PJRC Store

- 8051 Board, $79
- LCD 128x64 Pixel, $29
- LCD 16x2 Char, $14
- Serial Cable, $5
- 9 Volt Power, $6
- More Components...

8051 Tools

- Main Page
- **Software**
- **PAULMON Monitor**
- **Development Board**
- **Code Library**
  - Serial I/O, Polled
  - Automatic Baud Rate
  - Serial I/O, Intr.
  - Lexer
  - Random Numbers
  - Serial EEPROM
  - AMD 28F256 Flash
  - Xilinx 3000
  - **IDE Hard Drive**
    - Main Page
    - FAT32 Info
    - Peter Faasse
  - Tiny Basic
- **89C2051 Programmer**
- **Other Resources**

Understanding FAT32 Filesystems

This page is intended to help you understand how to access data on Microsoft FAT32 filesystems, commonly used on hard drives ranging in size from 500 megs to hundreds of gigabytes. FAT is a relatively simple and unsophisticated filesystem that is understood by nearly all operating systems, including Linux and MacOS, so it's usually a common choice for firmware-based projects that need to access hard drives. FAT16 and FAT12 are very similar and used on smaller disks. This page will concentrate on FAT32 only (to keep it simple), and briefly mention where these other two are different.

The official FAT specification is available from Microsoft, complete with a software "license agreement". Saddly, the document from Microsoft is hard to read if you do not already understand the FAT filesystem structure, and it lacks information about disk partitioning which also must be dealt with to properly use standard hard drives. While you may find the Microsoft spec useful, this page is meant to "stand alone"... and you can simply read it without suffering through 3 pages of legalese!

However, this page will intentionally "gloss over" many small details and omit many of the finer points, in an attempt to keep it simple and easy to read for anyone faced with learning FAT32 without any previous exposure.

Where To Start... How About At The Beginning?

The first sector of the drive is called the Master Boot Record (MBR). You can read it with LBA = 0. For any new projects, you should not even worry about accessing a drive in CHS mode, as LBA just numbers the sectors sequentially starting at zero, which is much simpler. All IDE drives support accessing sectors using LBA. Also, all IDE drives use sectors that are 512 bytes. Recent Microsoft operating systems refer to using larger sectors, but the drives still use 512 bytes per sector and MS is just treating multiple sectors as if they were one sector. The remainder of this page will only refer to LBA address of 512 byte sectors.

The first 446 bytes of the MBR are code that boots the computer. This is followed by a 64 byte partition table, and the last two bytes are always 0x55 and 0xAA. You should always check these last two bytes, as a simple "sanity check" that the MBR is ok.

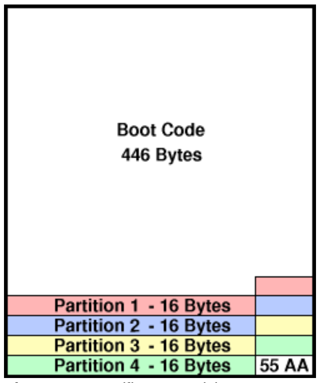


Figure 1: MBR (first sector) layout

The MBR can only represent four partitions. A technique called "extended" partitioning is used to allow more than four, and often times it is used when there are more than two partitions. All we're going to say about extended partitions is that they appear in this table just like a normal partition, and their first sector has another partition table that describes the partitions within its space. But for the sake of simply getting some code to work, we're going to not worry about extended partitions (and repartition and reformat any drive that has them....) The most common scenario is only one partition using the whole drive, with partitions 2, 3 and 4 blank.

Each partition description is just 16 bytes, and the good news is that you can usually just ignore most of them. The fifth byte is a *Type Code* that tells what type of filesystem is supposed to be contained within the partition, and the ninth through twelth bytes indicate the *LBA Begin* address where that partition begins on the disk.

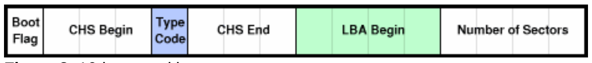


Figure 2: 16-byte partition entry

Normally you only need to check the *Type Code* of each entry, looking for either 0x0B or 0x0C (the two that are used for FAT32), and then read the *LBA Begin* to learn where the FAT32 filesystem is located on the disk.

TODO: add a table of known type codes, with the FAT32 ones colored

The *Number of Sectors* field can be checked to make sure you do not access (particularly write) beyond the end of the space that is allocated for the parition. However, the FAT32 filesystem itself contains information about its size, so this *Number of Sectors* field is redundant. Several of Microsoft's operating systems ignore it and instead rely on the size information embedded within the first sector of the filesystem. (yes, I have experimentally verified this, though unintentionally :) Linux checks the *Number of Sectors* field and properly prevents access beyond the allocated space. Most firmware will probably ignore it.

The Boot Flag, CHS Begin, and CHS End fields should be ignored.

Winhex Warning

Several people have attempted to read the MBR (LBA=0) with Winhex, and actually ended up reading the FAT Volume ID sector. When using

FAT32 Volume ID... Yet Another First Sector

Winhex, sector addressing may be relative to your windows partition.

The first step to reading the FAT32 filesystem is the read its first sector, called the Volume ID. The Volume ID is read using the LBA Begin address found from the partition table. From this sector, you will extract information that tells you everything you need to know about the physical layout of the FAT32 filesystem.

Microsoft's specification lists many variables, and the FAT32 Volume ID is slightly different than the older ones used for FAT16 and FAT12. Fortunately, most of the information is not needed for simple code. Only four variables are required, and three others should be checked to make sure they have the expected values.

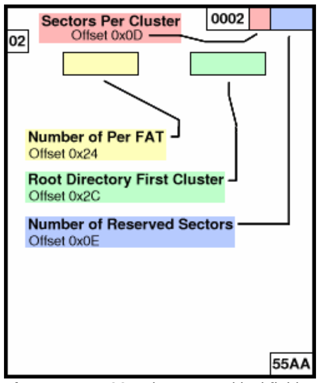


Figure 4: FAT32 Volume ID, critical fields

Field	Microsoft's Name	Offset	Size	Value
Bytes Per Sector	BPB_BytsPerSec	0x0B	16 Bits	Always 512 Bytes
Sectors Per Cluster	BPB_SecPerClus	0x0D	8 Bits	1,2,4,8,16,32,64,128
Number of Reserved Sectors	BPB_RsvdSecCnt	0x0E	16 Bits	Usually 0x20
Number of FATs	BPB_NumFATs	0x10	8 Bits	Always 2
Sectors Per FAT	BPB_FATSz32	0x24	32 Bits	Depends on disk size
Root Directory First Cluster	BPB_RootClus	0x2C	32 Bits	Usually 0x00000002
Signature	(none)	0x1FE	16 Bits	Always 0xAA55

After checking the three fields to make sure the filesystem is using 512 byte sectors, 2 FATs, and has a correct signature, you may want to "boil down" these variables read from the MBR and Volume ID into just four simple numbers that are needed for accessing the FAT32 filesystem. Here are simple formulas in C syntax:

```
(unsigned long)fat_begin_lba = Partition_LBA_Begin + Number_of_Reserved_Sectors;
(unsigned long)cluster_begin_lba = Partition_LBA_Begin + Number_of_Reserved_Sectors + (Number_of_FATs * Sectors_Per_FAT);
(unsigned char)sectors_per_cluster = BPB_SecPerClus;
(unsigned long)root_dir_first_cluster = BPB_RootClus;
```

As you can see, most of the information is needed only to learn the location of the first cluster and the FAT. You will need to remember the size of the clusters and where the root directory is located, but the other information is usually not needed (at least for simply reading files).

If you compare these formulas to the ones in Microsoft's specification, you should notice two differences. They lack "RootDirSectors", because FAT32 stores the root directory the same way as files and subdirectories, so RootDirSectors is always zero with FAT32. For FAT16 and FAT12, this extra step is needed to compute the special space allocated for the root directory.

Microsoft's formulas do not show the "Partition_LBA_Begin" term. Their formulas are all relative to the beginning of the filesystem, which they don't explicitly state very well. You must add the "Partition_LBA_Begin" term found from the MBR to compute correct LBA addresses for the IDE interface, because to the drive the MBR is at zero, not the Volume ID. Not adding Partition_LBA_Begin is one of the most common errors most developers make, so especially if you are using Microsoft's spec, do not forget to add this for correct LBA addressing.

The rest of this page will usually refer to "fat_begin_lba", "cluster_begin_lba", "sectors_per_cluster", and "root_dir_first_cluster", rather than the individual fields from the MBR and Volume ID, because it is easiest to compute these numbers when starting up and then you no longer need all the details from the MBR and Volume ID.

How The FAT32 Filesystem Is Arranged

The layout of a FAT32 filesystem is simple. The first sector is always the Volume ID, which is followed by some unused space called the reserved sectors. Following the reserved sectors are two copies of the FAT (File Allocation Table). The remainder of the filesystem is data arranged in "clusters", with perhaps a tiny bit of unused space after the last cluster.

Bad Sectors

In the old days, disk drives had "bad" sectors. Brand new drives would often have several bad sectors, and as the drive was used (or abused), additional sectors would become unusable.

The FAT filesystems are designed to handle bad sectors. This is done by

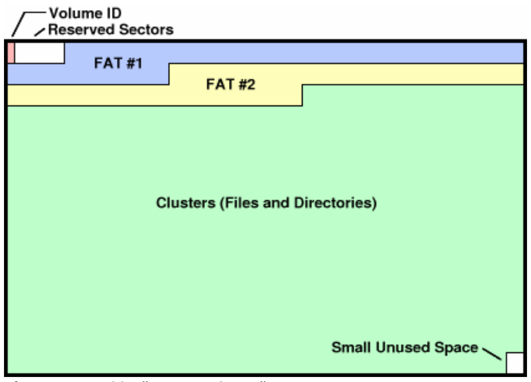


Figure 5: FAT32 Filesystem Overall Layout

The vast majority of the disk space is the clusters section, which is used to hold all the files and directories. The clusters begin their numbering at 2, so there is no cluster #0 or cluster #1. To access any particular cluster, you need to use this formula to turn the cluster number into the LBA address for the IDE drive:

```
lba_addr = cluster_begin_lba + (cluster_number - 2) * sectors_per_cluster;
```

Normally clusters are at least 4k (8 sectors), and sizes of 8k, 16k and 32k are also widely used. Some later versions of Microsoft Windows allow using even larger cluster sizes, by effectively considering the sector size to be some mulitple of 512 bytes. The FAT32 specification from Microsoft states that 32k is the maximum cluster size.

Now If Only We Knew Where The Files Were....

When you begin, you only know the first cluster of the root directory. Reading the directory will reveal the names and first cluster location of other files and subdirectories. A key point is that **directories only tell you how to find the first cluster number of their files and subdirectories**. You also obtain a variety of other info from the directory such as the file's length, modification time, attribute bits, etc, but a directory only tells you where the files begin. To access more than the first cluster, you will need to use the FAT. But first we need to be able to find where those files start.

In this section, we'll only briefly look at directories as much as is necessary to learn where the files are, then we'll look at how to access the rest of a file using the FAT, and later we'll revisit directory structure in more detail.

Directory data is organized in 32 byte records. This is nice, because any sector holds exactly 16 records, and no directory record will ever cross a sector boundry. There are four types of 32-byte directory records.

1. **Normal record with short filename** - Attrib is normal
2. **Long filename text** - Attrib has all four type bits set
3. **Unused** - First byte is 0xE5
4. **End of directory** - First byte is zero

Unused directory records are a result of deleting files. The first byte is overwritten with 0xE5, and later when a new file is created it can be reused. At the end of the directory is a record that begins with zero. All other records will be non-zero in their first byte, so this is an easy way to determine when you have reached the end of the directory.

Records that do not begin with 0xE5 or zero are actual directory data, and the format can be determined by checking the Attrib byte. For now, we are only going to be concerned with the normal directory records that have the old 8.3 short filename format. In FAT32, all files and subdirectories have short names, even if the user gave the file a longer name, so you can access all files without needing to decode the long filename records (as long as your code simply ignores them). Here is the format of a normal directory record:

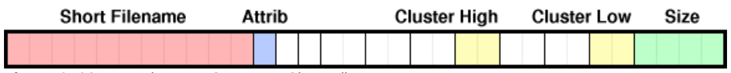


Figure 6: 32 Byte Directory Structure, Short Filename Format

Field	Microsoft's Name	Offset	Size
Short Filename	DIR_Name	0x00	11 Bytes
Attrib Byte	DIR_Attr	0x0B	8 Bits
First Cluster High	DIR_FstClusHI	0x14	16 Bits
First Cluster Low	DIR_FstClusLO	0x1A	16 Bits
File Size	DIR_FileSize	0x1C	32 Bits

storing two identical copies of the File Allocation Table. The idea is that if a sector within one FAT becomes bad, the corresponding sector in the other FAT will (hopefully) still be good.

Bad sectors within the clusters are handled by storing a special code within the FAT entry that represents the cluster that contains the bad sector. If a file happened to be using that cluster, well, you lost part of the file's data, but at least that cluster would be marked as bad so that it would never be used again. The initial formatting of the disk would write and read every sector and mark the bad clusters before they could be used, so data loss could only occur when a previously-good sector became bad.

Fortunately, bad sectors today are only a distant memory, at least on hard drives. Modern drives still internally experience media errors, but they store Reed-Solomon error correcting codes for every sector. Sector data can be interleaved and distributed over a wide area, so a physical defect (localized to one place) can damage only a small part of many sectors, rather than all of a few sectors. The error correction can easily recover the few missing bits of each sector. Error correction also dramatically increases storage capacity (despite using space to store redundant data), because the bits can be packed so close together that a small number of errors begin to occur, and the error correction fixes them.

All modern drives also include extra storage capacity that is used to automatically remap damaged sectors. When a sector is read and too many of the bits needed error correction, the controller in the drive will "move" that sector to a fresh portion of the space set aside for remapping sectors. Remapping also dramatically reduces the cost of disk drives, because small but common defects in manufacturing don't impact overall product quality.

Because all modern drives use sophisticated error correction to automatically detect and (almost always) correct for media errors, bad sectors are virtually never seen at the IDE interface level.

The Attrib byte has six bits defined, as shown in the table below. Most simple firmware will check the Attrib byte to determine if the 32 bytes are a normal record or long filename data, and to determine if it is a normal file or a subdirectory. Long filename records have all four of the least significant bits set. Normal files rarely have any of these four bits set.

Attrib Bit	Function	LFN	Comment
0 (LSB)	Read Only	1	Should not allow writing
1	Hidden	1	Should not show in dir listing
2	System	1	File is operating system
3	Volume ID	1	Filename is Volume ID
4	Directory	x	Is a subdirectory (32-byte records)
5	Archive	x	Has been changed since last backup
6	Ununsed	0	Should be zero
7 (MSB)	Ununsed	0	Should be zero

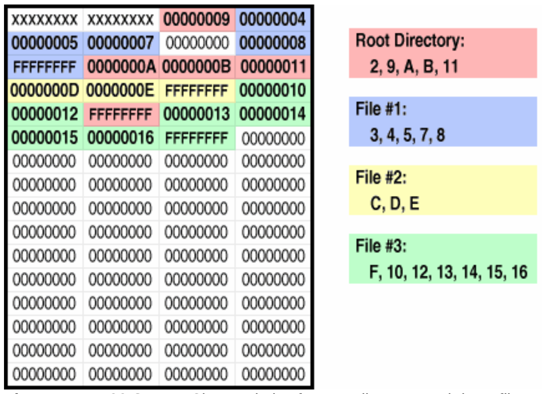
The remaining fields are relatively simple and straigforward. The first 11 bytes are the short filename (old 8.3 format). The extension is always the last three bytes. If the file's name is shorter than 8 bytes, the unused bytes are filled with spaces (0x20). The starting cluster number is found as two 16 bit sections, and the file size (in bytes) is found in the last four bytes of the record. In both, the least significant byte is first. The first cluster number tells you where the file's data begins on the drive, and the size field tells you how long the file is The actual space allocated on the disk will be an integer number of clusters, so the file size lets you know how much of the last cluster is the file's data.

File Allocation Table - Following Cluster Chains

The directory entries tell you where the first cluster of each file (or subdirectory) is located on the disk, and of course you find the first cluster of the root directory from the volume ID sector. For tiny files and directories (that fit inside just one cluster), the only information you obtain from the FAT is that there are no more clusters for that file. To access all the other clusters of a file beyond the first one, you need to use the File Allocation Table. The name FAT32 refers to this table, and the fact that each entry of the table is 32 bits. In FAT16 and FAT12, the entries are 16 and 12 bits. FAT16 and FAT12 work the same way as FAT32 (with the unpleasant exception that 12 bit entries do not always fall neatly within sector boundries, but 16 and 32 bit entries never cross sector boundries). We're only going to look at FAT32.

The File Allocation Table is a big array of 32 bit integers, where each one's position in the array corresponds to a cluster number, and the value stored indicates the next cluster in that file. The purpose of the FAT is to tell you where the next cluster of a file is located on the disk, when you know where the current cluster is at. Every sector of of the FAT holds 128 of these 32 bit integers, so looking up the next cluster of a file is relatively easy. Bits 7-31 of the current cluster tell you which sectors to read from the FAT, and bits 0-6 tell you which of the 128 integers in that sector contain is the number of the next cluster of your file (or if all ones, that the current cluster is the last).

Here is a visual example of three small files and a root directory, all allocated near within first several clusters (so that one one sector of the FAT is needed for the drawing). Please note that in this example, the root directory is 5 clusters... a very unlikely scenario for a drive with only 3 files. The idea is to show how to follow cluster chains, but please keep in mind that a small root directory (in only 1 cluster) would have "FFFFFFFF" at position 2 (or whereever the volume ID indicated as the first cluster), rather than "00000009" leading to even more clusters. However, with more files, the root directory would likely span several clusters, and rarely would a root directory occupy consecutive clusters due to all the allocation for files written before the directory grew to use more clusters. With that caveat in mind, here's that simple example:



xxxxxxxx	xxxxxxxx	00000009	00000004
00000005	00000007	00000000	00000008
FFFFFFFF	0000000A	0000000B	00000011
0000000D	0000000E	FFFFFFFF	00000010
00000012	FFFFFFFF	00000013	00000014
00000015	00000016	FFFFFFFF	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Root Directory:
2, 9, A, B, 11

File #1:
3, 4, 5, 7, 8

File #2:
C, D, E

File #3:
F, 10, 12, 13, 14, 15, 16

Figure 7: FAT32 Sector, Cluster chains for root directory and three files

In this example, the root directory begins at cluster 2, which is learned from the Volume ID. The number at position 2 in the FAT has a 9, so the second cluster is #9. Likewise, clusters A, B, and 11 hold the remainder of the root directory. The number at position 11 has 0xFFFFFFFF which indicates that this is the last cluster of the root directory. In practice, your code may not even attempt to read position 11 because the an end-of-directory marker (first of 32 bytes is zero) would be found. Likewise, a filesystem with a small root directory might fit entirely into one cluster, and the FAT could not be needed at all to read such a directory. But if you reach the end of the 32 byte entries within the first cluster without finding the end-of-directory marker, then the FAT must be used to find the remaining clusters.

Similarly, three files are shown. In each case, the FAT gives no indication of which cluster is the first... that must be extracted from the directory, and then the FAT is used to access subsequent clusters of the file. The number of clusters allocated to the file should always be enough to hold the number of bytes specified by the size field in the directory. A portion of the last cluster will be unused, except in the rare case where the file's size is an exact multiple of the cluster size. Files that are zero length do not have any clusters allocated to them, and the cluster number in the directory should be zero. Files that fit into just one cluster will have only the "FFFFFFFF" end of file marker in the FAT at their cluster position.

Shortcut Hint: One approach to the simplest possible code is to keep the root directory very small (few files, only 8.3 filenames), avoid subdirectories, and run the "defrag" program on a PC. That way, the root directory is found in just one cluster and every file occupies only consecutive clusters. Though this approach is very limiting, it means you can read files without using the FAT at all.

According to Microsoft's spec, the cluster numbers are really only 28 bits and the upper 4 bits of a cluster are "reserved". You should clear those top for bits to zeros before depending on a cluster number. Also, the end-of-file number is actually anything equal to or greater than 0xFFFFFFF8, but in practive 0xFFFFFFFF is always written. Zeros in the FAT mark clusters that are free space. Also, please remember that the cluster numbers are stored with their least significant byte first (Figure 7 shows them human-readable formatted, but they are actually stored LSB first).

The good news for a firmware designer working with limited resources is that FAT is really very simple. However, FAT's simplicity is also its weakness as a general purpose file system for high performance computing. For example, to append data to a file, the operating system must traverse the entire cluster chain. Seeking to random places within a file also requires many reads within the FAT. For the sake of comparison, unix filesystems use a tree-like structure, where a cluster (also called a "block") has either a list of blocks (often called "inode" in unix terminology), or a list of other inodes which in turn point to blocks. In this manner, the location on disk of any block can be found by reading 1, 2, or 3 (for huge files) inodes. The other weakness of FAT, which is mitigated by caching, is that it is physically located near the "beginning" of the disk, rather than physically distributed. Unix inode-based filesystems scatter the inode blocks evenly among the data blocks, and attempt to describe the location of data using nearby inode blocks.

Directories and Long Filenames In Detail

> TODO: write this section someday.... the basic idea is that a bunch of 32-byte entries preceeding the normal one are used to hold the long name that corresponds to that normal directory record.

Where's the Free Code ???

At this time, I do not have a free general-purpose FAT32 code library for you to easily drop into your project.

However, you can find FAT32 code in the mp3 player project. The 0.1.x and 0.5.x firmware used a simple one-sector-at-a-time approach that is very easy to understand and only needs a single 512 byte buffer, but is not very sophisticated. These old firmware revs do not properly follow cluster chains for file (they do for directories), so it is necessary to run defrag before using the drive with them.

The 0.6.x mp3 player firmware does use the FAT properly to follow cluster chains. It uses megabytes of memory from the SIMM to buffer cluster and sectors from the FAT. A table of recently read FAT sectors is maintained, and a binary search is implemented to quickly locate the memory associated with any cached FAT sector. The firmware manages the megabytes of RAM by dividing it into 4k chunks (which works together with the bank swapping implemented by the FPGA), and the first 32 hold a big array of 16-byte structs with info about what is held in each 4k block. These structs implement linked lists that for each file cached in memory. A file I/O API is built on top of this to allow the main program to access the cached data. The 0.6.x firmware also uses DMA, so data transfers are quite fast.

Both of the MP3 player project FAT32 implementations are read-only. Both are GPL'd, but both are quite specific to that project. You can choose either very simple without features and needing defrag, or quite complex with lots of features but heavily dependent on the FPGA and SIMM.

Perhaps someday I will write a more general purpose FAT32 implementation with nice documentation for this code library section. But this is not planned for the near future. Hopefully the description on this page will at least help you to understand how FAT32 works so you can write your own code or understand the project-specific code in the mp3 players.

Other Sites With FAT Filesystem Code For Microcontrollers

- Rob's Projects - FAT32 File I/O Library. Intended for AVR, but the code is in C.
- Nassif's ZipAmp - A mp3 player with FAT16 and FAT32 code.

Please Help....

If you have found this web page helpful, please create a direct link on your own project webpage (hopefully with at least "FAT32" in your link text). Visitors to your site will be able to find this page, and your link will help search engines to direct other developers searching for FAT32 information to this page.

Understanding FAT32 can be difficult, especially if you can only find poorly written documentation like Microsoft's specification. I wrote this page to help bridge the gap and make understanding FAT32 easier. Please help make it easier for others to find it.

Thanks.

Understanding FAT32 Filesystems, Paul Stoffregen
http://www.pjrc.com/tech/8051/ide/fat32.html
Last updated: February 24, 2005
Suggestions, comments, criticisms: <paul@pjrc.com>