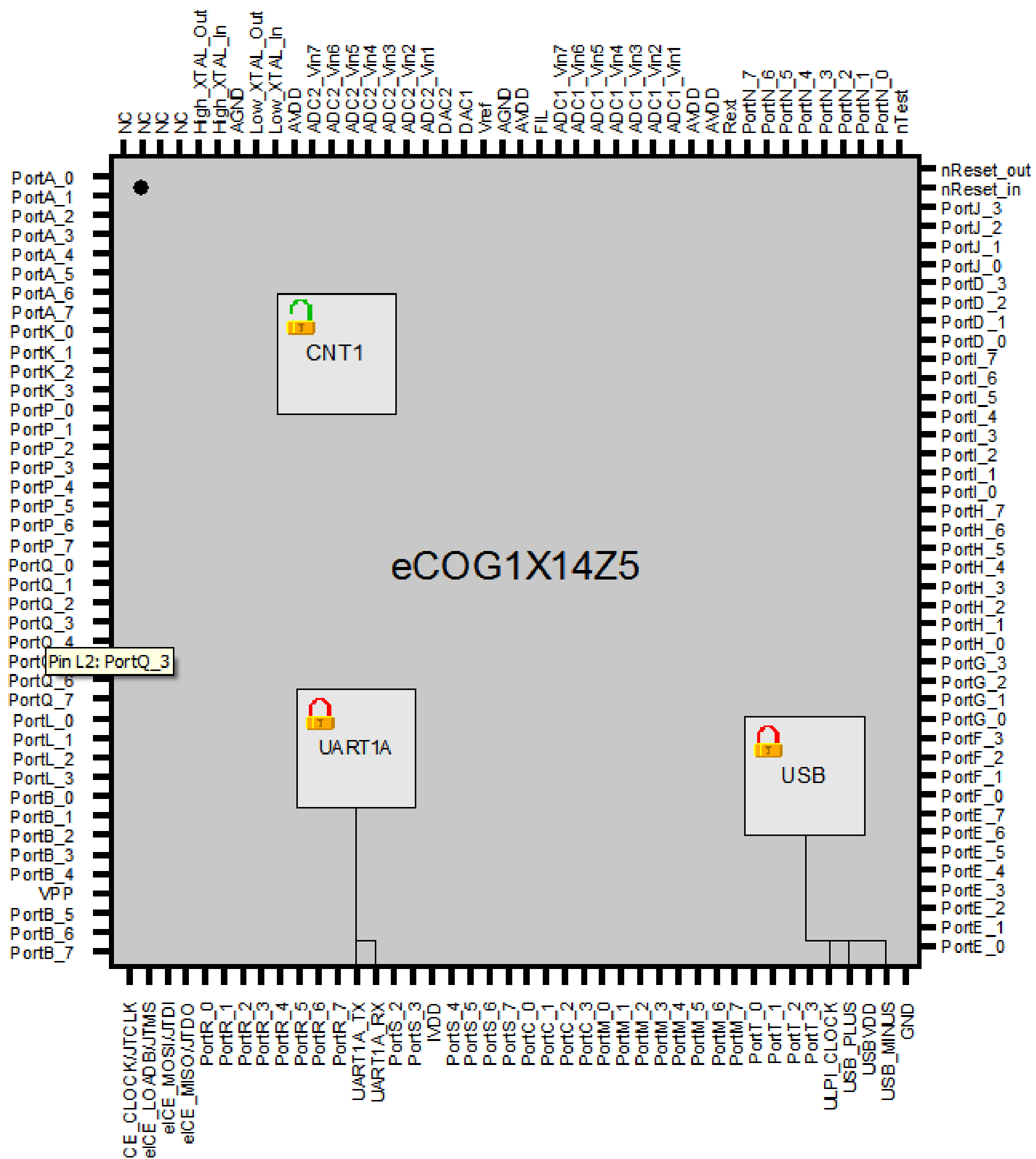
cyan www.cyantechnology.com

AN054 – eCOG1X USB Mass Storage with FAT File System

Version 1.1

This application note describes using the eCOG1X microcontroller with a USB Mass Storage Device (such as a flash memory stick) having a FAT 16 or FAT 32 compatible file system. The software can automatically detect and mount standard FAT16/32 partitions and volumes.



Confidential and Proprietary Information

©Cyan Technology Ltd, 2008

This document contains confidential and proprietary information of Cyan Technology Ltd and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.

Revision History

Version	Date	Notes
V1.0	12/06/2007	First release
V1.1	13/08/2007	Update to include notes about powering the USB and updated library.

Contents

1 Introduction 7
2 Glossary 7
3 Overview 7
4 FAT File System Organisation 8
5 Software Library 10
5.1 Installation 10
5.2 Using with CyanIDE 10
5.3 Endian Issues 10
5.4 Error Handling 10
5.5 Limitations 11
6 Examples 12
6.1 Mounting a Partition 12
6.2 Showing the Contents of a Directory 12
6.3 Reading a File 12
6.4 Writing a File 13
6.5 Navigating to a Subdirectory 13
7 USB Power 14
8 Example Software Operation 15
USB_INIT (–-) *15*
MOUNT (n –-) *15*
UNMOUNT (–-) *15*
DIR (–-) *15*
CD <NAME> (–-) *15*
MD <NAME> (–-) *16*
RD <NAME> (–-) *16*
DEL <NAME> (–-) *16*
PRINT <NAME> (–-) *16*
MAKE <NAME> (n –-) *16*
DUMP </PATH/NAME> (–-) *16*
Appendix A API Functions 17
A.1 Initialisation Functions 17
fat_global_init *17*
A.2 Disk Functions 18
fat_format *18*
fat_get_num_partitions *18*

fat_get_partition_info *18*
fat_mount *19*
fat_set_partition_info *19*
fat_unmount *19*
fat_get_free_space *19*
A.3 C-Library Style Functions 20
fat_fclose *20*
fat_fopen *20*
fat_fread *20*
fat_fseek *20*
fat_ftell 21
fat_fwrite *21*
A.4 File and Directory Functions 22
fat_create_directory *22*
fat_create_file *22*
fat_delete_file *22*
fat_delete_directory *22*
fat_expand_file *22*
fat_find_directory_entry *23*
fat_find_entry *23*
fat_get_attributes *23*
fat_get_dir_entry *23*
fat_get_num_dir_entries *23*
fat_read_file *24*
fat_rename *24*
fat_set_attributes *24*
fat_write_file *24*
fat_get_dir_parent *24*
A.5 Utility Functions 25
fat_get_error *25*
fat_filename_to_fatfilename *26*
fat_is_long_filename_entry *26*
fat_is_valid_filename *26*
fat_print_dir_entry *26*
fat_set_dir_entry_date *26*
fat_set_dir_entry_time *27*
fat_set_error *27*

A.6 Callback Functions 28
fat_read_block_raw *28*
fat_write_block_raw *28*
fat_read_file_block *28*
fat_write_file_block *29*

1 Introduction

This application note describes using the eCOG1X microcontroller with a USB Mass Storage Device (such as a flash memory stick) having a FAT 16 or FAT 32 compatible file system. The software can automatically detect and mount standard FAT16/32 partitions and volumes.

2 Glossary

A table of abbreviations and terms used in this document.

BPB	Bios Parameter Block.
Cluster	A collection of Sectors that form the basic storage unit within the FAT file system, allowing FAT types to address more data than the addressable sector count.
eCOG1X	Cyan Technology target micro controller.
EOF	End of File. Used as a marker in the FAT table.
FAT	File Allocation Table.
LBA	Logical Block Addressing.
MBR	Master Boot Record.
Sector	The basic storage unit on a physical disk. Usually 512 Bytes.
UART	Universal Asynchronous Receiver/Transmitter.

3 Overview

The FAT library provides a number of disk, file and directory manipulation routines for FAT16 and FAT32 disk formats. These disk formats are the most commonly used formats for solid-state memory cards that are used with digital cameras, MP3 players and other portable devices.

The library only requires simple block read and write access routines, allowing it to be used on top of lower level disk access routines; for example direct access of SD flash memory cards or control of a Mass Storage Device over USB.

The FAT library offers the following features:

- FAT16 and FAT32 support
- Read and write accesses
- Supports multiple partitions on a disk
- Supports nested subdirectories
- C-library style routines
  (`fopen(), fread(), fwrite(), fclose(), ftell(), fseek()`)
- Fast file block read and write routines
- File and directory delete
- Multiple open files at the same time
- Low level disk access routines
- Quick and full format
- Create and delete partitions
- Low RAM requirement (less than 1KB)
- Low program memory requirement

4 FAT File System Organisation

A typical FAT operating system, for a partitioned volume with a single partition, is organised as shown below. Currently this FAT Library supports only a single partition table in sector 0.

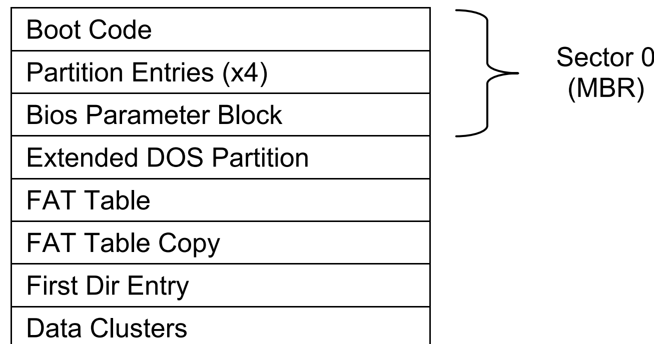


Figure 1. FAT file system organisation

Where a volume is partitioned, the disk or other storage media usually consists of a single partition, although the partition table of the master boot record (MBR) may contain up to four defined partitions. The DOS file system introduced the concept of an extended partition (which overcomes the limitation of a maximum of four partitions in the master boot record) using a singly linked list similar to the FAT table. This list can also be nested, although this is rarely supported. One example is Linux nested partition tables.

Where partitions exist, there are four main components to each partition:

1. The partition boot record is located in the first sector of the partition, and contains the bios parameter block, amongst details of data organisation and pointers to each of the three other components of the FAT file system.
2. The file allocation table (FAT) contains a list of 16 or 32 bit entries (depending on the file system type). Each entry is linked individually to a single allocation unit (cluster) within the data area of a volume, and forms part of a linked list that, in turn, forms the whole of a file entry. Additional special FAT entry markers are defined for EOF, bad, unused or deleted clusters. It is common for the disk to contain a second copy of the FAT, usually immediately after the first copy, for systems where it is possible for the data to become corrupt.
3. The root directory entry. This is a fixed length directory entry for FAT16 volumes, and directly precedes the data area. On FAT32 volumes, the root directory may be anywhere within the data area and is not fixed in length, although it is common for the root directory to immediately follow the FAT.
4. The data area, which contains sequentially numbered allocation units that share a 1:1 correspondence with the FAT entries. Data clusters are a fixed size on any one volume, although they can vary in size between volumes. They are used to store the contents of the files pointed to by the directory entries and FAT chain.

Where a volume is bootable, such as with a hard disk, both the MBR and/or the boot record for the volumes have a set structure and will contain the boot code for the volume. The boot record also contains a bios parameter block (BPB) that holds important information on the volume type, size and organisation.

Each cluster in the data area usually occupies a number of sectors (sectors are commonly 512 bytes in size). The size of each cluster depends on the size of the volume, and is set during formatting such that the entire volume is accessible within the 16 or 28-bit[1] FAT entry. For example, if a 1Gbyte disk is formatted using FAT16, each entry in the FAT table needs to access around 30 sectors of information of 512 bytes each, thus requiring 16K bytes per cluster. The large cluster sizes lead to an effect called *slack* where the final file segment does not completely fill the final cluster in the file chain. FAT32 is much more efficient than FAT16 as it can address a much larger number of clusters, allowing a smaller cluster size to be used.

Directories are stored in the data area of a disk, and always occupy an integer number of clusters in a linked list, much the same as files and FAT entries. Directories are identically structured to files, apart from having an attribute set that indicates it is a directory. Each directory contains a list of 32-byte directory entries that point to the first cluster of the file within the directory and contain file information such as size and creation date. The directory entry also contains a size field to store the byte count, so that the actual file size can be determined. The end of a directory is indicated by a directory entry beginning with the null (0x0) character to indicate that this directory entry is unused. Additionally, the first character of a directory entry (or in the case of long filenames, the first character in *each* directory entry) may be marked with the *deleted* character (0xE5) to indicate that the file has been deleted. A directory or file entry that contains no data (it is empty) uses the value zero as the start cluster. A nested directory entry has the same structure as a file entry, although the parent entry points to the first cluster in the child directory entry. Every directory except the root directory also contains two additional directory entries, termed the *dot* entries, that point to the current directory *('.')* and to the parent directory *('..').*

Long filenames were introduced into the FAT file system to overcome the limited amount of space available for filenames in the directory entries used in DOS[2] (8.3 format: eight character name plus three character extension). For long filenames, several directory entries are combined in a sequential list with illegal file attributes and a sequential sub-entry number. The long directory entry is preceded with a short directory entry to maintain compatibility. Using this method, it is possible for long and short filenames to co-exist within the same volume, while still maintaining compatibility with older operating systems.

---

1 FAT32 only actually uses 28 of the 32 bits in the FAT entry for file access. The remaining four bits are used to determine the type of FAT entry.

2 While long filename entries are supported by most operating systems, this example software does not support this feature due to patent restrictions.

5 Software Library

The FAT disk format is covered in the paper "Microsoft Extensible Firmware Initiative FAT32 File System Specification", published by Microsoft.

5.1 Installation

To install the FAT file system libraries:

- Open the zip file <AN042 FAT Libraries.zip>.
- Click the *Extract* icon, or select *Actions->Extract* from the main menu.
- Check that the *Use folder names* option in the dialogue box is ticked.
- Set the *Extract to:* directory to the CyanIDE peripheral libraries directory.
  Usually this is <*C:\Program Files\Cyan Technology\CyanIDE\libraries*>.
- Click the *Extract* button to extract the files to the CyanIDE peripheral libraries directory.

The zip file contains two copies of the FAT library files, one for eCOG1k and one for eCOG1X.

5.2 Using with CyanIDE

To use the FAT library with CyanIDE, add the library path to both the compiler include and linker library search path lists. See the CyanIDE user manual for instructions on adding additional libraries to projects.

The library functions may then be called by including the following statement at the top of any C source file that needs them:

```
#include <fat_lib.h>
```

Before calling any FAT library functions, the following function must be called in order to initialise the library:

```
fat_global_init()
```

The FAT library functions may now be called.

5.3 Endian Issues

The FAT file system uses little-endian integers, so care must be taken when accessing FAT structures directly. The core_lib library that is provided with CyanIDE contains efficient routines for byte-reversing 16- and 32-bit values: ***swap16()*** and ***swap32()*** respectively. See the core_lib library documentation for more details.

These functions should be present in the Core library directory for the target device, either eCOG1k or eCOG1X. CyanIDE V1.4.1 does not include these functions in the eCOG1k library; copy the files from the eCOG1X core library to the eCOG1k library as the source code is identical for the two devices.

Values returned from library functions are always big-endian and do not need any special treatment.

5.4 Error Handling

Most functions return an error status that indicates whether the function succeeded or failed. For more information about the error, call ***fat_get_error()*** to return a more detailed error code. This error code is updated on every error, but is not cleared on success; it always contains the last non-zero error code. This means that it cannot be used as an alternative to checking the function return values to test for success or failure.

5.5 Limitations

Users should be aware that there are a number of limitations in this implementation of the FAT file system library, in order to keep the required memory footprint to a minimum in both code and data space:

- The maximum number of files that can be opened at any one time is 4.
- The FAT library only supports disks with 512 byte sectors. This is the most common format and applies to almost all disks in use today.
- There is no direct support for long filenames, due to licensing issues.
- For all library functions that return a pointer to data, this data is only valid until the next library call. The FAT library only maintains a single block in memory and this block is likely to be changed with any further disk accesses.
- The C-library style access functions should only be used for the transfer of small amounts of data, as they require frequent accesses to the FAT in order to discover the next cluster in the file. Use ***fat_file_read()*** or ***fat_file_write()*** to perform faster accesses of large files.
- Only a single partition table is supported. Some disks support 4 or more partitions by using linked partition tables. Using linked partitions, only the first 3 are supported. Any of the 4 partitions may be used however.
- Only primary partitions are supported. Logical partitions are not recognised.

6 Examples

This section contains some examples of how to perform basic disk, file and directory access using the FAT library. For more comprehensive examples, see the example project that is included with this application note.

6.1 Mounting a Partition

To mount a partition, use:

```
if (!fat_mount(N))
{
      printf("Unable to mount partition\n\r");
}
```

where N is the partition number (0..3) to be mounted. The application code should first check that the disk contains partitions by checking for a non-negative number as a result of calling ***fat_get_num_partitions***().

6.2 Showing the Contents of a Directory

To display the contents of a directory, use:

```
int numDirectoryEntries;
int index;

// Get the number of entries in the root directory
numDirectoryEntries = fat_get_num_dir_entries(NULL);

// Print each entry out
for (index = 0; index < numDirectoryEntries; index++)
{
    fatDirEntry = fat_get_dir_entry(NULL, index);

    if (!fat_is_long_filename_entry(fatDirEntry))
    {
        fat_print_dir_entry(fatDirEntry);
    }
}
```

This example displays the contents of the root directory. To see how to show the contents of another directory, see "Navigating to a Subdirectory" below.

6.3 Reading a File

The following code reads the first 16 bytes of a file called "README.TXT" in the root directory:

```
fat_file_t *fatFile;
char buffer[16];
unsigned int numBytes;

fatFile = fat_fopen("/README.TXT", "rb+");

if (fatFile)
{
    numBytes = fat_fread(buffer, 1, 16, fatFile);

    printf("Read %d bytes\n\r", numBytes);

    fat_fclose(fatFile);
}
```

6.4 Writing a File

The following example writes the first 16 bytes of a file called "WRITEME.TXT" in a directory called "SUBDIR":

```
fat_file_t *fatFile;
char *buffer = "0123456789ABCDEF";
unsigned int numBytes;

fatFile = fat_fopen("/SUBDIR/WRITEME.TXT", "wb+");

if (fatFile)
{
    numBytes = fat_fwrite(buffer, 1, 16, fatFile);

    printf("Written %d bytes\n\r", numBytes);

    fat_fclose(fatFile);
}
```

6.5 Navigating to a Subdirectory

The following navigates the given path to find nested subdirectories:

```
fat_fat_dir_entry_t *dirEntry;
fat_fat_dir_entry_t dirEntryLocal;

dirEntry = fat_find_directory_entry("/FOLDER/SUBDIR/TEMP", NULL);

// Copy this entry so we can use it later
memcpy(&dirEntryLocal, dirEntry, sizeof(fat_fat_dir_entry_t));
```

It is a good idea to copy the returned FAT directory entry so that it can be used later. Without this copy, the returned data is only valid until the next call to the FAT library. To show this value being used, the "Showing Contents of a Directory" example above could be written as:

```
numDirectoryEntries = fat_get_num_dir_entries(&dirEntryLocal);

for (index = 0; index < numDirectoryEntries; index++)
{
    dirEntry = fat_get_dir_entry(&dirEntryLocal, index);

    if (!fat_is_long_filename_entry(dirEntry))
    {
        fat_print_dir_entry(dirEntry);
    }
}
```

This shows the contents of the /FOLDER/SUBDIR/TEMP directory.

7 USB Power

When acting as a Host, the eCOG1X must supply the VBus power to the peripheral. On the eCOG1X Development Board, this depends on which USB interface is used.

When the external ULPI PHY connection on S6 is used, the MAX5008 power supply is capable of supplying up to 125mA, which is sufficient for most flash memory based USB drives.

For the internal USB PHY connection on S5, the power supply output from the MAX3355 is capable of only 8mA, which is sufficient for OTG devices but not for most USB flash drives. To use this connection with a USB flash drive, it is recommended that a link is placed on the board to connect the USB VBus power directly to the +5V supply. For example, connect J33 pin 1 (VBus on the internal USB connector) and J18 pin 2 (+5V supply).

8 Example Software Operation

The example application uses the eCOG1X and the USB Mass Storage Device (MSD) Host software plugin, included with CyanIDE.

Two project files are given, one using the Internal USB Phy and the other using the External ULPI Phy. When using the Internal USB Phy, please read the USB Power section section above.

The FAT Library is available for both the eCOG1k and eCOG1X microcontrollers. See the Software Library Installation Section above.

The user interface is based on a cut down version of the Command Line Interface described in Application Note AN020.

The user interface provides a basic set of commands that can be used to initialise and mount a partition from a USB MSD, and then navigate the directory structure in that partition, creating and deleting files and directories.

A terminal program (such as HyperTerminal) is required for accessing the Command Line Interface. The terminal software should be configured to 9600baud, 8 data bits, no parity and one stop bit, and connected to the eCOG1X though DUART A on connector P1.

The descriptions in this section have the following format:

```
<Command Name>    ( <Initial Stack> -- <Final Stack> )
```

<Description>

The Command Name is the exact entry in the CLI dictionary.

The two stack states are the values on the stack before and after the command is executed. The top of the stack is to the right in both cases.

The description describes the function in words.

`USB_INIT       ( -- )`

This command initialises the USB MSD connected to the eCOG1X. This must be performed after reset and when a new memory device is connected to the USB.

`MOUNT          ( n -- )`

This command mounts the specified partition on the USB MSD.

`UNMOUNT        ( -- )`

This command unmounts the current partition.

`DIR            ( -- )`

This command displays the current directory contents.

`CD <NAME>      ( -- )`

This command changes to the specified sub directory of the current directory. “CD ..” changes to the parent directory.

MD <NAME> (--)

This command makes a new directory of the specified name in the current directory.

RD <NAME> (--)

This command removed the specified sub directory from the current directory, including any files and sub directories it contains.

DEL <NAME> (--)

This command deletes the specified file from the current directory.

PRINT <NAME> (--)

This command prints the contents of the specified file in the current directory to the terminal.

The implementation of this function gives an example of the use of the ***fat_read_file_block()*** callback function.

MAKE <NAME> (n --)

This command creates a file of the specified name and size (in kilobytes) in the current directory, and fills it with the string "The quick brown fox jumps over the lazy dog".

The implementation of this function gives an example of the use of the ***fat_write_file_block()*** callback function.

DUMP </PATH/NAME> (--)

This command displays a hexadecimal dump of the specified file in the following format:

```
54 68 65 20 71 75 69 63 6B 20 62 72 6F 77 6E 20  The quick brown
66 6F 78 20 6A 75 6D 70 73 20 6F 76 65 72 20 74  fox jumps over t
68 65 20 6C 61 7A 79 20 64 6F 67 0D 0A 54 68 65  he lazy dog..The
```

The implementation of this function gives an example of the use of the C style interface functions. Because of this, the name is taken with a path relative to the root directory, and not the current working directory.

Appendix A API Functions

A.1 Initialisation Functions

fat_global_init

```
#include <fat_lib.h>
void fat_global_init(void)
```

Initialises the FAT library, ready for use.
This function should be called before all other FAT library functions.

A.2 Disk Functions

fat_format

```
#include <fat_lib.h>
BOOL fat_format(unsigned int partition_index,
      const char *volume_name, BOOL full_format,
      BOOL blank_and_verify, unsigned int fat_type)
```

Formats the partition given by *partition_index* as either FAT16 or FAT32. Any mounted partition must be unmounted before performing a format.

Set *volume_name* to point to a volume name, which must be 11 characters or less. Set *volume_name* to NULL to name the disk with the default “NO NAME”.

To perform a quick format, set *full_format* to FALSE; this simply clears the FAT and creates an empty root directory. A full format is performed by setting *full_format* to TRUE; this recreates the partition’s BPB (BIOS Parameter Block), FAT and root directory.

Use quick format for quickly deleting all the files from the disk, full format when formatting a blank partition for the first time. Neither a quick format or a full format sets the data sectors to zero or checks for bad sectors, however the quick format retains any bad sectors that are currently marked in the FAT.

The *blank_and_verify* parameter is not used in the current version of the library and should be set to FALSE.

The fat_type parameter should be set to one of the following values:

Value	Description
FAT_TYPE_FAT16	Format the drive as FAT16
FAT_TYPE_FAT32	Format the drive as FAT32

The function returns TRUE on success, FALSE otherwise. Extended error information is available through the ***fat_get_error()*** call.

fat_get_num_partitions

```
#include <fat_lib.h>
int fat_get_num_partitions(void)
```

Counts the number of partitions on the drive. Returns:

Return value	Description
-1	Drive does not have a valid Master Boot Record (MBR) in sector 0. This is probably an unformatted drive, but may be an error; call `fat_get_error()` to check.
0	No partitions in the partition table, but a valid MBR in sector 0. This will be single partition.
1-4	1-4 partitions in a partition table.

fat_get_partition_info

```
#include <fat_lib.h>
BOOL fat_get_partition_info(unsigned int partition_index,
      fat_partition_table_entry_t *partition_table_entry)
```

Gets the partition table entry from the MBR and places it in *partition_table_entry*. No check is made that this information is valid. The *partition_index* parameter must be in the range 0-3.

fat_mount

```
#include <fat_lib.h>
BOOL fat_mount(unsigned int partition_index)
```

Mounts the partition given by *partition_index*. Returns TRUE on success, FALSE otherwise. The *partition_index* parameter must be in the range 0-3.

fat_set_partition_info

```
#include <fat_lib.h>
BOOL fat_set_partition_info(unsigned int partition_index,
      const fat_partition_table_entry_t *partition_table_entry)
```

Sets the partition table entry to the information given in *partition_table_entry*. No check is made that this information is valid. The *partition_index* parameter must be in the range 0-3.

fat_unmount

```
#include <fat_lib.h>
void fat_unmount(void)
```

Unmounts the currently mounted partition.

fat_get_free_space

```
#include <fat_lib.h>
unsigned long fat_get_free_space(void)
```

Returns the amount of free space on the mounted partition, in bytes. This is always a multiple of the cluster size.

A.3 C-Library Style Functions

---

fat_fclose

```
#include <fat_lib.h>
void fat_fclose(fat_file_t *fat_file)
```

Closes the file *fat_file*.

---

fat_fopen

```
#include <fat_lib.h>
fat_file_t *fat_fopen(char *filename, char *mode)
```

Opens a new file for read or write. The filename should be of the form:

```
/AAAAA/BBBBB/CCCCC/DDDDD.EXT
```

The following modes are valid:

mode	Description
"rb+"	Open a binary file for update (both reading and writing). The original data is left intact and the file seek position is placed at the start of the file. The file must exist for this mode to succeed.
"wb+"	Open a binary file for update (both reading and writing). The original data is deleted and the file seek position is placed at the start of the file. If the file does not exist, a new one is created.

Returns pointer to new *fat_file* on success, NULL on error.

---

fat_fread

```
#include <fat_lib.h>
unsigned int fat_fread(char *buffer, unsigned int size,
      unsigned int num, fat_file_t *fat_file)
```

Reads (*num* x *size*) bytes from *fat_file* to buffer. Returns the number of elements read. If the return value differs from the value of the *num* parameter, this is an error.

---

fat_fseek

```
#include <fat_lib.h>
unsigned int fat_fseek(fat_file_t *fat_file, long offset,
      unsigned int origin)
```

Seeks to the offset in the file, *offset* from the origin. Possible values for the origin are:

origin	Description
FAT_SEEK_SET	Seek to absolute position
FAT_SEEK_CUR	Seek from the current position

Returns zero on success, non-zero otherwise.

fat_ftell

```
#include <fat_lib.h>
long fat_ftell(fat_file_t *fat_file)
```

Returns the current position of the given file stream, or -1 on failure.

fat_fwrite

```
#include <fat_lib.h>
unsigned int fat_fwrite(const char *buffer, unsigned int size,
      unsigned int num, fat_file_t *fat_file)
```

Writes (*num* x *size*) bytes from a buffer into *fat_file*. Returns the number of elements written. If the return value differs from the value of the *num* parameter, this is an error.

A.4 File and Directory Functions

---

fat_create_directory

```
#include <fat_lib.h>
fat_fat_dir_entry_t *fat_create_directory(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry)
```

Creates a new empty directory called *filename*, in the parent directory provided. Set *parent_dir_entry* to NULL to create a new directory in the root directory.

Returns a pointer to the newly created directory on success, NULL on error.

The directory entry pointer that is returned is only valid until the next call to the FAT library; copy this data if you wish to keep it.

---

fat_create_file

```
#include <fat_lib.h>
fat_fat_dir_entry_t *fat_create_file(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry,
      unsigned long filesize)
```

Creates a new file called *filename*, in the parent directory provided. Set *parent_dir_entry* to NULL to create a new file in the root directory. Set *filesize* to the initial size of the file.

Returns a pointer to the newly created file on success, NULL on error.

The directory entry pointer that is returned is only valid until the next call to the FAT library; copy this data if you wish to keep it.

---

fat_delete_file

```
#include <fat_lib.h>
BOOL fat_delete_file(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry)
```

Deletes the given file, in the parent directory provided. Returns TRUE on success, FALSE otherwise.

---

fat_delete_directory

```
#include <fat_lib.h>
BOOL fat_delete_directory(const char *dirname,
      const fat_fat_dir_entry_t *parent_dir_entry)
```

Deletes the given directory, in the parent directory provided. The directory must be empty before deleting. Returns TRUE on success, FALSE otherwise.

---

fat_expand_file

```
#include <fat_lib.h>
BOOL fat_expand_file(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry,
      unsigned long filesize_increase)
```

Adds *filesize_increase* bytes to the file. Returns TRUE on success, FALSE otherwise.

`fat_find_directory_entry`

```
#include <fat_lib.h>
fat_fat_dir_entry_t *fat_find_directory_entry(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry)
```

Finds a diectory. The filename should be of the format:

```
/AAAAA/BBBBB/CCCCC/DDDDD
```

Searches relative to the *parent_dir_entry*. Returns the entry if found, NULL otherwise.

The directory entry pointer that is returned is valid only until the next call to the FAT library; copy this data if you wish to keep it.

`fat_find_entry`

```
#include <fat_lib.h>
fat_fat_dir_entry_t *fat_find_entry(const char *filename,
      unsigned int attributes, unsigned int attributes_mask,
      const fat_fat_dir_entry_t *parent_dir_entry,
      unsigned long *ret_sector_index)
```

Finds an entry with a given filename and attributes. Pass in *attributes_mask* to mask out particular attributes. If *ret_sector_index* is non-NULL, returns the sector index for the directory entry in *ret_sector_index*. Returns the entry if found, NULL otherwise.

The directory entry pointer that is returned is valid only until the next call to the FAT library; copy this data if you wish to keep it.

`fat_get_attributes`

```
#include <fat_lib.h>
unsigned int fat_get_attributes(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry)
```

Returns the attributes for the file, or FAT_FILEATTR_ERROR on any error.

`fat_get_dir_entry`

```
#include <fat_lib.h>
fat_fat_dir_entry_t *fat_get_dir_entry(
      const fat_fat_dir_entry_t *parent_dir_entry, unsigned int index)
```

Returns the dir entry with *index* on success, NULL otherwise. Use this function to iterate through all the entries in a directory. Use ***fat_get_num_dir_entries()*** to find out how many directory entries exist in the directory.

The directory entry pointer that is returned is valid only until the next call to the FAT library; copy this data if you wish to keep it.

`fat_get_num_dir_entries`

```
#include <fat_lib.h>
unsigned int fat_get_num_dir_entries(
      const fat_fat_dir_entry_t* parent_dir_entry)
```

Returns the number of entries in the directory of the disk starting at *cluster_index*. Returns zero on any error. Use ***fat_get_dir_entry()*** to retrieve details about a particular directory entry.

---

fat_read_file

```
#include <fat_lib.h>
BOOL fat_read_file(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry, void *user_ptr)
```

Reads the given file. Calls the ***fat_read_file_block()*** callback function with each sector of the file. Extended information can be given and returned from the callback function with the *user_ptr* pointer.

Returns TRUE on success, FALSE otherwise. If the user stops read, still returns TRUE.

---

fat_rename

```
#include <fat_lib.h>
BOOL fat_rename(const char *from_filename,
      const char *to_filename,
      const fat_fat_dir_entry_t *parent_dir_entry)
```

Renames the given file or directory from *from_filename* to *to_filename*. Returns TRUE on success, FALSE otherwise.

---

fat_set_attributes

```
#include <fat_lib.h>
BOOL fat_set_attributes(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry,
      unsigned int attributes, unsigned int attributes_mask)
```

Sets the attributes for the given file. Only modifies the attributes that are set in the *attributes_mask* parameter. To clear an attribute, set its bit in *attributes_mask* and leave it cleared in the *attributes* parameter.

Returns TRUE on success, FALSE otherwise.

---

fat_write_file

```
#include <fat_lib.h>
BOOL fat_write_file(const char *filename,
      const fat_fat_dir_entry_t *parent_dir_entry, void *user_ptr)
```

Writes the given file using routines that are optimised for creating large files. Calls the ***fat_write_file_block()*** callback function with each sector of the file. Extended information can be given and returned from the callback function with the *user_ptr* pointer.

Returns TRUE on success, FALSE otherwise. If the user stops write, returns FALSE.

---

fat_get_dir_parent

```
#include <fat_lib.h>
BOOL fat_get_dir_parent(const fat_fat_dir_entry_t *dir_entry,
      const fat_fat_dir_entry_t **parent_entry)
```

Finds the parent directory entry of the given *dir_entry*. Note that the *parent_entry* is returned by a double de-referenced pointer, as NULL is a valid return.

Returns TRUE on success, FALSE otherwise.

A.5 Utility Functions

fat_get_error

```
#include <fat_lib.h>
unsigned int fat_get_error(void)
```

Returns the last error number. The following error numbers can be returned:

No.	Error	Description
0	FAT_ERROR_NONE	No error.
1	FAT_ERROR_UNABLETOREADBLOCK	Unable to read block from the disk. May be caused by disk error, or attempting to read a block beyond the number available.
2	FAT_ERROR_UNABLETOWRITEBLOCK	Unable to write block. May be caused by attempting to write to a read-only disk, or a disk error.
3	FAT_ERROR_UNRECOGNISEDFAT	Disk is not FAT16 or FAT32 format.
4	FAT_ERROR_NOFREESPACE	No free space on the disk for the write operation.
5	FAT_ERROR_FILENOTFOUND	Cannot find the file specified.
6	FAT_ERROR_BADBLOCKFOUND	Bad disk block found.
7	FAT_ERROR_NOFREEFILEPTR	Unable to open any more files. The FAT library is limited to FAT_MAX_OPEN_FILES open files at any one time.
8	FAT_ERROR_INVALIDPATH	Unable to follow the given path. This is usually due to missing directories on the disk.
9	FAT_ERROR_INVALIDSECTORSIZE	The FAT library only supports disks with 512 byte sectors.
10	FAT_ERROR_NOMBR	The disk does not have a Master Boot Record. This is usually due to the disk being unformatted or having a non-FAT format.
11	FAT_ERROR_INVALIDFILENAME	The given filename is not valid. Check that it does not contain any invalid characters.
12	FAT_ERROR_ENTRYEXISTS	An entry with this name already exists in the given directory.
13	FAT_ERROR_NOTENOUGHCLUSTERS	There are not enough clusters available to format the disk in the required format. If you attempting to format as FAT32, try formatting as FAT16 instead.
14	FAT_ERROR_PARTITIONMOUNTED	Some operations cannot be performed while a partition is currently mounted. Unmount the partition using fat_unmount() before continuing.
15	FAT_ERROR_PATHTOOLONG	Some operations (such as fat_delete_directory_full()) have a limit to the maximum path length with which they can operate. By default this is 64 characters, but can be extended by #defining a new value for FAT_PATHMAX_LENGTH.

---

fat_filename_to_fatfilename

```
#include <fat_lib.h>
void fat_filename_to_fatfilename(char *fatfilename,
      const char *filename)
```

Converts a filename of the format "123456.ABC" to the fatfilename format "123456 ABC". The *filename* string must be zero terminated, and *fatfilename* is zero terminated.

---

fat_is_long_filename_entry

```
#include <fat_lib.h>
BOOL fat_is_long_filename_entry(
      const fat_fat_dir_entry_t *dir_entry)
```

Returns TRUE if *dir_entry* is part of a long filename, FALSE otherwise.

---

fat_is_valid_filename

```
#include <fat_lib.h>
BOOL fat_is_valid_filename(const char *filename)
```

Returns TRUE if *filename* is a valid short FAT filename, FALSE otherwise.

---

fat_print_dir_entry

```
#include <fat_lib.h>
void fat_print_dir_entry(const fat_fat_dir_entry_t *dir_entry)
```

Prints a directory entry in a nice format, showing filename, file size and attributes.

---

fat_set_dir_entry_date

```
#include <fat_lib.h>
void fat_set_dir_entry_date(unsigned int *date, unsigned int year,
      unsigned int month, unsigned int day)
```

Sets date for the entry. Pass in a pointer to the time entry to be set, either *DIR_CrtDate* (file creation date), *DIR_WrtDate* (file modification date), or *DIR_LstAccDate* (file access date). The following ranges are valid:

value	**range**
year	1980-2107
month	1-12
day	1-31

fat_set_dir_entry_time

```
#include <fat_lib.h>
void fat_set_dir_entry_time(unsigned int *time, unsigned int hours,
      unsigned int minutes, unsigned int seconds)
```

Sets time for the entry. Pass in a pointer to the time entry to be set, either *DIR_CrtTime* (file creation time) or *DIR_WrtTime* (file modification time). The following ranges are valid:

value	range
hours	0-23
minutes	0-59
seconds	0-59

fat_set_error

```
#include <fat_lib.h>
void fat_set_error(unsigned int error_num)
```

Sets the last error number. Usually set only by the FAT library routines.

A.6 Callback Functions

These functions must be implemented by the application to allow the library to access the storage on a sector-by-sector basis.

---

fat_read_block_raw

```
#include <fat_lib.h>
BOOL fat_read_block_raw(unsigned int *block_buffer,
      unsigned long block_index)
```

Called by the FAT library when it wants to read a block from the disk. The callback function should copy the block data into the buffer pointed to by block_buffer. The block buffer is FAT_BLOCKBUFFER_SIZE bytes in size. The block that is required to be read is given by *block_index*.

Return TRUE on success, FALSE to indicate an error.

---

fat_write_block_raw

```
#include <fat_lib.h>
BOOL fat_write_block_raw(const unsigned int *block_buffer,
      unsigned long block_index)
```

Called by the FAT library when it wants to write a block from the disk. The callback function should copy the block data from the buffer pointed to by block_buffer. The block buffer is FAT_BLOCKBUFFER_SIZE bytes in size. The block that is required to be written is given by *block_index*.

Return TRUE on success, FALSE to indicate an error.

---

fat_read_file_block

```
#include <fat_lib.h>
BOOL fat_read_file_block(fat_fat_dir_entry_t *dir_entry,
      unsigned int *block_buffer, unsigned long byte_offset,
      unsigned int bytes_valid, void *user_ptr)
```

Called by the FAT library to handle the data from the read, when reading a file using the optimised ***fat_read_file()*** routine.

Passed the file directory entry of the file that is being read, the number of bytes into the file (*byte_offset*) and the number of valid bytes in the buffer (*bytes_valid*). The data is pointed to by *block_buffer*.

The value *bytes_valid* is always equal to FAT_BLOCKBUFFER_SIZE, except when the last block is read, where it may be less than FAT_BLOCKBUFFER_SIZE. The *user_ptr* argument is a user-defined parameter which is set by calling ***fat_read_file()*** originally.

Return TRUE to continue reading the file, FALSE to stop.

fat_write_file_block

```
#include <fat_lib.h>
BOOL fat_write_file_block(fat_fat_dir_entry_t *dir_entry,
      unsigned int *block_buffer, unsigned long byte_offset,
      unsigned int bytes_valid, void *user_ptr)
```

Called by the FAT library to supply the data for the write, when writing a file using the optimised ***fat_write_file()*** routine.

Passed the file directory entry of the file that is being written, the number of bytes into the file (*byte_offset*) and the number of valid bytes in the buffer (*bytes_valid*). The data is pointed to by *block_buffer*.

The value *bytes_valid* is always equal to FAT_BLOCKBUFFER_SIZE, except when the last block is written, where it may be less than FAT_BLOCKBUFFER_SIZE. The *user_ptr* argument is a user-defined parameter which is set by calling ***fat_write_file()*** originally.

Return TRUE to continue writing the file, FALSE to stop.