FAT Filesystem

elm-chan.org/docs/fat_e.html

May 24, 2009
Update: October 31, 2020

This is a documentation about FAT filesystem written based on the FAT32 Filesystem Specification (FAT specs below) to know how FAT filesystem works and how to manage it. It is written according to the contents of the FAT specs and the behavior of the standard system (real DOS and Windows), however, there are many improvements and ommissions from the FAT specs. If the behavior of standard system is differ from the FAT specs, it is written in accordance with the real systems rather than FAT specs. And there is a possibility that there is any unintended or intended error in this documentation, so that you need to refer the primary sources of FAT filesystem and make sure behavior of the real systems when write FAT driver or utility. For exFAT filesystem, please refer here in addition to this documentation.

Introduction

The *filesystem* generally denotes the entire system to manage data stored on the storage, however, this document describes the data format of FAT filesystem on the storage device.

The FAT file system originated around 1980 and is the filesystem that was first supported by MS-DOS. It was originally developped a simple filesystem suitable for floppy disk less than 500k bytes in size. Over the time, its specs has been expanded to support laeger media as increasing its capacity. FAT is the abbreviation of File Allocation Table, which is the array to manage allocation of data area and the name of the file system itself. Currently there are three FAT sub-types, FAT12, FAT16 and FAT32. These are developped in order of the number and completely backward compatible with older one. (FAT16 always include FAT12, FAT32 includes all FAT types)

About Notations in this Document

Numbers starting with "0x" are assumed to be hexadecimal numbers, and others are decimal numbers.

The prefix K, M, G and T of each unit is assumed to be 2^{10}, 2^{20}, 2^{30} and 2^{40} respectively.

The fragment of program codes contained in this document is written assuming C language, but it is not strict to the syntax.

32-bit value and 16-bit value are arbitrary mixed in the fragment of the program code. The programmer needs be aware of the data loss due to the type conversion and how to avoid it. Also, every data type is assumed to be *unsigned*. Do not calculate in signed because it may result in unintended results.

Basics of FAT File System

Sector

Sector is the smallest unit of data block on the storage to read and write the storage device. The common sector size is 512 bytes and a larger sector size is sometimes used for some type of storage media. Each sector on the storage device is addressed by a sector number assgined in order from top of the storage device. Since volumes are not that always placed at top of the storage, in this document, "sector number" denotes the relative location origin from top of the volume and "physical sector number" denotes the absolute location origin from top of the storage device.

FAT Volume

A FAT file system completes itself is called logical volume (or logical drive). The FAT logical volume consists of three or four areas, each of them consists of one or more sectors and located on the volume in order of as follows. (FAT volume map)

1. Reserved area (volume configuration data)
2. FAT area (allocation table for data area)
3. Root directory area (not present on FAT32 volume)
4. Data area (contents of file and directory)

Data Forms on the Storage

The FAT file system was initially developed for IBM PC with x86 processors. The most important thing is that data structures in the FAT filesystem on the storage is stored in *little endian*. If the architecture of the platform to access the FAT filesystem is big endian, an endian conversion is required when accessing the structures of the FAT filesystem. Also, word data in multiple bytes are not that always aligned to the word boundaries. If the processor cannot access the unaligned word data, it will need to access the data in byte-by-byte. For this reason, accessing the FAT volume as C structure member discards the code portability. Accessing the FAT volume in byte-by-byte as simple byte array instead of C structure gives the best code portability.

Boot Sector and BPB

The most important data structure in the FAT volume is *BPB* (BIOS Parameter Block), where the configuration parameters of the FAT volume are stored. The BPB is placed in the *boot sector*. The boot sector is often referred to as VBR (Volume Boot Record) or PBR (Private Boot Record), but it is simply the first sector of the reserved area, the first sector of the volume.

BPB has often been changed as new feature of the FAT file system is added. The first confusion that occurred was due to the newly-establishment of BPB. In the MS-DOS Ver.1, there was no BPB in the boot sector. In the first version of the FAT file system, there is only a few disk format (single-sided and double-sided 5.25 inch floppy disk), and the disk format is determined by referring to the first byte (lower 8 bits of the first FAT item) of FAT starting at the next sector of the boot sector.

At the MS-DOS Ver.2, this method of determination of disk format is superseded by referring the BPB in the boot sector and the determination by referring the first byte of FAT has not been supported any longer. Now all FAT volumes must have a BPB in the boot sector. BPB was sometimes changed and it brought confusions about determination of disk format. (e.g. Which is the correct parameter? What means this parameter? How should I use this parameter?) The authorized volume recognization method is described in FAT specs a time later.

At the first time, problems about deterioration of disk usage efficiency and limitation of the number of files on a volume appered due to the disk size got lager by widespread use of hard disks and FAT16 has newly supported at MS-DOS Ver.3. However, a new problem occurred immediately after this change. Since the size of field to indicating the size of the volume was 16 bits, the supported volume size is less than 65536 sectors (32 MB at 512 bytes/sector). For this reason, a 32-bit field has been added to at MS-DOS Ver.3.31 and it can support 128 MB for FAT12 and 2 GB for FAT16 (at 32 KB/cluster).

The last change of BPB was at Windows 95 OSR2 where the FAT32 appeared. At that time, number of files and maximum capacity of the FAT16 volume had reached at some applications. The FAT32 as final version of FAT filesystem removed the limitations of FAT16 and spports upto 2 TB of volume size (at 512 bytes/sector). However Microsoft recommends any filesystem other than FAT, such as NTFS for fixed disk and exFAT for removable disk, for the volumes larger than 32 GB.

The following table shows the data fields of the boot sector. Any field named with heading *BPB_* are part of the BPB. Any field named with heading *BS_* is not a part of BPB but it only a part of the boot sector.

The fields in the first 36 bytes are common field for all FAT types and the fields from byte offset 36 depends on whether the FAT type is FAT32 or FAT12/FAT16. The method of determining FAT type is described at next section.

Field name	Offset	Size	Description
BS_JmpBoot	0	3	Jump instruction to the bootstrap code (x86 instruction) used by OS boot sequence. There are two type of formats for this field and the former format is prefered. `0xEB, 0x??, 0x90` (Short jump + NOP) `0xE9, 0x??, 0x??` (Near jump) ?? is the arbitrary value depends on where to jump is. In case of any format out of these formats, the volume will not be recognized by Windows.

BS_OEMName	3	8	"MSWIN 4.1" is recommended but also "MSDOS 5.0" is often used. There are many misconceptions about this field. This is only a name. Microsoft's OS does not pay any attention to this field, but some FAT drivers do some reference. This string is recommended because it is considered to minimize compatibility problems. You can set something else, but some FAT drivers may not recognize that volume. This field usually indicates name of the system created the volume.
BPB_BytsPerSec	11	2	Sector size in unit of byte. Valid values for this field are 512, 1024, 2048 or 4096. Microsoft's OS properly supports these sector sizes, but many FAT drivers assume the sector size is 512 and do not check this field. For this reason, 512 should be used for maximum compatibility. However, you should not misunderstand that it is only related to compatibility. This value must be the same as the sector size of the storage contains the FAT volume.
BPB_SecPerClus	13	1	Number of sectors per allocation unit. In the FAT file system, the allocation unit is called *Cluster*. This is a block of one or more consecutive sectors and the data area is managed in this unit. The number of sectors per cluster must be a power of 2. Therefore, valid values are 1, 2, 4,... and 128. However, any value whose cluster size (BPB_BytsPerSec * BPB_SecPerClus) exceeds 32 KB should not be used. Recent systems, such as Windows, supprts cluster size larger than 32 KB, such as 64 KB, 128 KB, and 256 KB, but such volumes will not be recognized correctly by MS-DOS or old disk utilities.
BPB_RsvdSecCnt	14	2	Number of sectors in reserved area. This field must not be 0 because there is the boot sector itself contains this BPB in the reserved area. To avoid compatibility problems, it should be 1 on FAT12/16 volume. This is because some old FAT drivers ignore this field and assume that the size of reserved area is 1. On the FAT32 volume, it is typically 32. Microsoft's OS properly supports any value of 1 or larger.

BPB_NumFATs	16	1	Number of FATs. The value of this field should always be 2. Also any value eaual to or greater than 1 is valid but it is strongly recommended not to use values other than 2 to avoid compatibility problem. Microsoft's FAT driver properly supports the values other than 2 but some tools and FAT drivers ignore this field and operate with number of FAT is 2. The standard value for this field 2 is to provide redudancy for the FAT data. The value of FAT entry is typically read from the first FAT and any change to the FAT entry is refrected to each FATs. If a sector in the FAT area is damaged, the data will not be lost because it is duplicated in another FAT. Therefore it can minimize risk of data loss. On the non-disk based storages, such as memory card, such redundancy is a useless feature, so that it may be 1 to save the disk space. But some FAT driver may not recognize such a volume properly.
BPB_RootEntCnt	17	2	On the FAT12/16 volumes, this field indicates number of 32-byte directory entries in the root directory. The value should be set a value that the size of root directory is aligned to the 2-sector boundary, `BPB_RootEntCnt * 32` becomes even multiple of `BPB_BytsPerSec`. For maximum compatibility, this field should be set to 512 on the FAT16 volume. For FAT32 volumes, this field must be 0.
BPB_TotSec16	19	2	Total number of sectors of the volume in old 16-bit field. This value is the number of sectors including all four areas of the volume. When the number of sectors of the FAT12/16 volumes is 0x10000 or larger, an invalid value 0 is set in this field, and the true value is set to `BPB_TotSec32`. For FAT32 volumes, this field must always be 0.
BPB_Media	21	1	The valid values for this field is 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE and 0xFF. 0xF8 is the standard value for non-removable disks and 0xF0 is often used for non partitioned removable disks. Other important point is that the same value must be put in the lower 8-bits of FAT[0]. This comes from the media determination of MS-DOS Ver.1 and not used for any purpose any longer.

BPB_FATSz16	22	2	Number of sectors occupied by a FAT. This field is used for only FAT12/16 volumes. On the FAT32 volumes, it must be an invalid value 0 and `BPB_FATSz32` is used instead. The size of the FAT area becomes `BPB_FATSz??` * `BPB_NumFATs` sectors.
BPB_SecPerTrk	24	2	Number of sectors per track. This field is relevant only for media that have geometry and used for only disk BIOS of IBM PC.
BPB_NumHeads	26	2	Number of heads. This field is relevant only for media that have geometry and used for only disk BIOS of IBM PC.
BPB_HiddSec	28	4	Number of hidden physical sectors preceding the FAT volume. It is generally related to storage accessed by disk BIOS of IBM PC, and what kind of value is set is platform dependent. This field should always be 0 if the volume starts at the beginning of the storage, e.g. non-partitioned disks, such as floppy disk.
BPB_TotSec32	32	4	Total number of sectors of the FAT volume in new 32-bit field. This value is the number of sectors including all four areas of the volume. When the value on the FAT12/16 volume is less than 0x10000, this field must be invalid value 0 and the true value is set to `BPB_TotSec16`. On the FAT32 volume, this field is always valid and old field is not used.

FAT12/16/32 common field (offset from 0 to 35)

Since the following fields change depending on whether the volume is FAT12/16 or FAT32, the FAT type must be determined prior to refer these fields. Also there are some fields exist in only FAT32 volumes and not exist in FAT12/16 volumes.

Field name	**Offset**	**Size**	**Description**
BS_DrvNum	36	1	Drive number used by disk BIOS of IBM PC. This field is used in MS-DOS bootstrap, 0x00 for floppy disk and 0x80 for fixed disk. Actually it depends on the OS.
BS_Reserved	37	1	Reserved (used by Windows NT). It should be set 0 when create the volume.

BS_BootSig	38	1	Extended boot signature (0x29). This is a signature byte indicates that the following three fields are present.
BS_VolID	39	4	Volume serial number used with `BS_VolLab` to track a volume on the removable storage. It enables to detect a wrong media change by FAT driver. This value is typically generated with current time and date on formatting.
BS_VolLab	43	11	This field is the 11-byte volume label and it matches volume label recorded in the root directory. FAT driver should update this field when the volume label in the root directory is changed. MS-DOS does it but Windows does not do it. When volume label is not present, `″NO NAME    ″` should be set in this field.
BS_FilSysType	54	8	`″FAT12   ″`, `″FAT16   ″` or `″FAT     ″`. Many people think that this string has any effect in determination of the FAT type but it is clearly a misrecognization. From the name of this field, you will find that this is not a part of BPB. Since this string is often incorrect or not set, Microsoft's FAT driver does not use this field to determine the FAT type. However, some old FAT drivers use this string to determine the FAT type, so that it should be set based on the FAT type of the volume to avoid compatibility problems.
BS_BootCode	62	448	Bootstrap program. It is platform dependent and filled with zero when not used.
BS_BootSign	510	2	0xAA55. A boot signature indicating that this is a valid boot sector.
	512		When the sector size is larger than 512 bytes, rest field in the sector should be filled with zero.

Fields for FAT12/16 volumes (offset from 36)

Field name	**Offset**	**Size**	**Description**
BPB_FATSz32	36	4	Size of a FAT in unit of sector. The size of the FAT area is `BPB_FATSz32 * BPB_NumFATs` sector. This is an only field needs to be referred prior to determine the FAT type while this field exists in only FAT32 volume. But this is not a problem because `BPB_FATSz16` is always invalid in FAT32 volume.

BPB_ExtFlags	40	2	Bit3-0: Active FAT starting from 0. Valid when bit7 is 1. Bit6-4: Reserved (0). Bit7: 0 means that each FAT are active and mirrored. 1 means that only one FAT indicated by bit3-0 is active. Bit15-8-4: Reserved (0).
BPB_FSVer	42	2	FAT32 version. Upper byte is major version number and lower byte is minor version number. This document describes FAT32 version 0.0. This field is for futuer extension of FAT32 volume to manage the filesystem verison. However, FAT32 volume will not be updated any longer.
BPB_RootClus	44	4	First cluster number of the root directory. It is usually set to 2, the first cluster of the volume, but it does not need to always be 2.
BPB_FSInfo	48	2	Sector of FSInfo structure in offset from top of the FAT32 volume. It is usually set to 1, next to the boot sector.
BPB_BkBootSec	50	2	Sector of backup boot sector in offset from top of the FAT32 volume. It is usually set to 6, next to the boot sector but 6 and any other value is not recommended.
BPB_Reserved	52	12	Reserved (0).
BS_DrvNum	64	1	Same as the description of FAT12/16 field.
BS_Reserved	65	1	Same as the description of FAT12/16 field.
BS_BootSig	66	1	Same as the description of FAT12/16 field.
BS_VolID	67	4	Same as the description of FAT12/16 field.
BS_VolLab	71	11	Same as the description of FAT12/16 field.
BS_FilSysType	82	8	Always ″FAT32 ″ and has not any effect in determination of FAT type.
BS_BootCode32	90	420	Bootstrap program. It is platform dependent and filled with zero when not used.
BS_BootSign	510	2	0xAA55. A boot signature indicating that this is a valid boot sector.
	512		When the sector size is larger than 512 bytes, rest part in the sector should be filled with zero.

Fields for FAT32 volumes (offset from 36)

There is another important thing about boot sector. The boot sector needs to be valid only if the boot signature (BS_Sign) contains 0xAA55. If not, the boot sector is invalid. Many FAT documents describe about this field, "Boot signature is put at the end of the boot sector". This is correct only if the sector size is 512, but it is incorrect in other case. The boot signature must always be put at offset 510 (also it is not bad for both offset 510 and the end of the sector contain this signature). Microsoft's disk formatter fills the rest part of boot sector with zeros if the sector size is larger than 512. BS_Sign is often not set in the volumes formatted with old version of MS-DOS.

Size of the volume, the value of BPB_TotSec??, might be smaller than the container (storage or partition) where the volume is contained in. It is not a problem at all. The FAT volume sometimes gets this state due to alignment or resize of volume. Such an alignment hole wastes disk space, but it does not mean insanity of the FAT volume itself.

However, if BPB_TotSec?? is larger than the volume's container, it can be said that the FAT volume is serious state, corrupted volume or incorrectly created volume. Any operation to such volume can result a catastrophic data loss, so that the FAT driver should reject the volume if it detected such condition.

Calculating Parameters

The offset and size of each area are calculated from the parameters in BPB as shown below.

Since the FAT area is next to the reserved area, its offset and size are:

```
FatStartSector = BPB_ResvdSecCnt;
FatSectors = BPB_FATSz * BPB_NumFATs;
```

The offset and size of the root directory are:

```
RootDirStartSector = FatStartSector + FatSectors;
RootDirSectors = (32 * BPB_RootEntCnt + BPB_BytsPerSec - 1) / BPB_BytsPerSec;
```

The 32 in the equation is the size of a directory entry. A remainder at the divsion is rounded up, but such configuration that gives remainder is not recommended. On the FAT32 volumes, BPB_RootEntCnt is always 0 and the root directory area is not exist. The data area becomes the rest of these areas and it is obtained as follows.

```
DataStartSector = RootDirStartSector + RootDirSectors;
DataSectors = BPB_TotSec - DataStartSector;
```

If the volume does not start from the top of the storage, for example when the storage is partitioned, these start sector numbers are not equal to the physical sector number.

FAT and Cluster

The another important area is FAT. What this structure does is define a linked list of the extents (cluster chain) of a file. Note that both directroy and file is contained in the file and nothing different on the FAT. The directory is really a file with a special attribute that indicates its content is a directory table.

The data area is divided into blocks of a certain number of sectors (`BPB_SecPerClus`) called *cluster* and the data area is managed in this unit. Each item of FAT is associated with each cluster in the data area and the FAT value indicates the state of the corresponding cluster. However, the top two FAT items, FAT[0] and FAT[1], are reserved and not associated with any cluster. The third FAT item, FAT[2], is the item associated with the first cluster of data area and the valid cluster number starts at 2. As for the data recorded on the FAT, see Association of File and Cluster.

FAT is usually duplicated for the redundancy because a damage of any FAT sector results a serious data loss. Number of FAT copyies indicated by `BPB_NumFATs` and the size of FAT area becomes `BPB_FATSz * BPB_NumFATs`. FAT driver typically refers only first FAT copy and any update to the FAT item is refrected every FAT copy.

Determination of FAT sub-type

There are three FAT types, FAT12, FAT16 and FAT32, need to be determined on mount a FAT volume. However there is considerable confusion over exactly how this works and it leads various degree of errors. It is really quite simple how this works.

The FAT type is determined by the count of clusters on the volume and NOTHING ELSE.

The count of clusters is that can exist in the data area, the quotient of size of the data area divided by cluster size. Remainder is ignored if it exist in the result.

```
CountofClusters = DataSectors / BPB_SecPerClus;
```

Once you know the count of clusters you can determine the FAT type. This is done as follows:

- A volume with count of clusters <=4085 is FAT12.
- A volume with count of clusters >=4086 and <=65525 is FAT16.
- A volume with count of clusters >=65526 is FAT32.

This is the only way to determine the FAT type. FAT12 volumes never have clusters more than 4085 and FAT16 volumes never have clusters less than 4086 or more than 65525. If you tried to create an illegal FAT volume out of this rule, the properly designed FAT driver will recognize it as a different FAT type and will not able to access such volume. Maximum cluster count for FAT32 volume is not defined and the practical limit is 268435445.

However, these boundaries are really not strictly settled. From the maximum possible values of the cluster number, it will gets as described above, but there are many variants (1, 2, 16 or more) over the existing documentations and software implementations. For example about maximum count of clusters of FAT12, the FAT specs says it is 4084, while MSDN page says it is 4085 and Windows works with 4085 (FAT driver) or 4086 (chkdsk). Even the authorized documentation and the standad system differ as to what the correct value is, so that it is recommended to avoid count of clusters close to the boundaries when create a FAT volume. The FAT specs says count of clusters should be at least 16 clusters off from the boundaries.

Also, some FAT driver written in out of this rule seem to determine the FAT type without the count of clusters but the string of `BS_FilSysType`. In order to support such the wrong FAT drivers, it is recommended to initialize `BS_FilSysType` with a proper string based on actual FAT type when create a FAT volume.

You will understand what the count of clusters determins the FAT type means which FAT type can be legal depends on the volume size. For example under the condition of cluster size of from 512 to 32768 bytes, it can be saied as follows:

FAT type	Volume size
FAT12	- 128 MB
FAT16	2 MB - 2 GB
FAT32	32 MB - 2 TB

Accessing FAT Entries

An important thing related to FAT is how to access the FAT entries. First of all, you need to know where the FAT entry is located in the FAT. At the FAT16/32, it is quite simple. FAT is a simple generic integer array. The difference from the on-memory array is that the FAT is not on the continuous memory but divided in multiple blocks and stored on the contiguous disk sectors origin from the first sector of the FAT. The location of the FAT entry FAT[N], the sector number and byte offset in the sector, can be got by following calculation.

```
FAT16 entry location:
    ThisFATSecNum = BPB_ResvdSecCnt + (N * 2 / BPB_BytsPerSec);
    ThisFATEntOffset = (N * 2) % BPB_BytsPerSec;

FAT32 entry location:
    ThisFATSecNum = BPB_ResvdSecCnt + (N * 4 / BPB_BytsPerSec);
    ThisFATEntOffset = (N * 4) % BPB_BytsPerSec;
```

The 2 (FAT16) or 4 (FAT32) byte word from the location is the FAT entry to access. The byte order is little endian. The FAT entries never across sector boundaries at FAT16/32.

There is another important thing about FAT32. The FAT entry of FAT32 volume occupies 32 bits, but its upper 4 bits are reserved, only lower 28 bits are valid. The reserved bits are initialized by zero when createing the FAT32 volume, and it should not be changed on the regular use. Therefore, when load the value from the FAT entry of the FAT32 volume, upper 4 bits needs to be and-masked with 0x0FFFFFFF. Also, when store a value into the FAT entry, the upper 4 bits in the FAT entry need to be preserved.

```
Load a value of FAT32 entry:
    ReadSector(SecBuff, ThisFATSecNum);
    ThisEntryVal = *(uint32*)&SecBuff[ThisFATEntOffset] & 0x0FFFFFFF;
```

```
Store a value of FAT32 entry:
    ReadSector(SecBuff, ThisFATSecNum);
    tmp = *(uint32*)&SecBuff[ThisFATEntOffset];
    tmp = (tmp & 0xF0000000) | (NewEntryVal & 0x0FFFFFFF);
    *(uint32*)&SecBuff[ThisFATEntOffset] = tmp;
    WriteSector(SecBuff, ThisFATSecNum);
```

It is a little difficult at the FAT12 volume. The FAT12 entry is in bit field and it needs a complicated operation.

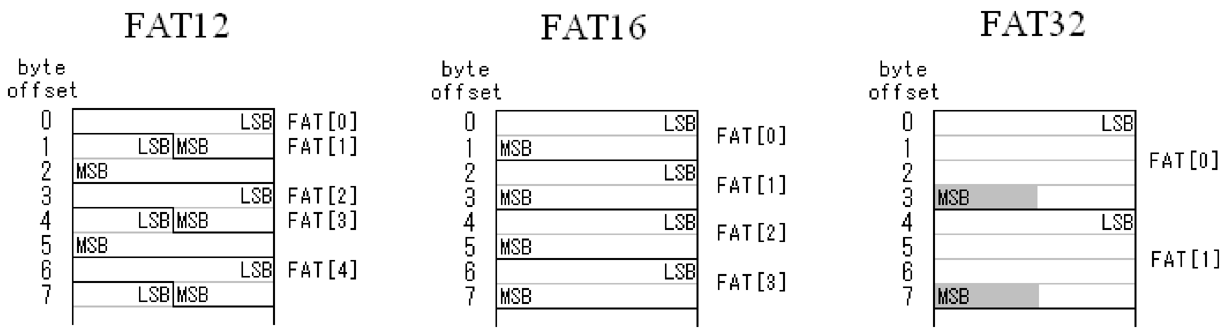
```
FAT12 entry location:
    ThisFATSecNum = BPB_ResvdSecCnt + ((N + (N / 2)) / BPB_BytsPerSec);
    ThisFATEntOffset = (N + (N / 2)) % BPB_BytsPerSec;

Load a value of FAT12 entry:
    ReadSector(SecBuff, ThisFATSecNum);
    if (N & 1) {    /* Odd entry */
        ThisEntryVal = (SecBuff[ThisFATEntOffset] >> 4)
                     | ((uint16)SecBuff[ThisFATEntOffset + 1] << 4);
    } else {        /* Even entry */
        ThisEntryVal = SecBuff[ThisFATEntOffset]
                     | ((uint16)(SecBuff[ThisFATEntOffset + 1] & 0x0F) << 8);
    }

Store a value of FAT12 entry:
    ReadSector(SecBuff, ThisFATSecNum);
    if (N & 1) {    /* Odd entry */
        SecBuff[ThisFATEntOffset] = (SecBuff[ThisFATEntOffset] & 0x0F)
                                  | (NewEntryVal << 4);
        SecBuff[ThisFATEntOffset + 1] = NewEntryVal >> 4;
    } else {        /* Even entry */
        SecBuff[ThisFATEntOffset] = NewEntryVal;
        SecBuff[ThisFATEntOffset + 1] = (SecBuff[ThisFATEntOffset + 1] & 0xF0)
                                      | ((NewEntryVal >> 8) & 0x0F);
    }
    WriteSector(SecBuff, ThisFATSecNum);
```

Unfortunately, this code does not work properly because the bit field spans over a sector boundary when `ThisFATEntOffset` points the last byte of the sector and it needs to be handled properly.

Following image shows the FAT usage of each FAT type



Association of File and Cluster

Files on the FAT volume are managed by directory, the array of 32-byte directory entry structures. Details of the directory entry is described below. The directory entry has the file name, file size, timestamp and the first cluster number of the file data. The cluster number is the entry point to follow the cluster chain of the file data. If the file size is zero, a zero is set to the first cluster number and no data cluster is allocated to the file.

As described above, cluster number 0 and 1 are reserved and valid cluster number starts from 2. The cluster number 2 corresponds to the first cluster of the data area. Therefore, valid cluster number is from 2 to N + 1 and count of FAT entries is N + 2 in the volume with N clusters. The location of a data cluster N is calculated as follows:

```
FirstSectorofCluster = DataStartSector + (N - 2) * BPB_SecPerClus;
```

If the file size is larger than the sector size, file data is spanning over multiple sectors in the cluster. If the file size is larger than the cluster size, file data is spanning over multiple clusters in the cluster chain. The value of the FAT entry indicates following cluster number if exist, so that the any byte offset in the file can be reached by following the cluster chain. The cluster chain cannot be followed backward. The FAT entry with last link of cluster chain has a special value (end of chain, EOC, mark), which is never matches any valid cluster number. The EOC mark for each FAT type is as follows:

- FAT12: `0xFF8 - 0xFFF` (typically `0xFFF`)
- FAT16: `0xFFF8 - 0xFFFF` (typically `0xFFFF`)
- FAT32: `0x0FFFFFF8 - 0x0FFFFFFF` (typically `0x0FFFFFFF`)

There is also a special value, bad cluster mark. The bad cluster mark indicates that there is a defective sector in the cluster and it cannot be used. The bad cluster found on the format, surface inspection or disk repair is recorded in the FAT as bad cluster mark. The value of bad cluster mark is `0xFF7` for FAT12, `0xFFF7` for FAT16 and `0x0FFFFFF7` for FAT32.

The value of bad cluster mark never be equal to any valid cluster number on the FAT12/16 volume. However, it can be equal to any allocatable cluster number because the maximum count of clusters is not defined in FAT32. Such FAT32 volumes can make disk utilities confuse, so that you should avoid creating such FAT32 volume. Therefore, the upper limit of cluster count of FAT32 volume is practicaly 268435445 (about 256 M clusters).

Some system has a limitation on the maximum cluster count due to implementation reasons. For example, Windows9X/Me supports the FAT size 16 MB maximum and it limits number of clusters about 4 M clusters maximum.

The initial value of each allocatable FAT entry, FAT[2] and followings, is zero, which indicats the cluster is not in use and free for a new allocation. If the value is not zero, it means the cluster is in use or bad. Free cluster count is not recoreded anywhere in the FAT12/16 volume and full FAT scan is needed to get this information. FAT32 supports FSInfo to store the free cluster count to avoid full FAT scan because of its very large FAT structure.

The top two FAT entry, FAT[0] and FAT[1], are reserved and not associated with any cluster. These FAT entries are initialized on creating the volume as follows:

- FAT12: `FAT[0] = 0xF??; FAT[1] = 0xFFF;`
- FAT16: `FAT[0] = 0xFF??; FAT[1] = 0xFFFF;`
- FAT32: `FAT[0] = 0xFFFFFF??; FAT[1] = 0xFFFFFFFF;`

?? in the value of FAT[0] is the same value of `BPB_Media` but the entry has not any function. Some bits in the FAT[1] records error history.

- Volume dirty flag: (FAT16: bit15、FAT32: bit31): Cleared on boot, restored on clean shutdown. Already cleared on boot indicates a dirty shutdown and possibility of logical error in the volume.
- Hard error flag: (FAT16: bit14、FAT32: bit30): Cleared on unrecoverable read/write error to indicates that a surface inspection is needed.

These flags indicates possibility of an error on the volume. Some OSs supporing this feature check these flags on boot and launch disk inspection tool automatically. Windows 9X family uses these flags. Windows NT family does not use these flags but alternatives in the BPB.

There are two more important things about the FAT area. One is that the last sector of a FAT may not be fully used. In most case, FAT ends at the middle of the sector. FAT driver should not have any assumption about unused area. It should be filled with zeros on formatting the volume and should not be changed afterwards. The other is that the `BPB_FATSz16/32` can indicates a value lager than the volume requires. In other wards, unused sectors can follow each FATs. It may a result of data area alignment or something. Also these sector are filled with zeros on formatting.

Following table shows the range of FAT values and meaning for each FAT type.

FAT12	FAT16	FAT32	Meaninig
`0x000`	`0x0000`	`0x00000000`	Free
`0x001`	`0x0001`	`0x00000001`	Reserved
`0x002 - 0xFF6`	`0x0002 - 0xFFF6`	`0x00000002 - 0x0FFFFFF6`	In use (value is link to next)
`0xFF7`	`0xFFF7`	`0x0FFFFFF7`	Bad cluster
`0xFF8 - 0xFFF`	`0xFFF8 - 0xFFFF`	`0x0FFFFFF8 - 0x0FFFFFFF`	In use (end of chain)

FSInfo Sector Structure and Backup Boot Sector

The size of FAT is up to 6 KB on FAT12 volume and up to 128 KB on FAT16 volume, but it typically reach several MB on FAT32 volume. For this reason, FAT32 volume supports FSInfo structure in order to avoid to read over an entire FAT to look for free clusters or get count of free clusters. This structure is put in the FSInfo sector indicated by `BPB_FSInfo`.

Field name	Offset	Size	Description
FSI_LeadSig	0	4	0x41615252. This is a lead signature used to validate that this is in fact an FSInfo sector.
FSI_Reserved1	4	480	Reserved. This field should be always initialized to zero.
FSI_StrucSig	484	4	0x61417272. Another signature that is more localized in the sector to the location of the fields that are used.
FSI_Free_Count	488	4	This field indicates the last known free cluster count on the volume. If the value is 0xFFFFFFFF, it is actually unknown. This is not necessarily correct, so that the FAT driver needs to make sure it is valid for the volume.
FSI_Nxt_Free	492	4	This field gives a hint for the FAT driver, the cluster number at which the driver should start looking for free clusters. Because a FAT32 FAT is large, it can be rather time consuming if there are a lot of allocated clusters at the start of the FAT and the driver starts looking for a free cluster starting at the first cluster. Typically this value is set to the last cluster number that the driver allocated. If the value is 0xFFFFFFFF, there is no hint and the driver should start looking at cluster 2. This may not be correct, so that the FAT driver needs to make sure it is valid for the volume.
FSI_Reserved2	496	12	Reserved. This field should be always initialized to zero.
FSI_TrailSig	508	4	0xAA550000. This trail signature is used to validate that this is in fact an FSInfo sector.
	512		When the sector size is larger than 512, rest bytes should be initialized to zero.

FAT32 FSInfo sector

Another feature of FAT32 volume is backup boot sector. This is a feature to provide redundancy for the only boot sector existing on the FAT volume. This can increase the possibility of volume recovery if the boot sector is corrupted for any reason. The location of backup boor sector is indicated by `BPB_BkBootSec`. 6 is strongly recommended for this field because the boot loader and FAT driver are hard coded to try reading the boot sector at sector 6 when it failed to load the main boot sector. The FAT32 boot sector is actually three 512-byte sectors long. There is a copy of all three of these sectors starting at the sector indicated by `BPB_BkBootSec`. A copy of the FSInfo sector is also there, even though the `BPB_FSInfo` field in this backup boot sector is set to the same value as the value in sector 0. All three sectors have boot signature, 0xAA55, at the offset 510.

FAT Directory

This section describes about only short file name (SFN), the basic feature of FAT volume. The directory is really a file with a special attribute. It contains table of directory entries that contain meta data of the files on the volume. Size of a directory entry is 32 byte long and it corresponds with a file or diretory on the volume. Maximum size of a directory is 2 MB (65536 entries).

The root directory is only a special directory needs to be always exist and it becomes top node of the hierarchy in the volume. On the FAT12/16 volume, the root directory is not a file but put on the root directory area separated form the data area. The count of root directory entries is determined on the formatting and indicated in `BPB_RootEntCnt`. On the FAT32 volume, there is no difference between the sub-directories except for it does not have any entry to indicate it and the start cluster number is indicated by `BPB_RootClus`.

Another difference from the sub-directory is that it does not contain dot entries (`″.″`, `″..″`) what always exist in the sub-directory and it can contain a volume label (an entry with `ATTR_VOLUME_ID` attribute). Following table shows directory entry structure

Field name	Offset	Size	Description
DIR_Name	0	11	Short file name (SFN) of the object.

DIR_Attr	11	1	File attribute in combination of following flags. Upper 2 bits are reserved and must be zero.
			`0x01: ATTR_READ_ONLY (Read-only)`
			`0x02: ATTR_HIDDEN (Hidden)`
			`0x04: ATTR_SYSTEM (System)`
			`0x08: ATTR_VOLUME_ID (Volume label)`
			`0x10: ATTR_DIRECTORY (Directory)`
			`0x20: ATTR_ARCHIVE (Archive)`
			`0x0F: ATTR_LONG_FILE_NAME (LFN entry)`

DIR_NTRes	12	1	Optional flags that indicates case information of the SFN. `0x08`: Every alphabet in the body is low-case. `0x10`: Every alphabet in the extensiton is low-case.

DIR_CrtTimeTenth	13	1	Optional sub-second information corresponds to `DIR_CrtTime`. The time resolution of `DIR_CrtTime` is 2 seconds, so that this field gives a count of sub-second and its valid value range is from 0 to 199 in unit of 10 miliseconds. If not supported, set zero and do not change afterwards.
DIR_CrtTime	14	2	Optional file creation time. If not supported, set zero and do not change afterwards.
DIR_CrtDate	16	2	Optional file creation date. If not supported, set zero and do not change afterwards.
DIR_LstAccDate	18	2	Optional last accesse date. There is no time information about last accesse time, so that the resolution of last accesse time is 1 day. If not supported, set zero and do not change afterwards.
DIR_FstClusHI	20	2	Upeer part of cluster number. Always zero on the FAT12/16 volume. See `DIR_FstClusLO`.
DIR_WrtTime	22	2	Last time when any change is made to the file (typically on closeing).
DIR_WrtDate	24	2	Last data when any change is made to the file (typically on closeing).
DIR_FstClusLO	26	2	Lower part of cluster number. When the file size is zero, no cluster is assigned and this item must be zero. Always an valid value if it is a directory.
DIR_FileSize	28	4	Size of the file in unit of byte. Not used when it is a directroy and the value must be always zero.

Directory entry structure

The first byte of `DIR_Name` field, `DIR_Name[0]`, is an impotant data to indicates state of the directory entry. When the value is 0xE5, it indicates that the entry is not used (free for new allocation). When the value is 0x00, it indicates that the entry is not used (same as 0xE5) and in addition, there is no allocated entry after this one (all of the `DIR_Name[0]` in all of the entries after this one are also set to 0). Any other value in the `DIR_Name[0]` indicates the entry in in use. There is an exception about the file name with heading character 0xE5. In this case, 0x05 is set instead.

`DIR_Name` field is a 11-byte string and divided in two parts, body and extension. The file name is stored in 8-byte body + 3-byte extension. The dot in the file name to separate body and exitension is removed on the directory entry. If any part of the name does not fit to the part, rest bytes in the part is filled with spaces (`0x20`). The code page used for the file name depends on the system.

```
FileName           DIR_Name[]       Description
"FILENAME.TXT"    "FILENAMETXT"     Dot is removed.
"DOG.JPG"         "DOG     JPG"     Each part is padded with spaces.
"file.txt"        "FILE    TXT"     Low-case characters are up-converted.
"蜃気楼.JPG"       "・気楼   JPG"     The first byte of "蜃", 0xE5, is replaced with 0x05
"NOEXT"           "NOEXT      "     No extension
".cnf"                              (illegal) Any name without body is not allowed
"new file.txt"                      (illegal) Space is not allowed.
"file[1].2+2"                       (illegal) [ ] + are not allowed.
"longext.jpeg"                      (illegal) Out of 8.3 format.
"two.dots.txt"                      (illegal) Out of 8.3 format.
```

Allowable charactes for the file name are
0〜9 A〜Z ! # $ % & ’ ( ) - @ ^ _ ` { } ~
in ASCII characters and extended characters (\x80 - \xFF). Low-case ASCII characters (a-z) in the input file name are replaced with up-case characters prior to matching and recording. As for the extended characters, there are many differnce on the replacement between each system, such as Ää→ÄÄ (CP852) and Ää→AA (CP850). Therefore, using extended characters can cause compatibility problem in the different systems (e.g. file open failur) even if with the same name binary. As for the DBCS extended characters in Japanese environment, refer to the Compatibility described below.

Every file names in a directory is unique each other. Any other enrty with the same name never exists. `DIR_Attr` field indicates the attribute of the entry.

Flag	Meaning
ATTR_READ_ONLY	Read-only File. Any changes to the file or delete should be rejected.
ATTR_HIDDEN	Normal directoly listing should not show this file. (system dependent)
ATTR_SYSTEM	Indicates this is an system file. (system dependent)
ATTR_DIRECTORY	Indicates this is a container of a directory.
ATTR_ARCHIVE	This is for backup utilities. Set by FAT driver on new creation, modification or renaming to the file is made. The backup utilities able to easily find the file to be backed up and it clears the attribute after the back up process.
ATTR_VOLUME_ID	An entry with this attribute has the volume label of the volume. Only one entry can be exist in the root directory. `DIR_FstClusHI`, `DIR_FstClusLO` and `DIR_FileSize` field must be always zero. Some system may set `ATTR_ARCHIVE`, but it has no meaning.
ATTR_LONG_NAME	This combination of attributes indicates the entry is a part of long file name. Details are described below.

File attribute

Directory Operations

Creating File

To create a file, FAT driver finds a free entry in the directory to create in. If no free entry is found in the directry, stretch the directry a cluster to allocate a new free entry, but size of a directory cannot exceed 2 MB (65536 entries). Size of static directory (root directory on the FAT12/16 volume) is fixed and cannot be changed. The new entry has its name in the `DIR_Name`, `ATTR_ARCHIVE` flag in the `DIR_Attr` and `DIR_FstClusHI`, `DIR_FstClusLO`, `DIR_FileSize` has 0 for the initial value. When any data is written to the file and file size changed form 0, a new cluster chain is created and the first cluster numner is stored to `DIR_FstClusHI`, `DIR_FstClusLO`. The cluster chain is streached as file size increse.

Creating Sub-directory

To create a sub-directory, FAT driver creates a directory entry as creating a file. The entry needs to have `ATTR_DIRECTORY` attribute. The sub-directry has no size information and `DIR_FileSize` field must be always zero. A cluster is initially allocated to the sub-directory and the cluster number is set to the `DIR_FstClusHI`, `DIR_FstClusLO` field. Each entries in the cluster is initialized to zero. When the directory table gets full, the cluster chain is stretched and cluster is initialized to zero. The maximum length of a direcotry is 2 MB (64 K entries).

Sub-directory has two special entries (dot entry) at top of the directory, `DIR[0]` as `″.          ″` and `DIR[1]` as `″..         ″`. These entries have `ATTR_DIRECTORY` attribute, however, they do not have any cluster but point another directory's cluster instead. `″.″` entry points this directoy and `″..″` entry points the parent directory. If the parent directory is the root directory, set zero to the `DIR_FstClusHI`, `DIR_FstClusLO` field even if at FAT32 volume.

Deleting File

To remove a file, set 0xE5 to the `DIR_Name[0]` to free the entry. If the file has a cluster chain, also the chain needs to be removed from the FAT.

Deleting Sub-directory

It is same as deleting a file. All nodes below the directory needs to be scanned and all files and directories in the directory need to be deleted prior to delete the directory otherwise those objects' clusters get lost clusters.

Volume Label

FAT volume can have a its own name called volume label, which is recorded as a directory entry with `ATTR_VOLUME_ID` attribute in the root directory. The volume label is not a file but only a name of the volume. Its name space is independent of the files and the name can be duplicated with any other file in the directory. Allowable characters for the volume label is similar to the SFN entry but it can contain spaces anywhere the name and cannot contain dot.

The LFN extension is not applied to the volume label. When any change to the volume label is made, it should be reflected to `BS_VolLab`, but Windows does not do it. Windows has a problem on the behavior at the volume label beginning with a 0xE5. It does not replace it with 0x05 and the change will have no effect, so that such volume label shoud not be used.

Timestamp

There are some fields related to the time and data in the directory entry. Most FAT driver supports only `DIR_WrtTime`, `DIR_WrtDate` field which is mandatory to be supported and optional fields are not supprted. The non supporting field should be initialized to zero on creation of entry and do not chane afterwards. The format of the time and date is described as follows:

Field name	Bit fields
DIR_WrtDate DIR_CrtDate DIR_LstAccDate	Bit 15-9: Count of years from 1980 in range of from 0 to 127 (1980-2107). Bit 8-5: Month of year in range of from 1 to 12. Bit 4-0: Day of month in range of from 1 to 31.
DIR_WrtTime DIR_CrtTime	Bit 15-11: Hours in range of from 0 to 23. Bit 10-5: Minutes in range from 0 to 59. Bit 4-0: 2 second count in range of form 0 to 29 (0-58 seconds).

Long File Name

When add the long file name (*LFN*) as a new feature to the FAT filesystem, a backward compatibility with the existing systems is required. Following are concrete examples required.

- The existence of LFN needs to be invisible on the existing systems, especially any file API on the MS-DOS and Windows.
- LFN needs to be located physically near the direcroty entry of the corresponding file to prevent bad effect to the performance.
- If disk utility found the LFN information recorded somewhere on the FAT volume, the filesystem needs to keep sanity and be not affected.

To achieve these requirement, LFN information is recorded as directory entry with a special attribute. As described above, the attribute for the LFN entry (`ATTR_LONG_NAME`) is defined in combination of existing attribute bits (`ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID`), and mask value (`ATTR_LONG_NAME_MASK` = `ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID | ATTR_DIRECTORY | ATTR_ARCHIVE`) is also defined. When `DIR_Attr` masked with (`ATTR_LONG_NAME_MASK` matched with `ATTR_LONG_NAME`, the entry is an LFN entry and its field is defined as below.

Field name	Offset	Size	Description

LDIR_Ord	0	1	Sequence number (1-20) to identify where this entry is in the sequence of LFN entries to compose an LFN. One indicates the top part of the LFN and any value with `LAST_LONG_ENTRY` flag (0x40) indicates the last part of the LFN.
LDIR_Name1	1	10	Part of LFN from 1st character to 5th character.
LDIR_Attr	11	1	LFN attribute. Always `ATTR_LONG_NAME` and it indicates this is an LFN entry.
LDIR_Type	12	1	Must be zero.
LDIR_Chksum	13	1	Checksum of the SFN entry associated with this entry.
LDIR_Name2	14	12	Part of LFN from 6th character to 11th character.
LDIR_FstClusLO	26	2	Must be zero to avoid any wrong repair by old disk utility.
LDIR_Name3	28	4	Part of LFN from 12th character to 13th character.

Directory entry structuer for LFN

LFN entry is always associated with the corresponding SFN entry in order to add an LFN to the file. LFN entries never exist independent of SFN. Therfore each file has only SFN or both of SFN and LFN. LFN entry has only name information in it and nothing else about the file. If an LFN entry without association with the SFN entry is exist, such stray LFN entry is invalid and considered garbage. This is for backward compatibility with old system. If an LFN is given to the file, the LFN is the primary name of the file and the SFN is an alternative. Old systems without support for LFN do not recognize LFN entry, but it can access the files with SFN. LFN system can access the file with LFN or SFN. Following table shows how the set of LFN and SFN is recorded on the directory.

Location	First byte	Name field	Attribute	Content
DIR[N-3]	0x43	ary.pdf	--VSHR	LFN 3rd part (lfn[26..38])
DIR[N-2]	0x02	d System Summ	--VSHR	LFN 2nd part (lfn[13..25])
DIR[N-1]	0x01	MultiMediaCar	--VSHR	LFN 1st part (lfn[0..12])
DIR[N]	'M'	MULTIM~1PDF	A-----	Associated SFN entry

Association of LFN "MultiMediaCard System Summary.pdf" with a file

If the LFN is longer than 13 characters, it is divided into some LFN entries. The maximum name length for LFN is 255, so that an LFN occupies upto 20 LFN entries. LFN is put on the directory at just before the associated SFN entry. For the example shown above, an LFN with 33 character in length consist of 3 LFN entries that have `0x43, 0x02, 0x01` in `LDIR_Ord`. The character code used for the LFN is Unicode in UTF-16 encoding while the character code for SFN is ANSI/OEM code in local code page depends on the system. If the last part does not fit 13 characters, it is terminated with a null character (`U+0000`) and rest of name field must be filled with `U+FFFF`. `LDIR_Ord` must start at 1 and be recorded in descending order. The block of LFN+SFN are recorded on a contiguous entries. If any of these condition about LFN entry is not met, the LFN is invalid any longer.

Furthermore, a check sum is used to make sure of relevance between LFN and SFN. Each LFN entry has a check sum of associated SFN in `LDIR_Chksum`. The check sum is generated in the algorithm shown below.

```
uint8_t create_sum (const DIR* entry)
{
    int i;
    uint8_t sum;

    for (i = sum = 0; i < 11; i++) { /* Calculate sum of DIR_Name[] field */
        sum = (sum >> 1) + (sum << 7) + entry->DIR_Name[i];
    }
    return sum;
}
```

If any check sum in the LFN entries does not match, the LFN is invalid. This is to prevent wrong association due to any changes (delete, create or rename) to the direcotry by the non-LFN system. However, stray LFN entries continue to occupy the directory and the disk usage gets worse. This will be a problem at the fixed length directory (root directory on FAT12/16 volume). These garbage entries are removed by disk utirities.

Namespace

Short File Name

SFN, often called the 8.3 format name, is a traditional style file name originally used on the MS-DOS in format of body (1-8 character) plus optional extension (1-3 characters). These two parts are separated with a dot (.). The allowable charactes for the SFN are ASCII alphanumerics, some ASCII marks (`$%'-_@~`` `!(){}^#&`) and extended characters (\x80 - \xFF).

SFN is stored in the SFN entry in OEM code set (used on MS-DOS) depends on the system locale. Low-case character in the file name is converted to up-case and then stored and matched, so that the case information of SFN is lost.

Long File Name

Length of LFN can be upto 255 characters. The allowable characters for the LFN are white space and some ASCII marks (`+,;=[]`) in addition to the SFN characters. Dots can be embedded anywhere in the file name except the trailing dots and spaces are treated as end of the name and truncated off on the file API. Preceding spaces and dots are valid, but some user interface, such as Windows common dialog, rejects such file name.

LFN is stored in the LFN entry without up-case conversion. Character code used by LFN is in Unicode.

Because different character codes are used by SFN (OEM code set) and LFN (Unicode), generic implementation needs to convert those codes. This is not the matter on the OEM code is single byte code, however, when the OEM code page is in double byte character set (DBCS), a huge (several hundreds KB) conversion table is needed, so that it is difficult to implement LFN in the small embedded systems with a limited memory.

Name Matching

Every file names is unique in the directory and never matchs with any other name no matter it is between LFN and SFN. LFN can contain low-case characters and it is matched in case insensitive, so that these three names, `"LongFileName.Txt"`, `"longfilename.txt"`, `"LONGFILENAME.TXT"`, are treated as the same name. Therfore name matching on directory search is always done in case insensitive.

When find an input file name in 8.3 format, both LFN and SFN of the files in the directory are compared. When the file name is out of 8.3 format, only LFN is compared.

When list the file names in a directory, only LFN is output unless SFN is specified. If the file doesn't have LFN or any character in the LFN could not be converted into OEN code, this is the case when OEM code is used on the API, SFN is output instead.

Generating SFN

In every FAT filesystem with LFN extension, file names given to the file API need to be treated as LFN. SFN needs to be generated from the input file name and the API should not allow to specify LFN and SFN individually. Theoretically, any arbitrary SFN can be used unless it collides with another name in the directory, but a certain rule for name generation is needed in order to achive consistency of the file name and to avoid confusions about file name between users or applications. The SFN is generated as *body(+numeric-tail)(+extension)* in following procedure.

1. Convert low-case characters includs extended characters into up-case.
2. If any space is exist, remove it and set lossy conversion flag.
3. If heading dots are exist, remove them and set lossy conversion flag.
4. If two or more dots are exist, remove them except last one and set lossy conversion flag.
5. If any character not allowed for SFN is exist, replace it with an underscore (_) and set lossy conversion flag.
6. When the input file name is in Unicode, convert it to ANSI/OEM code. If any character could not be converted to ANSI/OEM code, replace it with an underscore and set lossy conversion flag.
7. If length of the body is longer than 8 bytes, truncate it to 8 bytes and set lossy conversion flag.

8. If length of the extention, if exist, is longer than 3 bytes, truncate it to 3 bytes and set lossy conversion flag.

Here an SFN in body(+extension) is created. If lossy conversion flag has been set, it means the input file name is out of 8.3 format. The lossy converted SFN needs to be modified to suggest the SFN differs from LFN to user. To generate a unique name, a numeric-tail (`~N`, a tilde followed by 1-6 digits of numeral) is added to the body. The body will need to be truncated to add the numeric-tail. The unique name not collide with any other name in the directory is typically searched in ascending order from N = 1, but it depends on the implementation. For example, Windows NT family OSs totally re-create the body with some hash value at 5th trial. Therefore what SFN is generated cannot be determinable and it depends on the existing name in the directory and locale settings.

LFN(input)	SFN
File.txt	FILE.TXT
foo.tar.gz	FOOTAR~1.GZ
.conf	CONF~1
a+b=c	A_B_C~1
Asakura Otome.jpeg	ASAKUR~1.JPE
Asakura Yume.jpeg	ASAKUR~2.JPE

Example of generated SFN

Suppressing LFN Entry

When the file name is in 8.3 format, generated SFN is the exactly same as LFN (except for up-case conversion). In this case, the LFN entry can be suppressed and not created if the follwing conditions each is true.

- Any low-case character is not contained.
- Any extended character is not contained.

For example, `"HELLO.TXT"` is the case. On the Window NT family OSs, also following condition is tested.

- Either or both of body and extension contains low-case characters with not in mixed-case.
- Any extended character is not contained.

If it is the case, low-capital information is recorded in the `DIR_NTRes` of SFN entry and the LFN entry is suppressed. The file names satisfy this condition, such as `"lower12.dll"`, `"system32"`, `"FDCMD.exe"`, are very common for most existing files. This enables to save the number of direcrory entries. On the some LFN aware systems, such as Windows 9X family, DIR_NTRes field is not supported and these entries are recognized as simple SFN entries and the files will appear in the directory listing as "LOWER12.DLL", "SYSTEM32", "FDCMD.EXE". It is not a problem because name matching is done in case insensitive on the FAT file system. However, it can confuse some Unix based application running on the Windows 9X family OSs when it refers a volume written by Window NT family OS.

Compatibility

Non-LFN Aware System

The support of LFN is most important on the fixed disks, however it is supported on removable media as well. The removavle media is shared by various systems with or without support for LFN, so that the downward compatibility is important for the implementation of LFN and it provides support for LFN without breaking compatibility with the existing FAT format. An FAT volume with any LFN exists can be read by down level systems without any compatibility problems. An exsisting FAT volume does not need to go through any comversion process prior to start using LFN and any current files remain not modified. The LFN entry is added on a long name is created. When rename a current file with LFN, the SFN entry will be moved within the directory to create an entry block. The LFN entries are hidden at the generic file APIs on the down level systems and it does not cause any problem on geneic use. The user can read any file with 8.3 format name and put a new file without any side effect.

Down level systems do not aware of existence of LFN, so that some LFN entry can be broken by a directory operation. For instance, when a file with LFN is renamed, LDIR_Chksum in the LFN entries gets mismatch and as the result the LFN will be lost. Renaming or deleting volume label can break some LFN entry as well. This is unfortunate, but the file data itself is kept safe.

Up-case Conversion in Japanese MS-DOS

Up-case conversion for SFN is applied to extended characters as well. However, it is not applied to extended characters in the Japanese MS-DOS and they are recorded and searched without up-case conversion. It is changed at Windows NT family OS and it creates SFN entry with up-case conversion for all charactres. As the result, it brings a serious compatibility problem. For instance, create a file "Ｆａｔ.TXT" (name body is in full-width character) on the MS-DOS. And then mount the volume on the Windows XP, and the file is no longer accessible. This is because Windows NT family OSs find the SFN entry with up-case converted name "ＦＡＴ.TXT" and it never matches the SFN created by MS-DOS. The only workaround is to avoid using such file name. Also Windows 9X family OSs create the SFN without up-case converson, but it is not the problem because LFN entry is created and it can be opened on the Windows NT family OSs.

Differnet OEM Code Page

Some OEM code pages are in DBCS. The trailing byte of a double-byte character can match some illigal character for file name, especially most FAT driver uses \ (\x5C) for directory separator for the path name. If the file name contains such character bytes, the file will not able to be accessed on the system with SBCS. There is similar ploblem between SBCS systems due to up-case conversions of extended characters which differ between each SBCS systems. When exchange files between the systems with different code page, any file name with extended character should not be used.

There is no problem on the LFN because the file name is stored on the directory in Unicode. However, when character code on the file API is OEM code (this is incomplete support for LFN), some problems related to different code page can occuer.

Mac OS X

Mac OS X supports the file names with trailing dots or spaces and the OS can mount the FAT volume as well. However, such file name is not allowed on the FAT volume. In case of the file to be created on the FAT volume is that condition, Mac OS X replace the last character with an escape character (space:U+F028, dot:U+F029). Application program needs to consider this replacement when exchange the file between Mac and another systems via a removable media.

Physical Drive Partitioning

This is not in the scope of FAT filesystem. However, it is a generic knowledge about disk usage that everybody need to know when use storage devices in embedded systems.

To use the huge disk space of harddisk drive efficiently, it is often used in multiple partitions on a physical drive. For example, a 100 GB harddisk is divided into 3 partitions, 10, 30 and 60 GB, and create the volumes for system, data and cache. In generic Windwos PCs, two partitions, one for system and the other for recovery, will be exist on the harddisk.

There are two partitioning rules, MBR format (aka FDISK format) and SFD format (Super-floppy Disk). The MBR format is usually used for harddisk and memory card. It can divide a physical drive into one or more partitions with a partition table on the MBR (Maser Boot Record, the LBA 0 of the physical drive). The SFD format is non-partitioned disk format. The FAT volume starts at the LBA 0 of the physical drive without any disk partitioning. It is usually used for floppy disk, optical disk and most type of super-floppy media.

Some combinations of systems and media type support only either one of the two formats and the other is not supported. Windows OS does not support second partition on the removable drive and SFD format on the harddisk (the former has been supported at Windows 10 1703).

Field name	Offset	Size	Description
MBR_bootcode	0	446	Boot program. Depends on the system. Filled with zeros when not used.
MBR_Partation1	446	16	Partition table enrty 1. Indicates partition type and status.
MBR_Partation2	462	16	Partition table enrty 2.
MBR_Partation3	478	16	Partition table enrty 3.

MBR_Partation4	494	16	Partition table enrty 4.
MBR_Sig	510	2	0xAA55. Indicates this is a valid MBR.

MBR and Partition table

Following table shows the field of partition table entry and upto four entry can be recorded in the MBR. This means a storage device can be divided four partitions. There is a type of partition which can contain some partitions in it, but for details of it, prease refer to others.

Field name	Offset	Size	Description
PT_BootID	0	1	Boot indicator. Not bootable (0x00) or Bootable (0x80). Bootable is to make the system boots from this partition, but it is system dependent. Only one partition can be set to bootable.
PT_StartHd	1	1	Head number of partition start sector in CHS form (0 – 254).
PT_StartCySc	2	2	Cylinder number (bit9-0: 0-1023) and sector number in the cylinder (bit15-10: 1-63) of partition start sector in CHS form.
PT_System	4	1	Type of this partition. Typical values are: 0x00: Blank entry. Any other field must be zero. 0x01: FAT12 (CHS/LBA, < 65536 sectors) 0x04: FAT16 (CHS/LBA, < 65536 sectors) 0x05: Extended partition (CHS/LBA) 0x06: FAT12/16 (CHS/LBA, >= 65536 sectors) 0x07: HPFS/NTFS/exFAT (CHS/LBA) 0x0B: FAT32 (CHS/LBA) 0x0C: FAT32 (LBA) 0x0E: FAT12/16 (LBA) 0x0F: Extended partition (LBA)
PT_EndHd	1	1	Head number of partition end sector in CHS form (0 – 254).
PT_EndCySc	2	2	Cylinder number (bit9-0: 0-1023) and sector number in the cylinder (bit15-10: 1-63) of partition end sector in CHS form.
PT_LbaOfs	8	4	Partition start sector in 32-bit LBA (1 – 0xFFFFFFFF).
PT_LbaSize	12	4	Partition size in unit of sector (1 – 0xFFFFFFFF).

Partition table entry

Ecah partition occupy a part of drive without overlapping each other and the first sector of the partition is the VBR. In most case, only first entry is used and rest of entries are left blanked.

There are two formart to express allocation of the partitions, CHS and LBA. Both parameters are stored on the entry. CHS field is used for the drives that have geometry, but this is actually depends on the system. LBA field is used for the drives controled in LBA. If the partition overlaps an area where cannot be expressed in CHA (8 GB and above), CHS field is no longer valid and only LBA field can be used.