Introduction to FAT

Concepts

FAT file system was first implemented in the year 197? by a student Bill Gates. Then it grew into the most used file system among IBM PC computers. I will explain the FAT idea in this paper. FAT implementations will be described in later documents.

In FAT file system, storage device is of a fixed size. It is logically divided into chunks of data of equal size called **clusters**. Any **file** takes up a natural number of clusters. Disk map, called File Allocation Table (**FAT**), is located at a known position on disk. FAT is an array of the entries of equal size. Number of entries is equal to number of clusters on disk, so there exists a unique relationship between FAT entries and clusters. FAT entry is one of the following:

Value	*Meaning*
0	Free Cluster
-8..-1	End of File
-9	Bad Cluster
-10h..-0ah	Reserved Cluster
Other	Next Cluster In Chain

Free clusters are those free for allocation. The cluster marked as *EOF* (End Of File) is the last cluster for the file chain. Although several values may serve as EOF, only `-1` is used by DOS. *Bad clusters* are physically damaged clusters. They cannot contain any data. *Reserved clusters* are reserved for DOS file system usage. They should not be used by anyone but their creator. In every FAT, the very first entry is reserved

If FAT entry is none of the above values, it should contain the number of the next cluster for this file. As you see, FAT is just a variation of a linked list.

Another FAT file system concept is *directory*, or *folder*. Directory is a file that contains file descriptions. File description consists of file name, date, flags, size, and the first cluster in the file's cluster chain. Directory itself is stored just like a normal file. It also has a FAT chain. Directory entry may point to a nested directory, which leads to directory hierarchy. But where does this all start? Most FAT variations expect the first-level, or root, directory at a certain fixed location on disk.

Example

To make things clear, let us read `512` bytes of the file `"C:\DOC\INTEL\INTEL386.TXT"` starting from offset `2048` relative to the beginning of this file. Filename should first be split into meaningful tokens. They are `"C:"`,`"DOC"`, `"INTEL"`, and `"INTEL386.TXT"` in this order. The first token tells us to select the first primary partition on the first hard drive for reading. `"DOC"` is name of the directory that is in the root directory of `"C:"`. Since we know where the root directory is on disk (and root directory size too), let us look through it and find the entry named `"DOC"`. As I said earlier, one of the fields of this entry is `FirstCluster`. Now we know where the FAT chain for the directory `"DOC"` starts.

> Let us read the first cluster of this chain. Now let us assign the value of `FirstCluster` to `NextCluster` - you will later understand why. We should now scan the cluster that we have read for the entry with the name`"INTEL"`. If we have not found it, let us read the next cluster in the chain.
>
> To do it, we look at the value at `FAT[NextCluster]`. If it is `EOF`, there are no more clusters in the chain. If it contains any other special value, FAT chain has been corrupted. Otherwise, it contains the next cluster, so we write `NextCluster=FAT[NextCluster]`. Let us read `NextCluster` into memory and scan it for `"INTEL"`. We repeat reading next cluster in the chain and scanning it for `"INTEL"` until we either find it, or reach `EOF` or any other reserved value in the FAT chain.

Hopefully, we have found the subdirectory `"INTEL"`. We know the first cluster of this subdirectory, so let us look for the entry `"INTEL386.TXT"` in the same manner as we have just looked for subdirectory.

Now we should know `FirstCluster` of our file. Let's imagine that cluster size for our disk is `1024` bytes. Thus, to reach offset `2048` we must skip two `1024`-byte blocks, or two clusters. You know how to do it, don't you? Hint: look through the FAT chain. Finally, let's read our `512` bytes.

The example was intentionally oversimplified, but I hope, it showed the idea. As I said, implementation will be discussed in later documents.

Limitations of FAT

If you carefully followed the example, you noticed that I cheated. I did not check all possible error conditions, and some errors in FAT might have caused troubles, up to infinite loops. Let us take a closer look at FAT limitations.

First of all, FAT system is terribly inefficient for large files. You can imagine how many FAT entries I will have to scan to access some file at offset 96 MB. Even if I have all FAT cached, time to look through the chain in memory is not acceptable. This situation can be somewhat corrected by keeping current position in FAT, but this will only help for sequential access patterns.

Secondly, FAT may create high file fragmentation. Because today's storage devices are slow to move heads across the disk surface, space for files should be allocated carefully. This is especially seen under multitasking systems, when many files are written to at the same time. So, when the file is being expanded, it is wise to allocate a number of clusters for it in advance.

Thirdly, FAT clustering eats off space. If cluster size is 8 KB, it is reasonable to expect that each file will take up 4 KB more of disk space than it actually needs. If there are 10,000 files on disk, 40 MB of space is waisted. However, FAT32 increased the number of available clusters and reasonably reduced cluster size.

Fourthy, FAT is very sensitive to errors. It is almost impossible to restore the file whose chain was corrupted. Consider the following examples:

- The easiest to diagnose error is the chain that contains some reserved value other from EOF. Unless you are a special disk analyzer, the best solution is to set the previous chain entry to EOF, thus truncating the file.
- The variation of the above error is a valid FAT entry which points to the physically defective or out of bounds cluster. File system should allocate another cluster and plug it in the chain, replacing the invalid one. If any data can be retrieved from the invalid cluster, it should be copied to the new cluster.
- A FAT chain that will throw my above algorithm into endless loop is a self-linked chain. Consider the following chain: `(2->3->4->8->7->4-..)`. The last node points to the cluster already in chain. Some caution should be taken to avoid looping infinitely in this case. Note that the following simple sanity check will identify this kind of problem. Let us go through the chain and compare each new entry with all entries that we have already passed. If any of them equal, the chain is self-linked. But this check is extremely expensive. If the chain contains n clusters, the order of the number of comparisons is n squared. The situation can be improved by allocating a bitmap, each bit representing a cluster, and setting bits for the clusters we have passed. If the corresponding bit is already set, then the chain is self-linked. The order of such a check is n, but it demands extra memory. In any case, it is impractical to do sanity check after each step through the chain. Norton Disk Editor did it after a fixed number of steps, and this is the best technique I can suggest. After all, let us hope that 4G file size limitation won't make it terribly slow. If this is a file we are accessing (not a directory), sanity check may be performed only when file size via the cluster chain becomes larger than file size in directory (rounded up to the nearest multiple of cluster size). Once the self-linked chain is identified, the entry that is pointing back should be set to EOF, thus truncating the file.

- Two or more FAT chains may be cross-linked. This happens when FAT chain entries for different files point to the same cluster. In this case files share the tail. It is even worse when a file is cross-linked with directory. The latter case will have a fairly strange outcome, especially if the cross-linked directory was cached. The order of checking for cross-linked files is at best number of clusters on disk. This is achieved by using the bitmap for all clusters, much like described above. But the problem is, you also have to go through all directory structure to retrieve starting clusters for all files. This makes checking for cross-linked files the privilege of special analyzing programs. When this problem is detected, separate tails are saved for each of the cross-linked files. There is no easy and quick way of detecting cross-linked files, so if you have this problem, expect weird results from accessing such files at the same time. The results are usually unpredictable because of caching.

Correcting any of these problems will likely corrupt the file, but leaving them uncorrected will sooner or later hang the system. FAT provides no ways for insuring data integrity, compression, encryption, or any other nice perversions. With FAT you cannot even detect that a file was corrupted - forget about correction. Besides the problems that are related to the FAT concept itself, there exists a large group of problems related to specific FAT implementations. For example, earlier FAT systems expected much vital data at fixed disk addresses, which made disks unusable because of corruption of just a few sectors. FAT32 finally made root directory floating and gave a choice of selecting the working copy of FAT, but File Allocation Tables are still at fixed physical addresses. Besides, directory format of FAT, especially of VFAT, is a mess. This mess is even hard to handle normally - any error handling becomes nightmare.

Conclusion

Next documents will describe FAT implementations in detail. I had a choice of either describing them separately from each other, or merging together. I will use the later method for several reasons. First of all, there are many similarities between different types of FAT. Most structures are transferred without any modifications or with minor extensions to next versions. Secondly, implementing all types of FAT at once is probably more productive than implementing them separately. And thirdly, I will try to build a base for integrating different file systems under common protocols. This base will, hopefully, save you much time for coding in future.

FAT Filenames

Format

This document is for those who are already familiar with DOS files. It is intended to remind you of DOS conventions.

DOS uses the following pattern for forming *file paths*:

```
[disk]:\[directory]\[subdirectories]\[filename].[extension]
```

`[disk]` is one letter of the Latin alphabet. Valid letters are `A` through `Z`. How disk letters are assigned to physical devices was discussed earlier.

`[directory]` and `[subdirectories]` are strings. They specify the location of the file in the *directory tree*. None, either or both are omitted depending on file location.

`[filename]` is also a string that identifies name of the file. `[extension]` usually contains some information regarding type of the file. It is a string. Note that directories may also have extensions. Because file and directory names are of the same format, everything below that applies to filenames also applies to directory names.

Short and Long Names

There are two types of filenames: **long** and **short**, or **aliases**. Short filenames are subject to the infamous 8.3 limitation. Thus, `[filename]` is one to eight characters long, and `[extension]` is zero to three characters long. This limitation also applies to directory names. Many sources say that paths are limited to 80 or 128 characters, but these limitations are due to DOS peculiarities, but not FAT format. With FAT file system, one can have infinitely long paths. The recommended (by me) maximum length of a path is 256 characters, including terminating null character. Short filenames use ASCII character set. Thus, each character takes up exactly one byte. According to DOS manuals, short filenames are case insensitive, and the following characters can be used:

- Letters `A` through `Z`.
- Digits `0` through `9`.
- Characters with ASCII codes greater than 127. Note that these characters depend on current code table. Also, their handling is case sensitive.
- Space. Note that many applications do not recognize this character and it is not used in creating aliases for long filenames.
- `$ % - _ @ ~ ` ! ( ) { } ^ # &`

Case insensitivity is achieved by converting the name to uppercase when the file is accessed or created. The following characters have special meaning:

- `.` (dot) is the delimiter between `[name]` and `[extension]`. There should be only one delimiter for each file or directory name.
- `\` (back slash) is the delimiter between directory and file names.
- `:` (colon) is the delimiter between disk letter and the rest of the path.

The following characters are called wild cards. They are used in search operations:

- `?` (question mark) denotes any character.
- `*` (asterisk) denotes blank or any number of any characters.

You are best advised not to allow any other characters in filenames, and to use special characters according to their meaning. If the existing filename has any illegal characters in it, it should either be ignored or the invalid characters should be replaced by the valid characters.

Long names are up to 256 characters long, including extension and the terminating null character. This limitation is artificial, and the long names on the disk can actually be longer than 256 characters. Again, some limitations were created by the software that serves FAT. The maximum length of a directory path, including drive letter, column, and leading slash, but excluding trailing slash, null terminator, filename and extension, is 246 bytes. The maximum length of a full path is 260 characters, including null terminator. This is four characters more than I recommend.

Long filenames are stored in unicode. Each character is two bytes long. There are two important things to remember about unicode:

- For only the characters in the low half of the ASCII table, the high byte of unicode character is zero, and the low byte is the same as in ASCII.
- Unicode system supports case insensitive operations for all characters, not just Latin.

All characters that are valid for short filenames are also valid for long filenames. In addition, the following characters can be used:

- `+ , ; = [ ]`
- `.` (dot) can occur more than once in a filename. Extension is the substring after the last dot.

Long filenames are what I call half case sensitive. The case of the characters is preserved when creating the file, but other from that long filenames are case insensitive. For example, "File" and "file" are treated as the same string by file system. They cannot co-exist in the same folder, and any of these names can be used to reference the file.

Aliasing

Whenever a file with a long filename is created, its alias with short filename is also created. The converse is usually not the case. If the long filename fits in the standard 8.3 scheme and contains only valid for short filenames characters, the following rules are applied:

- If one or more of the characters are not upper case, the alias created is the same as the long filename, only converted to upper case.
- If all characters are uppercase, then only the alias is created, but the directory entry for the long filename itself is not created.

If the long name is either too long to fit in 8.3 or contains illegal for short filenames characters, the following rules are applied:

- First, illegal for short filenames characters are deleted from long filename. All dots except the last one are also deleted.
- Then, first six (or less, if the name is shorter) characters are used as base, and `~` (tilde) and the number `1` are used as tail. Base and tail are concatenated to form the alias' filename. Alias' extension is formed by the first three (or less) letters from the long filename's extension.
- If the alias with such a name already exists, the number in the tail is increased. This process is repeated until we arrive at a unique alias. Note that when the number reaches the power of ten, its decimal representation takes up one more byte. In this case one last character is stripped off the base. The situation when this loop overflows (base is empty, but number needs to be increased) is very unlikely. After all, where did you see more than 9,999,999 files in one directory?

Needless to say, there is no way to tell the alias by just looking at the long name, and there is no way to retrieve the long name by looking at the alias. Special directory structure insures that they are associated with each other. That is why when the file is accessed via its long filename and is edited (usually, deleted and re-created), the alias may change. It may especially change if the file is copied to a different folder.

One can access the file using its alias.

- When the file is created using its alias, the corresponding long name is not created, with the exception of one case below.
- When the file is deleted, either via its alias or via its long name, all directory entries for the long name and the alias are marked as deleted.
- Some extra steps are taken to preserve the long filename. The long filename is obviously lost when the file is copied using its alias. But there are less obvious things. For example, an application may delete and re-create the file using its alias. Not to loose the long filename, Windows 95 keeps the information about the deleted via aliases files for fifteen seconds, and if an an attempt was made to re-create the file during this time using the same alias, the old long filename is associated with this alias.

Finally, only VFAT filesystems support long filenames and aliasing.

FAT Boot Sector

Introduction

This paper describes the FAT boot sector. I will try to unify the boot sectors of the FAT32, FAT16, and FAT12 file systems.

Structure

Boot sector is always the very first sector in the partition. Validity check is performed by comparing the 16 bit word at offset 1FE to AA55. For FAT systems, this sector always contains the Bios Parameter Block (BPB) at offset 0B. The structure of the boot sector is below.

Offset in Boot Sector	*Length in Bytes*	*Mnemonic*
03	8	OEM_Identifier
0B	2	BytesPerSector
0D	1	SectorsPerCluster
0E	2	ReservedSectors
10	1	NumberOfFATs
11	2	RootEntries
13	2	NumberOfSectors
15	1	MediaDescriptor
16	2	SectorsPerFAT
18	2	SectorsPerHead
1A	2	HeadsPerCylinder
1C	4	HiddenSectors
20	4	BigNumberOfSectors
24	4	BigSectorsPerFAT
28	2	ExtFlags
2A	2	FSVersion
2C	4	RootDirectoryStart
30	2	FSInfoSector
32	2	BackupBootSector
34	12d	Reserved

`OEM_Identifier` is the eight-byte ASCII string that identifies the system that formatted the disk. All eight characters are meaningful. Spaces or zeroes are appended if the name is less than eight characters long. As any other OEM name, this string is nice to display near to the other disk information, but it is absolutely useless for any other purpose.

`BytesPerSector` is how many bytes long the physical sector is. All disks I have ever seen contained 512 in this field. The gossips are, DOS supports disks with different sector sizes. Other gossips are, early versions of DOS silently set this field to 512 and ignored whatever the original value was. Anyway, as of today, you will not loose much if you compare this value to 512 and refuse to work with the disks that have a different sector size.

`SectorsPerCluster` is how many sectors are in one logical cluster. What is cluster was described earlier. This value should not be zero.

`ReservedSectors` are reserved, starting from the LBA sector `(0)` relative to current partition. LBA `(ReservedSectors)` is the beginning of the first FAT. For FAT12 and FAT16 this value is usually 1. For FAT32 it is 20h. At least one sector must always be reserved.

`NumberOfFATs` is the number of File Allocation Tables. This value is usually two. FATs are consecutive on the disk: the second copy of FAT goes right after the first copy. At least one copy of FAT should be present.

`RootEntries` contains the number of the entries in the root directory if root directory is fixed. It is zero if the root directory is not fixed. FAT32 disks should contain zero in this field, indicating that the root directory can be arbitrarily long. Otherwise, this field usually contains 512. Each directory entry takes up 32 bytes. To avoid wasting space, `RootEntries*32` should be divisible by `BytesPerSector`.

`NumberOfSectors` is the total number of sectors on the disk. If the number of sectors is greater than 65535, then this field is set to zero and the dword at `BigNumberOfSectors` contains the actual number of sectors. By`NumberOfSectors` I will refer to that of `NumberOfSectors` and `BigNumberOfSectors`, which is used. Note that this field should contain the same or lesser value as the corresponding field in the partition table. If the values are not equal, the lesser of them should be used. `NumberOfSectors` should be large enough to contain at least the reserved sectors, all FAT copies, and the root directory, if any. Disk layout is described below.

`MediaDescriptor` describes the device:

Value	*DOS version*	*Meaning*
FF	1.1	5 1/4 floppy, 320KB
FE	1.0	5 1/4 floppy, 160KB
FD	2.0	5 1/4 floppy, 360KB
FC	2.0	5 1/4 floppy, 180KB
F9	3.0	5 1/4 floppy, 1.2MB
F9	3.2	3 1/2 floppy, 720KB
F8	2.0	Any Hard Drive
F0	3.3	3 1/2 floppy, 1.44MB

As of today, no other values are defined.

`SectorsPerFAT` contains the number of sectors in one FAT. This field is zero for FAT32 drives, and `BigSectorsPerFAT` contains the actual value. Note that the Microsoft FAT32 boot loader will not work with the FAT32 drives if `SectorsPerFAT` is not zero, and FAT12 and FAT16 loaders will not work with the drives if `SectorsPerFAT` is zero. As with `NumberOfSectors`, by `SectorsPerFAT` I will mean the appropriate value. It goes without saying, FAT should be long enough to contain the information about all clusters on the disk.

`SectorsPerHead` is the number of sectors grouped under one head. `HeadsPerCylinder` is also what you think it is. If this partition is a CHS partition, these values must be the same as those returned by BIOS. If they are not the same, the disk was misconfigured and the partition should not be used. Note that the Microsoft boot loader alteres the BIOS Diskette Parameters table by setting the `SectorsPerTrack` field of this structure to`SectorsPerHead` read from the boot disk. The values in these fields do not matter for LBA partitions.

`HiddenSectors` is the number of sectors between the beginning of this partition and the partition table. This field should be the same as "number of sectors preceding the partition" in the partition table. Note that it is not necessarily the physical LBA address of the first sector because there exist secondary partitions. If `HiddenSectors` is not the same as in the partition table, boot sector was corrupted and the partition should not be used. Also note that the high word contains garbage for old versions of DOS.

`ExtFlags`, and all fields described below, are defined only for FAT32 disks. They are defined differently for FAT12 and FAT16 (that issue is discussed below). If the left-most bit of `ExtFlags` value is set then only the*active* copy of FAT is changed. If the bit is cleared then FATs will be kept in synchronization. Disk analyzing programs should set this bit only if some copies of the FAT contain defective sectors. Low four bits define which copy should be active. As you see, only the first sixteen copies of the FAT may be selected active, so the disk is usable if and only if among the first sixteen copies at least one is usable. For this flag, FAT numbers start from zero. Sanity check insures that the active FAT is less than `NumberOfFATs`.

`FSVersion` is the version of the file system. The high byte is the major version, the low byte is the minor version. Both are set to zero on my Windows 95 OSR2. I do not think that this value should be checked. However, the Microsoft boot loader does check it *in certain cases*, and it complains if it is not zero.

`RootDirectoryStart` contains the number of the first cluster for the root directory. Yes, finally the root directory became stored like any other directory, in the cluster chain. This also implies that it may grow as needed. The value in this field should be at least two. It is two on my system.

`FSInfoSector` is the sector number for the file system information sector. This sector is new to FAT32. Its structure is below:

Offset in Sector	*Size*	*Meaning*
00	4	Signature, should be 41615252h (?)
1E4	4	Signature, should be 61417272h
1E8	4	Number of free clusters on the drive, or -1 if unknown
1EC	4	Number of the most recently allocated cluster
1F0	12d	Reserved
1FE	2	Signature AA55

All the other bytes are set to zero. As you see, these values are introduced to improve performance. Make sure that `FSInfoSector` is at least one and it lies within the reserved disk area. Also make sure that this value is not the same as `BackupBootSector`. If it does not satisfy these conditions, do not use this sector, but the file system should still be usable. Do not use the information in this sector if its signature is incorrect. Note that only the second signature is documented by Microsoft. Normally, the number of free clusters is checked only by special disk analysis programs. `FSInfoSector` is usually one.

`BackupBootSector` is the sector number for the backup copy of the boot sector. This copy can be used if the main copy was corrupted. It is also nice to compare the two copies on startup. If they do not match, a warning should be issued. They may not be in tact because of corruption or a boot virus. If this field contains zero or the number greater than or equal to `ReservedSectors` or the same value as `FSInfoSector`, the backup sector should not be used.

My experience says, there are two complete copies of the boot information. The first copy starts from the very first sector in the partition and is three sectors long. The first sector is being described now. The second sector is usually the FS Information Sector, as described above. The third sector is some extra code for the boot loader. The third sector also contains the `AA55` signature at `1FE`. Then, starting from `BackupBootSector`, goes another triple. The first and the third sectors of this triple are identical to those of the main one. The second sector also contains the FS Information Sector, but it is not kept in synchronization with the main FS Information Sector. It is not documented by Microsoft. Its "Number of Free Clusters" is set to `-1` and "The Most Recently Allocated Cluster" is two, which suggests that it was touched only by the formatting program. All the other reserved sectors contain zeroes.

Notes

Note that some of the values in the boot sector have a different meaning in FAT12/16 systems as compared to the FAT32 system. These fields form the Extended BIOS Parameter Block, or EBPB. EBPB is the same for FAT12/16 and FAT32, but it starts from the different offsets in the boot sector. For the sake of completeness I will describe it here:

Offset for FAT12/16	*Offset for FAT32*	*Length in Bytes*	*Meaning*
24	40	1	BIOS drive number
25	41	1	Reserved
26	42	1	Extended Boot Record signature = 29h
27	43	4	Serial Number
2B	47	11	Volume label
36	52	8	System Identifier (FAT12, FAT16, or FAT32)

In my humble opinion, you do not need these fields even for FAT12 or FAT16. However, there is one compatibility issue. If you have detected that FAT12 or FAT16 is used, and the extended boot record signature is not`29h`, you should ignore all the values in the BPB starting from the offset `1E`. This is not a typo: `HiddenSectors` is really split by half, and if the extended boot record signature is incorrect then only the lower word of`HiddenSectors` is valid.

Sanity check might also insure that System Identifier is correct for the detected file system. It should be `"FAT12", "FAT16"`, or `"FAT32"` according to the file system. Strings are padded with spaces to fit in eight bytes.

Finally, the presense of the EBPB in FAT32 is not documented by Microsoft.

Now it is high time to explain how clusters are mapped to sectors. First, consider the layout of the FAT disk:

LBA Location	*Length in Sectors*	*Description*
`0`	`ReservedSectors`	`Boot Sector(s), File System Info Sector`
`ReservedSectors`	`NumberOfFATs*SectorsPerFAT`	`File Allocation Tables`
`RootStart *`	`(RootEntries*32)/BytesPerSector`	`Root Directory, if any`
`ClustersStart`	`NumberOfClusters*SectorsPerCluster`	`Data Clusters`

* Note: `RootStart` does not make sense for FAT32 partitions. However, the formulae below are still valid for FAT32.

Some of the values in the table are in the BPB. Let us calculate the rest of them:

```
RootStart=ReservedSectors+NumberOfFATs*SectorsPerFAT
ClustersStart=RootStart+(RootEntries*32) div BytesPerSector
* Note: if (RootEntries*32) mod BytesPerSector then
ClustersStart=ClustersStart+1
NumberOfClusters=2+(NumberOfSectors-ClustersStart) div SectorsPerCluster
```

To convert cluster address to LBA address use the formula:

```
LBA=ClustersStart+(Cluster-2)*SectorsPerCluster
```

Now we are ready to detect which file system is used.

- If `NumberOfClusters<4087` then FAT12 is used.
- If `4087<=NumberOfClusters<65,527` then FAT16 is used.
- If `65,527<=NumberOfClusters<268,435,457` then FAT32 is used.
- Otherwise, none of the above is used.

I fairly warn you: I have not yet found two sources that agree on these values. So, be careful if the number of clusters is close to the border value. One might have noticed that the maximum `NumberOfClusters` for FAT32 looks odd. Since only 28 of 32 bits are currently used, the FAT32 partition can have no more than 268,435,456 clusters. Looking at `NumberOfClusters` is the **only** recommended by Microsoft way of detecting the FAT entry size.

Since the type of the file system depends on the cluster size, which can be set arbitrarily by the formatting program, let me introduce the Microsoft recommendations. Note that these are only guidelines. The partition size is **not** used directly to determine the FAT type. It is first converted to `NumberOfClusters` as described above.

Partition Size	*Type of FAT*
10MB or less	FAT12
10MB through 512MB (?)	FAT16
512MB (?) through 2TB	FAT32

The following table contains the ranges for the partition sizes. They do overlap.

Min Partition Size	*Max Partition Size*	*Type of FAT*
1.5KB	510.75MB	FAT12
2.0435MB	8190.75MB	FAT16
~32MB	32TB	FAT32

File Allocation Table

This document will be surprisingly short. By now, you should already know what File Allocation Table is, how to use it, and where it is located. I can only add a couple of notes.

FAT is just an array of values. Values are either 12 or 16 or 32 bits wide. Indices start from zero. The very first value, FAT[0] is reserved and it contains the media description byte, sign-extended to the necessary width. Because of the way this byte is chosen, the resulting value falls right within the range of the reserved values in FAT. This entry is a symbolic representation of a boot sector and other reserved sectors. It is always followed by EOF. Thus, FAT[1]=EOF. This entry represents FAT itself. Therefore, the entries with the indices `2` through `NumberOfClusters-1` are available. You should always watch for the cluster numbers that are out of range.

FAT disks usually contain two copies of FAT. The next copy of FAT immediately follows the previous one. In FAT12 and FAT16 systems all copies of FAT are kept in synch with each other, but only the first copy is ever read. Microsoft claims that the next copies of FAT are used when the first one is physically unusable, but it is not true for all versions of their systems. In FAT32 there exists a field in the BPB that tells which copy to use and whether to synchronize all copies.

Finally, data in FAT is stored in Intel little endian order. Note that the high four bits in FAT32 are reserved and should be masked out when reading from FAT. They should not be changed when writing to FAT either. It also concerns EOF and other special values. In FAT12, the entries are one and a half bytes long. So, use the following steps to read the value from FAT12:

- Multiply the cluster number by three and divide it by two. Read the 16 bit word at this offset in FAT.
- If the cluster number was even, mask it with `0FFFh`, otherwise shift the value right by four.

Writing the value to FAT12:

- First, mask your value with `0FFFh` or, better, check if it is in the allowed range.
- Multiply the cluster number by three and divide it by two. Read the 16 bit word at this offset in FAT.
- If the cluster number was even, `and` the word you got with `0F000h`, `or` it with your value, and write it back.
- If the cluster number was odd, shift your value left by four, `and` the word you got with `0Fh`, or it with your value, and write it back.

For FAT16 and FAT32 just multiply the cluster by the size of the FAT entry and use it as an offset in FAT. Don't forget to mask out the high four bits when working with FAT32.

Directory Structure

Introduction

Welcome to hell. This document is devoted to the structure of the special type of DOS files called directories.

Each directory consists of a number of fixed-size entries. Each entry is 32 bytes long. The number of sectors in the directory is fixed for the root directory on FAT12 and FAT16 partitions, and sectors are consecutive on disk. For non-root directories, as well as for the root directory on the FAT32 partitions, the number of sectors is not fixed, and the directory is stored according to the normal cluster chain.

There are two different types of the entries for long filenames and for aliases.

Aliases

The entry for an alias is:

Offset in the Entry	*Length in Bytes*	*Description*
00	8	Filename
08	3	Extension
0B	1	Attribute
0C	1	Case
0D	1	Creation time in ms
0E	2	Creation time
10	2	Creation date
12	2	Last access date
14	2	High word of starting cluster for FAT32
16	2	Time stamp
18	2	Date stamp
1A	2	Starting cluster
1C	4	Size of the file

Filename and extension are left-justified and blank-padded. Note that filename cannot consist solely of spaces, but extension can. Watch for illegal characters in filenames. My humble suggestion is replacing any illegal characters with underscores.

Some characters in the filename have special meaning. If the first character has the code `05`, then actually the first character has the code `E5` and it is not a special character. If the first character has the code `E5`, then the file was deleted. You may save some time when going through the directory structure by checking the first character in the filename. If it is zero, there are no more entries in current directory.

Two entries have a special meaning. They are present only in subdirectories, but not in the root directory. The entry with the name consisting of exactly one dot is the pointer to the root directory. Its starting cluster is the first cluster of the root directory, which is usually two. You are best advised to ignore this value because the location of the root directory can be easily calculated otherwise. The entry with the name made up of exactly two dots points to the next higher-level directory in the hierarchy. Its starting cluster is the first cluster in that directory. These entries should be the first and the second one in the directory, correspondingly. Their attributes should be `10h` (Directory). They are created at the time the subdirectory is created. There are no corresponding long names for them.

Attribute is a collection of bit flags:

Value	*Meaning*
01	Read Only
02	Hidden
04	System
08	Volume Label
10	Directory
20	Archive
40	Unused
80	Unused

Read Only, Hidden, and System are pretty self-explanatory. I will only note that neither Hidden nor System files should be moved during defragmentation or any other disk service. If you remember, I recommended certain actions when file corruption is detected. You are best advised not to try any corrections on Hidden or System files. Also, Hidden files should not be returned by search commands unless they were explicitly asked to do so.

Volume label attribute means that this entry contains the disk label in the filename and extension fields. Volume label is valid only in the root directory. Common sense says, there should be only one volume label per disk. For the entry to really contain the volume label, the attribute should be exactly `08`. *If Attribute is equal to* `0Fh` *(Read Only, Hidden, System, Volume Label) then this entry does not contain the alias, but it is used as part of the long filename or long directory name.*

Directory bit is set if the entry is a subdirectory. In this case the starting cluster contains the beginning cluster for the subdirectory, and the file size field is ignored (set to zero). Directories can also be Read Only, Hidden, System, or Archive. Directory bit is not set for the long directory name entries.

Archive bit is somewhat symbolic. It should be set if the file was not archived by the backup utility. Never in my life I have seen the use of this bit.

Two values are unused, which means that the entries with either of these bits set should be considered invalid. Another invalid combination is when both, Directory and Volume Label bits are set. Unless you are a disk analyzing tool, the best technique is to ignore the entries with the invalid attribute.

Case is zero if the filename and extension need to be converted to upper case. This field is used only by Windows NT.

Time stamp and creation time have the following format:

Bits	*Range*	*Translated Range*	*Valid Range*	*Description*
`0..4`	`0..31`	`0..62`	`0..59`	`Seconds/2`
`5..10`	`0..63`	`0..63`	`0..59`	`Minutes`
`11..15`	`0..31`	`0..31`	`0..23`	`Hours`

Date stamp, last accessed date, and creation date have the following format:

Bits	*Range*	*Translated Range*	*Valid Range*	*Description*
`0..4`	`0..31`	`0..31`	`1..28 up to 1..31`	`Day, blame Julian for complexity`
`5..8`	`0..15`	`0..15`	`1..12`	`Month`
`9..15`	`0..127`	`1980..2107`	`1980..2107`	`Year, add 1980 to convert`

Generally, creation time and date say when the file was created. Accessed time and date say when the file was last modified. Time and date stamps are set to the time that applications want you to think is the time of the last modification.

Starting cluster is the beginning cluster for the file or directory cluster chain. For FAT32, this value consists of the two 16-bit words, and the high four bits of the high word should be masked out. I have never seen any documentation regarding this, but a couple of hours of playing with FAT32 convinced me that this is the case.

Size of the file specifies the real file size in bytes. This value might be in conflict with the file size calculated by going through the cluster chain. Whenever they are in conflict, the smaller value takes over.

Long Filenames

The entries for long filenames look pretty odd because Microsoft tried to maintain compatibility with the older software. However, their format has not changed in FAT32, which sounds somewhat ironic because FAT32 is in no way compatible with the older software. You may notice that these entries look much similar to those of aliases. The difference is, they use the strange combination of attributes, so they are likely to be skipped by the older software. The word "slot" was used for these entries by Galen C. Hunt, and I will stay with his terminology.

Offset in Entry	*Length in Bytes*	*Description*
00	1	Sequence number for the slot
01	10d	First five characters in filename
0B	1	Attribute
0C	1	Reserved, always zero
0D	1	Alias checksum
0E	12d	Next six characters in filename
1A	2	Starting cluster
1C	4	Last two characters in filename

The starting cluster number is always zero, and the attribute is always 0Fh.

Slots are always positioned right before the alias in the directory. The closest to the alias slot contains the first thirteen characters of the long filename. The slot above it contains the next thirteen characters, and so on, up to 256 characters. Additionally, the sequence number of the slot contains its number in the slot chain, starting from one. The sequence number for the last slot in the chain is or'ed with 40h to indicate end of chain.

Slot Number	*Sequence Number*	*Characters*
3	43h	me.text
2	02	y long filena
1	01	This is a ver
Alias	Alias	THISIS~1.TEX

If the length of the filename is not the multiple of thirteen, the name in null-terminated. Otherwise, it is not null-terminated. If after null termination there are any characters left in the slot, they are filled with `FFFF`.

Checksum contains the checksum for the corresponding alias. It is calculated in the following way:

```
unsigned char sum, i;
for(sum=i=0; i<11; i++)
    sum=(((sum&1)<<7)|((sum&0xFE)>>1))+name[i];
```

In a more common language, they rotate the sum right with cycling and add the next character at each iteration. Note that the checksum is case-sensitive.

When the file is deleted, all entries for the long filename start with `E5`.

What can go wrong with long filenames? Give some space to your imagination...

Operations

Let us perform some operations with long filenames and aliases. I do not describe the steps that are related to cluster chain management because they were already described. As for search operations, they should really be performed in a different way to be any quick. However, the directory structure itself does not suggest anything other than a sequential search.

Find File by Alias

1. Start from the top of the directory.
2. Check the first byte of the entry. If it is zero, finish.
3. Then check the attribute. Skip the entry if the attribute is invalid, `0F`, or does not correspond to the specific search demands.
4. Finally, compare the filename and extension with the search pattern. Don't forget about wildcards.
5. If you have not found the needed entry yet, go to the next entry. Watch for the end of the directory. If not end of the directory, go to step two.

Find File by Long Name

1. Start from the top of the directory.
2. Check the first byte of the entry. Exit the loop if it is zero. Skip to the next entry if it is not one and not `41h`.
3. Retrieve the long name. Skip to the next entry in case of an error at this stage. Compare the long name against your search string. If you search operation has any restrictions for attributes, date, etc. use the fields of the corresponding alias for comparison. Skip to the next entry if the current one doesn't match.
4. If you have not found the needed entry, go to the next one. Watch for end of directory.

Retrieve the Long Name

1. Start from the entry with the sequence number one or `41h`. Start with an empty string. The string must be at least 512 bytes long. Any error in the first entry should be interpreted as an invalid long name. Any error in the next entries should end up in the valid but truncated long name. Invalid long name is signaled by the empty string on return.
2. Check the attribute. It must be `0Fh`.
3. If sequence number is one or `41h`, check the next entry in the directory. It must be the alias for the long name. Check if its attribute is valid. Calculate the checksum.
4. Compare the checksum in the slot with the checksum calculated for the alias. They must be equal.
5. Concatenate the current string and the strings in the slot. Remember that strings may be null-terminated. Watch for the invalid characters and replace them with underscores.
6. If sequence number and `40h`, or the string in the slot is null-terminated, or out of buffer space, return your current string.
7. Else go to the previous entry in the directory. Watch for the beginning of the directory.

Allocate New Entry

When you need to allocate a new entry in the directory, look through the entries from the beginning. Use the first one that is marked as deleted or the first one that has zero as the first character of the filename. Expand the directory if no entries are free. Also follow this simple algorithm to allocate the block of consecutive entries. You will need it for aliasing. You can make it a bit smarter by overwriting the deleted entries only after all free entries are used up, but because the probability that deleted files will be recovered is low and memory for the directory cache is expensive, I do not recommend it.

Delete and Undelete File or Directory

To delete the file:

1. Set the first character of the alias to `E5`.
2. Set the first byte for each of the slots for the long name to `E5`.
3. Fill the cluster chain with zeroes.

To delete the directory, follow these very steps, but first check if the directory to be deleted is empty. Empty directory consists of no entries but `"."`, `".."`, and, possibly, deleted entries. Do not delete the directory unless it is empty.

Because the directory entry is otherwise untouched, recovery is possible. However, recovery is not guaranteed. Furthermore, it is not guaranteed that the recovered file will have the same contents as the original file. To recover the deleted file:

1. Find the directory entry marked as deleted. Begin from the starting cluster and check the number of the consecutive FAT entries corresponding to the file size. If they all are zero, build a file chain through them and conclude that undeleting is completed. If any is marked as physically bad, just skip it. Otherwise, conclude that undeleting is impossible.
2. Ask the user for the first character in the filename, or get the first character from the long filename. Write the first character to the directory entry (for the alias).
3. Read the previous directory entry, if any. If this entry is also marked as deleted and its attribute is `0F`, this is probably the first slot for the long filename. Continue going up the directory while these conditions are true. Build the chain for the long filename.

Create New Directory

1. Allocate the necessary number of entries. You should have enough space to store the alias and, possibly, the long filename. The case when the long filename is not created was mentioned earlier.
2. Allocate at least one cluster. The recommended number of clusters is such a number that can contain 512 directory entries.
3. Write the first two required entries "." and "..".
4. Write the alias and the long filename. Set the beginning cluster field of the alias to the just allocated cluster chain.

Physical Interfaces

Introduction

I assume that you are already familiar with what disks are as physical devices. The purpose of these documents is to describe disk logical structures, but not disk interfaces. To understand the material better, I recommend that you familiarize yourself with EIDE or SCSI disk controllers. Only the things that are absolutely necessary for understanding further material are explained here.

CHS and LBA

In a nutshell, disk consists of sectors. Sector is a group of bytes (octets), the minimal unit of transaction between the disk and the system. Although sector size can theoretically be pretty anything, it is safe to assume that one sector is 512 bytes long.

There are two major modes for addressing sectors on disk. One is called **CHS** mode. Sectors in this mode are grouped under heads. How many sectors are under one head is determined by the device parameter called`SectorsPerHead`. Further, heads are grouped under cylinders. How many heads are under one cylinder is determined by another device parameter, `HeadsPerCylinder`. The remaining disk parameter is `NumberOfCylinders`, which, together with `HeadsPerCylinder` and `SectorsPerHead` unambiguously determines disk capacity in CHS mode:

```
CHSCapacity=NumberOfCylinders*HeadsPerCylinder*SectorsPerHead*512
```

Needless to say, neither of disk parameters should be equal to zero. If any of them is equal to zero, such disk should be considered invalid

You need `Cylinder`, `Head`, and `Sector` to identify the sector in CHS mode. Identification is unique as long as you stay in the same CHS translation mode. However, it is not unique across different translation modes. Devices, in conjunction with BIOS, may translate `Cylinder` and `Head` values to make it possible for BIOS to access large amounts of data. The mechanism of this translation is discussed below. Head and cylinder numbers always start from zero, sector numbers start from one. You will likely perform some arithmetic operations on CHS values. The formulas below will help you. Note that you can use them only if you know disk parameters.

To add S sectors to some CHS address follow these steps:

```
Sector=Sector+S;
Head=Head+(Sector div SectorsPerHead);
Sector=Sector mod SectorsPerHead;
Cylinder=Cylinder+(Head div HeadsPerCylinder);
Head=Head mod HeadsPerCylinder;
```

You should always check if the resulting `Cylinder` is less than `NumberOfCylinders`. If this is not the case, the data on disk that made you do this calculation was corrupted.

To subtract one sector from the given CHS address follow these steps:

```
if Sector=0
  Sector=SectorsPerHead-1;
  if Head=0
    Head=HeadsPerCylinder-1;
    if Cylinder=0
      error, data was corrupted
    else
      Cylinder=Cylinder-1;
  else
    Head=Head-1;
else
  Sector=Sector-1;
```

The second mode for addressing sectors is called **LBA**. It stands for Linear Block Addressing, which means exactly that. You need only sector number to address the sector. Linear sector numbers start from zero. Disk capacity in LBA mode is determined by `NumberOfSectors`, which is certainly device parameter. The following equality must always hold:

```
LBA=(Cylinder*HeadsPerCylinder+Head)*SectorsPerHead+Sector-1
```

Thus, CHS `(0, 0, 1)` corresponds to LBA `(0)`. One might think that CHS `(NumberOfCylinders-1, HeadsPerCylinder-1, SectorsPerHead)` corresponds to LBA `(NumberOfSectors-1)`, but this is not the case. In practice, one can usually address slightly more sectors in LBA mode than he does in CHS mode. According to my rules described earlier, you should ignore extra LBA sectors that might exist in CHS partitions. Refer to EIDE or SCSI documentation for the ways of determining physical disk parameters. Refer to your BIOS manual for the way of determining logical disk parameters.

There are two key operations that can be performed on sectors in any mode: read and write. Refer to the manuals described above for the ways of performing these operations. From now on, you must be able to read and write sectors given their LBA or CHS address.

Limitations

As you might have noticed reading documentation, there are some limitations on how much space is addressable in LBA and CHS modes. For example, ATA standard reserves two 16 bit words for `NumberOfSectors`, which means that the highest possible disk capacity in LBA mode is 2 TB. ATA standard also reserves 16 bit word for each of the disk parameters in CHS mode, thus the highest disk capacity in CHS mode is 131072 TB. In reality, `HeadsPerCylinder` is usually limited to 16 and `SectorsPerHead` is limited to 64, so the real highest disk capacity for CHS mode is only 32 GB. As will be shown later, BIOS imposes its own limitations:`NumberOfCylinders` cannot exceed 1024, `SectorsPerHead` is limited to 64, and `HeadsPerCylinder` is limited to 256. While

BIOS itself can support up to 8G disks, comparing BIOS and EIDE limitations yields the unpleasant 512 MB barrier in CHS mode (BIOS barrier in LBA mode is 2 TB). How this problem is dealt with is the subject of the next section.

CHS Translation

Since many current EIDE drives still have 16 or less heads and more than 1024 cylinders, and far not all drives support the necessary hardware CHS translation modes, BIOS provides software CHS translation. It does some arithmetic on `HeadsPerCylinder` and `NumberOfCylinders` to make more than 512 MB accessible. BIOS maps logical CHS address to physical CHS address by multiplying logical `Cylinder` by the power of two and dividing logical `Head` by the same power of two. Conversely, one can translate physical CHS address to logical CHS address by dividing physical `Cylinder` by the power of two and multiplying physical `Head` by the same power of two. Certainly, the same arithmetic is performed on `NumberOfCylinders` and `HeadsPerCylinder`. `Sector` and `SectorsPerHead` remain untouched by the translation. The corresponding power of two is the *minimal*power of two that makes logical `NumberOfCylinders` less than or equal to 1024. You can pick it from the table:

Physical Number of Cylinders	*Physical Heads Per Cyl*	*Physical Sectors Per Head*	*Power of Two for Translation*
`1..1024`	`1..256`	`1..64`	`1`
`1025..2048`	`1..128`	`1..64`	`2`
`2049..4096`	`1..64`	`1..64`	`4`
`4097..8192`	`1..32`	`1..64`	`8`
`8193..16384`	`1..16`	`1..64`	`16`
`16385..32768`	`1..8`	`1..64`	`32`
`32769..65536`	`1..4`	`1..64`	`64`

It is obvious that with some values of `NumberOfCylinders` disk may have different capacity in different CHS modes. However, I have never seen this in practice. The situation when physical disk parameters do not fit in this table is only possible if disk capacity is more than 8 GB or when disk has more than 64 sectors per head. Such disks are not yet handled by BIOS correctly. That is why I cannot provide any guidelines for these cases.

LARGE, NORMAL, and LBA

Finally, I would like to say some words about BIOS settings for EIDE drives. BIOS routines can use different addressing modes for accessing drives. **NORMAL** BIOS mode uses CHS addressing mode with no software translation. **LARGE** mode uses software CHS translation as described above. It is identical to NORMAL mode for the drives that have 1024 or less cylinders. **LBA** mode uses LBA addressing. In my experience, this mode also provides software CHS translation for the requests that are made in CHS notation. **AUTO** selects the best mode automatically.

Partitions

Introduction

This document concerns only hard drives. Floppy disks, CD-ROM, and ZIP drives are not normally split into partitions. You should familiarize yourself with CHS and LBA sector addressing modes before reading this document.

Here we go

The first sector of the hard disk must contain the information about disk structure. If the disk is booted from, it must also contain a boot loader. As you remember, this sector is at CHS `(0, 0, 1)` or LBA `(0)`. To verify that this sector is valid, one should read a 16 bit word at offset `1FE` from the beginning of this sector and compare it to `AA55` (needless to say, the word is in Intel little endian notation). If this word is different from `AA55`, the hard disk should not be considered valid. The typical behavior of decent operating systems in this case is to silently ignore this hard disk.

The information about disk partitions starts from the predefined offset `1BE` from the beginning of the first sector. There are four partition table entries at this offset, each taking up 10h bytes:

Partition	*Offset*
1	1BE
2	1CE
3	1DE
4	1FE

The structure of each entry is below:

Offset in entry	*Size in bytes*	*Meaning*
0	1	Boot indicator
1	1	Beginning head number
2	1	Beginning sector and high cylinder number
3	1	Beginning low cylinder number
4	1	System indicator
5	1	Ending head number
6	1	Ending sector and high cylinder number
7	1	Ending low cylinder number
8	4	Number of sectors preceding the partition
C	4	Number of sectors in the partition

Let us take a closer look at each field and its meaning.

Boot indicator is `80h` for bootable partitions and `zero` for other valid partitions. If it is neither `80h` nor `zero`, the corresponding partition should be considered invalid. Invalid partitions are silently ignored by most of the systems. For the disk to be bootable, exactly one entry must contain `80h` in this field. However, the condition when more than one entry is marked as bootable should not be considered fatal error. If two or more of the partitions are bootable, all should be treated as valid partitions.

Beginning cylinder, head, and sector numbers point to the first sector of the partition in CHS notation. Two high bits of "`beginning sector and high cylinder number`" are two high bits of cylinder number. Similarly,*ending cylinder, head, and sector* point to the last sector in the partition. They are in exactly the same format.

```
Sector="sector and high cylinder number" and 4F
Cylinder="low cylinder number"+(("sector and high cylinder number" shr 6) shl 8)
```

Thus, BIOS imposes the following limitations:

CHS	*Minimum value*	*Maximum value*	*Number of bits*	*Number of values*
Sector	`1`	`64`	`6`	`64`
Head	`0`	`255`	`8`	`256`
Cylinder	`0`	`1023`	`10`	`1024`

If you multiply the values in the last column out and multiply the result by sector size, you will get the hard disk size limitation of 8 GB. If you consider that older IDE implementations supported up to 16 heads, you will get the infamous 512 MB barrier. How this barrier is dealt with was described in previous documents.

If disk was partitioned in LARGE or LBA modes, CHS addresses in these fields are logical CHS addresses. Therefore, you should better use BIOS services to access CHS partitions. But since BIOS is terribly inefficient, you may take a risk of providing the same software CHS translation as it does. To do it, get physical device parameters from the device and logical device parameters from BIOS. If any of them equals to zero, the disk is misconfigured and should not be used. Then compare physical `SectorsPerHead` against logical `SectorsPerHead`. If they are not equal, BIOS is using some strange CHS translation mode, and you should use BIOS services to access the partition. If they are equal, divide physical `NumberOfCylinders` by logical `NumberOfCylinders` and remember the result. Then divide logical `HeadsPerCylinder` by physical `HeadsPerCylinder`. If the quotient is not the same as the result of the previous division, or if the quotient is not the power of two, you should use BIOS services for accessing the partition. Otherwise use this quotient to provide BIOS-like CHS translation.

Certainly, sanity check on these CHS values is absolutely necessary. Systems vary in the ways how they handle this stage. I recommend the following simple algorithm. I insist that you copy partition table to the temporary buffer with easier to handle format. Zero-extend `Cylinder`, `Head`, and `Sector` values to 16 bit words and use signed arithmetic. First of all, make sure that none of the partitions starts at CHS `(0, 0, 1)`. Remove all that do. Then make sure that none of the partitions overlap. If any two partitions do overlap, my best guess would be truncating the preceding partition so that they don't overlap any more. Do this by setting ending sector address of the preceding partition to the beginning sector address of the next partition minus one. Do not forget that arithmetic operations on CHS addresses have peculiarities. After that check that none of the ending sector addresses exceeds physical disk size. Truncate ending sector addresses, if necessary, by setting them to CHS `(NumberOfCylinders-1, HeadsPerCylinder-1, SectorsPerHead)`. Finally, make sure that each partition has at least one sector in it. If beginning sector address is greater than or equal to ending sector address, such partition should be considered invalid. By now you should have valid partition information in your temporary buffer. Use this information to access partitions. But unless you are a disk analyzer program, do not write any changes to the partition table itself.

Now I would like to say a couple of words about compatibility. Many programs expect that the very first partition starts at CHS `(0, 1, 1)` and that all partitions except the first one start at even cylinder boundaries, CHS `(2*k, 0, 1)`. They also expect that partitions cover the disk completely and are consecutive on the disk. If you are paranoid, also make sure that the very first partition is bootable.

Number of sectors preceding the partition is its LBA address. *Number of sectors in the partition* is its LBA length. Note that these values are essentially duplicating the CHS values that are described above. They must define exactly the same partition as CHS values. When LBA and CHS values are in conflict, some heuristics is necessary. Check system indicator to find out which values should be used. Give precedence to LBA values in LBA partitions and to CHS values in CHS partitions. Do not make any assumptions about the partitions you don't know. You should not access them anyway.

The same sanity check should be performed on LBA values, as was on CHS values. Remove the partitions that start at LBA `(0)`. If any two partitions overlap, truncate the preceding partition. Make sure that none of the sums of partition address and size exceeds disk capacity and truncate partition size if necessary. Remove partitions that have zero size.

Note that sanity check across CHS and LBA partitions would be very nice. If CHS and LBA addresses are consistent for each partition, it is easier to perform this check by converting all addresses to LBA, doing the check, and then converting the results back to CHS. Unfortunately, if CHS and LBA addresses of any partition are in conflict, sanity check can do more harm than good because of different CHS translation modes. My recommendation in this case is to do separate sanity checks between CHS partitions and LBA partitions, but not across them.

Some of the *system indicators* are in the table below:

Value	*System*	*Capacity*	*Translation mode*
00	Unknown		
01	DOS FAT12, 16 bit sector number	<10MB	CHS
02	XENIX		
04	DOS FAT16, 16 bit sector number	<32MB	CHS
05	DOS Extended partition		CHS
06	DOS 4.0 FAT16, 32 bit sector number	>=32MB	CHS
0B	DOS FAT32	<2TB	CHS
0C	DOS FAT32	<2TB	LBA
0E	DOS FAT16, 32 bit sector number	>=32MB	LBA
0F	DOS Extended partition		LBA
51h	Ontrack extended partition		
64h	Novell		

Note that 2 TB limit for partition type `0C` is not in contradiction with BIOS 8 GB limitation because LBA addressing is used. However, I don't know where the 2 TB limit for partition type `0B` comes from. Well, it comes from one of the documents in the Microsoft Knowledge Base, but I fail to see how the CHS partition larger than 8 GB can be implemented.

This table does not mean to be exhaustive. New operating systems and utilities appear and die here and there, so this table will always be incomplete. Common sense says that when you find an unfamiliar value in the table, you should not attempt to modify any of the sectors in this partition. The best policy is to silently ignore such partition.

DOS Extended Partitions

Two of the system indicators deserve special attention. They are extended DOS partitions `05` and `0F`. Extended DOS partition contains a secondary partition table and some space for nested partitions. The secondary partition table is right at the beginning sector of the DOS extended partition. Its format is almost the same as that of the main partition table described above. However, some things must be taken into consideration.

Extended partition must have one or two entries. I do not encourage you to handle the situation when extended partition has more than two entries because this partition is very DOS specific and unlikely to be changed. I think, it is safe to check only the first two entries of the extended partition.

One of the entries should define a DOS partition. For this entry, CHS and LBA addresses are relative to the sector containing the secondary partition table. Another entry may define the deeper nested extended DOS partition. In this case, CHS and LBA addresses are relative to the beginning of the physical disk. This also implies that there can be many nested extended partitions, so the function that looks through them should keep track of used resources and handle the situation when it runs out of buffers. All nested partitions must fall within the mother extended partition. Truncate them if they don't. None of the partitions can be bootable, but this should be taken into consideration only by special disk analyzing programs.

Finally, primary partition may have multiple extended partition entries.

Other DOS Notes

The information below is specific to DOS operating system.

Everything that is not DOS extended partition is called primary partition. For the unclear reasons DOS manuals say that every partition table should contain only one primary partition. But DOS does support multiple primary partitions correctly, and you should too.

Letters for DOS disks are assigned in the following order. Letters 'A' and 'B' are assigned to floppy drives. If there is one floppy drive, it is assigned 'A'. The letter 'C ' is assigned to the first primary partition on the first hard drive. The next letters are assigned to first primary partitions on all hard drives. Then go volumes inside the extended partitions on all hard drives, in the order they appear. First, all extended partitions on the first drive are given letters, then - on the second drive, etc. Then the remaining primary partitions on all drives are assigned letters. Finally, CD-ROM drives take their share of letters. Note that the letters that CD-ROM drives have can be configured by software.

Author: Alex Verstak 3/11/1998