incremented every time a record is read from the file, allowing `showdata` to always know where to seek to next.

```
   waitfor (fat_Seek(&file, readto, SEEK_SET) != -EBUSY);
```

The rest of `showdata` is a `while` loop inside of a `while` loop. The inner `while` loop is where each record is read from the file into the buffer and then displayed in the Stdio window with the `printf()` call. Since `fat_Read()` may return less than the requested number of bytes, the `while` loop is needed to make sure that the function will be called repeatedly until all bytes have been read. When the full record has been read, it will then be displayed to the Stdio window.

The outer `while` loop controls when to stop reading records from the file. After the last record is read, the `fat_Read()` function is called once more, returning an end-of-file error. This causes the `if` statements that are checking for this error to return `TRUE`, which resets `filestate` to zero, breaking out of the outer `while` loop and freeing the lock for the `putdata` costatement to use.

```
while (filestate){
   icount = 0;
   iptr = ibuf;
   while (icount < REC_LEN) {
      waitfor((rc = fat_Read(&file, iptr, REC_LEN-icount)) != -EBUSY);
      if (rc < 0)
      {
         if (rc == -EEOF)
         {
            filestate = 0;
            break;
         }
         printf("fat_Read: rc = %d\n",rc);
         while ((rc=fat_UnmountDevice(first_part->dev)) == -EBUSY);
         return rc;
      }
      iptr += rc;
      icount += rc;
   }                                    // end of inner while loop
   if (filestate)
   {
      printf("%s", ibuf);
      readto += REC_LEN;
   }
}                                       // end of outer while loop
```

The other costatement in the endless `while` loop is the one that blinks the LED. It illustrates that while using the file system in non-blocking mode, there is still plenty of time for other tasks.

4. FAT Operations

There are some basic groups of operations involved in using the Dynamic C FAT library. These are described at length in the following sections.

Section 4.1 "Format and Partition the Device"

- Default Partitioning
- Creating Multiple FAT Partitions
- Preserving Existing Partitions

Section 4.2 "File and Directory Operations"

- Open and Close Operations
- Read and Write Operations
- Going to a Specified Position in a File
- Creating Files and Subdirectories
- Reading Directories
- Deleting Files and Directories

4.1 Format and Partition the Device

The flash device must be formatted before its first use. Formatting it after its first use may destroy information previously placed on it.

4.1.1 Default Partitioning

As a convenience, `Samples/FileSystem/Fmt_Device.c` is provided to format the flash device. This program can format individual FAT 12/16 partitions, or can format all FAT 12/16 partitions found on a device. If no FAT 12/16 partitions are found, it offers the option of erasing the entire device and formatting it with a single FAT 16 partition. Be aware that this will destroy any data on the device, including that contained on FAT 32 partitions. This is an easy way to format new media that may contain an empty FAT32 partition spanning the entire device, such as a new SD or XD card.

After the device has been formatted with `Fmt_Device.c`, an application that wants to use the FAT file system just has to call the function `fat_Init()` (replaced in FAT version 2.01) or `fat_AutoMount()`. If you are calling `fat_AutoMount()` refer to Section 2.1 for an example of its use. Note that if you call `fat_AutoMount()` using the configuration flag `FDDF_DEV_FORMAT`, you may not need to run `Fmt_Device.c`.

4.1.2 Creating Multiple Partitions

To create multiple partitions on the flash device use the sample program `FAT_Write_MBR.c`, which will allow you to easily create as many as four partitions. This program does require that the device be "erased" before being run. This can be done with the appropriate sample program: `sdflash_inspect.c`, `sflash_inspect.c` or `nflash_inspect.c`. You only need to clear the first three pages on SD cards or serial flash, or the first page on NAND flash or XD cards. Once this is done, run FAT_Write_MBR and it will display the total size of the device in MegaBytes and allow you to specify the size of each partition until all the space is used. If you specify an amount larger than the space remaining, then all remaining space will be used for that partition. Once all space is specified, it will ask approval to write the new partition structure. This utility does not format the partitions, it merely creates their defini-

tions. Run Fmt_device.c afterwards and use the 0 or 1 option to format the full device and all partitions will be formatted. Be forewarned that on removable media, using multiple partitions will typically make the device unusable with PC readers.

The sample program FAT_Write_MBR.c is distributed with FAT module version 2.13. It is also compatible with FAT versions 2.01, 2.05 and 2.10. If you have one of these earlier versions of the FAT module and would like a copy of FAT_Write_MBR.c, please contact Technical Support either by email to support@rabbitsemiconductor.com or using the online form available on the Rabbit website: www.rabbitsemiconductor.com/support/questionSubmit.shtml.

There is a way to create multiple partitions without using the utility FAT_Write_MBR.c; this auxiliary method is explained in A.3.5.

4.1.3 Preserving Existing Partitions

If the flash device already has a valid partition that you want to keep, you must know where it is so you can fit the FAT partition onto the device. This requires searching the partition table for both available partitions and available space. An available partition has the partsecsize field of its mbr_part entry equal to zero.

Look in lib/.../RCM3300/RemoteApplicationUpdate/downloadmanager.lib for the function dlm_initserialflash() for an example of searching through the partition table for available partitions and space. See the next section for more information on the download manager (DLM) and how to set up coexisting partitions.

4.1.4 FAT and DLM Partitions

The RabbitCore RCM3300 comes with a download manager utility that creates a partition on a serial flash device, which is then used by the utility to remotely update an application. You can set up a device to have both a DLM partition and a FAT partition.

Run the program Samples/RCM3300/RemoteApplicationUpdate/DLM_FAT_FORMAT.C. This program must be run on an unformatted serial flash, i.e., a flash with no MBR. To remove an existing MBR, first run the program Samples/RCM3300/SerialFlash/SFLASH_INSPECT.C to clear the first three pages.

The program DLM_FAT_FORMAT.C will set aside space for the DLM partition and use the rest of the device to create a FAT partition. Then, when you run the DLM software, it will be able to find space for its partition and will coexist with the FAT partition. This shows the advantage to partitions: Partitions set hard boundaries on the allocation of space on a device, thus neither FAT nor the DLM software can take space from the other.

4.2 File and Directory Operations

The Dynamic C FAT implementation supports the basic set of file and directory operations. Remember that a partition must be mounted before it can be used with any of the file, directory or status operations.

4.2.1 Open and Close Operations

The `fat_Open()` function opens a file or a directory. It can also be used to create a file or a directory. When using the non-blocking FAT, check the return code and call it again with the same arguments until it returns something other than `-EBUSY`..

```
rc = fat_Open(my_part, "DIR\\FILE.TXT", FAT_FILE, FAT_CREATE,
               &my_file, &prealloc);
```

The first parameter, `my_part`, points to a partition structure. This pointer must point to a mounted partition. Some of the sample programs, like `fat_create.c`, declare a local pointer and then search for a partition pointer in the global array `fat_part_mounted[]`. Other sample programs, like `fat_shell.c`, define an integer to be used as an index into `fat_part_mounted[]`. Both methods accomplish the same goal of gaining access to a partition pointer.

The second parameter contains the file name, including the directory (if applicable) relative to the root directory. All paths in Dynamic C must specify the full directory path explicitly, e.g., `DIR1\\FILE.EXT` or `DIR1/FILE.EXT`. The direction of the slash in the pathname is a backslash by default. If you use the default backslash for the path separator, you must always precede it with another backslash, as shown in the above call to `fat_Open()`. This is because the backslash is an escape character in a Dynamic C string. To use the forward slash as the path separator, define the macro `FAT_USE_FORWARDSLASH` in your application (or in `FAT.LIB` to make it the system default).

The third parameter determines whether a file or directory is opened (`FAT_FILE` or `FAT_DIR`).

The fourth parameter is a flag that limits `fat_Open()` to the action specified. `FAT_CREATE` creates the file (or directory) if it does not exist. If the file does exist, it will be opened, and the position pointer will be set to the start of the file. If you write to the file without moving the position pointer, you will overwrite existing data. Use `FAT_MUST_CREATE` if you know the file does not exist; this last option is also a fail-safe way to avoid opening and overwriting an existing file since an `-EEXIST` error message will be returned if you attempt to create a file that already exists.

The fifth parameter, `&my_file`, is an available file handle. After a file or directory is opened, its handle is used to identify it when using other API functions, so be wary of using local variables as your file handle.

The final parameter is an initial byte count if the object needs to be created. It is only used if the `FAT_CREATE` or `FAT_MUST_CREATE` flag is used and the file or directory does not already exist. The byte count is rounded up to the nearest whole number of clusters greater than or equal to 1. On return, the variable `prealloc` is updated to the number of bytes allocated. Pre-allocation is used to set aside space for a file, or to speed up writing a large amount of data as the space allocation is handled once.

Pass NULL as the final parameter to indicate that you are opening the file for reading or that a minimum number of bytes needs to be allocated to the file at this time. If the file does not exist and you pass NULL, the file will be created with the minimum one cluster allocation.

Once you are finished with the file, you must close it to release its handle so that it can be reused the next time a file is created or opened.

```
rc = fat_Close(&my_file);
```

Remember to check the return code from `fat_Close()` since an error return code may indicate the loss of data. Once you are completely finished, call `fat_UnmountDevice()` to make sure any data stored in the cache is written to the flash device.

4.2.2 Read and Write Operations

Use `fat_Read()` to read a file.

```
rc = fat_Read(&my_file, buf, sizeof(buf));
```

The first parameter, `&my_file`, is a pointer to the file handle already opened by `fat_Open()`. The parameter `buf` points to a buffer for reading the file. The `sizeof(buf)` parameter is the number of bytes to be read into the buffer. It does not have to be the full size of the buffer. If the file contains fewer than `sizeof(buf)` characters from the current position to the end-of-file marker (EOF), the transfer will stop at the EOF. If the file position is already at the EOF, 0 is returned. The maximum number of characters read is 32767 bytes per call.

The function returns the number of characters read or an error code. Characters are read beginning at the current position of the file. If you have just written to the file that is being read, the file position pointer will be where the write left off. If this is the end of the file and you want to read from the beginning of the file you must change the file position pointer. This can be done by closing the file and reopening it, thus moving the position pointer to the start of the file. Another way to change the position pointer is to use the `fat_Seek()` function. This function is explained in Section 4.2.3.

Use `fat_ReadDir()` to read a directory. This function is explained in Section 4.2.5.

Use `fat_Write()` or `fat_xWrite()` to write to a file. The difference between the two functions is that `fat_xWrite()` copies characters from a string stored in extended memory.

```
rc = fat_Write(&my_file, "Write data\r\n", 12);
```

The first parameter, `&my_file`, is a pointer to the file handle already opened by `fat_Open()`. Because `fat_Open()` sets the position pointer to the start of the file, you will overwrite any data already in the file. You will need to call `fat_Seek()` if you want to start the write at a position other than the start of the file (see Section 4.2.3).

The second parameter contains the data to write to the file. Note that `\r\n` (carriage return, line feed) appear at the end of the string in the function. This is essentially a FAT (or really, DOS) convention for text files. It is good practice to use these standard line-end conventions. (If you only use `\n`, the file will read just fine on Unix systems, but some DOS-based programs may have difficulties.) The third parameter specifies the number of characters to write. Select this number with care since a value that is too small will result in your data being truncated, and a value that is too large will append any data that already exists beyond your new data.

Remember that once you are finished with a file you must close it to release its handle. You can call the `fat_Close()` function, or, if you are finished using the file system on a particular partition, call `fat_UnmountPartition()`, which will close any open files and then unmount the partition. If you

are finished using the device, it is best to call fat_UnmountDevice(), which will close any open FAT files on the device and unmount all mounted FAT partitions. Unmounting the device is the safest method for shutting down after using the device.

4.2.3 Going to a Specified Position in a File

The position pointer is at the start of the file when it is first opened. Two API functions, fat_Tell() and fat_Seek(), are available to help you with the position pointer.

```
fat_Tell(&my_file, &pos);
fat_Seek(&my_file, pos, SEEK_SET);
```

The fat_Tell() function does not change the position pointer, but reads its value (which is the number of bytes from the beginning of the file) into the variable pointed to by &pos. Zero indicates that the position pointer is at the start of the file. The first parameter, &my_file, is the file handle already opened by fat_Open().

The fat_Seek() function changes the position pointer. Clusters are allocated to the file if necessary, but the position pointer will not go beyond the original end of file (EOF) unless doing a SEEK_RAW. In all other cases, extending the pointer past the original EOF will preallocate the space that would be needed to position the pointer as requested, but the pointer will be left at the original EOF and the file length will not be changed. If this occurs, the error code -EEOF is returned to indicate the space was allocated but the pointer was left at the EOF. If the position requires allocating more space than is available on the device, the error code -ENOSPC is returned.

The first parameter passed to fat_Seek() is the file handle that was passed to fat_Open(). The second parameter, pos, is a long integer that may be positive or negative. It is interpreted according to the value of the third parameter. The third parameter must be one of the following:

- SEEK_SET - pos is the byte position to seek, where 0 is the first byte of the file. If pos is less than 0, the position pointer is set to 0 and no error code is returned. If pos is greater than the length of the file, the position pointer is set to EOF and error code -EEOF is returned.
- SEEK_CUR - seek pos bytes from the current position. If pos is less than 0 the seek is towards the start of the file. If this goes past the start of the file, the position pointer is set to 0 and no error code is returned. If pos is greater than 0 the seek is towards EOF. If this goes past EOF the position pointer is set to EOF and error code -EEOF is returned.
- SEEK_END - seek to pos bytes from the end of the file. That is, for a file that is x bytes long, the statement:

  ```
  fat_Seek (&my_file, -1, SEEK_END);
  ```

  will cause the position pointer to be set at x-1 no matter its value prior to the seek call. If the value of pos would move the position pointer past the start of the file, the position pointer is set to 0 (the start of the file) and no error code is returned. If pos is greater than or equal to 0, the position pointer is set to EOF and error code -EEOF is returned.
- SEEK_RAW - is similar to SEEK_SET, but if pos goes beyond EOF, using SEEK_RAW will set the file length and the position pointer to pos. This adds whatever data exists on the allocated space onto the end of the file..

4.2.4 Creating Files and Subdirectories

While the `fat_Open()` function is versatile enough to not only open a file but also create a file or a subdirectory, there are API functions specific to the tasks of creating files and subdirectories.

The `fat_CreateDir()` function is used to create a subdirectory one level at a time.

```
rc = fat_CreateDir(my_part, "DIR1");
```

The first parameter, `my_part`, points to a partition structure. This pointer must point to a mounted partition. Some of the sample programs, like `fat_create.c`, declare a local pointer and then search for a partition pointer in the global array `fat_part_mounted[]`. Other sample programs, like `fat_shell.c`, define an integer to be used as an index into `fat_part_mounted[]`. Both methods accomplish the same goal of gaining access to a partition pointer.

The second parameter contains the directory or subdirectory name relative to the root directory. If you are creating a subdirectory, the parent directory must already exist.

Once `DIR1` is created as the parent directory, a subdirectory may be created, and so on.

```
rc = fat_CreateDir(my_part, "DIR1/SUBDIR");
```

Note that a forward slash is used in the pathname instead of a backslash. Either convention may be used. The backslash is used by default. To use a forward slash instead, define `FAT_USE_FORWARDSLASH` in your application or in `FAT.LIB`.

A file can be created using the `fat_CreateFile()` function. All directories in the path must already exist.

```
rc = fat_CreateFile(my_part, "DIR1/SUBDIR/FILE.TXT", &prealloc,
   &my_file);
```

The first parameter, `my_part`, points to the static partition structure set up by `fat_AutoMount()`.

The second parameter contains the file name, including the directories (if applicable) relative to the root directory. All paths in the FAT library are specified relative to the root directory.

The third parameter indicates the initial number of bytes to pre-allocate. At least one cluster will be allocated. If there is not enough space beyond the first cluster for the requested allocation amount, the file will be allocated with whatever space is available on the partition, but no error code will be returned. If no clusters can be allocated, the `-ENOSPC` error code will return. Use NULL to indicate that no bytes need to be allocated for the file at this time. Remember that pre-allocating more than the minimum number of bytes necessary for storage will reduce the available space on the device.

The final parameter, `&my_file`, is a file handle that points to an available file structure. If NULL is entered, the file will be closed after it is created.

4.2.5 Reading Directories

The `fat_ReadDir()` function reads the next directory entry from the specified directory. A directory entry can be a file, directory or a label. A directory is treated just like a file.

```
fat_ReadDir(&dir, &dirent, mode);
```

The first parameter specifies the directory; `&dir` is an open file handle. A directory is opened by a call to `fat_OpenDir()` or by passing `FAT_DIR` in a call to `fat_Open()`. The second parameter, `&dirent`, is a pointer to a directory entry structure to fill in. The directory entry structure must be declared in your application, for example:

```
fat_dirent dirent;
```

Search Conditions

The last parameter, `mode`, determines which directory entry is being requested, a choice that is built from a combination of the macros described below. To understand the possible values for `mode`, the first thing to know is that a directory entry can be in one of three states: empty, active or deleted. This means you must choose one of the default flags described below, or one or more of the following macros:

- `FAT_INC_ACTIVE` - include active entries. This is the default setting if other `FAT_INC_*` macros are not specified; i.e., active files are included unless `FAT_INC_DELETED`, `FAT_INC_EMPTY`, or `FAT_INC_LNAME` is set.
- `FAT_INC_DELETED` - include deleted entries
- `FAT_INC_EMPTY` - include empty entries
- `FAT_INC_LNAME` - include long name entries (this is included for completeness, but is not used since long file names are not supported)

The above macros narrow the search to only those directory entries in the requested state. The search is then refined further by identifying particular attributes of the requested entry. This is done by choosing one or more of the following macros:

- `FATATTR_READ_ONLY` - include read-only entries
- `FATATTR_HIDDEN` - include hidden entries
- `FATATTR_SYSTEM` - include system entries
- `FATATTR_VOLUME_ID` - include label entries
- `FATATTR_DIRECTORY` - include directory entries
- `FATATTR_ARCHIVE` - include modified entries

Including a `FATATTR_*` macro means you do not care whether the corresponding attribute is turned on or off. Not including a `FATATTR_*` macro means you only want an entry with that particular attribute turned off. Note that the FAT system sets the archive bit on all new files as well as those written to, so including `FATATTR_ARCHIVE` in your mode setting is a good idea.

For example, if mode is (`FAT_INC_ACTIVE`) then the next directory entry that has all of its attributes turned off will be selected; i.e., an entry that is not read only, not hidden, not a system file, not a directory or a label, and not archived. In other words, the next writable file that is not hidden, system or already archived is selected.

But, if you want the next active file and do not care about the file's other attributes, mode should be (FAT_INC_ACTIVE | FATATTR_READ_ONLY | FATATTR_HIDDEN | FATATTR_SYSTEM | FATATTR_ARCHIVE). This search would only exclude directory and label entries.

Now suppose you want only the next active read-only file, leaving out hidden or system files. The next group of macros allows this search by filtering on whether the requested attribute is set. The filter macros are:

- FAT_FIL_RD_ONLY - filter on read-only attribute
- FAT_FIL_HIDDEN - filter on hidden attribute
- FAT_FIL_SYSTEM - filter on system attribute
- FAT_FIL_LABEL - filter on label attribute
- FAT_FIL_DIR - filter on directory attribute
- FAT_FIL_ARCHIVE - filter on modified attribute

If you set mode to (FAT_INC_ACTIVE | FATATTR_READ_ONLY | FAT_FIL_RD_ONLY | FATATTR_ARCHIVE), the result will be the next active file that has its read-only attribute set (and has the archive attribute in either state).

> **Note:** If you have FAT module version 2.05 or earlier, you do not have access to the FAT_FIL_* macros.

Default Search Flags

To make things easier, there are two predefined mode flags. Each one may be used alone or in combination with the macros already described.

- FAT_INC_ALL - selects any directory entry of any type.
- FAT_INC_DEF - selects the next active file or directory entry, including read-only or archived files. No hidden, system, label, deleted, or empty directories or files will be selected. This is typically what you see when you do a directory listing on your PC.

Search Flag Examples

Here are some more examples of how the flags work.

1. If you want the next hidden file or directory:

   Start with the FAT_INC_DEF macro default flag. This flag does not allow hidden files, so we need FATATTR_HIDDEN. Then to narrow the search to consider only a hidden file or directory, we need the macro FAT_FIL_HIDDEN to filter on files or directories that have the hidden attribute set. That is, mode is set to:

   FAT_INC_DEF | FATATTR_HIDDEN | FAT_FIL_HIDDEN

2. If you want the next hidden directory:

   Again, start with the FAT_INC_DEF macro default flag. To narrow the search to directories only, we want entries with their directory attribute set; therefore, OR the macros FATATTR_DIRECTORY and FAT_FIL_DIR. Then OR the macros FATATTR_HIDDEN and FAT_FIL_HIDDEN to search only for directories with their hidden attribute set. That is, set mode to:

   FAT_INC_DEF | FATATTR_DIRECTORY | FAT_FIL_DIR | FATATTR_HIDDEN | FAT_FIL_HIDDEN

3. If you want the next hidden file (no directories):

   Start with the predefined flag, `FAT_INC_DEF`. This flag allows directories, which we do not want, so we do an AND NOT of the `FATATTR_DIRECTORY` macro.

   Next we want to narrow the search to only entries that have their hidden attribute set. The default flag does not allow hidden flags, so we need to OR the macros `FATTR_HIDDEN` and `FAT_FIL_HIDDEN`.

   That is, set `mode` to:

   `FAT_INC_DEF & ~FATATTR_DIRECTORY | FATATTR_HIDDEN | FAT_FIL_HIDDEN`

4. If you want the next non-hidden file (no directories):

   First, select the `FAT_INC_DEF` filter default flag. This flag allows directories, which we do not want, so we do an AND NOT of the `FATATTR_DIRECTORY` macro. The default flag already does not allow hidden files, so we are done. That is, set `mode` to:

   `FAT_INC_DEF & ~FATATTR_DIRECTORY`

5. Finally let's see how to get the next non-empty entry of any type.

   Start with the predefined flag, `FAT_INC_ALL`. This flag selects any directory entry of any type. Since we do not want empty entries, we have to remove that search condition from the flag, so we do an AND NOT for the `FAT_INC_EMPTY` macro to filter out the empty entries. That means `mode` is the bitwise combination of the macros:

   `mode = FAT_INC_ALL & ~FAT_INC_EMPTY`

4.2.6 Deleting Files and Directories

The `fat_Delete()` function is used to delete a file or directory. The second parameter sets whether a file or directory is being deleted. Only one file or directory may be deleted at any one time—this means that you must call `fat_Delete()` at least twice to delete a file and its associated directory (if the directory has no other files or subdirectories since a directory must be empty to be deleted).

```
fat_Delete(my_part, FAT_FILE, "DIR/FILE.TXT");
```

The first parameter, `my_part`, points to the static partition structure that was populated by `fat_AutoMount()`. The second parameter is the file type, `FAT_FILE` or `FAT_DIR`, depending on whether a file or a directory is to be deleted. The third parameter contains the file name, including the directory (if applicable) relative to the directory root. All paths in the FAT library are specified relative to the root directory.

4.3 Error Handling

Most routines in the FAT library return an int value error code indicating the status of the requested operation. Table 2 contains a list of error codes specific to the FAT file system. Most of these codes, along with some other error codes, are defined in `/Lib/../ERRNO.LIB`.

Table 2. FAT-Specific Error Codes

Code	Value	Description
EOF	231	End of File Encountered
EEOF	41	End-of-file marker reached
ETYPE	232	Incorrect Type
EPATHSTR	233	Invalid Path String
EROOTFULL	234	Root Directory is Full
EUNFORMAT	235	Unformatted Volume
EBADPART	236	Invalid Partition
ENOPART	237	Unpartitioned / Unformatted Media
ENOTEMPTY	238	Open Files in Partition / Directory to be Deleted
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
EIO	5	I/O error
EBUSY	16	Device or resource busy
EEXIST	17	File exists
ENODEV	19	No such device
ENOSPC	28	No space left on device
ENOTEMPTY	39	Directory is not empty
ENOMEDIUM	123	No medium found

5. FAT API Functions

The FAT API functions are described in this section. The table below groups the functions by category and provides links to the function descriptions.

Function Category	Function Names	
Device and partition operations	fat_AutoMount fat_Init fat_EnumDevice fat_FormatDevice fat_UnmountDevice	fat_EnumPartition fat_FormatPartition fat_PartitionDevice fat_MountPartition fat_UnmountPartition
File operations	fat_Close fat_CreateFile fat_Delete fat_Open fat_Read	fat_Seek fat_Split fat_Truncate fat_Write fat_xWrite
Directory operations	fat_CreateDir fat_OpenDir	fat_ReadDir
Status operations	fat_Free fat_FileSize fat_GetAttr fat_GetName	fat_SetAttr fat_Status fat_Tell nf_XD_Detect
Administrative	fat_CreateTime fat_InitUCOSMutex fat_LastAccess fat_LastWrite	fat_SyncFile fat_SyncPartition fat_tick

fat_AutoMount

```
int fat_AutoMount( word flags );
```

DESCRIPTION

Initializes the drivers in the default drivers configuration list in `fat_config.lib` and enumerates the devices in the default devices configuration list, then mounts partitions on enumerated devices according to the device's default configuration flags, unless overridden by the specified run time configuration flags. Despite its lengthy description, this function makes initializing multiple devices using the FAT library as easy as possible. The first driver in the configuration list becomes the primary driver in the system, if one is not already set up.

After this routine successfully returns, the application can start calling directory and file functions for the devices' mounted partitions.

If devices and/or partitions are not already formatted, this function can optionally format them according to the device's configuration or run time override flags.

This function may be called multiple times, but will not attempt to (re)mount device partitions that it has already mounted. Once a device partition has been mounted by this function, unmounts and remounts must be handled by the application.

There are two arrays of data structures that are populated by calling `fat_AutoMount()`. The array named `fat_part_mounted[]` is an array of pointers to `fat_part` structures. A `fat_part` structure holds information about a specific FAT partition. The other array, `_fat_device_table[]`, is composed of pointers to `mbr_dev` structures. An `mbr_dev` structure holds information about a specific device. Partition and device structures are needed in many FAT function calls to specify the device and partition to be used.

An example of using `fat_part_mounted[]` was shown in the sample program `fat_create.c`. FAT applications will need to scan `fat_part_mounted[]` to locate valid FAT partitions. A valid FAT partition must be identified before any file and directory operations can be performed. These pointers to FAT partitions may be used directly by indexing into the array or stored in a local pointer. The `fat_shell.c` sample uses an index into the array, whereas most other sample programs make a copy of the pointer.

An example of using `_fat_device_table[]` is in the sample program `fat_shell.c`. This array is used in FAT operations of a lower level than `fat_part_mounted[]`. Specifically, when the device is being partitioned, formatted and/or enumerated. Calling `fat_AutoMount()` relieves most applications of the need to directly use `fat_device_table[]`.

PARAMETERS

flags

Run-time device configuration flags to allow overriding the default device configuration flags. If not overriding the default configuration flags, specify FDDF_USE_DEFAULT. To override the default flags, specify the ORed combination of one or more of the following:

- FDDF_MOUNT_PART_0: Mount specified partition
- FDDF_MOUNT_PART_1:
- FDDF_MOUNT_PART_2:
- FDDF_MOUNT_PART_3:
- FDDF_MOUNT_PART_ALL: Mount all partitions
- FDDF_MOUNT_DEV_0: Apply to specified device
- FDDF_MOUNT_DEV_1:
- FDDF_MOUNT_DEV_2:
- FDDF_MOUNT_DEV_3:
- FDDF_MOUNT_DEV_ALL: Apply to all available devices
- FDDF_NO_RECOVERY: Use norecovery if fails first time
- FDDF_COND_DEV_FORMAT: Format device if unformatted
- FDDF_COND_PART_FORMAT: Format partition if unformatted
- FDDF_UNCOND_DEV_FORMAT: Format device unconditionally
- FDDF_UNCOND_PART_FORMAT: Format partition unconditionally

Note: The FDDF_MOUNT_PART_* flags apply equally to all FDDF_MOUNT_DEV_* devices which are specified. If this is a problem, call this function multiple times with a single DEV flag bit each time.

Note: Formatting the device creates a single FAT partition covering the entire device. It is recommended that you always set the *_PART_FORMAT flag bit if you set the corresponding *_DEV_FORMAT flag bit.

RETURN VALUE

0: success
-EBADPART: partition is not a valid FAT partition
-EIO: Device I/O error
-EINVAL: invalid prtTable
-EUNFORMAT: device is not formatted
-ENOPART: no partitions exist on the device
-EBUSY: For non-blocking mode only, the device is busy. Call this function again to complete the close.

Any other negative value means that an I/O error occurred when updating the directory entry. In this case, the file is forced to close, but its recorded length might not be valid.

LIBRARY

FAT.LIB

SEE ALSO

fat_EnumDevice, fat_EnumPartition, fat_MountPartition

fat_Close

```
fat_Close( FATfile *file );
```

DESCRIPTION

Closes a currently open file. You should check the return code since an I/O needs to be performed when closing a file to update the file's EOF offset (length), last access date, attributes and last write date (if modified) in the directory entry. This is particularly critical when using non-blocking mode.

PARAMETERS

file	Pointer to the open file to close.

RETURN VALUE

0: success.
-EINVAL: invalid file handle.
-EBUSY: For non-blocking mode only, the device is busy. Call this function again to complete the close.

Any other negative value means that an I/O error occurred when updating the directory entry. In this case, the file is forced to close, but its recorded length might not be valid.

LIBRARY

FAT.LIB

SEE ALSO

fat_Open, fat_OpenDir

fat_CreateDir

```
fat_CreateDir( fat_part *part, char *dirname );
```

DESCRIPTION

Creates a directory if it does not already exist. The parent directory must already exist.

In non-blocking mode, only one file or directory can be created at any one time, since a single static `FATfile` is used for temporary storage. Each time you call this function, pass the same `dirname` pointer (not just the same string contents).

PARAMETERS

part	Handle for the partition being used.
dirname	Pointer to the full path name of the directory to be created.

RETURN VALUE

`0`: success.
`-EINVAL`: invalid argument. Trying to create volume label.
`-ENOENT`: parent directory does not exist.
`-EPERM`: the directory already exists or is write-protected.
`-EBUSY`: the device is busy (only if non-blocking).
`-EFSTATE`: if non-blocking, but a previous sequence of calls to this function (or `fat_CreateFile()`) has not completed and you are trying to create a different file or directory. You must complete the sequence of calls for each file or directory i.e., keep calling until something other than `-EBUSY` is returned.

Other negative values are possible from `fat_Open()`/`fat_Close()` calls.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_ReadDir`, `fat_Status`, `fat_Open`, `fat_CreateFile`

fat_CreateFile

```
int fat_CreateFile( fat_part *part, char *filename, long alloc_size,
    FATfile *file );
```

DESCRIPTION

Creates a file if it does not already exist. The parent directory must already exist.

In non-blocking mode, if `file` is NULL, only one file or directory can be created at any one time, since a single static `FATfile` is used for temporary storage. Each time you call this function, pass the same `dirname` pointer (not just the same string contents).

PARAMETERS

part	Pointer to the partition being used.
filename	Pointer to the full path name of the file to be created.
alloc_size	Initial number of bytes to pre-allocate. Note that at least one cluster will be allocated. If there is not enough space beyond the first cluster for the requested allocation amount, the file will be allocated with whatever space is available on the partition, but no error code will be returned. If not even the first cluster is allocated, the `-ENOSPC` error code will return. This initial allocation amount is rounded up to the next whole number of clusters.
file	If not NULL, the created file is opened and accessible using this handle. If NULL, the file is closed after it is created.

RETURN VALUE

`0`: success.
`-EINVAL`: `part`, `filename`, `alloc_size`, or `file` contain invalid values.
`-ENOENT`: the parent directory does not exist.
`-ENOSPC`: no allocatable sectors were found.
`-EPERM`: write-protected, trying to create a file on a read-only partition.
`-EBUSY`: the device is busy (non-blocking mode only).
`-EFSTATE`: if non-blocking, but a previous sequence of calls to this function (of fat_CreateFile) has not completed but you are trying to create a different file or directory. You must complete the sequence of calls for each file or directory i.e. keep calling until something other than `-EBUSY` is returned. This code is only returned if you pass a NULL file pointer, or if the file pointer is not NULL and the referenced file is already open.

Other negative values indicate I/O error, etc.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_Open`, `fat_ReadDir`, `fat_Write`

fat_CreateTime

```
fat_CreateTime( fat_dirent *entry, struct tm *t );
```

DESCRIPTION

This function puts the creation date and time of the entry into the system time structure `t`. The function does not fill in the `tm_wday` field in the system time structure.

PARAMETERS

`entry`	Pointer to a directory entry
`t`	Pointer to a system time structure

RETURN VALUE

0: success.
-EINVAL: invalid directory entry or time pointer

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_ReadDir`, `fat_Status`, `fat_LastAccess`, `fat_LastWrite`

fat_Delete

```
fat_Delete( fat_part *part, int type, char *name );
```

DESCRIPTION

Deletes the specified file or directory. The `type` must match or the deletion will not occur. This routine inserts a deletion code into the directory entry and marks the sectors as available in the FAT table, but does not actually destroy the data contained in the sectors. This allows an undelete function to be implemented, but such a routine is not part of this library. A directory must be empty to be deleted.

PARAMETERS

part	Handle for the partition being used.
type	Must be a FAT file (`FAT_FILE`) or a FAT directory (`FAT_DIR`), depending on what is to be deleted.
name	Pointer to the full path name of the file/directory to be deleted.

RETURN VALUE

0: success.
`-EIO`: device I/O error.
`-EINVAL`: `part`, `type`, or `name` contain invalid values.
`-EPATHSTR`: `name` is not a valid path/name string.
`-EPERM`: the file is open, write-protected, hidden, or system.
`-ENOTEMPTY`: the directory is not empty.
`-ENOENT`: the file/directory does not exist.
`-EBUSY`: the device is busy. (Only if non-blocking.)
`-EPSTATE`: if the partition is busy; i.e., there is an allocation in progress. (Only if non-blocking.)

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_Open`, `fat_OpenDir`, `fat_Split`, `fat_Truncate`, `fat_Close`

fat_EnumDevice

```
fat_EnumDevice(mbr_drvr *driver, mbr_dev *dev, int devnum, char *sig,
    int norecovery);
```

DESCRIPTION

This routine is called to learn about the devices present on the driver passed in. The device will be added to the linked list of enumerated devices. Partition pointers will be set to NULL, indicating they have not been enumerated yet. Partition entries must be enumerated separately.

The signature string is an identifier given to the write-back cache, and must remain consistent between resets so that the device can be associated properly with any battery-backed cache entries remaining in memory.

This function is called by `fat_AutoMount()` and `fat_Init()`.

PARAMETERS

driver	Pointer to an initialized driver structure set up during the initialization of the storage device driver.
dev	Pointer to the device structure to be filled in.
devnum	Physical device number of the device.
sig	Pointer to a unique signature string. Note that this value **must** remain the same between resets.
norecovery	Boolean flag - set to True to ignore power-recovery data. True is any value except zero.

RETURN VALUE

`0`: success.
`-EIO`: error trying to read the device or structure.
`-EINVAL`: `devnum` invalid or does not exist.
`-ENOMEM`: memory for page buffer/RJ is not available.
`-EUNFORMAT`: the device is accessible, but not formatted. You may use it provided it is formatted/partitioned by either this library or by another system.
`-EBADPART`: the partition table on the device is invalid.
`-ENOPART`: the device does not have any FAT partitions. This code is superseded by any other error detected.
`-EEXIST`: the device has already been enumerated.
`-EBUSY`: the device is busy (nonblocking mode only).

LIBRARY

FAT.LIB

SEE ALSO

fat_AutoMount, fat_Init, fat_EnumPartition

fat_EnumPartition

```
fat_EnumPartition(mbr_dev *dev, int pnum, fat_part *part);
```

DESCRIPTION

This routine is called to enumerate a partition on the given device. The partition information will be put into the FAT partition structure pointed to by `part`. The partition pointer will be linked to the device structure, registered with the write-back cache, and will then be active. The partition must be of a valid FAT type.

This function is called by `fat_AutoMount()` and `fat_Init()`.

PARAMETERS

dev	Pointer to an MBR device structure.
pnum	Partition number to link and enumerate.
part	Pointer to an FAT partition structure to be filled in.

RETURN VALUE

`0`: success.
`-EIO`: error trying to read the device or structure.
`-EINVAL`: partition number is invalid.
`-EUNFORMAT`: the device is accessible, but not formatted.
`-EBADPART`: the partition is not a FAT partition.
`-EEXIST`: the partition has already been enumerated.
`-EUNFLUSHABLE`: there are no flushable sectors in the write-back cache.
`-EBUSY`: the device is busy (Only if non-blocking.).

LIBRARY

FAT.LIB

SEE ALSO

`fat_EnumDevice`, `fat_FormatPartition`, `fat_MountPartition`

fat_FileSize

```
fat_FileSize( FATfile *file, unsigned long *length );
```

DESCRIPTION

Puts the current size of the file in bytes into `length`.

PARAMETERS

`file`	Handle for an open file.
`length`	Pointer to the variable where the file length (in bytes) is to be placed.

RETURN VALUE

`0`: success.
`-EINVAL`: `file` is invalid.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_Open`, `fat_Seek`

fat_FormatDevice

```
fat_FormatDevice( mbr_dev *dev, int mode );
```

DESCRIPTION

Formats a device. The device will have a DOS master boot record (MBR) written to it. Existing partitions are left alone if the device was previously formatted. The formatted device will be registered with the write-back cache for use with the FAT library. The one partition mode will instruct the routine to create a partition table, with one partition using the entire device. This mode only works if the device is currently unformatted or has no partitions.

If needed (i.e., there is no MBR on the device), this function is called by `fat_AutoMount()` if its flags parameter allows it.

PARAMETERS

dev Pointer to the data structure for the device to format.

mode Mode:

0 = normal (use the partition table in the device structure)
1 = one partition using the entire device (errors occur if there are already partitions in the device structure)
3 = force one partition for the entire device (overwrites values already in the device structure)

RETURN

`0`: success.
`-EIO`: error trying to read the device or structure.
`-EINVAL`: device structure is invalid or does not exist.
`-ENOMEM`: memory for page buffer/RJ is not available.
`-EEXIST`: the device is already formatted.
`-EPERM`: the device already has mounted partition(s).
`-EBUSY`: the device is busy. (Only if non-blocking.)

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_AutoMount`, `fat_Init`, `fat_EnumDevice`, `fat_PartitionDevice`, `fat_FormatPartition`

fat_FormatPartition

```
fat_FormatPartition( mbr_dev *dev, fat_part *part, int pnum,
   int type, char *label, int (*usr)() );
```

DESCRIPTION

Formats partition number `pnum` according to partition type. The partition table information in the device must be valid. This will always be the case if the device was enumerated. The partition type must be a valid FAT type. Also note that the partition is ***not*** mounted after the partition is formatted. If `-EBUSY` is returned, the partition structure must not be disturbed until a subsequent call returns something other than `-EBUSY`.

If needed (i.e., `fat_MountPartition()` returned error code `-EBADPART`), this function is called by `fat_AutoMount()`.

PARAMETERS

dev	Pointer to a device structure containing partitions.
part	Pointer to a FAT partition structure to be linked. Note that `opstate` ***must*** be set to zero before first call to this function if the library is being used in the non- blocking mode.
pnum	Partition number on the device (0–3).
type	Partition type.
label	Pointer to a partition label string.
usr	Pointer to a user routine.

RETURN VALUE

`0`: success.
`-EIO`: error in reading the device or structure.
`-EINVAL`: the partition number is invalid.
`-EPERM`: write access is not allowed.
`-EUNFORMAT`: the device is accessible, but is not formatted.
`-EBADPART`: the partition is not a valid FAT partition.
`-EACCES`: the partition is currently mounted.
`-EBUSY`: the device is busy (Only if non-blocking.).

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_AutoMount`, `fat_Init`, `fat_FormatDevice`, `fat_EnumDevice`, `fat_PartitionDevice`, `fat_EnumPartition`

fat_Free

```
fat_Free( fat_part *part );
```

DESCRIPTION

This function returns the number of free clusters on the partition.

PARAMETERS

part Handle to the partition.

RETURN VALUE

Number of free clusters on success

0: partition handle is bad or partition is not mounted.

LIBRARY

FAT.LIB

SEE ALSO

fat_EnumPartition, fat_MountPartition

fat_GetAttr

```
fat_GetAttr( FATfile *file );
```

DESCRIPTION

This function gets the given attributes to the file. Use the defined attribute flags to check the value:

- FATATTR_READ_ONLY
- FATATTR_HIDDEN
- FATATTR_SYSTEM
- FATATTR_VOLUME_ID
- FATATTR_DIRECTORY
- FATATTR_ARCHIVE
- FATATTR_LONG_NAME

PARAMETERS

file	Handle to the open file.

RETURN VALUE

Attributes on success
-EINVAL: invalid file handle.

LIBRARY

FAT.LIB

SEE ALSO

fat_Open, fat_Status

fat_GetName

```
int fat_GetName( fat_dirent *entry, char *buf, word flags );
```

DESCRIPTION

Translates the file or directory name in the `fat_dirent` structure into a printable name. FAT file names are stored in a strict fixed-field format in the `fat_dirent` structure (returned from `fat_Status`, for example). This format is not always suitable for printing, so this function should be used to convert the name to a printable null-terminated string.

PARAMETERS

entry Pointer to a directory entry obtained by `fat_Status()`.

buf Pointer to a `char` array that will be filled in. This array must be at least 13 characters long.

flags May be one of the following:

- 0 - standard format, e.g., `AUTOEXEC.BAT` or `XYZ.GIF`
- `FAT_LOWERCASE` - standard format, but make lower case.

RETURN VALUE

`0`: success.
`-EINVAL`: invalid (NULL) parameter(s).

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_ReadDir`, `fat_Status`

fat_Init

```
fat_Init( int pnum, mbr_drvr *driver, mbr_dev *dev, fat_part *part,
   int norecovery );
```

DESCRIPTION

Initializes the default driver in `MBR_DRIVER_INIT`, enumerates device 0, then enumerates and mounts the specified partition. This function was replaced with the more powerful `fat_AutoMount()`.

`fat_Init()` will only work with device 0 of the default driver. This driver becomes the primary driver in the system.

The application can start calling any directory or file functions after this routine returns successfully.

The desired partition must already be formatted. If the partition mount fails, you may call the function again using a different partition number (`pnum`). The device will not be initialized a second time.

PARAMETERS

pnum	Partition number to mount (0-3).
driver	Pointer to the driver structure to fill in.
dev	Pointer to the device structure to fill in.
part	Pointer to the partition structure to fill in.
norecovery	Boolean flag - set to True to ignore power-recovery data. True is any value except zero.

RETURN VALUE

`0`: success.
`-EIO`: device I/O error.
`-EINVAL`: `pnum`, `driver`, or `device`, or `part` is invalid.
`-EUNFORMAT`: the device is not formatted.
`-EBADPART`: the partition requested is not a valid FAT partition.
`-ENOPART`: no partitions exist on the device.
`-EBUSY`: the device is busy. (Only if non-blocking.)

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_AutoMount`, `fat_EnumDevice`, `fat_EnumPartition`, `fat_MountPartition`

fat_InitUCOSMutex

`fat_InitUCOSMutex( int mutexPriority );`

DESCRIPTION

This function was introduced in FAT version 2.10. Prior versions of the FATfile system are compatible with µC/OS-II only if FAT API calls are confined to one µC/OS-II task. The FAT API is not reentrant from multiple tasks without the changes made in FAT version 2.10. If you wish to use the FAT file system from multiple µC/COS tasks, you must do the following:

1. The statement `#define FAT_USE_UCOS_MUTEX` must come before the statement:

   `#use FAT.LIB`

2. After calling `OSInit()` and before starting any tasks that use the FAT, call `fat_InitUCOSMutex(mutexPriority)`. The parameter `mutexPriority` is a µC/OS-II task priority that *must* be higher than the priorities of all tasks that call FAT API functions.
3. You must not call low-level, non-API FAT or write-back cache functions. Only call FAT functions appended with "fat_" and with public function descriptions.
4. Run the FAT in blocking mode (`#define FAT_BLOCK`).

Mutex timeouts or other errors will cause a run-time error `-ERR_FAT_MUTEX_ERROR`.

µC/OS-II may raise the priority of tasks using mutexes to prevent priority inversion.

The default mutex time-out in seconds is given by `FAT_MUTEX_TIMEOUT_SEC`, which defaults to 5 seconds if not defined in the application before the statement `#use FAT.LIB`.

PARAMETERS

`mutexPriority` A µC/OS-II task priority that MUST be higher than the priorities of all tasks that call FAT API functions.

RETURN VALUE

None. A run-time error causes an exception and the application will exit with the error code `-ERR_FAT_MUTEX_ERROR`.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_AutoMount`, `fat_Init`

fat_LastAccess

```
fat_LastAccess( fat_dirent *entry, struct tm *t );
```

DESCRIPTION

Puts the last access date of the specified entry into the system time structure `t`. The time is always set to midnight. The function does ***not*** fill in the `tm_wday` field in the system time structure.

PARAMETERS

`entry`	Pointer to a directory entry
`t`	Pointer to a system time structure

RETURN VALUE

`0`: success.
`-EINVAL`: invalid directory entry or time pointer

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_ReadDir`, `fat_Status`, `fat_CreateTime`, `fat_LastWrite`

fat_LastWrite

```
fat_LastWrite( fat_dirent *entry, struct tm *t );
```

DESCRIPTION

Puts the date and time of the last write for the given entry into the system time structure `t`. The function does not fill in the `tm_wday` field in the system time structure.

PARAMETERS

entry	Pointer to a directory entry
t	Pointer to a system time structure

RETURN VALUE

`0`: success.
`-EINVAL`: invalid directory entry or time pointer

LIBRARY

FAT.LIB

SEE ALSO

fat_ReadDir, fat_Status, fat_CreateTime, fat_LastAccess

fat_MountPartition

```
fat_MountPartition( fat_part *part );
```

DESCRIPTION

Marks the enumerated partition as mounted on both the FAT and MBR level. The partition MUST be previously enumerated with `fat_EnumPartition()`.

This function is called by `fat_AutoMount()` and `fat_Init()`.

PARAMETER

part	Pointer to the FAT partition structure to mount.

RETURN VALUE

`0`: success.
`-EINVAL`: device or partition structure or `part` is invalid.
`-EBADPART`: the partition is not a FAT partition.
`-ENOPART`: the partition does not exist on the device.
`-EPERM`: the partition has not been enumerated.
`-EACCESS`: the partition is already linked to another `fat_part` structure.
`-EBUSY`: the device is busy. (Only if non-blocking.)

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_EnumPartition`, `fat_UnmountPartition`

fat_Open

```
int fat_Open( fat_part *part, char *name, int type, int ff,
   FATfile *file, long *prealloc );
```

DESCRIPTION

Opens a file or directory, optionally creating it if it does not already exist. If the function returns `-EBUSY`, call it repeatedly with the same arguments until it returns something other than `-EBUSY`.

PARAMETERS

part Handle for the partition being used.

name Pointer to the full path name of the file to be opened/created.

type `FAT_FILE` or `FAT_DIR`, depending on what is to be opened/created.

ff File flags, must be one of:

- `FAT_OPEN` - Object must already exist. If it does not exist, `-ENOENT` will be returned.
- `FAT_CREATE` - Object is created only if it does not already exist
- `FAT_MUST_CREATE` - Object is created, and it must not already exist.
- `FAT_READONLY` - No write operations (this flag is mutually exclusive with any of the `CREATE` flags).
- `FAT_SEQUENTIAL` - Optimize for sequential reads and/or writes. This setting can be changed while the file is open by using the `fat_fcntl()` function.

file Pointer to an empty FAT file structure that will act as a handle for the newly opened file. Note that you must `memset` this structure to zero when you are using the non-blocking mode before calling this function the first time. Keep calling until something other than `-EBUSY` is returned, but do not change anything in any of the parameters while doing so.

prealloc An initial byte count if the object needs to be created. This number is rounded up to the nearest whole number of clusters greater than or equal to 1. This parameter is only used if one of the `*_CREATE` flag is set and the object does not already exist. On return, `*prealloc` is updated to the actual number of bytes allocated. May be `NULL`, in which case one cluster is allocated if the call is successful.

RETURN VALUE

0: success.
-EINVAL: invalid arguments. Trying to create volume label, or conflicting flags.
-ENOENT: file/directory could not be found.
-EEXIST: object existed when FAT_MUST_CREATE flag set.
-EPERM: trying to create a file/directory on a read-only partition.
-EMFILE - too many open files. If you get this code, increase the FAT_MAXMARKERS definition in the BIOS.

Other negative values indicate I/O error, etc.

Non-blocking mode only:

-EBUSY: the device is busy (nonblocking mode only).
-EFSTATE - file structure is not in a valid state. Usually means it was not zerod before calling this function for the first time (for that file) struct, when in non-blocking mode; can also occur if the same file struct is opened more than once.

LIBRARY

FAT.LIB

SEE ALSO

fat_ReadDir, fat_Status, fat_Close

fat_OpenDir

```
fat_OpenDir( fat_part *part, char *dirname, FATfile *dir );
```

DESCRIPTION

Opens a directory for use, filling in the `FATfile` handle.

PARAMETERS

part	Pointer to the partition structure being used.
dirname	Pointer to the full path name of the directory to be opened or created.
dir	Pointer to directory requested.

RETURN VALUE

`0`: success
`-EINVAL`: invalid argument.
`-ENOENT`: the directory cannot be found.
`-EBUSY`: the device is busy (Only if non-blocking).

Other negative values are possible from the `fat_Open()` call.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_ReadDir`, `fat_Status`, `fat_Open`, `fat_Close`

fat_PartitionDevice

```
fat_PartitionDevice( mbr_dev *dev, int pnum );
```

DESCRIPTION

This function partitions the device by modifying the master boot record (MBR), which could destroy access to information already on the device. The partition information contained in the specified `mbr_dev` structure must be meaningful, and the sizes and start positions must make sense (no overlapping, etc.). If this is not true, you will get an `-EINVAL` error code. The device being partitioned must already have been formatted and enumerated.

This function will only allow changes to one partition at a time, and this partition must either not exist or be of a FAT type.

The validity of the new partition will be verified before any changes are done to the device. All other partition information in the device structure (for those partitions that are not being modified) must match the values currently existing on the MBR. The type given for the new partition must either be zero (if you are deleting the partition) or a FAT type.

You may not use this function to create or modify a non-FAT partition.

PARAMETERS

dev	Pointer to the device structure of the device to be partitioned.
pnum	Partition number of the partition being modified.

RETURN VALUE

`0`: success.
`-EIO`: device I/O error.
`-EINVAL`: `pnum` or device structure is invalid.
`-EUNFORMAT`: the device is not formatted.
`-EBADPART`: the partition is a non-FAT partition.
`-EPERM`: the partition is mounted.
`-EBUSY`: the device is busy (Only if non-blocking).

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_FormatDevice`, `fat_EnumDevice`, `fat_FormatPartition`

fat_Read

```
fat_Read( FATfile *file, char *buf, int len );
```

DESCRIPTION

Given file, buf, and len, this routine reads len characters from the specified file and places the characters into buf. The function returns the number of characters actually read on success. Characters are read beginning at the current position of the file and the position pointer will be left pointing to the next byte to be read. The file position can be changed by the fat_Seek() function. If the file contains fewer than len characters from the current position to the EOF, the transfer will stop at the EOF. If already at the EOF, 0 is returned. The len parameter must be positive, limiting reads to 32767 bytes per call.

PARAMETERS

file	Handle for the file being read.
buf	Pointer to the buffer where data are to be placed.
len	Length of data to be read.

RETURN VALUE

Number of bytes read: success. May be less than the requested amount in non-blocking mode, or if EOF was encountered.
-EEOF: stating position for read was at (or beyond) end-of-file.
-EIO: device I/O error.
-EINVAL: file, buf, or len, contain invalid values.
-EPERM: the file is locked.
-ENOENT: the file/directory does not exist.
-EFSTATE: file is in inappropriate state (Only if non-blocking).

LIBRARY

FAT.LIB

SEE ALSO

fat_Open, fat_Write, fat_Seek

fat_ReadDir

```
int fat_ReadDir( FATfile *dir, fat_dirent *entry, int mode );
```

DESCRIPTION

Reads the next directory entry of the desired type from the given directory, filling in the entry structure.

PARAMETERS

dir Pointer to the handle for the directory being read.

entry Pointer to the handle to the entry structure to fill in.

mode 0 = next active file or directory entry including read only (no hidden, sys, label, deleted or empty)

A nonzero value sets the selection based on the following attributes:

- FATATTR_READ_ONLY - include read-only entries
- FATATTR_HIDDEN - include hidden entries
- FATATTR_SYSTEM - include system entries
- FATATTR_VOLUME_ID - include label entries
- FATATTR_DIRECTORY - include directory entries
- FATATTR_ARCHIVE - include modified entries
- FAT_FIL_RD_ONLY - filter on read-only attribute
- FAT_FIL_HIDDEN - filter on hidden attribute
- FAT_FIL_SYSTEM - filter on system attribute
- FAT_FIL_LABEL - filter on label attribute
- FAT_FIL_DIR - filter on directory attribute
- FAT_FIL_ARCHIVE - filter on modified attribute

The FAT_INC_* flags default to FAT_INC_ACTIVE if none set:

- FAT_INC_DELETED - include deleted entries
- FAT_INC_EMPTY - include empty entries
- FAT_INC_LNAME - include long name entries
- FAT_INC_ACTIVE - include active entries

The following predefined filters are available:

- FAT_INC_ALL - returns ALL entries of ANY type
- FAT_INC_DEF - default (files and directories including read-only and archive)

Note: Active files are included by default unless FAT_INC_DELETED, FAT_INC_EMPTY, or FAT_INC_LNAME is set. Include flags become the desired filter value if the associated filter flags are set.

EXAMPLES OF FILTER BEHAVIOR

```
mode = FAT_INC_DEF | FATFIL_HIDDEN | FATATTR_HIDDEN
```

would return the next hidden file or directory (including read-only and archive)

```
mode = FAT_INC_DEF|FAT_FIL_HIDDEN|FAT_FIL_DIR|FATATTR_HIDDEN
```

would return next hidden directory (but would not return any hidden file)

```
mode = FAT_INC_DEF|FAT_FIL_HIDDEN|FAT_FIL_DIR|
       FATATTR_HIDDEN & ~FATATTR_DIRECTORY
```

would return next hidden file (but would not return any hidden directory)

```
mode = FAT_INC_ALL & ~FAT_INC_EMPTY
```

would return the next non-empty entry of any type

RETURN VALUE

0: success.
-EINVAL: invalid argument.
-ENOENT: directory does not exist
-EEOF: no more entries in the directory
-EFAULT: directory chain has link error
-EBUSY: the device is busy (non-blocking mode only)

Other negative values from the `fat_Open()` call are also possible.

LIBRARY

FAT.LIB

SEE ALSO

fat_OpenDir, fat_Status

fat_Seek

```
fat_Seek( FATfile *file, long pos, int whence );
```

DESCRIPTION

Positions the internal file position pointer. `fat_Seek()` will allocate clusters to the file if necessary, but will not move the position pointer beyond the original end of file (EOF) unless doing a `SEEK_RAW`. In all other cases, extending the pointer past the original EOF will preallocate the space that would be needed to position the pointer as requested, but the pointer will be left at the original EOF and the file length will not be changed. If this occurs, an EOF error will be returned to indicate the space was allocated but the pointer was left at the EOF.

PARAMETERS

`file` Pointer to the file structure of the open file.

`pos` Position value in number of bytes (may be negative). This value is interpreted according to the third parameter, `whence`.

`whence` Must be one of the following:

- `SEEK_SET` - `pos` is the byte position to seek, where 0 is the first byte of the file. If `pos` is less than 0, the position pointer is set to 0 and no error code is returned. If pos is greater than the length of the file, the position pointer is set to EOF and error code `-EEOF` is returned.
- `SEEK_CUR` - seek `pos` bytes from the current position. If `pos` is less than 0 the seek is towards the start of the file. If this goes past the start of the file, the position pointer is set to 0 and no error code is returned. If pos is greater than 0 the seek is towards EOF. If this goes past EOF the position pointer is set to EOF and error code `-EEOF` is returned.
- `SEEK_END` - seek to `pos` bytes from the end of the file. That is, for a file that is x bytes long, the statement:

  ```
  fat_Seek (&my_file, -1, SEEK_END);
  ```

  will cause the position pointer to be set at x-1 no matter its value prior to the seek call. If the value of pos would move the position pointer past the start of the file, the position pointer is set to 0 (the start of the file) and no error code is returned. If `pos` is greater than or equal to 0, the position pointer is set to EOF and error code `-EEOF` is returned..
- `SEEK_RAW` - is similar to `SEEK_SET`, but if `pos` goes beyond EOF, using `SEEK_RAW` will set the file length and the position pointer to `pos`.

RETURN VALUE

0: success.
-EIO: device I/O error.
-EINVAL: file, pos, or whence contain invalid values.
-EPERM: the file is locked or writes are not permitted.
-ENOENT: the file does not exist.
-EEOF: space is allocated, but the pointer is left at original EOF.
-ENOSPC: no space is left on the device to complete the seek.
-EBUSY: the device is busy (Only if non-blocking).
-EFSTATE: if file in inappropriate state (Only if non-blocking).

LIBRARY

FAT.LIB

SEE ALSO

fat_Open, fat_Read, fat_Write, fat_xWrite

fat_SetAttr

```
fat_SetAttr( FATfile *file, int attr );
```

DESCRIPTION

This function sets the given attributes to the file. Use defined attribute flags to create the set values.

PARAMETERS

file Handle to the open file.

attr Attributes to set in file. May be one or more of the following:

- FATATTR_READ_ONLY
- FATATTR_HIDDEN
- FATATTR_SYSTEM
- FATATTR_VOLUME_ID
- FATATTR_DIRECTORY
- FATATTR_ARCHIVE
- FATATTR_LONG_NAME

RETURN VALUE

0: Success
-EIO: on device IO error
-EINVAL: invalid open file handle
-EPERM: if the file is locked or write not permitted
-EBUSY: if the device is busy. (Only if non-blocking)

LIBRARY

FAT.LIB

SEE ALSO

fat_Open, fat_Status

fat_Split

```
fat_Split( FATfile *file, long where, char *newfile );
```

DESCRIPTION

Splits the original file at `where` and assigns any left over allocated clusters to `newfile`. As the name implies, `newfile` is a newly created file that must not already exist. Upon completion, the original file is closed and the file handle is returned pointing to the created and opened new file. The file handle given must point to a file of type `FAT_FILE`. There are internal static variables used in this function, so only one file split operation can be active. Additional requests will be held off with `-EBUSY` returns until the active split completes.

PARAMETERS

`file` Pointer to the open file to split.

`where` May be one of the following:

- ≥ 0 - absolute byte to split the file. If the absolute byte is beyond the EOF, file is split at EOF.
- `FAT_BRK_END` - split at EOF.
- `FAT_BRK_POS` - split at current file position.

`newfile` Pointer to the absolute path and name of the new file created for the split.

RETURN VALUE

0: success.
`-EIO`: device I/O error.
`-EINVAL`: `file` has invalid references.
`-EPATHSTR`: `newfile` is not a valid path/name string.
`-EEOF`: no unused clusters are available for `newfile`. `file` will be unchanged and open, `newfile` is not created.
`-EPERM`: `file` is in use, write-protected, hidden, or system.
`-ENOENT`: `file` does not exist.
`-ETYPE`: `file` is not a FAT file type.
`-EBUSY`: the device is busy (Only non-blocking mode).
`-EFSTATE`: if file in inappropriate state (Only non-blocking mode).

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_Open`, `fat_OpenDir`, `fat_Delete`, `fat_Truncate`, `fat_Close`

fat_Status

```
int fat_Status( fat_part *part, char *name, fat_dirent *entry );
```

DESCRIPTION

Scans for the specified entry and fills in the entry structure if found without opening the directory or entry.

PARAMETERS

part	Pointer to the partition structure being used.
name	Pointer to the full path name of the entry to be found.
entry	Pointer to the directory entry structure to fill in.

RETURN VALUE

`0`: success.
`-EIO`: device I/O error.
`-EINVAL`: `part`, `filepath`, or `entry` are invalid.
`-ENOENT`: the file/directory/label does not exist.
`-EBUSY`: the device is busy (Only non-blocking mode). If you get this error, call the function again without changing any parameters.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_ReadDir`

fat_SyncFile

```
fat_SyncFile( FATfile *file );
```

DESCRIPTION

Updates the directory entry for the given file, committing cached size, dates, and attribute fields to the actual directory. This function has the same effect as closing and re-opening the file.

PARAMETERS

file	Pointer to the open file.

RETURN VALUE

0: success.
-EINVAL: file is invalid.
-EPERM - this operation is not permitted on the root directory.
-EBUSY: the device is busy (Only if non-blocking). Call function again to complete the update.
-EFSTATE - file not open or in an invalid state.

Any other negative value: I/O error when updating the directory entry.

LIBRARY

FAT.LIB

SEE ALSO

fat_Close, fat_Open, fat_OpenDir

fat_SyncPartition

```
fat_SyncPartition( fat_part *part );
```

DESCRIPTION

Flushes all cached writes to the specified partition to the actual device.

PARAMETER

part	Pointer to the partition to be synchronized.

RETURN VALUE

0: success.
-EINVAL: part is invalid.
-EBUSY: the device is busy (Only if non-blocking). Call function again to complete the sync.

Any other negative value: I/O error when updating the device.

LIBRARY

FAT.LIB

SEE ALSO

fat_Close, fat_SyncFile, fat_UnmountPartition

fat_Tell

```
fat_Tell( FATfile *file, unsigned long *pos );
```

DESCRIPTION

Puts the value of the position pointer (that is, the number of bytes from the beginning of the file) into `pos`. Zero indicates the position pointer is at the beginning of the file.

µC/OS-II USERS:

- The FAT API is not reentrant. To use the FAT from multiple µC/OS-II tasks, put the following statement in your application:

  `#define FAT_USE_UCOS_MUTEX`

- Mutex timeouts or other mutex errors will cause the run-time error `ERR_FAT_MUTEX_ERROR`. The default mutex timeout is 5 seconds and can be changed by #define'ing a different value for `FAT_MUTEX_TIMEOUT_SEC`.
- You MUST call `fat_InitUCOSMutex()` after calling `OSInit()` and before calling any other FAT API functions.
- You must run the FAT in blocking mode (`#define FAT_BLOCK`).
- You must not call low-level, non-API FAT or write-back cache functions. Only call FAT functions appended with "`fat_`" and with public function descriptions.

PARAMETERS

file	Pointer to the file structure of the open file
pos	Pointer to the variable where the value of the file position pointer is to be placed.

RETURN VALUE

`0`: success.
`-EIO`: position is beyond EOF.
`-EINVAL`: `file` is invalid.

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_Seek`, `fat_Read`, `fat_Write`, `fat_xWrite`

fat_tick

```
int fat_tick( void )
```

DESCRIPTION

Drive device I/O completion and periodic flushing. It is not generally necessary for the application to call this function; however, if it is called regularly (when the application has nothing else to do) then file system performance may be improved.

RETURN VALUE

Currently always 0.

LIBRARY

`FATWTC.LIB`

fat_Truncate

```
fat_Truncate( FATfile *file, long where );
```

DESCRIPTION

Truncates the file at `where` and frees any left over allocated clusters. The file must be a `FAT_FILE` type.

PARAMETERS

file	Pointer to the open file to truncate.
where	One of the following: • ≥ 0 - absolute byte to truncate the file. The file is truncated at EOF if the absolute byte is beyond EOF. • `FAT_BRK_END` - truncate at EOF. • `FAT_BRK_POS` - truncate at current file position.

RETURN VALUE

`0`: success.
`-EIO`: device I/O error.
`-EINVAL`: `file` is invalid.
`-EPERM`: `file` is in use, write-protected, hidden, or system.
`-ENOENT`: the file does not exist.
`-ETYPE`: `file` is not a FAT file type.
`-EBUSY`: the device is busy (Only if non-blocking).
`-EFSTATE`: if file in inappropriate state (Only if non-blocking)

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_Open`, `fat_OpenDir`, `fat_Delete`, `fat_Split`

fat_UnmountDevice

```
fat_UnmountDevice( mbr_dev * dev );
```

DESCRIPTION

Unmounts all FAT partitions on the given device and unregisters the device from the cache system. This commits all cache entries to the device and prepares the device for power down or removal. The device structure given must have been enumerated with `fat_EnumDevice()`.

This function was introduced in FAT module version 2.06. Applications using prior versions of the FAT module would call `fat_UnmountPartition()` instead.

PARAMETER

dev	Pointer to a FAT device structure to unmount.

RETURN VALUE

`0`: success.
`-EINVAL`: device structure (`dev`) is invalid.
`-EBUSY`: the device is busy (Only if non-blocking).

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_EnumDevice`, `fat_AutoMount`, `fat_UnmountPartition`

fat_UnmountPartition

```
fat_UnmountPartition( fat_part *part );
```

DESCRIPTION

Marks the enumerated partition as unmounted on both the FAT and the master boot record levels. The partition must have been already enumerated using `fat_EnumPartition()` (which happens when you call `fat_AutoMount()`).

To unmount all FAT partitions on a device call `fat_UnmountDevice()`, a function introduced with FAT version 2.06. It not only commits all cache entries to the device, but also prepares the device for power down or removal.

> **Note:** The partitions on a removable device must be unmounted in order to flush data before removal. Failure to unmount a partition that has been written could cause damage to the FAT file system.

PARAMETERS

part	Pointer to a FAT partition structure to unmount.

RETURN VALUE

`0`: success.
`-EINVAL`: device or partition structure or `pnum` is invalid.
`-EBADPART`: the partition is not a FAT partition.
`-ENOPART`: the partition does not exist on the device.
`-EPERM`: the partition has not been enumerated.
`-EBUSY`: the device is busy (only if non-blocking).

LIBRARY

`FAT.LIB`

SEE ALSO

`fat_EnumPartition`, `fat_MountPartition`, `fat_UnmountDevice`

fat_Write

```
fat_Write( FATfile *file, char *buf, int len );
```

DESCRIPTION

Writes characters into the file specified by the file pointer beginning at the current position in the file. Characters will be copied from the string pointed to by `buf`. The `len` variable controls how many characters will be written. This can be more than one sector in length, and the write function will allocate additional sectors if needed. Data is written into the file starting at the current file position regardless of existing data. Overwriting at specific points in the file can be accomplished by calling the `fat_Seek()` function before calling `fat_Write()`.

PARAMETERS

file	Handle for the open file being written.
buf	Pointer to the buffer containing data to write.
len	Length of data to be written.

RETURN VALUE

Number of bytes written: success (may be less than `len`, or zero if non-blocking mode)
`-EIO`: device I/O error.
`-EINVAL`: `file`, `buf`, or `len` contain invalid values.
`-ENOENT`: file does not exist.
`-ENOSPC`: no space left on the device to complete the write.
`-EFAULT`: problem in file (broken cluster chain, etc.).
`-EPERM`: the file is locked or is write-protected.
`-EBUSY`: the device is busy (only if non-blocking).
`-EFSTATE`: file is in inappropriate state (only if non-blocking).

LIBRARY

FAT.LIB

SEE ALSO

`fat_Open`, `fat_Read`, `fat_xWrite`, `fat_Seek`

fat_xWrite

fat_xWrite(FATfile *file, long xbuf, int len);

DESCRIPTION

Writes characters into the file specified by the file pointer beginning at the current position in the file. Characters will be copied from the xmem string pointed to by xbuf. The len variable controls how many characters will be written. This can be more than one sector in length, and the write function will allocate additional sectors if needed. Data will be written into the file starting at the current file position regardless of existing data. Overwriting at specific points in the file can be accomplished by calling the fat_Seek() function before calling fat_xWrite().

PARAMETERS

file	Handle for the open file being written.
xbuf	xmem address of the buffer to be written.
len	Length of data to write.

RETURN VALUE

Number of bytes written: success. (may be less than len, or zero if non-blocking mode)
-EIO: device I/O error.
-EINVAL: file, xbuf, or len contain invalid values.
-ENOENT: the file/directory does not exist.
-ENOSPC: there are no more sectors to allocate on the device.
-EFAULT: there is a problem in the file (broken cluster chain, etc.).
-EPERM: the file is locked or write-protected.
-EBUSY: the device is busy (only if non-blocking).
-EFSTATE: file is in inappropriate state (only if non-blocking).

LIBRARY

FAT.LIB

SEE ALSO

fat_Open, fat_Read, fat_Write, fat_Seek

nf_XD_Detect

```
long nf_XD_Detect(int debounceMode)
```

DESCRIPTION

This function attempts to read the xD card ID and searches the internal device table for that ID.

This function assumes that there is only one XD card present.

WARNING! - This should not be called to determine if it is safe to do write operations if there is a chance a removable device might be pulled between calling it and the write. It is best used to determine if a device is present to proceed with an automount after a device has been unmounted in SW and removed.

PARAMETERS

debounceMode

0 - no debouncing

1 - busy wait for debouncing interval

2 - for use if the function is to be called until the debouncing interval is done, e.g.,

```
waitfor(rc = nf_XD_Detect(1) != -EAGAIN);
```

`-EAGAIN` will be returned until done.

RETURN VALUE

`>0`: The ID that was found on the device and in the table
`-EBUSY`: NAND flash device is busy
`-ENODEV`: No device found
`-EAGAIN`: if `debounceMode` equals 2, then not done debouncing, try again

LIBRARY

NFLASH_FAT.LIB

Appendix A. More FAT Information

The FAT file system stores and organizes files on a storage device such as a hard drive or a memory device.

A.1 Clusters and Sectors

Every file is stored on one or more *clusters*. A cluster is made up of a contiguous number of bytes called *sectors* and is the smallest unit of allocation for files. The Dynamic C FAT implementation supports a sector size of 512 bytes. Cluster sizes depend on the media. The table below gives the cluster sizes used for some of our RabbitCore modules.

Table 3. Cluster Sizes on Flash Devices

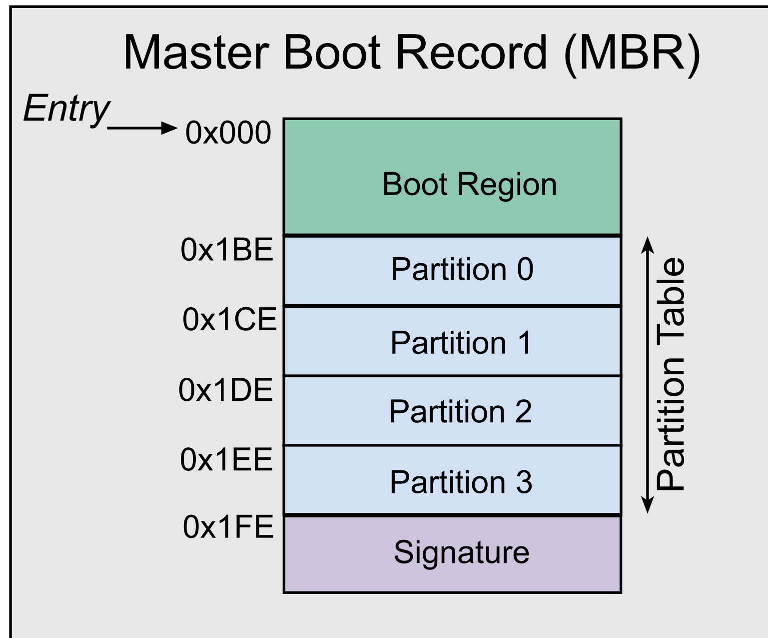
RabbitCore Model	Flash Device	Number of Sectors per Cluster
RCM 3700	1 MB Serial Flash	1
RCM 3300	4 and 8 MB Serial Flash	2
RCM3360/70	NAND Flash	32
RCM4000	NAND Flash	32
RCM4200/10	8 and 4 MB Serial Flash	2
RCM4300/10	SD Card	Varies

The cluster size for a NAND device corresponds to its page size. Note that a file or directory takes at minimum one cluster. On a NAND device the page size is 16K bytes; therefore, while it is allowable to write very small files to the FAT file system on a NAND device, it is not space efficient. Even the smallest file takes at least 16,000 bytes of storage. Cluster sizes for SD cards vary with the size of the card inserted. To determine the number of sectors per cluster on an SD card, divide the size of the card by 32MB.

A.2 The Master Boot Record

The *master boot record* (MBR) is located on one or more sectors at the physical start of the device. Its basic structure is illustrated in Figure 3. The boot region of the MBR contains DOS boot loader code, which is written when the device is formatted (but is not otherwise used by the Dynamic C FAT file system). The partition table follows the boot region. It contains four 16-byte entries, which allows up to four partitions on the device. Partition table entries contain some critical information: the partition type (Dynamic C FAT recognizes partition types FAT12 and FAT16) and the partition's starting and ending sector numbers. There is also a field denoting the total number of sectors in the partition. If this number is zero, the corresponding partition is empty and available.

Figure 3. High-Level View of an MBR

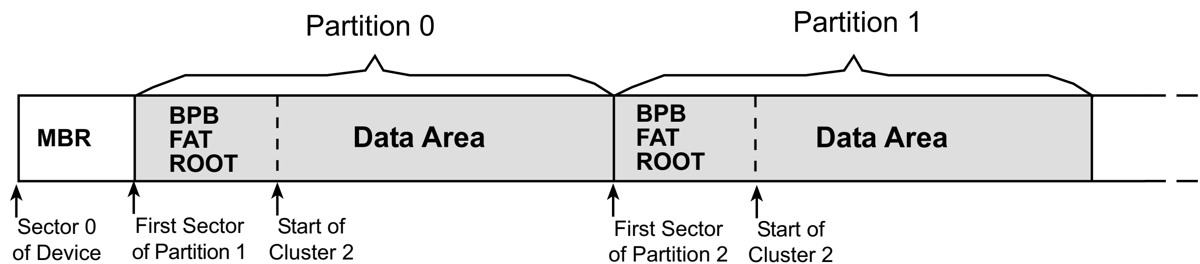


Note: Some devices are formatted without an MBR and, therefore, have no partition table. This configuration is not currently supported in the Dynamic C FAT file system.

A.3 FAT Partitions

The first sector of a valid FAT file system partition contains the *BIOS parameter block* (BPB); this is followed by the *file allocation table* (FAT), and then the *root directory.* The figure below shows a device with two FAT partitions.

Figure 4. Two FAT Partitions on a Device



A.3.1 BPB

The fields of the BPB contain information describing the partition:

- the number of bytes per sector
- the number of sectors per cluster (see Table 3)
- the total count of sectors on the partition
- the number of root directory entries
- plus additional information not mentioned here

The FAT type (FAT12 or FAT16) is determined by the count of clusters on the partition. The "12" and "16" refer to the number of bits used to hold the cluster number. The FAT type is calculated using information found in the BPB. Information from a BPB on a mounted partition is stored in the partition structure (of type `fat_part`) populated by `fat_AutoMount()`.

Partitions greater than or equal to 2 MB will be FAT16. Smaller partitions will be FAT12. To save code space, you can compile out support for either FAT type. Find the lines

```
#define FAT_FAT12          // comment out to disable FAT12 support
#define FAT_FAT16          // comment out to disable FAT16 support
```

in `LIB/../FAT.LIB`, make your change, and then recompile your application.

A.3.2 FAT

The file allocation table is the structure that gives the FAT file system its name. The FAT stores information about cluster assignments. A cluster is either assigned to a file, is available for use, or is marked as bad. A second copy of the FAT immediately follows the first.

A.3.3 Root Directory

The root directory has a predefined location and size. It has 512 entries of 32 bytes each. An entry in the root directory is either empty or contains a file or subdirectory name (in 8.3 format), file size, date and time of last revision and the starting cluster number for the file or subdirectory.

A.3.4 Data Area

The data area takes up most of the partition. It contains file data and subdirectories. Note that the data area of a partition must, by convention, start at cluster 2.

A.3.5 Creating Multiple FAT Partitions

FAT version 2.13 introduces `FAT_Write_MBR.c`, a utility that simplifies the creation of multiple partitions. It is distributed with FAT module version 2.13. It is also compatible with FAT versions 2.01, 2.05 and 2.10. If you have one of these earlier versions of the FAT module and would like a copy of `FAT_Write_MBR.c`, please contact Technical Support either by email to support@rabbitsemiconductor.com or by using the online form available on the Rabbit website: www.rabbitsemiconductor.com/support/questionSubmit.shtml. See Section 4.1.2 for information on running this utility.

Without the use of `FAT_Write_MBR.c`, creating multiple FAT partitions on the flash device requires a little more effort than the default partitioning. If the flash device does not contain an MBR, i.e., the device is not formatted, both `fat_Init()` and `fat_AutoMount()` return an error code (`-EUNFORMAT`) indicating this fact. So the next task is to write the MBR to the device. This is done with a call to `fat_FormatDevice()`. Since we want more than one partition on the flash device, `fat_FormatDevice()` must be called with a `mode` parameter of zero.

Before calling `fat_FormatDevice()`, partition specific information must be set in the `mbr_part` entries for each partition you are creating. The following code shows possible information for partition 0 where `MY_PARTITION_SIZE` is equal to the size of the desired partition in bytes, 512 is the flash sector size, and `dev` points to the `mbr_part` structure.

```
memset(dev->part, 0, sizeof(mbr_part));
dev->part[0].starthead = 0xFE;
dev->part[0].endhead = 0xFE;
dev->part[0].startsector = 1;
dev->part[0].partsecsize = (MY_PARTITION_SIZE / 512 ) + 1;
dev->part[0].parttype = (dev->part[0].partsecsize < SEC_2MB) ? 1 : 6;
```

The `memset()` function is used to initialize the entry to zero. The values for `starthead` and `endhead` should be 0xFE to indicate that the media uses LBA (Logical Block Addressing) instead of head and cylinder addressing. The FAT library uses LBA internally. The values for the `startsector`, `partsecsize` and `parttype` fields determine where the partition starts, how many sectors it contains and what partition type it is. The number of sectors in the partition is calculated by dividing the number of raw bytes in the partition by the sector size of the flash. The number of raw bytes in the partition includes not only bytes for file storage, but also the space needed by the BPB and the root directory. One is added to `dev->partsecsize` to ensure an extra sector is assigned if `MY_PARTITION_SIZE` is not evenly divisible by the size of a flash sector. The partition type (`.parttype`) is determined by the partition size: 1 indicates FAT12 and 6 indicates FAT16. Fill in an `mbr_part` structure for each partition you are creating. The remaining entries should be zeroed out.

When laying out partitions, there are three basic checks to make sure the partitions fit in the available device space and do not overlap.

1. No partition can start on a sector less than 1.
2. Each partition resides on sectors from `startsector` through `startsector+partsecsize-1`. No other partition can have a `startsector` value within that range.
3. No partition ending sector (`startsector+partsecsize-1`) can be greater than or equal to the total sectors on the device.

The partition boundaries are validated in the call to `fat_FormatDevice()` and the function will return an error if any of the partition boundaries are invalid. If `fat_FormatDevice()` returns success, then call `fat_AutoMount()` with flags of `FDDF_COND_PART_FORMAT` | `FDDF_MOUNT_DEV_#` | `FDDF_MOUNT_PART_ALL`; where # is the device number for the device being partitioned. This will format and mount the newly created partitions.

A.4 Directory and File Names

File and directory names are limited to 8 characters followed by an optional period (.) and an extension of up to 3 characters. The characters may be any combination of letters, digits, or characters with code point values greater than 127. The following special characters are also allowed:

```
$ % ' - _ @ ~ ` ! ( ) { } ^ # &
```

File names passed to the file system are always converted to upper case; the original case value is lost.

The maximum size of a directory is limited by the available space. It is recommended that no more than ten layers of directories be used with the Dynamic C FAT file system.

A.5 µC/OS-II and FAT Compatibility

Versions of the FAT file system prior to version 2.10 are compatible with µC/OS-II only if FAT API calls are confined to one µC/OS-II task. To make the FAT API reentrant from multiple tasks, you must do the following:

- Use FAT version 2.10
- #define `FAT_USE_UCOS_MUTEX` before #use'ing `FAT.LIB`
- Call the function `fat_InitUCOSMutex(priority)` after calling `OSInit()` and before calling FAT APIs or beginning multitasking; the parameter "priority" MUST be a higher priority than all tasks using FAT APIs
- Call only high-level fat APIs with names that begin with "fat_"

See the function description for `fat_InitUCOSMutex` for more details, and the sample program `Samples/FileSystem/FAT_UCOS.C` for a demonstration of using FAT with µC/OS-II.

A.6 SF1000 and FAT Compatibility

There are two macros that need to be defined for the FAT module to work with the SF1000 Serial Flash Expansion Board.

```
#define SF_SPI_DIVISOR 5
#define SF_SPI_INVERT_RX
```

A.7 Hot-Swapping an xD Card

Hot-swapping is currently supported on the RCM3365 and the RCM3375. FAT version 2.10 or later is required. Two sample programs are provided in `Samples/FileSystem` to demonstrate this feature: `FAT_HOT_SWAP.C` and `FAT_HOT_SWAP_336x0.C`. The samples are mostly identical: they both test for a keyboard hit to determine if the user wants to hot-swap the xD card, but in addition, the sample program `FAT_HOT_SWAP_336x0.C` also checks for a switch press and indicates a ready-to-mount condition with an LED.

As demonstrated in the sample programs, an xD card should only be removed after it has unmounted with `fat_UnmountDevice()` and no operations are happening on the device. Only `fat_AutoMount()` should be used to remount xD cards. In addition, the function `nf_XD_Detect()` should be called to verify xD card presence before attempting to remount an xD card.

xD cards formatted with versions of the FAT prior to 2.10 did not have unique volume labels. If there is a chance that two such cards may be swapped, call `fat_autoMount()` with the `FDDF_NO_RECOVERY` flag set. This means that if there is a write cache entry to be written, it will not be written. The function `fat_UnmountDevice()` flushes the cache (i.e., writes all cache entries to the device) before unmounting, so this should not generally be a problem if the device was properly unmounted.

A.8 Hot-Swapping an SD Card

Hot-swapping is currently supported on the RCM4300 and the RCM4310. FAT version 2.14 or later is required. A sample program is provided in `Samples/FileSystem` to demonstrate this feature: `FAT_HOT_SWAP_SD.C`. The sample tests for a keyboard hit to determine if the user wants to hot-swap the SD card. The LED near the SD socket on the RCM 4300/4310 will go out when the SD card is unmounted and safe to remove.

Hot-swapping an SD card requires that you unmount the device before removal, as the FAT filesystem employs a cache system that may not have written all information to the device unless unmounted. This is easy to see with both the RCM4300 and RCM4310 because the FAT system turns on the LED during the mount process, and turns it off when the card is unmounted. It is possible to have the LED left on during an error condition. This may require you to restart the system and mount the card again, then unmount to ensure all cached entries have been written.

As demonstrated in the sample program, the SD card should only be removed after it has unmounted with `fat_UnmountDevice()` and no operations are happening on the device. Only `fat_AutoMount()` should be used to remount SD cards. In addition, the function `sdspi_debounce()` should be called to verify SD card presence before attempting to remount an SD card.

A.9 Unsupported FAT Features

At this time, the Dynamic C FAT file system does not support the following.

- Single-volume drives (they do not have an MBR)
- FAT32 or long file or directory names
- Sector sizes other than 512 bytes
- Direct parsing of relative paths
- Direct support of a “working directory”
- Drive letters (the FAT file system is not DOS)

A.10 References

There are a number of good references regarding FAT file systems available on the Internet. Any reasonable search engine will bring up many hits if you type in relevant terms, such as “FAT,” “file system,” “file allocation table,” or something along those lines. At the time of this writing, the following links provided useful information.

1. This link is to Microsoft’s “FAT32 File System Specification,” which is also applicable to FAT12 and FAT16.

   www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

2. This article gives a brief history of FAT.

   http://en.wikipedia.org/wiki/File_Allocation_Table

3. These tutorials give lots of details plus links to more information.

   www.serverwatch.com/tutorials/article.php/2239651
   www.pcguide.com/ref/hdd/file/fat.htm

Appendix B. Custom Configurations

The configuration library `fat_config.lib` is brought in when fat.lib is #use'd in the code. This configuration library recognizes the macro `_DRIVER_CUSTOM` as a flag that a custom hardware configuration or custom device driver is being used in the hardware/device driver arrangement set up by fat_config.lib.

B.1 Adding SD Card Interface

As an example of a custom hardware configuration, consider the task of designing an SD card interface on a board that uses a core module that does not natively support such an interface. This is just one example of connecting an SD card socket to an RCM4400W core module. A device driver already exists for the SD card interface with FAT module version 2.13 and Dynamic C version 10.21. The desired driver, which in this case is `SD_FAT.LIB`, must be identified before `fat.lib` is #use'd in the application. One strategy is to create a new configuration library that will be #use'd at the top of your application program. The bulk of this library can be taken from existing configuration setups and modified for the custom application. In this case, we get this setup section from the RCM43xx.lib file, which sets up the SD card for the RCM43xx series of core modules. There is a fairly long parameter block which consists of defines setting which port, pins and shadow registers are used to access the control and serial lines needed to control the SD card.

Once this type of configuration library is created, you can use any FAT-based sample with the custom hardware by simply replacing the #use "FAT.lib" statement in the sample with #use "customSD.lib", where customSD.lib is the name of the new configuration library.

B.1.1 Example Code

The configuration library would have to include something similar to the following code:

```
// Tells fat_config.lib that a custom driver/configuration is to be used
#define _DRIVER_CUSTOM  "SD_FAT.LIB"

// Configuration definitions modified to include custom initialization function
// Signature strings have been set to NULL because the SD card is removable media
#define PC_COMPATIBLE
#define _DRIVER_CUSTOM_INIT { "SD", sd_custom_init, NULL, },
#define _DEVICE_CUSTOM_0 { sd_custom_init, NULL, 0, 0, \
                     FDDF_MOUNT_PART_ALL|FDDF_MOUNT_DEV_0, NULL, },

// SD hardware parameter block copied from RCM43xx.lib
//*** SD hardware definitions

// Sets up the following port connections for the SD driver
// This would use external buffer and power line control similar to the SD card circuitry on the RCM4300
// Directly connecting these pins to the SD card would have possible line drive issues
// Port C, Pin 2   TxC Serial Transmit line
// Port D, Pin 0   Card Select line, active low
// Port D, Pin 1   SD Card Active LED, active high
// Port D, Pin 2   SclkC Serial Clock line
// Port D, Pin 3   TxC Serial Receive line
// Port E, Pin 3   Card Detect line, active low

// This example does not use write protect or power control
```

```
// If not using power control, it is recommended that the detect switch on the SD card socket
// be tied to appropriate circuitry for applying power to the SD card.

#define SD_CD_PORT            PEDR              // Card Detect set to pin PE3
#define SD_CD_PORT_FR         PEFR
#define SD_CD_PORT_FRSHADOW   &PEFRShadow
#define SD_CD_PORT_DDR        PEDDR
#define SD_CD_PORT_DDRSHADOW &PEDDRShadow
#define SD_CD_PIN             3

// No Write Protect input
#define SD_WP_PORT            0                 // WP input not used
#define SD_WP_PORT_FR         0
#define SD_WP_PORT_FRSHADOW   NULL
#define SD_WP_PORT_DDR        0
#define SD_WP_PORT_DDRSHADOW NULL
#define SD_WP_PIN             0

#define SD_CS_PORT            PDDR              // Card Select set to pin PD0
#define SD_CS_PORT_DRSHADOW   &PDDRShadow
#define SD_CS_PORT_DDR        PDDDR
#define SD_CS_PORT_FR         PDFR
#define SD_CS_PORT_FRSHADOW   &PDFRShadow
#define SD_CS_PORT_DDRSHADOW &PDDDRShadow
#define SD_CS_PORT_DCR        PDDCR
#define SD_CS_PORT_DCRSHADOW &PDDCRShadow
#define SD_CS_PIN             0
#define SD_CS_PORT_OD         0

#define SD_TX_PORT_DR         PCDR              // TxC set to pin PC2
#define SD_TX_PORT_DRSHADOW   &PCDRShadow
#define SD_TX_PORT_FR         PCFR
#define SD_TX_PORT_FRSHADOW   &PCFRShadow
#define SD_TX_PORT_DDR        PCDDR
#define SD_TX_PORT_DDRSHADOW &PCDDRShadow
#define SD_TX_PORT_DCR        PCDCR
#define SD_TX_PORT_DCRSHADOW &PCDCRShadow
#define SD_TX_PIN             2
#define SD_TX_PORT_OD         0

#define SD_PWR_PORT_DR         0                // Power control pin not used
#define SD_PWR_PORT_DRSHADOW   NULL
#define SD_PWR_PORT_FR         0
#define SD_PWR_PORT_FRSHADOW   NULL
#define SD_PWR_PORT_DDR        0
#define SD_PWR_PORT_DDRSHADOW NULL
#define SD_PWR_PORT_DCR        0
#define SD_PWR_PORT_DCRSHADOW NULL
```

```
#define SD_PWR_PIN            0
#define SD_PWR_PORT_OD        0
#define SD_PWR_PORT_ON        0

#define SD_LED_PORT_DR        PDDR                // LED Output set to pin PD1
#define SD_LED_PORT_DRSHADOW  &PDDRShadow
#define SD_LED_PORT_FR        PDFR
#define SD_LED_PORT_FRSHADOW  &PDFRShadow
#define SD_LED_PORT_DDR       PDDDR
#define SD_LED_PORT_DDRSHADOW &PDDDRShadow
#define SD_LED_PORT_DCR       PDDCR
#define SD_LED_PORT_DCRSHADOW &PDDCRShadow
#define SD_LED_PIN            1
#define SD_LED_PORT_OD        1
#define SD_LED_PORT_ON        0

#define SD_RX_PORT_DR         PDDR                //  RxC set to pin PD3
#define SD_RX_PORT_FR         PDFR
#define SD_RX_PORT_FRSHADOW   &PDFRShadow
#define SD_RX_PORT_DDR        PDDDR
#define SD_RX_PORT_DDRSHADOW  &PDDDRShadow
#define SD_RX_PIN             3

#define SD_CLK_PORT_DR        PDDR                //  SclkC set to pin PD2
#define SD_CLK_PORT_FR        PDFR
#define SD_CLK_PORT_FRSHADOW  &PDFRShadow
#define SD_CLK_PORT_DDR       PDDDR
#define SD_CLK_PORT_DDRSHADOW &PDDDRShadow
#define SD_CLK_PORT_DCR       PDDCR
#define SD_CLK_PORT_DCRSHADOW &PDDCRShadow
#define SD_CLK_PIN            2
#define SD_CLK_PORT_OD        0

// Setup clock & control registers for serial port
#define SD_SPI_TACRSHADOW &TACRShadow
#define SD_SPI_SERPORT SCDR
#define SD_SPI_TCREG TACR
#define SD_SPI_TCRSHADOW &TACRShadow
#define SD_SPI_TCRVALUE 0
#define SD_SPI_SERSHADOW &SCERShadow
#define SD_SPI_SERVALUE SD_SPI_CLOCK_MODE
#define SD_SPI_SCRSHADOW &SCCRShadow
#define SD_SPI_SCRVALUE SD_SPI_CONTROL_VALUE
#define SD_SPI_DIVREG TAT6R
#define SD_SPI_DIVREGSHADOW &TAT6RShadow
#define SD_SPI_DIVISOR 0
```

```
// Macros for enabling and disabling the Card Select control line
#define SD_ENABLECS(DI) BitWrPortI(DI->csport,DI->csportdrShadow, 0,
DI->cspin)

#define SD_DISABLECS(DI) BitWrPortI(DI->csport, DI->csportdrShadow, 1,
        DI->cspin)

#define SD_ENABLEPOW(DI) DI           // Power enable not used
#define SD_DISABLEPOW(DI) DI

//SD serial port register offsets
#define SD_AR_OFFSET 1
#define SD_SR_OFFSET 3
#define SD_CR_OFFSET 4
#define SD_ER_OFFSET 5


#define SD_SPI_CONTROL_VALUE 0x1c     // Selects Port D as RxC alt. input
#define SD_SPI_TXMASK 0x80            // Control bits for starting TX or RX operations
#define SD_SPI_RXMASK 0x40
#define SD_SPI_CLOCK_MODE 0x08        // Sets Reverse Data Bit operation (MSB first)


#use "FAT.lib"                        // Use the necessary libraries for FAT operation
#use "SD_FAT.lib"

/*** EndHeader */

/*** BeginHeader sd_custom_init */
// This is a custom initialization function; it adds alternate routing of SCLKC to Port D pin 2
// before calling the standard SD initialization function.
// This function would only be required if an alternate routing issue could not be set by the
// definitions above.
int sd_custom_init(mbr_drvr *driver, void *device_list);
/*** EndHeader */

int sd_custom_init(mbr_drvr *driver, void *device_list)
{
   // Select SCLKC output on Port D pin 2
   WrPortI(PDALR, &PDALRShadow, PDALRShadow & 0xCF);
   sd_InitDriver(driver, device_list);
}
```

Index

Symbols

\\ 18
\n 8
\r 8
_fat_device_table 27
µC/OS-II compatibility 44, 74

Numerics

2nd copy of FAT 72

A

attributes of a file 11

B

back slash 18
blocking 1
blocking a non-blocking function 13
blocking mode 5
BPB 71
bringing up the FAT file system 4

C

cached write 9, 61
carriage return 8
clusters
  assignments 72
  available amount 40
  definition 70
compatibility with µC/OS-II 44, 74
configuration library 5
costatements 13–15
creating a file 8
custom configurations 76
custom device driver 5

D

data area 72
device 1
device structure 27
directory 2
  create 21
  default search 23
  delete 24
  entry structure 22
  names 73
  root 72
  search conditions 22
DLM and FAT 17
download manager 17
driver 1
Dynamic C version 1

E

error codes 2, 5
escape character 18

F

FAT and DLM 17
FAT API Functions
  fat_AutoMount 27
  fat_Close 30
  fat_CreateDir 31
  fat_CreateFile 32
  fat_CreateTime 33
  fat_Delete 34
  fat_EnumDevice 35
  fat_EnumPartition 36
  fat_FileSize 37
  fat_FormatDevice 38
  fat_FormatPartition 39
  fat_Free 40
  fat_GetAttr 41
  fat_GetName 42
  fat_Init 43
  fat_InitUCOSMutex 44
  fat_LastAccess 45
  fat_LastWrite 46
  fat_MountPartition 47
  fat_Open 48
  fat_OpenDir 50
  fat_PartitionDevice 51
  fat_Read 52
  fat_ReadDir 53
  fat_Seek 55
  fat_SetAttr 57
  fat_Split 58
  fat_Status 59
  fat_SyncFile 60
  fat_SyncPartition 61

fat_Tell ... 62
fat_tick ... 63
fat_Truncate ... 64
fat_UnmountDevice ... 65
fat_UnmountPartition ... 66
fat_Write ... 67
fat_xWrite ... 68
nf_XD_Detect ... 69
FAT module version ... 1
fat_AutoMount ... 6
fat_config.lib ... 5
fat_dirent ... 22
fat_Init ... 6
fat_part ... 6, 27
fat_part_mounted ... 6, 21, 27
FAT_USE_FORWARDSLASH ... 18
file
attributes ... 11, 41, 57
create ... 8, 21
delete ... 24
names ... 73
open ... 8, 18
read ... 9, 19
seek ... 20
size ... 37
state ... 13
write ... 8, 19
flash types supported ... 2
flush cached file information ... 60
flush cached writes ... 61
forward slash ... 18

H

hot-swapping
SD card ... 75
xD card ... 74

I

initialization ... 6

L

line feed ... 8

M

max number of characters read ... 9
MBR ... 70
mbr_dev ... 27
Micro C/OS-II ... 44, 74
multitasking compatibility ... 44, 74

N

names ... 73
non-blocking ... 1, 12
num_fat_devices ... 6

O

opening a file ... 8

P

partition ... 1
partition structure ... 6, 27
partitioning ... 16–17
path separator ... 18
prealloc ... 5

R

reading a file ... 9
reading max number of characters ... 9
removable device advice ... 9
reserving file space ... 5
result code (rc) ... 5

S

sector ... 70
SF1000 ... 74
shell program ... 10
software version ... 1
state of file ... 13
subdirectory ... 2
supported flash types ... 2

U

ucos2 ... 44, 74
udppages.c ... 10
unsupported FAT features ... 75
using the FAT file system ... 7

V

version of software ... 1

W

write-back cache ... 9
writing a file ... 8