FAT

The **File Allocation Table** (**FAT**) was the native file system of MS-DOS. FAT was originally introduced by Marc McDonald in Stand-alone Disk BASIC with 8-bit FAT entries and 16 byte directory entries. The better known FAT12 variant, with 12-bit FAT entries and 32 byte directory entries, was introduced with DOS. FAT is a very simple file system -- nothing more than a singly-linked list of clusters in a gigantic table. A FAT file system uses very little memory (unless the OS caches the whole allocation table in memory) and is one of, if not the, most basic file system in use today.

Filesystems

Virtual Filesystems

VFS

Disk Filesystems

FAT 12/16/32, VFAT, ExFAT
Ext 2/3/4
LEAN
HPFS
NTFS
HFS
HFS+
MFS
ReiserFS
FFS (Amiga)
FFS (BSD)/UFS
BeFS
BFS
XFS
SFS
ZDSFS
ZFS
USTAR

CD/DVD Filesystems

ISO 9660
Joliet
UDF

Network Filesystems

NFS
RFS
AFS

Flash Filesystems

JFFS2
YAFFS

Contents

Overview

There are several different versions of the FAT file system. Each version was designed for a different size of storage media.

FAT 12

FAT 12 was designed for floppy disks and can manage a maximum size of 16 megabytes because it uses 12 bits to address the clusters.

FAT 16

FAT 16 was designed for early hard disks and could handle a maximum size of 64K clusters * the cluster size. The larger the hard disk, the larger the cluster size would be, which leads to large amounts of "slack space" on the disk.

FAT 32

FAT 32 was introduced to us by Windows95-B and Windows98. FAT32 solved some of FAT's problems. No more 64K max clusters! Although FAT32 uses 32 bits per FAT entry, only the bottom 28 bits are actually used to address clusters on the disk (top 4 bits are reserved). With 28 bits per FAT entry, the filesystem can address a maximum of about 270 million clusters in a partition. This enables very large hard disks to still maintain reasonably small cluster sizes and thus reduce slack space between files.

ExFAT

Main article: ExFAT

ExFAT is the filesystem used on SDXC cards, created by Microsoft. It is FAT32 with actually 32 bits per FAT entry, with the ability to indicate a file is fully consecutive on disk (allowing you to skip reading the FAT), some more advanced features and a fully redesigned file entry system. Since it's so similar to FAT32, please merge any bits of info from the exFAT article into this one.

Microsoft has published the official specification at https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification .

VFAT

VFAT is an extension to the FAT file system that has the ability to use long filenames (up to 255 characters). First introduced by Windows 95, it uses a "kludge" whereby long filenames are marked with a "volume label" attribute and filenames are subsequently stored in 11 byte chunks in sequential directory entries. (This is a bit of an oversimplification, but close enough).

Implementation Details

The FAT file system views the storage media as a flat array of clusters. If the physical media does not address its data as a flat list of sectors (really old hard disks and floppy disks) then the cluster numbers will need to be translated before being sent to the disk. The storage media is organized into three basic areas.

- The boot record
- The File Allocation Table (FAT)
- The directory and data area

Boot Record

The boot record occupies one sector, and is always placed in logical sector number zero of the "partition". If the media is not divided into partitions, then this is the beginning of the media. This is the easiest sector on the partition for the computer to locate when it is loaded. If the storage media is partitioned (such as a hard disk), then the beginning of the actual media contains an MBR (x86) or other form of partition information. In this case each partition's first sector holds a Volume Boot Record.

BPB (BIOS Parameter Block)

The boot record contains both code and data, mixed together. The data that isn't code is known as the BPB.

Offset (decimal)	Offset (hex)	Size (in bytes)	Meaning
0	0x00	3	The first three bytes EB 3C 90 disassemble to JMP SHORT 3C NOP. (The 3C value may be different.) The reason for this is to jump over the disk format information (the BPB and EBPB). Since the first sector of the disk is loaded into ram at location 0x0000:0x7c00 and executed, without this jump, the processor would attempt to execute data that isn't code. Even for non-bootable volumes, code matching this pattern (or using the E9 jump opcode) is required to be present by both Windows and OS X. To fulfil this requirement, an infinite loop can be placed here with the bytes EB FE 90.
3	0x03	8	OEM identifier. The first 8 Bytes (3 - 10) is the version of DOS being used. The next eight Bytes 29 3A 63 7E 2D 49 48 and 43 read out the name of the version. The official FAT Specification from Microsoft says that this field is really meaningless and is ignored by MS FAT Drivers, however it does recommend the value "MSWIN4.1" as some 3rd party drivers supposedly check it and expect it to have that value. Older versions of dos also report MSDOS5.1, linux-formatted floppy will likely to carry "mkdosfs" here, and FreeDOS formatted disks have been observed to have "FRDOS5.1" here. If the string is less than 8 bytes, it is padded with spaces.
11	0x0B	2	The number of Bytes per sector (remember, all numbers are in the little-endian format).
13	0x0D	1	Number of sectors per cluster.
14	0x0E	2	Number of reserved sectors. The boot record sectors are included in this value.
16	0x10	1	Number of File Allocation Tables (FAT's) on the storage media. Often this value is 2.
17	0x11	2	Number of root directory entries (must be set so that the root directory occupies entire sectors).
19	0x13	2	The total sectors in the logical volume. If this value is 0, it means there are more than 65535 sectors in the volume, and the actual count is stored in the Large Sector Count entry at 0x20.
21	0x15	1	This Byte indicates the media descriptor type (https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#BPB20_OFS_0Ah).
22	0x16	2	Number of sectors per FAT. FAT12/FAT16 only.
24	0x18	2	Number of sectors per track.
26	0x1A	2	Number of heads or sides on the storage media.
28	0x1C	4	Number of hidden sectors. (i.e. the LBA of the beginning of the partition.)
32	0x20	4	Large sector count. This field is set if there are more than 65535 sectors in the volume, resulting in a value which does not fit in the *Number of Sectors* entry at 0x13.

Note: the "geometry" of the media (sectors per track, heads, and perhaps the number of bytes in a sector) is not necessarily known correctly by the program that originally formats the media. Also, if the media is moved (from the computer that formatted it) to another machine with a different BIOS -- then the new BIOS may specify a different geometry for the same media. So it is generally a very bad idea to trust the "SPT" or "heads" numbers. Get them from the BIOS instead, if possible.

Note2: many of the values in the BPB are not correctly "aligned". That is, word-sized values are not stored on word ("even" address) boundaries. On some architectures, accessing misaligned words may cause the code to crash. Making a copy of the BPB (somewhere else in memory and shifted up one byte) may solve the problem.

Extended Boot Record

The extended boot record information comes right after the BPB. The data at the beginning is known as the EBPB. It contains different information depending on whether this partition is a FAT 12, FAT 16, or FAT 32 filesystem. Immediately following the EBPB is the actual boot code, then the standard 0xAA55 boot signature, to fill out the 512-byte boot sector. Offsets shows are from the start of the standard boot record.

FAT 12 and FAT 16

Offset (decimal)	Offset (hexadecimal)	Length (in bytes)	Meaning
36	0x024	1	Drive number. The value here should be identical to the value returned by BIOS interrupt 0x13, or passed in the DL register; i.e. 0x00 for a floppy disk and 0x80 for hard disks. This number is useless because the media is likely to be moved to another machine and inserted in a drive with a different drive number.

37	0x025	1	Flags in Windows NT. Reserved otherwise.
38	0x026	1	Signature (must be 0x28 or 0x29).
39	0x027	4	VolumeID 'Serial' number. Used for tracking volumes between computers. You can ignore this if you want.
43	0x02B	11	Volume label string. This field is padded with spaces.
54	0x036	8	System identifier string. This field is a string representation of the FAT file system type. It is padded with spaces. The spec says never to trust the contents of this string for any use.
62	0x03E	448	Boot code.
510	0x1FE	2	Bootable partition signature 0xAA55.

FAT 32

Offset (decimal)	Offset (hexadecimal)	Length (in bytes)	Meaning
36	0x024	4	Sectors per FAT. The size of the FAT in sectors.
40	0x028	2	Flags.
42	0x02A	2	FAT version number. The high byte is the major version and the low byte is the minor version. FAT drivers should respect this field.
44	0x02C	4	The cluster number of the root directory. Often this field is set to 2.
48	0x030	2	The sector number of the FSInfo structure.
50	0x032	2	The sector number of the backup boot sector.
52	0x034	12	Reserved. When the volume is formated these bytes should be zero.
64	0x040	1	Drive number. The values here are identical to the values returned by the BIOS interrupt 0x13. 0x00 for a floppy disk and 0x80 for hard disks.
65	0x041	1	Flags in Windows NT. Reserved otherwise.
66	0x042	1	Signature (must be 0x28 or 0x29).
67	0x043	4	Volume ID 'Serial' number. Used for tracking volumes between computers. You can ignore this if you want.
71	0x047	11	Volume label string. This field is padded with spaces.
82	0x052	8	System identifier string. Always "FAT32 ". The spec says never to trust the contents of this string for any use.
90	0x05A	420	Boot code.
510	0x1FE	2	Bootable partition signature 0xAA55.

FSInfo Structure (FAT32 only)

Offset (decimal)	Offset (hexadecimal)	Length (in bytes)	Meaning
0	0x0	4	Lead signature (must be 0x41615252 to indicate a valid FSInfo structure)
4	0x4	480	Reserved, these bytes should never be used
484	0x1E4	4	Another signature (must be 0x61417272)
488	0x1E8	4	Contains the last known free cluster count on the volume. If the value is 0xFFFFFFFF, then the free count is unknown and must be computed. However, this value might be incorrect and should at least be range checked (<= volume cluster count)

492	0x1EC	4	Indicates the cluster number at which the filesystem driver should start looking for available clusters. If the value is 0xFFFFFFFF, then there is no hint and the driver should start searching at 2. Typically this value is set to the last allocated cluster number. As the previous field, this value should be range checked.
496	0x1F0	12	Reserved
508	0x1FC	4	Trail signature (0xAA550000)

exFat boot record

For exFAT the whole boot record was recreated from scratch instead of extending the existing FAT12/16/32 boot records even further. You can recognize exFAT by noticing that in the FAT12/16/32 boot record, the "bytes per sector" is zero.

Offset (decimal)	Offset (hex)	Size (in bytes)	Meaning
0	0x00	3	The first three bytes EB 3C 90 disassemble to JMP SHORT 3C NOP. (The 3C value may be different.) The reason for this is to jump over the disk format information (the BPB and EBPB). Since the first sector of the disk is loaded into ram at location 0x0000:0x7c00 and executed, without this jump, the processor would attempt to execute data that isn't code. Even for non-bootable volumes, code matching this pattern (or using the E9 jump opcode) is required to be present by both Windows and OS X. To fulfil this requirement, an infinite loop can be placed here with the bytes EB FE 90.
3	0x03	8	OEM identifier. This contains the string "EXFAT ". Not to be used for filesystem determination, but it's a nice hint.
11	0x0B	53	Set to zero. This makes sure any FAT driver will not be able to load it.
64	0x40	8	Partition offset. No idea why the partition itself would have this, but it's here. Might be wrong. Probably best to just ignore.
72	0x48	8	Volume length.
80	0x50	4	FAT offset (in sectors) from start of partition.
84	0x54	4	FAT length (in sectors).
88	0x58	4	Cluster heap offset (in sectors).
92	0x5C	4	Cluster count
96	0x60	4	Root directory cluster. Typically 4 (but just read this value).
100	0x64	4	Serial number of partition.
104	0x68	2	Filesystem revision
106	0x6A	2	Flags
108	0x6C	1	Sector shift
109	0x6D	1	Cluster shift
110	0x6E	1	Number of FATs
111	0x6F	1	Drive select
112	0x70	1	Percentage in use
113	0x71	7	Reserved (set to 0).

To read the filesystem, find out how big a 'sector' and a 'cluster' are. A sector is (1 << sectorshift) bytes, a cluster is (1 << (sectorshift + clustershift)) bytes. Then, find the start of the FAT and the start of the cluster heap (note that the first cluster is *still* cluster 2).

```
// This allows you to zero-index clusters:
uint64_t clusterArray = clusterheapoffset * sectorsize - 2 * clustersize;
uint64_t fatOffset = fatoffset * sectorsize;
uint64_t usablespace = clustercount * clustersize;
```

Note that all values in the BPB are now naturally aligned and that this code is *significantly* simpler than FAT32's BPB reading.

File Allocation Table

The File Allocation Table (FAT) is a table stored on the storage media that indicates the status and location of all data clusters that are on the disk. It can be considered the "table of contents" of a disk. The cluster may be available for use, it may be reserved by the operating system, it may be unavailable due to a bad sector on the disk, or it may be in use by a file. The clusters of a file need not be right next to each other on the disk. In fact it is likely that they are scattered widely throughout the disk. The FAT allows the operating system to follow the "chain" of clusters in a file.

FAT 12

FAT 12 uses 12 bits to address the clusters on the disk. Each 12 bit entry in the FAT points to the next cluster of a file on the disk. Given a valid cluster number, here is how you extract the value of the next cluster in the cluster chain:

```
unsigned char FAT_table[sector_size * 2]; // needs two in case we straddle a sector
unsigned int fat_offset = active_cluster + (active_cluster / 2);// multiply by 1.5
unsigned int fat_sector = first_fat_sector + (fat_offset / sector_size);
unsigned int ent_offset = fat_offset % sector_size;

//at this point you need to read two sectors from disk starting at "fat_sector" into "FAT_table".

unsigned short table_value = *(unsigned short*)&FAT_table[ent_offset];

table_value = (active_cluster & 1) ? table_value >> 4 : table_value & 0xfff;

//the variable "table_value" now has the information you need about the next cluster in the chain.
```

If "table_value" is greater than or equal to (>=) 0xFF8 then there are no more clusters in the chain. This means that the whole file has been read. If "table_value" equals (==) 0xFF7 then this cluster has been marked as "bad". "Bad" clusters are prone to errors and should be avoided. If "table_value" is not one of the above cases then it is the cluster number of the next cluster in the file.

The entries under index 0 and 1 are reserved. Index 0 is used as a value in other entries signifying that the given cluster is free, with the corresponding first entry in the table holding the value of the BPB_Media field in its low 8 bits and 0xf in its top 4 bits. For example, if BPB_Media is 0xF8, then the zeroth entry should hold the value 0xFF8. The second entry (index 1) is unused but must hold the value 0xFFF.

FAT12 uses an entry size that is not evenly divisible by 8 bits. This has some consequences.

First is storage in the table. Consider successive entries with values 0x123 and 0x456. In the bytes of the table, they'll be stored 0x23 0x61 0x45. Note that if you do little-endian 16-bit loads, you get 0x6123 at offset 0 and 0x4561 at offset 1, letting you recover the original two entry values with the shifts, masks, and offsets seen in the above code block.

The second is that, as seen above with the offsets used being 0 and 1, those word bytes might not be 16-bit aligned. That usually just means the x86 takes a slower path to load the word if you do e.g. `*(unsigned short *)bytes`, but if you're use something like UBSan to avoid undefined behavior, those UB-catching routines can be triggered (usually resulting in a panic) if you don't load the two bytes separately and stick them together yourself.

The third consequence is that the word bytes might not be *sector* aligned. Which means if your code loads a single sector of the table, it needs a special case where it loads two if the entry straddles the sector-size boundary. Or you can just load two sectors every time as seen above.

FAT 16

FAT 16 uses 16 bits to address the clusters on the disk. Because of this, it is much easier to extract the values out of a 16 bit File Allocation Table. Here is how it is done:

```
unsigned char FAT_table[sector_size];
unsigned int fat_offset = active_cluster * 2;
unsigned int fat_sector = first_fat_sector + (fat_offset / sector_size);
unsigned int ent_offset = fat_offset % sector_size;

//at this point you need to read from sector "fat_sector" on the disk into "FAT_table".

unsigned short table_value = *(unsigned short*)&FAT_table[ent_offset];
```

```
//the variable "table_value" now has the information you need about the next cluster in the chain.
```

If "table_value" is greater than or equal to (>=) 0xFFF8 then there are no more clusters in the chain. This means that the whole file has been read. If "table_value" equals (==) 0xFFF7 then this cluster has been marked as "bad". "Bad" clusters are prone to errors and should be avoided. If "table_value" is not one of the above cases then it is the cluster number of the next cluster in the file.

The entries under index 0 and 1 are reserved. The zeroth entry is reserved because index 0 is used as value of other entries signifying that the given cluster is free. Zeroth entry has to hold value of the BPB_Media field from in the low 8 bits, and the rest of the bits have to be set to zero. For example, if BPB_Media is 0xF8, then the zeroth entry should hold the value 0xFFF8. The first entry is reserved for the future and must hold the value 0xFFFF.

FAT 32 and exFAT

FAT 32 uses 28 bits to address the clusters on the disk. The highest 4 bits are reserved. This means that they should be ignored when read and unchanged when written. exFAT uses the full 32 bit to encode sector numbers. Similar to the same operation on a 16 bit FAT:

```
unsigned char FAT_table[sector_size];
unsigned int fat_offset = active_cluster * 4;
unsigned int fat_sector = first_fat_sector + (fat_offset / sector_size);
unsigned int ent_offset = fat_offset % sector_size;

//at this point you need to read from sector "fat_sector" on the disk into "FAT_table".

//remember to ignore the high 4 bits.
unsigned int table_value = *(unsigned int*)&FAT_table[ent_offset];
if (fat32) table_value &= 0x0FFFFFFF;

//the variable "table_value" now has the information you need about the next cluster in the chain.
```

If "table_value" is greater than or equal to (>=) 0x0FFFFFF8 (or 0xFFFFFFF8 for exFAT) then there are no more clusters in the chain. This means that the whole file has been read. If "table_value" equals (==) 0x0FFFFFF7 (or 0xFFFFFFF7 for exFAT) then this cluster has been marked as "bad". "Bad" clusters are prone to errors and should be avoided. If "table_value" is not one of the above cases then it is the cluster number of the next cluster in the file.

The entries under index 0 and 1 are reserved. The zeroth entry is reserved because index 0 is used as value of other entries signifying that the given cluster is free. Zeroth entry has to hold value of the BPB_Media field from in the low 8 bits, and the rest of the bits have to be set to zero. For example, if BPB_Media is 0xF8, then the zeroth entry should hold the value 0xFFFFFFF8. The first entry is reserved for the future and must to hold the value 0xFFFFFFFF.

Note that on exFAT, some files are not written out into the FAT. In the case that a file is fully contiguous, exFAT allows the operating system to encode this information and not update the FAT for this file. Unlike FAT32 therefore, the FAT table is not used for allocation status of a cluster; instead there is an allocation bitmap to handle that. See below under directory entries for that.

Directories on FAT12/16/32

A directory entry simply stores the information needed to know where a file's data or a folder's children are stored on the disk. It also holds information such as the entry's name, size, and creation time. There are two types of directories in a FAT file system. Standard 8.3 directory entries, which appear on all FAT file systems, and Long File Name directory entries which are optionally present to allow for longer file names.

Standard 8.3 format

Offset (in bytes)	Length (in bytes)	Meaning
0	11	8.3 file name. The first 8 characters are the name and the last 3 are the extension.
11	1	Attributes of the file. The possible attributes are: `READ_ONLY=0x01 HIDDEN=0x02 SYSTEM=0x04 VOLUME_ID=0x08 DIRECTORY=0x10 ARCHIVE=0x20 LFN=READ_ONLY|HIDDEN|SYSTEM|VOLUME_ID` (LFN means that this entry is a long file name entry)

12	1	Reserved for use by Windows NT.
13	1	Creation time in hundredths of a second, although the official FAT Specification from Microsoft says it is tenths of a second. Range 0-199 inclusive. Based on simple tests, Ubuntu16.10 stores either 0 or 100 while Windows7 stores 0-199 in this field.
14	2	The time that the file was created. Multiply Seconds by 2. Hour: 5 bits Minutes: 6 bits Seconds: 5 bits
16	2	The date on which the file was created. Year: 7 bits Month: 4 bits Day: 5 bits
18	2	Last accessed date. Same format as the creation date.
20	2	The high 16 bits of this entry's first cluster number. For FAT 12 and FAT 16 this is always zero.
22	2	Last modification time. Same format as the creation time.
24	2	Last modification date. Same format as the creation date.
26	2	The low 16 bits of this entry's first cluster number. Use this number to find the first cluster for this entry.
28	4	The size of the file in bytes.

Long File Names

Long file name entries *always* have a regular 8.3 entry to which they belong. The long file name entries are always placed immediately before their 8.3 entry. Here is the format of a long file name entry.

Offset (in bytes)	Length (in bytes)	Meaning
0	1	The order of this entry in the sequence of long file name entries. This value helps you to know where in the file's name the characters from this entry should be placed.
1	10	The first 5, 2-byte characters of this entry.
11	1	Attribute. Always equals 0x0F. (the long file name attribute)
12	1	Long entry type. Zero for name entries.
13	1	Checksum generated of the short file name when the file was created. The short filename can change without changing the long filename in cases where the partition is mounted on a system which does not support long filenames.
14	12	The next 6, 2-byte characters of this entry.
26	2	Always zero.
28	4	The final 2, 2-byte characters of this entry.

Here is an example of what a regular 8.3 entry with one long file name entry preceding it might look like in a hex editor:

```
41 62 00 69 00 6E 00 00 00 FF FF 0F 00 7F FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 FF FF FF FF
42 49 4E 20 20 20 20 20 20 20 20 10 00 00 F7 01 D5 38 D5 38 00 00 F7 01 D5 38 03 00 00 00 00 00
```

And in a text editor:

```
Ab.i.n.........................
BIN        ......8.8.....8......
```

The first line is the long file name entry (the second line is the regular 8.3 entry). The very first byte (41) tells us two important pieces of information. First, the one (01) tells us that this is the first long file name entry for the regular 8.3 entry. Second the forty (40) part tells us that this is also the *last* long file name entry for this regular 8.3 entry. The next 10 bytes spell out the first part of the long file name. In this case they read:

```
b 00 i 00 n 00 00 00 FF FF
```

Notice that each character is two bytes long and that the name is null terminated. The two FF's at the end are the padding at the end of the long file name. This is also what the other FF's in the long file name entry are. The final important thing to notice about the long file name entry is it's attribute byte at offset 11. the 0x0F attribute allows us to verify that this is indeed a long file name entry.

Directories on exFAT

exFAT redesigned these directory entries from the ground up.

Offset (in bytes)	Length (in bytes)	Meaning
0	1	Entry type
1	31	Rest of entry.

The base for every entry is that they are all still 32 bytes, and they all start with the type in the first byte. The types I've encountered that are relevant for reading files from disk:

File entry

Offset (in bytes)	Length (in bytes)	Meaning
0	1	Entry type = 0x85
1	1	Count of secondary entries.
2	2	Checksum of entry set
4	2	File attributes
6	2	Reserved
8	4	Creation date and time
12	4	Modification date and time
16	4	Access date and time
20	1	Creation time in hundredths of a second (0-199) to be added to the FAT style date/time for more accuracy. See FAT12 entry for format of date/time.
21	1	Modification time in hundredths of a second (0-199).
22	1	UTC offset for creation time
23	1	UTC offset for modification time
24	1	UTC offset for access time
25	7	Reserved.

Stream "extension" entry

It's called an extension, but it's 100% required to exist directly after the "file" entry.

Offset (in bytes)	Length (in bytes)	Meaning
0	1	Entry type = 0xC0
1	1	Secondary flags
2	1	Reserved

3	1	Name length
4	2	Name hash
6	2	Reserved
8	8	Valid data length. When writing large files, exFAT allocates the whole file first, and then incrementally updates this as data is written. Not sure what you're supposed to do with this, if it's not dataLength yell at the user?
16	4	Reserved
20	4	First cluster.
24	8	Data length.

File name entry

Offset (in bytes)	Length (in bytes)	Meaning
0	1	Entry type = 0xC1
1	1	flags
2	30	File name characters (15 UTF16 code units).

To actually use these, they typically come in the order:

- File entry - Stream extension entry - File name entry - (Additional file name entries)

The file entry has the file metadata info, the stream extension tells you how it's stored and the file name entries tell you what it's called. There is no 8.3 name any more.

When reading the file, the second bit in the stream extension secondary flags indicates if it's stored as extent, or if you need to use the FAT table. If it is set, the file is contiguous and the FAT is not up to date, if it is clear, the FAT is accurate and needs to be used (but could still say it's contiguous).

Long File Names

Long file name entries *always* have a regular 8.3 entry to which they belong. The long file name entries are always placed immediately before their 8.3 entry. Here is the format of a long file name entry.

Offset (in bytes)	Length (in bytes)	Meaning
0	1	The order of this entry in the sequence of long file name entries. This value helps you to know where in the file's name the characters from this entry should be placed.
1	10	The first 5, 2-byte characters of this entry.
11	1	Attribute. Always equals 0x0F. (the long file name attribute)
12	1	Long entry type. Zero for name entries.
13	1	Checksum generated of the short file name when the file was created. The short filename can change without changing the long filename in cases where the partition is mounted on a system which does not support long filenames.
14	12	The next 6, 2-byte characters of this entry.
26	2	Always zero.
28	4	The final 2, 2-byte characters of this entry.

Programming Guide

This section is intended to give you information about common functions that are preformed on a FAT file system.

Reading the Boot Sector

The Boot Sector is always placed at logical sector number zero. You can either read the boot sector into an array and access it's members that way or you can read it into a structure and access it through the structure. Either way, the values in the boot sector need to be readily available in order to do much of anything with a FAT file system.

Here is an example of some boot sector structures in C.

```
typedef struct fat_extBS_32
{
    //extended fat32 stuff
    unsigned int        table_size_32;
    unsigned short      extended_flags;
    unsigned short      fat_version;
    unsigned int        root_cluster;
    unsigned short      fat_info;
    unsigned short      backup_BS_sector;
    unsigned char       reserved_0[12];
    unsigned char       drive_number;
    unsigned char       reserved_1;
    unsigned char       boot_signature;
    unsigned int        volume_id;
    unsigned char       volume_label[11];
    unsigned char       fat_type_label[8];

}__attribute__((packed)) fat_extBS_32_t;

typedef struct fat_extBS_16
{
    //extended fat12 and fat16 stuff
    unsigned char       bios_drive_num;
    unsigned char       reserved1;
    unsigned char       boot_signature;
    unsigned int        volume_id;
    unsigned char       volume_label[11];
    unsigned char       fat_type_label[8];

}__attribute__((packed)) fat_extBS_16_t;

typedef struct fat_BS
{
    unsigned char       bootjmp[3];
    unsigned char       oem_name[8];
    unsigned short      bytes_per_sector;
    unsigned char       sectors_per_cluster;
    unsigned short      reserved_sector_count;
    unsigned char       table_count;
    unsigned short      root_entry_count;
    unsigned short      total_sectors_16;
    unsigned char       media_type;
    unsigned short      table_size_16;
    unsigned short      sectors_per_track;
    unsigned short      head_side_count;
    unsigned int        hidden_sector_count;
    unsigned int        total_sectors_32;

    //this will be cast to it's specific type once the driver actually knows what type of FAT this is.
    unsigned char       extended_section[54];

}__attribute__((packed)) fat_BS_t;
```

Important pieces of information that can be extracted from the boot sector include:

Total sectors in volume (including VBR):

```
total_sectors = (fat_boot->total_sectors_16 == 0)? fat_boot->total_sectors_32 : fat_boot->total_sectors_16;
```

FAT size in sectors:

```
fat_size = (fat_boot->table_size_16 == 0)? fat_boot_ext_32->table_size_16 : fat_boot->table_size_16;
```

The size of the root directory (unless you have FAT32, in which case the size will be 0):

```
root_dir_sectors = ((fat_boot->root_entry_count * 32) + (fat_boot->bytes_per_sector - 1)) / fat_boot->bytes_per_sector;
```

This calculation will round up. 32 is the size of a FAT directory in bytes.

The first data sector (that is, the first sector in which directories and files may be stored):

```
first_data_sector = fat_boot->reserved_sector_count + (fat_boot->table_count * fat_size) + root_dir_sectors;
```

The first sector in the File Allocation Table:

```
first_fat_sector = fat_boot->reserved_sector_count;
```

The total number of data sectors:

```
data_sectors = total_sectors - (fat_boot->reserved_sector_count + (fat_boot->table_count * fat_size) + root_dir_sectors);
```

The total number of clusters:

```
total_clusters = data_sectors / fat_boot->sectors_per_cluster;
```

This rounds down.

The FAT type of this file system:

```
if (sectorsize == 0)
{
   fat_type = ExFAT;
}
else if(total_clusters < 4085)
{
   fat_type = FAT12;
}
else if(total_clusters < 65525)
{
   fat_type = FAT16;
}
else
{
   fat_type = FAT32;
}
```

Reading Directories

The first step in reading directories is finding and reading the root directory. On a FAT 12 or FAT 16 volumes the root directory is at a fixed position immediately after the File Allocation Tables:

```
first_root_dir_sector = first_data_sector - root_dir_sectors;
```

In FAT32 and exFAT, root directory appears in data area on given cluster and can be a cluster chain. In exFAT it cannot be encoded as extent and will always be present in the FAT.

```
root_cluster_32 = extBS_32->root_cluster;
```

For each given cluster number we can calculate the first sector of it (relative to the partition's offset):

```
first_sector_of_cluster = ((cluster - 2) * fat_boot->sectors_per_cluster) + first_data_sector;
```

After the correct cluster has been loaded into memory, the next step is to read and parse all of the entries in it. Each entry is 32 bytes long. For each 32 byte entry this is the flow of execution:

1. If the first byte of the entry is equal to 0 then there are no more files/directories in this directory. FirstByte==0, finish. FirstByte!=0, goto 2.
2. If the first byte of the entry is equal to 0xE5 then the entry is unused. FirstByte==0xE5, goto 8, FirstByte!=0xE5, goto 3.
3. Is this entry a long file name entry? If the 11'th byte of the entry equals 0x0F, then it is a long file name entry. Otherwise, it is not. 11thByte==0x0F, goto 4. 11thByte!=0x0F, goto 5.
4. Read the portion of the long filename into a temporary buffer. Goto 8.
5. Parse the data for this entry using the table from further up on this page. It would be a good idea to save the data for later. Possibly in a virtual file system structure. goto 6
6. Is there a long file name in the temporary buffer? Yes, goto 7. No, goto 8
7. Apply the long file name to the entry that you just read and clear the temporary buffer. goto 8
8. Increment pointers and/or counters and check the next entry. (goto number 1)

This process should be repeated until all of the entries have been read from the cluster. You should then check to see if there is another cluster following this one in the cluster chain or if this is the last cluster in the chain. See section below and FAT section for more information. You should do the above process for each cluster in the chain, following it until there are no more clusters left in the chain. Then you can check if any of the entries that you just read are directories. If the are they should each be read in the same way starting with their first cluster number which is stored in the entry.

Following Cluster Chains

There are two basic steps to following cluster chains. The first step is to find out if there is another "link" (cluster) following the current one in the chain. The second step is to actually use the value read from the FAT to read the next sector. Here is the basic idea:

1. Extract the value from the FAT for the _current_ cluster. (Use the previous section on the File Allocation Table for details on how exactly to extract the value.) goto number 2
2. Is this cluster marked as the last cluster in the chain? (again, see the above section for more details) Yes, goto number 4. No, goto number 3
3. Read the cluster represented by the extracted value and return for more directory parsing.
4. The end of the cluster chain has been found. Our work here is finished. :)

Reading extents

On exFAT, files can have a bit set in their flags that indicate it is stored as extent-based. This means that the whole file is contiguous, and that the file size plus the first cluster indicate where the (whole) file is. The FAT entries will contain garbage and are not to be trusted.

To read this, do the same calculation as above, except you may in every step assume that the next cluster is the numerically next cluster and that enough sectors have been allocated for the file size.

Creating a fresh FAT filesystem

Typically during development you want to create a disk image with a FAT filesystem. There are two common approaches for this, either by using a utility that works directly on images, or by using a Loopback Device and using the OS' own driver to work on the image. A less common alternative is to have an actual disk in your drive.

The most buildscript-friendly tool is MTools - which can do all operations directly on a disk image using the -i argument and supplies every DOS command related to files in this fashion, only prefixed with an `m`. It can also use a configuration file to access drives in their DOS fashion, allowing you to use for instance A: and C: as actual drives. The tool can be built out of the box for Windows and is included in many a linux package manager.

Linux-only developers can, often with a bit of sudo and permission magic, automate the Loopback Device in combination with `mkdosfs` or `mkfs.vfat` as well as partition editing. This method is less portable as the commands often can't be reused outside of Linux. Several developers also make the error of passing -F to mkdosfs in an attempt to choose a FAT size, which often has the effect of creating a corrupt filesystem since the result doesn't follow the official rule for FAT sizes anymore.

Windows users can make use of VFD for loopback devices. It comes with a GUI, but at the cost of not being properly automatable in a script.

See Also

Threads

- from raw bits to directory listing (code posted) in the forum
- Public Domain FAT32 code in the forum
- FAT12/FAT16 bootsector code in the forum

External Links

- FAT32 File System Specification (http://www.osdever.net/downloads/docs/fatgen103.zip) - from Cottontail OS Development Library
- FAT32 File System Specification (http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc) - from Microsoft (documentation, but in tutorial style with many code examples)
- About an error in the above specification (http://board.flatassembler.net/topic.php?t=12680)
- http://scottie.20m.com/fat.htm
- http://www.maverick-os.dk/FileSystemFormats/FAT12_FileSystem.html
- http://www.pjrc.com/tech/8051/ide/fat32.html
- Intro into Sectors and Addressing (http://web.archive.org/web/20170112194555/http://www.viralpatel.net/taj/tutorial/fat.php)
- http://elm-chan.org/fsw/ff/00index_e.html - simple (V)FAT12/16/32 read/write library with good documentation
- http://gitorious.org/unix-stuff/fat-util - Utility to read, remove and extract files on FAT12, 16 and 32
- http://www.larwe.com/zws/products/dosfs/index.html - fat12/16/32 compatible fs driver
- http://www.isdaman.com/alsos/protocols/fats/nowhere/FAT.HTM