# Appendix A - FAT12 overview

FAT12 is the file system that my 32 MB SD card was formatted with (all SD cards > 4 GB are formatted with FAT32). FAT16 is almost identical and FAT32 is quite similar. The 12, 16 and 32 suffixes are the size of the File Allocation Table (FAT) entries. More to that later on. I developed a FAT12 library in which I could only read files. In this overview I want to share with you some of the things that I learned.

Before I write any further I will recommend you to also check Wikipedia and especially this tutorial (the latter explains FAT32 so be aware that there are some differences).

## Creative Commons License



## Master Boot Record

Among the biggest problems I had was that I mixed up the Master Boot Record (MBR) with the Volume Boot Record (VBR).

A Master Boot Record is the 512 byte boot sector which is the first sector (LBA sector 0) of a partitioned storage device, such as a hard drive or in our case an SD card[1]. One of the MBR's tasks is to hold the partition table and that is the only thing we care about. The partition table has four entries (each 16 byte in size) and they tell us where on the device we can find the partitions.

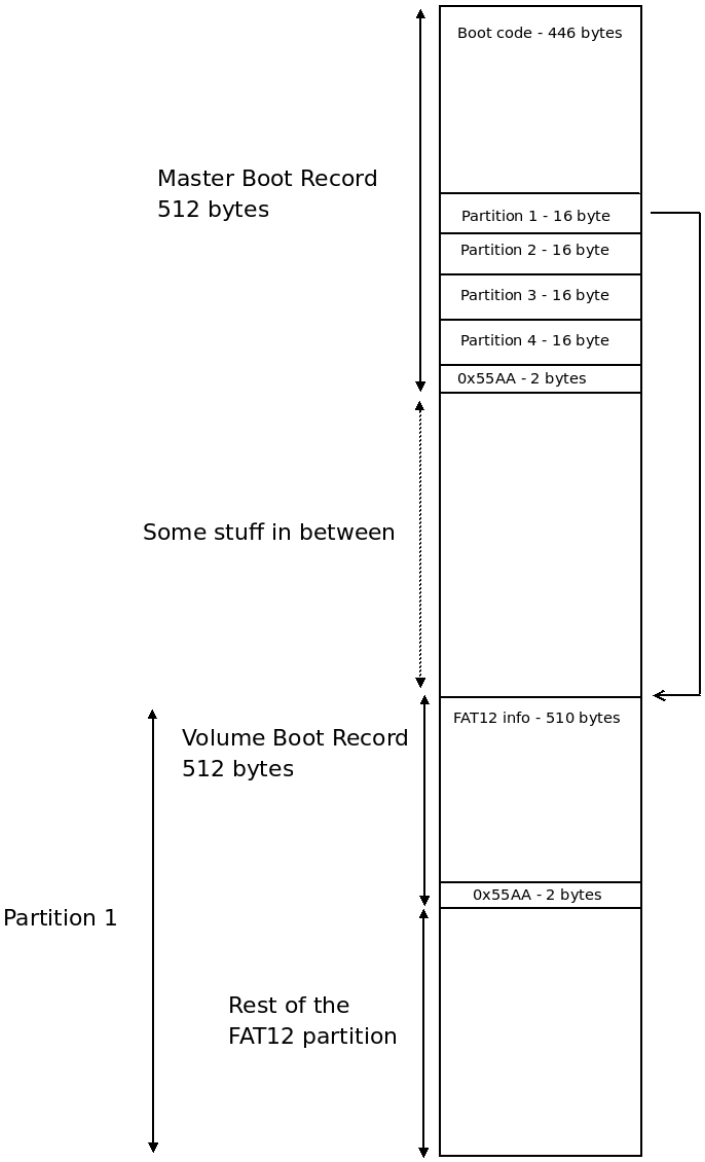
The Volume Boot Record however is the first sector of a non-partitioned storage device (such as a floppy) [2].

In other words, it's easier if you want to read the file system on a non-partitioned device because then you don't have to worry about the MBR. If you have a partitioned device (like an SD card, USB memory stick, etc.) you first have to read the MBR, look at the first partition table entry, jump to the sector it tells you, and there you will find the Volume Boot Record.

To make this as clear as possible, here is an image of the structure of a device which is partitioned.

---

1. http://en.wikipedia.org/wiki/Master_boot_record
2. http://en.wikipedia.org/wiki/Volume_boot_record




So, if you have a partitioned device, you first have to read the Master Boot Record (and look into the partition table entries) to find out where the first partition begins. The partition begins with the Volume Boot Record (aka boot sector, aka Volume ID). If you don't have a partitioned device (such as a floppy disk) then you know that the first sector is the Volume Boot Record.

If you want to examine the contents of an SD card or a USB memory stick by dumping the device content to an image file on disk, make sure that you dump *the whole* device and not only the partition. In Linux, if your USB stick has been mounted from /dev/sda1 do:

$ dd if=/dev/sda of=dump.img

Like that you will get the whole device, starting with the Master Boot Record. If you do the following:

$ dd if=/dev/sda1 of=dump.img

then you will only dump the content of partition1 on the device.

### Partition table entry

Each entry in the partition table is 16 byte in size and has the following structure

| Offset | Field length (bytes) | Description |
|---|---|---|
| 0x00 | 1 | status (0x80 = bootable (*active*), 0x00 = non-bootable, other = invalid) |
| 0x01 | 3 | CHS address of first block in partition.<br>The format is described in the next 3 bytes. |
| 0x01 | 1 | head |
| 0x02 | 1 | sector is in bits 5–0; bits 9–8 of cylinder are in bits 7–6 |
| 0x03 | 1 | bits 7–0 of cylinder |
| **0x04** | **1** | **partition type** |
| 0x05 | 3 | CHS address of last block in partition.<br>The format is described in the next 3 bytes. |
| 0x05 | 1 | head |
| 0x06 | 1 | sector is in bits 5–0; bits 9–8 of cylinder are in bits 7–6 |
| 0x07 | 1 | bits 7–0 of cylinder |
| **0x08** | **4** | **LBA of first sector in the partition** |
| 0x0C | 4 | number of blocks in partition, in little-endian format |

We are interested in the LBA of first sector in the partition. LBA is short for Logical Block Addressing and is in most case the same as sector address. Normally (but not always) one sector or on Logical Block is 512 bytes. If we multiply the LBA with 512 bytes, then we know at what address the first partition starts.

It could also be a good idea to check the partition type. For FAT12 the type is 0x01. FAT16 is 0x04, 0x06 or 0x0E. For FAT32 it's 0x0B or 0x0C.

## General overview of FAT12

The layout of the file system is as follows:

- Reserved sectors. The reserved sectors start with the boot sector (aka Volume ID). Then follows a number of other sectors we don't care about.
- File Allocation Table 1
- File Allocation Table 2 (identical to FAT1)
- Root directory
- Data region

What differs FAT32 from FAT12 and FAT16 may be interesting to know, in FAT32 there is no special place for the root directory, it is stored in the start of the data region. Because FAT32 has no special region for its root directory, its data region starts where FAT12 and

FAT16s root directory would start. So one might say that for all version of FAT, the root directory starts after the File Allocation Tables.

## Volume Boot Record (aka Volume ID)

Now when we know were the partition begins, we know where the FAT file system begins. And the FAT file system begins with a sector called the Volume ID or simply the boot sector. This sector contains a lot of information. Most of it doesn't matter to much for us. The things we care about are in bold. Be aware that we only handle FAT12. Some things like "Sector per file allocation table" are for FAT32 stored in a different place.

| Byte Offset | Length (bytes) | Description |
|---|---|---|
| 0x00 | 3 | Jump instruction. This instruction will be executed and will skip past the rest of the (non-executable) header if the partition is booted from. See Volume Boot Record. If the jump is two-byte near jmp it is followed by a NOP instruction. |
| 0x03 | 8 | OEM Name (padded with spaces). This value determines in which system disk was formatted. MS-DOS checks this field to determine which other parts of the boot record can be relied on. Common values are `IBM  3.3` (with two spaces between the "IBM" and the "3.3"), `MSDOS5.0`, `MSWIN4.1`and `mkdosfs`. |
| **0x0b** | **2** | **Bytes per sector.** A common value is 512, especially for file systems on IDE (or compatible) disks. The *BIOS Parameter Block* starts here. |
| **0x0d** | **1** | **Sectors per cluster.** Allowed values are powers of two from 1 to 128. However, the value must not be such that the number of bytes per cluster becomes greater than 32 KB. |
| **0x0e** | **2** | **Reserved sector count.** The number of sectors before the first FAT in the file system image. Should be 1 for FAT12/FAT16. Usually 32 for FAT32. |
| **0x10** | **1** | **Number of file allocation tables.** Almost always 2. |
| **0x11** | **2** | **Maximum number of root directory entries.** Only used on FAT12 and FAT16, where the root directory is handled specially. Should be 0 for FAT32. This value should always be such that the root directory ends on a sector boundary (i.e. such that its size becomes a multiple of the sector size). 224 is typical for floppy disks. |
| 0x13 | 2 | Total sectors (if zero, use 4 byte value at offset 0x20) |
| 0x15 | 1 | Media descriptor<br>0xF0: 3.5" Double Sided, 80 tracks per side, 18 or 36 sectors per track (1.44MB or 2.88MB). 5.25" Double Sided, 15 sectors per track (1.2MB). Used also for other media types.<br>0xF8: Fixed disk (i.e. Hard disk).<br>0xF9: 3.5" Double sided, 80 tracks per side, 9 sectors per track (720K). 5.25" Double sided, 40 tracks per side, 15 sectors per track (1.2MB)<br>0xFA: 5.25" Single sided, 80 tracks per side, 8 sectors per track (320K)<br>0xFB: 3.5" Double sided, 80 tracks per side, 8 sectors per track (640K)<br>0xFC: 5.25" Single sided, 40 tracks per side, 9 sectors per track (180K)<br>0xFD: 5.25" Double sided, 40 tracks per side, 9 sectors per track (360K). Also used for 8". |

| | | |
|---|---|---|
| | | 0xFE: 5.25" Single sided, 40 tracks per side, 8 sectors per track (160K). Also used for 8".<br>0xFF: 5.25" Double sided, 40 tracks per side, 8 sectors per track (320K)<br>Same value of media descriptor should be repeated as first byte of each copy of FAT. Certain operating systems (MSX-DOS version 1.0) ignore boot sector parameters altogether and use media descriptor value from the first byte of FAT to determine file system parameters. |
| **0x16** | **2** | **Sectors per File Allocation Table** for FAT12/FAT16 |
| 0x18 | 2 | Sectors per track |
| 0x1a | 2 | Number of heads |
| 0x1c | 4 | Hidden sectors |
| 0x20 | 4 | Total sectors (if greater than 65535; otherwise, see offset 0x13) |

From this we can now calculate where the FATs begin, where the root directory begins and where the data region begins:

- Address of first FAT: (Start sector for partition 1 + Reserved sector count) * Bytes per sector
- Address of root directory: Address of first FAT + Number of FATs * Sectors per FAT
- Address of data region: Address of root directory + Maximum number of root directory entries * 32

# Root directory

I will start to explain the root directory. This is because you don't really need the File Allocation Table... if all your files are smaller than the cluster size. However, if your files are bigger than the cluster size, you need the FAT to know where the rest of your file continues so that you can "link them together". But that will be explained later on.

A **directory table** is a special type of file that represents a directory (also known as a folder). Each file or directory stored within it is represented by a 32-byte entry in the table. Each entry records the name, extension, attributes (archive, directory, hidden, read-only, system and volume), the date and time of creation, the address of the first cluster of the file/directory's data and finally the size of the file/directory. Aside from the Root Directory Table in FAT12 and FAT16 file systems, which occupies the special *Root Directory Region* location, all Directory Tables are stored in the Data Region. [3]

The number of entries in the root directory in FAT12 and FAT16 is limited. However, this isn't a problem unless you have a lot of files.

**Note:** To support long file names, a trick has been used. Before each entry, one or multiple entries can be stored. These entries however, will have a file attributes that is 0x0F (a combination that won't occur for real entries). So if you don't care about long file name support, you can just search for an entry that has file attributes that differs from 0x0F.

Directory table entries, both in the Root Directory Region and in sub directories, are of the following format

---

3. http://en.wikipedia.org/wiki/File_Allocation_Table#Directory_table

<table>
<tr><th>Byte Offset</th><th>Length</th><th>Description</th></tr>
<tr><td>0x00</td><td>8</td><td>DOS file name (padded with spaces)<br>The first byte can have the following special values:<table>
<tr><td>0x00</td><td>Entry is available and no subsequent entry is in use</td></tr>
<tr><td>0x05</td><td>Initial character is actually 0xE5. 0x05 is a valid kanji lead byte, and is used for support for filenames written in kanji.</td></tr>
<tr><td>0x2E</td><td>'Dot' entry; either '.' or '..'</td></tr>
<tr><td>0xE5</td><td>Entry has been previously erased and is available. File undelete utilities must replace this character with a regular character as part of the undeletion process.</td></tr>
</table></td></tr>
<tr><td>0x08</td><td>3</td><td>DOS file extension (padded with spaces)</td></tr>
<tr><td>0x0b</td><td>1</td><td>File Attributes<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x01</td><td>Read Only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volume Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Archive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device (internal use only, never found on disk)</td></tr>
<tr><td>7</td><td>0x80</td><td>Unused</td></tr>
</table>An attribute value of 0x0F is used to designate a long file name entry.</td></tr>
<tr><td>0x0c</td><td>1</td><td>Reserved; two bits are used by NT and later versions to encode case information (see below); otherwise 0</td></tr>
<tr><td>0x0d</td><td>1</td><td>Create time, fine resolution: 10ms units, values from 0 to 199.</td></tr>
<tr><td>0x0e</td><td>2</td><td>Create time. The hour, minute and second are encoded according to the following bitmap:<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>15-11</td><td>Hours (0-23)</td></tr>
<tr><td>10-5</td><td>Minutes (0-59)</td></tr>
<tr><td>4-0</td><td>Seconds/2 (0-29)</td></tr>
</table>Note that the seconds is recorded only to a 2 second resolution. Finer resolution for file creation is found at offset 0x0d.</td></tr>
<tr><td>0x10</td><td>2</td><td>Create date. The year, month and day are encoded according to the following bitmap:<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>15-9</td><td>Year (0 = 1980, 127 = 2107)</td></tr>
<tr><td>8-5</td><td>Month (1 = January, 12 = December)</td></tr>
<tr><td>4-0</td><td>Day (1 - 31)</td></tr>
</table></td></tr>
<tr><td>0x12</td><td>2</td><td>Last access date; see offset 0x10 for description.</td></tr>
<tr><td>0x14</td><td>2</td><td>EA-Index (used by OS/2 and NT) in FAT12 and FAT16, High 2 bytes of first cluster number in FAT32</td></tr>
<tr><td>0x16</td><td>2</td><td>Last modified time; see offset 0x0e for description.</td></tr>
<tr><td>0x18</td><td>2</td><td>Last modified date; see offset 0x10 for description.</td></tr>
</table>

| | | |
|---|---|---|
| **0x1a** | **2** | **First cluster in FAT12 and FAT16.** Low 2 bytes of first cluster in FAT32. Entries with the Volume Label flag, subdirectory ".." pointing to root, and empty files with size 0 should have first cluster 0. |
| **0x1c** | **4** | **File size in bytes.** Entries with the Volume Label or Subdirectory flag set should have a size of 0. |

What we care about are the parts in bold:

- The filename
- The file attributes (if it's 0x0F it means that the entry is a long file name entry, skip those if you don't care about long file names)
- First cluster
- File size

So if we want to look up a file, we first have to examine that the entry is "real". So we want to skip all entries that has file attributes = 0x0F because those are fake entries that contains data to support long file name. Those entries have a different format. We also want to check that the file hasn't been deleted. So we check the first byte/character of the filename. If it's 0xE5 it means that the file has been deleted and then the entry is useless.

When we have found an entry which is OK, then we'll have a look at the first cluster. The memory address for the first cluster is calculated as follows, where Cluster number is the number from the directory table entry (First cluster in FAT12 and FAT16):

Start address for file: Address of data region + (Cluster number-2) * Sectors per cluster * Bytes per sector

The "-2" is because the first cluster of the Data region is cluster #2 [4].

Now, if you know that all your files are smaller than the cluster size, you don't have to worry about the File Allocation Table. If you know that, you can stop reading right now. However, if the size of your files are larger, then you have to check the File Allocation Table to see where the file continues. It's all quite simple. Let's have a look at the famous FAT.

## File Allocation Table

As we have said before, the FAT is for checking where your file continues. Since the partition is divided into clusters (of sizes of about 2 KB - 32 KB), a file can either fit inside a cluster or not. We can check the file size in the directory table entry to know if it will fit but we'll need the FAT to know where it continues (if it does). A file can consists of one or multiple clusters. The File Allocation Table is a table of entries that each corresponds to a cluster on the partition. The first entry corresponds to the first cluster, the second entry corresponds to the second entry, etc. This table is used to "chain together" the clusters that the file consists of. Each entry record on of these thing:

| Entry value | Description |
|---|---|
| 0x000 | Free Cluster |
| 0x001 | Reserved value; do not use |

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

| | |
|---|---|
| 0x002–0xFEF | Used cluster; value is the cluster number for the next cluster in the file chain |
| 0xFF0–0xFF6 | Reserved values; do not use. |
| 0xFF7 | Bad sector in cluster or reserved cluster |
| 0xFF8–0xFFF | Last cluster in file |

As you can see, each entry is 12 bit in size. This is strange and annoying since one entry is 1.5 byte long. One must be careful when reading entries from the FAT because it's easy to get it wrong. It is consistently little-endian: if you consider the 3 bytes as one little-endian 24-bit number, the 12 least significant bits are the first entry and the 12 most significant bits are the second[5].

## FAT example - No files

In a file system that has been newly created and has no files in it, the FAT would look like this

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0xFFF** | **0xFF0** | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 |
| 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 |

The first cluster of the Data Region is cluster #2. That leaves the first two entries of the FAT unused. But we don't care about those first two entries because they don't tell us stuff we want to know.

## FAT example - One small file

If we now create a file, file1.txt with the content "Hello", we will get the following table:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0xFFF | 0xFF0 | 0x000 | **0xFFF** | 0x000 | 0x000 | 0x000 | 0x000 |
| 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 |

This means that the file that uses cluster number 3 (that is file1.txt) has no more clusters in its cluster chain. That is only natural since the content "Hello" is of course less than the cluster size (which is approximately 2 KB - 32 KB).

## FAT example - One big file

If we create a that contains a lot of information (or at least a lot more than "Hello") we will get a FAT that looks like:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0xFFF | 0xFF0 | 0x000 | **0x004** | **0x005** | **0x006** | **0x007** | **0x008** |
| **0x009** | **0xFFF** | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 | 0x000 |

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

So, the file starts at cluster number 3 (which corresponds to the 4th entry in the FAT), continues in 4, etc. And we can see that it also uses cluster number 9 but when we look at the entry for that cluster we see 0xFFF which means that the file has no more clusters.

### FAT example - 3 files and fragmented

It's not always so that the files clusters are store adjacent to each other. They may also me *fragmented*. If we create 3 files in this order: file1.txt, file2.txt and file3.txt, we will get the starting cluster numbers as follows:

- Starting cluster for file1.txt: 3
- Starting cluster for file2.txt: 4
- Starting cluster for file3.txt: 5

If we then make file1.txt big so that it spans over many clusters we can see that the cluster chain for file1.txt looks like in this FAT

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0xFFF | 0xFF0 | 0x000 | **0x006** | 0xFFF | 0xFFF | **0x007** | **0x008** |
| **0x009** | **0x00A** | **0x00B** | **0xFFF** | 0x000 | 0x000 | 0x000 | 0x000 |

We can see that file2.txt and file3.txt don't use any more clusters (since the 5th and the 6th entry are 0xFFF).

## Code example for reading the MBR and VBR

Since getting the FAT12 information is the first step if you want to read files, here is some code. This code is under the public domain as far as is lawfully possible.

In C, this code would read the FAT12 information into the program. The function load_int(void *ptr) reads an integer starting at the address ptr and load_short(void *ptr) does the same but with a short.

```
struct {
	/* General Volume Boot Record */
	unsigned short		bytes_per_sector;
	unsigned char		sectors_per_cluster;
	unsigned short		reserved_sector_count;
	unsigned char		number_of_fats;
	unsigned short		max_root_dir_entries;

	unsigned short		sectors_per_fat;
	unsigned int		root_dir_sector;
	unsigned char		fat_type[8];
	unsigned short		signature;

	unsigned int		fat_base;		// What sector does the FATs start at
	unsigned int		data_start_addr;	// What address does the data regions
start at
```

```
} fat12;

char buffer[512];
unsigned char partition1_type;
unsigned int partition1_begin_sector;
unsigned int global_offset;


/* read MBR */
seek_from_start(0);
read_from_io(buffer, 1, 512);

partition1_begin_sector = load_int((buffer+446+8));

global_offset = partition1_begin_sector*512;

/* read VBR into buffer */
seek_from_start(global_offset);
read_from_io(buffer, 1, 512);

/* VBR: volume boot record */
fat12.bytes_per_sector = load_short(buffer+0x0b);
fat12.sectors_per_cluster = buffer[0x0d];
fat12.reserved_sector_count = load_short(buffer+0x0e);
fat12.number_of_fats = buffer[0x10];
fat12.max_root_dir_entries = load_short(buffer+0x11);
fat12.sectors_per_fat = load_short(buffer+0x16);
fat12.signature = load_short(buffer+0x1fe);

/* At what sector does the fat table begin? */
fat12.fat_base = partition1_begin_sector + fat12.reserved_sector_count;

/* At what sector does the root directory start at? */
fat12.root_dir_sector = fat12.fat_base + fat12.sectors_per_fat*fat12.number_of_fats;

/* At what address does the data region start at? */
fat12.data_start_addr = fat12.root_dir_sector*fat12.bytes_per_sector +
fat12.max_root_dir_entries*32;
```