Microsoft

Microsoft FAT Specification

Microsoft Corporation

August 30 2005

Table of Contents

Microsoft
..... 1
Microsoft FAT Specification 1
Table of Contents 2
Overview and Purpose 3
Section 1: Definitions and Notations 4
1.1 Definitions: 4
1.2 Notations 4
Section 2: Volume Structure 6
Section 3: Boot Sector and BPB 7
3.1 BPB structure common to FAT12, FAT16, and FAT32 implementations 7
3.2 Extended BPB structure for FAT12 and FAT16 volumes 10
3.3 Extended BPB structure for FAT32 volumes 11
3.4 Initialization of a FAT volume 12
3.5 Determination of FAT type when mounting the volume 14
3.6 Backup BPB Structure 15
Section 4: FAT 16
4.1: Determination of FAT entry for a cluster 17
4.2 Reserved FAT entries 19
4.3 Free space determination 20
4.4 Other points to note 20
Section 5: File System Information (FSInfo) Structure 21
Section 6: Directory Structure 23
6.1 File/Directory Name (field DIR_Name) 24
6.2 File/Directory Attributes 25
6.3 Date / Time 26
6.4 File/Directory Size 27
6.5 Directory creation 27
6.6 Root Directory 28
6.7 File allocation 28
Section 7: Long File Name Implementation (optional) 30
7.1 Ordinal Number Generation 31
7.2 Checksum Generation 32
7.3 Example illustrating persistence of a long name 32
7.4 Rules governing name creation and matching 33
Section 8: EA Support (optional) 35
Appendix 1: Example BPB for 512 byte sector-size 128 MB HDD media formatted FAT16 36
Appendix 2: Example BPB for 512 byte sector-size 128 MB HDD media formatted FAT32 37

Overview and Purpose

This document describes the on-media FAT file system format. This document is written to help guide development of FAT implementations that are compatible with those provided by Microsoft.

This document does not describe all algorithms contained in the Microsoft FAT file system driver implementation neither does it describe all algorithms contained in associated utilities (Microsoft *format* and *chkdsk* utilities). However, it is expected that the reader will refer to the Microsoft Corporation FAT file system reference source code for additional clarification as/when needed.

There are three variants of the FAT on-disk format, namely:

- FAT12
- FAT16
- FAT32

Data structures comprising all three variants are described here. Also provided are descriptions of specific algorithms that will be useful to the engineer implementing a FAT driver for reading and/or writing to media.

It is expected that the FAT Test Specification document will be used by the reader to validate interoperability and correctness of the resultant FAT implementation.

Section 1: Definitions and Notations

1.1 Definitions:

- **byte**

  A string of binary digits operated upon as a unit.

- **bad (defective) sector**

  A sector whose contents cannot be read or one that cannot be written.

- **file**

  A named stream of bytes representing a collection of information.

- **sector**

  A unit of data that can be accessed independently of other units on the media

- **cluster**

  A unit of allocation comprising a set of logically contiguous sectors. Each cluster within the volume is referred to by a cluster number "**N**". All allocation for a file must be an integral multiple of a cluster.

- **partition**

  An extent of sectors within a volume.

- **volume**

  A logically contiguous sector address space as specified in the relevant standard for recording.

1.2 Notations

- Numerical Notation

  Numbers in decimal notation are represented by decimal digits, namely 0 to 9.

  Numbers in hexadecimal notation are represented as a sequence of one or more hexadecimal digits namely 0 to 9 and A to F, prefixed by "0x".

  Zero represents a single bit with the value 0.

- Arithmetic Notation

  The notation $ip(x)$ shall mean the integer part of x.

  The notation $ceil(x)$ shall mean the minimum integer that is greater than x.

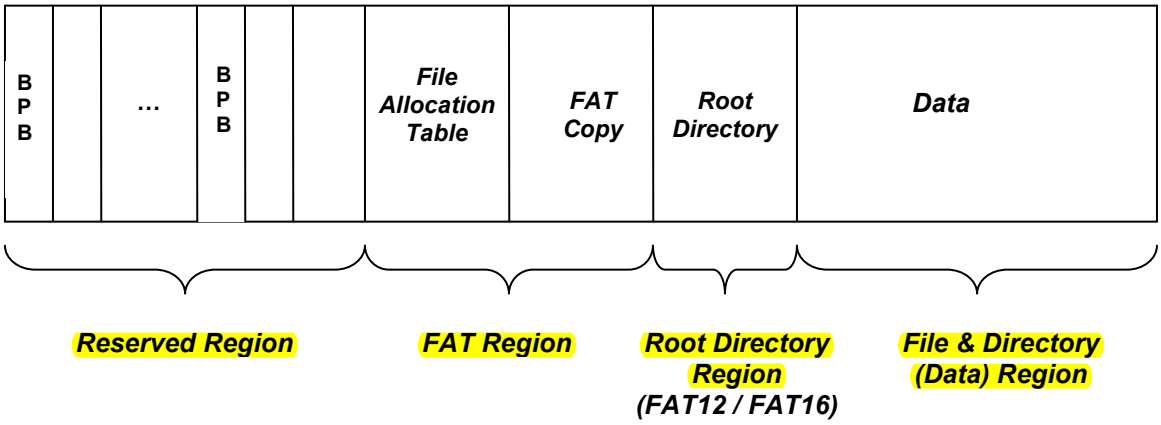
The notation *rem(x,y)* shall mean the remainder of the integer division of *x* by *y*.

Section 2: Volume Structure

A FAT file system volume is composed of four basic regions, which are laid out in this order on the volume:

0 – Reserved Region
1 – FAT Region
2 – Root Directory Region (doesn't exist on FAT32 volumes)
3 – File and Directory Data Region

The below figure illustrates the four regions in a volume formatted FAT:



All of the FAT file systems were originally developed for the IBM PC machine architecture. Hence, on disk data structures for the FAT format are all "little endian."

To illustrate, below is an instance of a 32-bit FAT entry stored on disk as a series of four 8-bit bytes—the first being byte[0] and the last being byte[3]. The 32 bits numbered 00 through 31 are stored as follows (00 being the least significant bit):

byte[3] 3 3 2 2 2 2 2 2
1 0 9 8 7 6 5 4

byte[2] 2 2 2 2 1 1 1 1
3 2 1 0 9 8 7 6

byte[1] 1 1 1 1 1 1 0 0
5 4 3 2 1 0 9 8

byte[0] 0 0 0 0 0 0 0 0
7 6 5 4 3 2 1 0

Section 3: Boot Sector and BPB

The BPB (BIOS Parameter Block) is located in the first sector of the volume in the *Reserved Region*. This sector is sometimes called the "*boot sector*" or the "*0th sector*". The important fact to note is that this sector is simply the first sector of the volume.

All FAT volumes must have a BPB in the boot sector. The BPB has evolved over the years as follows:

- The FAT implementation in MS-DOS 1.x did not include a BPB (introduced in MS-DOS 2.x)
- The BPB in MS-DOS 3.x was modified to allow > 64K sectors
- For FAT32, the BPB differs from that for FAT16/FAT12 beginning at offset 36

The BPB in the boot sector of a FAT volume must always have all of the BPB fields for either the FAT12/FAT16 or FAT32 BPB type. This ensures maximum compatibility of the FAT volume and will also ensure that all FAT file system drivers understand and support the volume correctly.

NOTE: In the following description, all the fields whose names start with BPB_ are part of the BPB. All the fields whose names start with BS_ are part of the boot sector and not really part of the BPB.

The fields comprising the BPB are described further in this section.

3.1 BPB structure common to FAT12, FAT16, and FAT32 implementations

The below table describes the fields in the BPB that are common to all FAT variants.

Descriptive name of field	Offset (byte)	Size (bytes)	Description
BS_jmpBoot	0	3	Jump instruction to boot code. This field has two allowed forms: **jmpBoot[0] = 0xEB, jmpBoot[1] = 0x??, jmpBoot[2] = 0x90** and **jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??** **0x??** indicates that any 8-bit value is allowed in that byte. What this forms is a three-byte Intel x86 unconditional branch (jump) instruction that jumps to the start of the operating system bootstrap code. This code typically occupies the rest of sector 0 of the volume following the BPB and possibly other sectors. Either of these forms is acceptable. **JmpBoot[0] = 0xEB** is the more frequently used format.

BS_OEMName	3	8	OEM Name Identifier. Can be set by a FAT implementation to any desired value. Typically this is some indication of what system formatted the volume.
BPB_BytsPerSec	11	2	Count of bytes per sector. This value may take on only the following values: 512, 1024, 2048 or 4096.
BPB_SecPerClus	13	1	Number of sectors per allocation unit. This value must be a power of 2 that is greater than 0. The legal values are 1, 2, 4, 8, 16, 32, 64, and 128.
BPB_RsvdSecCnt	14	2	Number of reserved sectors in the reserved region of the volume starting at the first sector of the volume. This field is used to align the start of the data area to integral multiples of the cluster size with respect to the start of the partition/media. This field must not be 0 and can be any non-zero value. This field should typically be used to align the start of the data area (cluster #2) to the desired alignment unit, typically cluster size.
BPB_NumFATs	16	1	The count of file allocation tables (FATs) on the volume. A value of 2 is recommended although a value of 1 is acceptable.
BPB_RootEntCnt	17	2	For FAT12 and FAT16 volumes, this field contains the count of 32-byte directory entries in the root directory. For FAT32 volumes, this field must be set to 0. For FAT12 and FAT16 volumes, this value should always specify a count that when multiplied by 32 results in an even multiple of `BPB_BytsPerSec`. For maximum compatibility, FAT16 volumes should use the value 512.
BPB_TotSec16	19	2	This field is the old 16-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. This field can be 0; if it is 0, then `BPB_TotSec32` must be non-zero. For FAT32 volumes, this field must be 0. For FAT12 and FAT16 volumes, this field contains the sector count, and `BPB_TotSec32` is 0 if the total sector count “fits” (is less than 0x10000).

BPB_Media	21	1	The legal values for this field are 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, and 0xFF. 0xF8 is the standard value for "fixed" (non-removable) media. For removable media, 0xF0 is frequently used.
BPB_FATSz16	22	2	This field is the FAT12/FAT16 16-bit count of sectors occupied by one FAT. On FAT32 volumes this field must be 0, and `BPB_FATSz32` contains the FAT size count.
BPB_SecPerTrk	24	2	Sectors per track for interrupt 0x13. This field is only relevant for media that have a geometry (volume is broken down into tracks by multiple heads and cylinders) and are visible on interrupt 0x13.
BPB_NumHeads	26	2	Number of heads for interrupt 0x13. This field is relevant as discussed earlier for `BPB_SecPerTrk`. This field contains the one based "count of heads". For example, on a 1.44 MB 3.5-inch floppy drive this value is 2.
BPB_HiddSec	28	4	Count of hidden sectors preceding the partition that contains this FAT volume. This field is generally only relevant for media visible on interrupt 0x13. This field must always be zero on media that are not partitioned. NOTE: Attempting to utilize this field to align the start of data area is incorrect.
BPB_TotSec32	32	4	This field is the new 32-bit total count of sectors on the volume. This count includes the count of all sectors in all four regions of the volume. This field can be 0; if it is 0, then `BPB_TotSec16` must be non-zero. For FAT12/FAT16 volumes, this field contains the sector count if `BPB_TotSec16` is 0 (count is greater than or equal to 0x10000). For FAT32 volumes, this field must be non-zero.

At this point, the BPB/boot sector for FAT12 and FAT16 differs from the BPB/boot sector for FAT32.

NOTE: It is recommended that the first sector of the partition (if any) be aligned to the required alignment unit.

3.2 Extended BPB structure for FAT12 and FAT16 volumes

If the volume has been formatted FAT12 or FAT16, the remainder of the BPB structure must be as described below:

Descriptive name of field	Offset (byte)	Size (bytes)	Description
BS_DrvNum	36	1	Interrupt 0x13 drive number. Set value to 0x80 or 0x00.
BS_Reserved1	37	1	Reserved. Set value to 0x0.
BS_BootSig	38	1	Extended boot signature. Set value to 0x29 if either of the following two fields are non-zero. This is a signature byte that indicates that the following three fields in the boot sector are present.
BS_VolID	39	4	Volume serial number. This field, together with `BS_VolLab`, supports volume tracking on removable media. These values allow FAT file system drivers to detect that the wrong disk is inserted in a removable drive. This ID should be generated by simply combining the current date and time into a 32-bit value.
BS_VolLab	43	11	Volume label. This field matches the 11-byte volume label recorded in the root directory. **NOTE:** FAT file system drivers must ensure that they update this field when the volume label file in the root directory has its name changed or created. The setting for this field when there is no volume label is the string "**NO NAME** ".
BS_FilSysType	54	8	One of the strings "**FAT12** ", "**FAT16** ", or "**FAT** ". **NOTE:** This string is informational only and does not determine the FAT type.
-	62	448	Set to 0x00
Signature_word	510	2	Set to 0x55 (at byte offset 510) and 0xAA (at byte offset 511)
-	512	All remaining bytes in sector	Set to 0x00 (only for media where BPB_BytsPerSec > 512

3.3 Extended BPB structure for FAT32 volumes

If the volume has been formatted FAT32, the remainder of the BPB structure must be as described below:

Descriptive name of field	Offset (byte)	Size (bytes)	Description
BPB_FATSz32	36	4	This field is the FAT32 32-bit count of sectors occupied by one FAT. Note that BPB_FATSz16 must be 0 for media formatted FAT32.
BPB_ExtFlags	40	2	Set as described below: Bits 0-3 -- Zero-based number of active FAT. Only valid if mirroring is disabled. Bits 4-6 -- Reserved. Bit 7 -- 0 means the FAT is mirrored at runtime into all FATs. -- 1 means only one FAT is active; it is the one referenced in bits 0-3. Bits 8-15 -- Reserved.
BPB_FSVer	42	2	High byte is major revision number. Low byte is minor revision number. This is the version number of the FAT32 volume. Must be set to 0x0.
BPB_RootClus	44	4	This is set to the cluster number of the first cluster of the root directory, This value should be 2 or the first usable (not bad) cluster available thereafter.
BPB_FSInfo	48	2	Sector number of `FSINFO` structure in the reserved area of the FAT32 volume. Usually 1. **NOTE:** There is a copy of the `FSINFO` structure in the sequence of backup boot sectors, but only the copy pointed to by this field is kept up to date (i.e., both the primary and backup boot record point to the same `FSINFO` sector).
BPB_BkBootSec	50	2	Set to 0 or 6. If non-zero, indicates the sector number in the reserved area of the volume of a copy of the boot record.
BPB_Reserved	52	12	Reserved. Must be set to 0x0.

BS_DrvNum	64	1	Interrupt 0x13 drive number. Set value to 0x80 or 0x00.
BS_Reserved1	65	1	Reserved. Set value to 0x0.
BS_BootSig	66	1	Extended boot signature. Set value to 0x29 if either of the following two fields are non-zero. This is a signature byte that indicates that the following three fields in the boot sector are present.
BS_VolID	67	4	Volume serial number. This field, together with `BS_VolLab`, supports volume tracking on removable media. These values allow FAT file system drivers to detect that the wrong disk is inserted in a removable drive. This ID should be generated by simply combining the current date and time into a 32-bit value.
BS_VolLab	71	11	Volume label. This field matches the 11-byte volume label recorded in the root directory. **NOTE:** FAT file system drivers must ensure that they update this field when the volume label file in the root directory has its name changed or created. The setting for this field when there is no volume label is the string "**NO NAME** ".
BS_FilSysType	82	8	Set to the string:"**FAT32** ". **NOTE:** This string is informational only and does not determine the FAT type.
-	90	420	Set to 0x00
Signature_word	510	2	Set to 0x55 (at byte offset 510) and 0xAA (at byte offset 511)
-	512	All remaining bytes in sector	Set to 0x00 (only for media where `BPB_BytsPerSec` > 512

3.4 Initialization of a FAT volume

Determination of FAT type (FAT12, FAT16, or FAT32) is dependent on the number of clusters (described in *Section 3.5* below).

This section describes the determination of values for specific fields in the BPB during volume initialization for a given target volume. FAT implementations must be capable of mounting media with all legal values for fields comprising the BPB.

- Floppy disks are formatted as FAT12. A simple (pre-specified) table determines the BPB structure contents for floppy media. Note that media size for FAT12 volumes must be <= 4MB.

- **For 512 byte sector sized media**: if volume size is < 512 MB, the volume is formatted FAT16 else it is formatted FAT32. It is possible to override the default FAT type selection.

  The below tables are used by the Microsoft Corporation FAT format utility - an entry in these tables is selected based on the size of the volume in 512 byte sectors (the value that will go in `BPB_TotSec16` or `BPB_TotSec32`), and the value that this table sets is the `BPB_SecPerClus` value:

```
struct DSKSZTOSECPERCLUS {
        DWORD   DiskSize;
        BYTE    SecPerClusVal;
    };

 /*
*This is the table for FAT16 drives. NOTE that this table includes
* entries for disk sizes larger than 512 MB even though typically
* only the entries for disks < 512 MB in size are used.
* The way this table is accessed is to look for the first entry
* in the table for which the disk size is less than or equal
* to the DiskSize field in that table entry.  For this table to
* work properly BPB_RsvdSecCnt must be 1, BPB_NumFATs
* must be 2, and BPB_RootEntCnt must be 512. Any of these values
* being different may require the first table entries DiskSize value
* to be changed otherwise the cluster count may be to low for FAT16.
   */
    DSKSZTOSECPERCLUS DskTableFAT16 [] = {
        {8400,   0}, /* disks up to  4.1 MB, the 0 value for SecPerClusVal trips an error */
        {32680,   2},  /* disks up to   16 MB,  1k cluster */
        {262144,   4},   /* disks up to 128 MB,  2k cluster */
        {524288,    8},   /* disks up to 256 MB,  4k cluster */
        {1048576,  16},     /* disks up to 512 MB,  8k cluster */
        /* The entries after this point are not used unless FAT16 is forced */
        {2097152,  32},     /* disks up to     1 GB, 16k cluster */
        {4194304,  64},     /* disks up to     2 GB, 32k cluster */
        {0xFFFFFFFF, 0} /*any disk greater than 2GB, 0 value for SecPerClusVal trips an error */
    };

/*
* This is the table for FAT32 drives. NOTE that this table includes
* entries for disk sizes smaller than 512 MB even though typically
* only the entries for disks >= 512 MB in size are used.
* The way this table is accessed is to look for the first entry
* in the table for which the disk size is less than or equal
* to the DiskSize field in that table entry. For this table to
* work properly BPB_RsvdSecCnt must be 32, and BPB_NumFATs
* must be 2. Any of these values being different may require the first
* table entries DiskSize value to be changed otherwise the cluster count
* may be to low for FAT32.
*/
    DSKSZTOSECPERCLUS DskTableFAT32 [] = {
        {66600,   0},  /* disks up to 32.5 MB, the 0 value for SecPerClusVal trips an error */
        {532480,   1},   /* disks up to 260 MB,  .5k cluster */
        {16777216,   8},     /* disks up to     8 GB,    4k cluster */
        {33554432, 16},      /* disks up to   16 GB,    8k cluster */
        {67108864, 32},      /* disks up to   32 GB,  16k cluster */
        {0xFFFFFFFF, 64}/* disks greater than 32GB, 32k cluster */
   };
```

  Given a disk size and a FAT type of FAT16 or FAT32, the above determines the `BPB_SecPerClus` value.

The below computes how many sectors the FAT takes up to enable setting the BPB_FATSz16 or BPB_FATSz32 fields (see *Section 4* for a description of the FAT). At this point, it is assumed that BPB_RootEntCnt, BPB_RsvdSecCnt, and BPB_NumFATs are appropriately set. It is also assumed that DskSize is the size of the volume that will be reflected in BPB_TotSec32 or BPB_TotSec16.

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec;
TmpVal1 = DskSize – (BPB_ResvdSecCnt + RootDirSectors);
TmpVal2 = (256 * BPB_SecPerClus) + BPB_NumFATs;
If(FATType == FAT32)
    TmpVal2 = TmpVal2 / 2;
FATSz = (TMPVal1 + (TmpVal2 – 1)) / TmpVal2;
If(FATType == FAT32) {
    BPB_FATSz16 = 0;
    BPB_FATSz32 = FATSz;
} else {
    BPB_FATSz16 = LOWORD(FATSz);
    /* there is no BPB_FATSz32 in a FAT16 BPB */
}
```

NOTE: The above math does not work perfectly. It will occasionally set a FATSz that is up to 2 sectors too large for FAT16, and occasionally up to 8 sectors too large for FAT32. It will never compute a FATSz value that is too small, however. Because it is OK to have a FATSz that is too large, at the expense of wasting a few sectors, the fact that this computation is surprisingly simple more than makes up for it being off in a safe way in some cases.

The unused sectors within a FAT must be set to 0x0.

3.5 Determination of FAT type when mounting the volume

The FAT type is determined solely by the count of clusters on the volume (*CountOfClusters*).

The following steps describe the computation of the count of clusters:

1. First, determine the count of sectors occupied by the root directory:

   ```
   RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec
   ```

   Note that on a FAT32 volume, the BPB_RootEntCnt value is always 0. Therefore, on a FAT32 volume, RootDirSectors is always 0.

2. Next, determine the count of sectors in the data region of the volume:

   ```
   If(BPB_FATSz16 != 0)
       FATSz = BPB_FATSz16;
   Else
       FATSz = BPB_FATSz32;

   If(BPB_TotSec16 != 0)
       TotSec = BPB_TotSec16;
   Else
       TotSec = BPB_TotSec32;

   DataSec = TotSec – (BPB_ResvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors);
   ```

3. Lastly, determine the count of clusters as:

   ```
   CountofClusters = DataSec / BPB_SecPerClus;
   ```

The computation rounds *down*.

To determine the FAT type, the following algorithm is used:

```
If(CountofClusters < 4085) {
/* Volume is FAT12 */
} else if(CountofClusters < 65525) {
    /* Volume is FAT16 */
} else {
    /* Volume is FAT32 */
}
```

The above algorithm determines the FAT type. Note the following:

- A FAT12 volume cannot contain *more than 4084 clusters*.
- A FAT16 volume cannot contain *less than 4085 clusters* or *more than 65,524 clusters*.

It is recommended that volumes not be created with a count of clusters that is exactly equal to the boundary values listed above (i.e. 4085 clusters or 65525 clusters).[1]

The Maximum Valid Cluster Number (**MAX**) for the volume is (`CountofClusters + 1`).

The "count of clusters including the two reserved clusters" is (`CountofClusters + 2`).

3.6 Backup BPB Structure

Loss (inability to retrieve contents) of sector #0 (containing the BPB) could lead to catastrophic data loss due to possible inability to mount the volume. On FAT32 formatted volumes, sector #6 must contain a copy of the BPB.

The `BPB_BkBootSec` field contains the value 6 for both copies of the BPB (the one in sector 0 and the backup copy located in sector 6).

Volume repair utilities are expected to retrieve contents of the BPB from sector #6 in a situation when sector #0 is unreadable.

NOTE: Starting at the `BPB_BkBootSec` sector is a *complete* boot record. The Microsoft FAT32 "boot sector" is actually three sectors long. There is a copy of all three of these sectors starting at the `BPB_BkBootSec` sector. A copy of the *`FSInfo`* sector (see *Section 5*) is also there, although the `BPB_FSInfo` field in this backup boot sector is set to the same value as is stored in the sector #0 BPB.

[1] To avoid hitting the boundary condition which may lead to computation errors in the FAT implementation, ensure count of clusters is at least 16 clusters greater than or less than these values.

Section 4: FAT

Each valid entry in the File Allocation Table (FAT) represents the state of a cluster from the set of clusters comprising the Root Directory Region (when applicable) and the File and Directory Data Region. The size of each (packed) entry in the FAT is as follows:

- For FAT12 volumes, each FAT entry is 12 bits in length
- For FAT16 volumes, each FAT entry is 16 bits in length
- For FAT32 volumes, each FAT entry is 32 bits in length

As stated earlier, a FAT may be larger than that required to describe all allocatable sectors comprising the Root/Data regions. Extra entries in the FAT (located at the tail/end of the FAT) must be set to a value of 0.

The FAT defines a singly linked list of the "extents" (clusters) of a file and thereby maps the data region of the volume by cluster number. The first data cluster in the volume is cluster #2.

NOTE: A FAT directory or file container is simply a regular file with a special attribute indicating the directory type. The contents of the directory are a series of 32 byte *directory entries* (described in *Section 6*)).

FAT entry values must be written as described in the below table:

FAT Entry Values			Comments
FAT12	**FAT16**	**FAT32**	
0x000	0x0000	0x0000000	Cluster is free.
0x002 to `MAX`	0x0002 to `MAX`	0x0000002 to `MAX`	Cluster is allocated. Value of the entry is the cluster number of the next cluster following this corresponding cluster. MAX is the Maximum Valid Cluster Number
(`MAX + 1`) to 0xFF6	(`MAX + 1`) to 0xFFF6	(`MAX + 1`) to 0xFFFFFF6	Reserved and must not be used.
0xFF7	0xFFF7	0xFFFFFF7	Indicates a bad (defective) cluster.
0xFF8 to 0xFFE	0xFFF8 to 0xFFFE	0xFFFFFF8 to 0xFFFFFFE	Reserved and should not be used. May be interpreted as an allocated cluster and the final cluster in the file (indicating *end-of-file* condition).
0xFFF	0xFFFF	0xFFFFFFF	Cluster is allocated and is the final cluster for the file (indicates *end-of-file*).

NOTE: The high 4 bits of a FAT32 FAT entry *are reserved*. All FAT implementations must preserve the current value of the high 4 bits when modifying any FAT32 FAT entry except during volume initialization (formatting) when the entire FAT table must be set to 0.

NOTE: No FAT32 volume should ever be configured containing cluster numbers available for allocation >= 0xFFFFFF7.

NOTE: The size of the FAT is limited to 6K sectors on volumes formatted FAT12. The size of the FAT is limited to 128K sectors on volumes formatted FAT16. There is no limit on the size of the FAT on volumes formatted FAT32.

4.1: Determination of FAT entry for a cluster

Given any valid cluster number **N**, this section describes the algorithm used to determine the entry in the FAT(s) for that cluster number.

FAT 16 and FAT 32

The FAT entry in the first FAT is determined as described below:

```
If(BPB_FATSz16 != 0)
     FATSz = BPB_FATSz16;
 Else
    FATSz = BPB_FATSz32;

If(FATType == FAT16)
    FATOffset = N * 2;
Else if (FATType == FAT32)
    FATOffset = N * 4;

ThisFATSecNum = BPB_ResvdSecCnt + (FATOffset / BPB_BytsPerSec);
ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);[2]
```

`ThisFATSecNum` is the sector number of the FAT sector that contains the entry for cluster `N` in the first FAT.

To compute the sector number in any other FAT:

```
SectorNumber = (FatNumber * FATSz) + ThisFATSecNum
```

where, `FatNumber` will be 2 for the second FAT, 3 for the third FAT, and so on.

The contents of the FAT entry can be extracted from the sector contents (once the sector has been read from media) as per below:

```
If(FATType == FAT16)
    FAT16ClusEntryVal = *((WORD *) &SecBuff[ThisFATEntOffset]);
Else
    FAT32ClusEntryVal = (*((DWORD *) &SecBuff[ThisFATEntOffset])) & 0x0FFFFFFF;
```

To modify an in-memory instance of a FAT entry contained within the sector, use the following logic:

```
If(FATType == FAT16)
    *((WORD *) &SecBuff[ThisFATEntOffset]) = FAT16ClusEntryVal;
Else {
     FAT32ClusEntryVal = FAT32ClusEntryVal & 0x0FFFFFFF;
    *((DWORD *) &SecBuff[ThisFATEntOffset]) =
        (*((DWORD *) &SecBuff[ThisFATEntOffset])) & 0xF0000000;
    *((DWORD *) &SecBuff[ThisFATEntOffset]) =
        (*((DWORD *) &SecBuff[ThisFATEntOffset])) | FAT32ClusEntryVal;
}
```

NOTE: The type WORD is defined as an unsigned 16-bit quantity. The type DWORD is defined as a 32-bit unsigned quantity.

NOTE: FAT16/FAT32 FAT entries cannot span a sector boundary.

[2] REM is the remainder operator.

FAT 12

Each FAT12 FAT entry has a length of 1.5 bytes (12-bits). The FAT entry in the first FAT is determined as described below:

```
    if (FATType == FAT12)
        FATOffset = N + (N / 2);
/* Multiply by 1.5 without using floating point, the divide by 2 rounds DOWN */

    ThisFATSecNum = BPB_ResvdSecCnt + (FATOffset / BPB_BytsPerSec);
    ThisFATEntOffset = REM(FATOffset / BPB_BytsPerSec);
```

The logic below checks for the entry spanning a sector boundary and comments on an approach to deal with this case:

```
If(ThisFATEntOffset == (BPB_BytsPerSec - 1)) {
    /* This cluster access spans a sector boundary in the FAT      */
    /* There are a number of strategies to handling this. The      */
    /* easiest is to always load FAT sectors into memory           */
    /* in pairs if the volume is FAT12 (if you want to load        */
    /* FAT sector N, you also load FAT sector N+1 immediately      */
    /* following it in memory unless sector N is the last FAT      */
    /* sector). It is assumed that this is the strategy used here  */
    /* which makes this if test for a sector boundary span         */
    /* unnecessary.                                                */
}
```

`ThisFATSecNum` is the sector number of the FAT sector that contains the entry for cluster `N` in the first FAT.

To compute the sector number in any other FAT:

```
SectorNumber = (FatNumber * FATSz) + ThisFATSecNum
```

where, `FatNumber` will be 2 for the second FAT, 3 for the third FAT, and so on.

The contents of the FAT entry can be extracted from the sector contents (once the sector has been read from media) as per below:

```
FAT12ClusEntryVal = *((WORD *) &SecBuff[ThisFATEntOffset]);
If(N & 0x0001)
    FAT12ClusEntryVal = FAT12ClusEntryVal >> 4;     /* Cluster number is ODD */
Else
    FAT12ClusEntryVal = FAT12ClusEntryVal & 0x0FFF; /* Cluster number is EVEN */
```

NOTE: For even numbered clusters, the logic above correctly obtains the low 12 bits describing the FAT entry. For odd numbered clusters, the logic correctly obtains the high 12 bits.

To modify an in-memory instance of a FAT entry contained within the sector, use the following logic:

```
If(N & 0x0001) {
    FAT12ClusEntryVal = FAT12ClusEntryVal << 4;      /* Cluster number is ODD */
    *((WORD *) &SecBuff[ThisFATEntOffset]) =
        (*((WORD *) &SecBuff[ThisFATEntOffset])) & 0x000F;
} Else {
    FAT12ClusEntryVal = FAT12ClusEntryVal & 0x0FFF;  /* Cluster number is EVEN */
    *((WORD *) &SecBuff[ThisFATEntOffset]) =
        (*((WORD *) &SecBuff[ThisFATEntOffset])) & 0xF000;
}
*((WORD *) &SecBuff[ThisFATEntOffset]) =
    (*((WORD *) &SecBuff[ThisFATEntOffset])) | FAT12ClusEntryVal;
```

NOTE: It is assumed that the >> operator shifts a bit value of 0 into the high 4 bits and that the << operator shifts a bit value of 0 into the low 4 bits.

4.2 Reserved FAT entries

The first two entries in the FAT are reserved.

The first reserved entry (FAT[0]), contains the `BPB_Media` byte value in its low 8 bits, and all other bits are set to 1. For example, if the `BPB_Media` value is 0xF8, then:

- for FAT12, FAT[0] = 0x0FF8
- for FAT16, FAT[0] = 0xFFF8
- for FAT32 FAT[0] = 0x0FFFFFF8

The second reserved entry (FAT[1]), is set by the format utility to the EOC value.

- On FAT12 volumes, this entry is not modified subsequent to the format and, therefore, always contains an EOC mark.
- For FAT16 and FAT32, the file system driver implementation on Microsoft Windows may use the high two bits of the FAT[1] entry for dirty volume flags (note that all other bits, are always left set to 1). Also note that the bit location is different for FAT16 and FAT32, because they are the high 2 bits of the entry.

  For FAT16:

  ```
  ClnShutBitMask  = 0x8000;
  HrdErrBitMask   = 0x4000;
  ```

  For FAT32:

  ```
  ClnShutBitMask  = 0x08000000;
  HrdErrBitMask   = 0x04000000;
  ```

  Bit `ClnShutBitMask`: If bit is 1, the volume is “clean”. The volume can be mounted for access. If bit is 0, the volume is “dirty” indicating that a FAT file system driver was unable to dismount the volume properly (during a prior mount operation). The volume contents should be scanned for any damage to file system metadata.

  Bit `HrdErrBitMask`: If this bit is 1, no disk read/write errors were encountered. If this bit is 0, the file system driver implementation encountered a disk I/O error on the volume the last time it was mounted, which is an indicator that some sectors may have gone bad. The volume contents should be scanned with a disk repair utility that does surface analysis on it looking for new bad sectors.

4.3 Free space determination

The file system driver implementation must scan through all FAT entries to construct a list of free/available clusters. A free cluster is identified by the following value recorded in the corresponding FAT entry:

- FAT12: 0x000
- FAT16: 0x0000
- FAT32: 0x0000000 (note that the highest order 4 bits are reserved and ignored)

Note that the list of free clusters is not stored on the volume. On FAT32 volumes, the `BPB_FSInfo` sector *may* contain a valid count of free clusters on the volume (see *Section 5*).

4.4 Other points to note

- The last sector of the FAT is not necessarily all part of the FAT. The FAT stops at the cluster number in the last FAT sector that corresponds to the entry for cluster number (`CountOfClusters + 1`), and this entry is not necessarily at the end of the last FAT sector.

  FAT implementations must not make any assumptions about what the contents of the last FAT sector are after the (`CountOfClusters + 1`) FAT entry. When initializing (formatting) a volume, the implementation must zero all bytes after this last FAT entry.

- The number of sectors reserved for each FAT (count of sectors in the `BPB_FATSz16` or `BPB_FATSz32` fields) *may* be bigger than the actual number of sectors required for containing the entire FAT. Therefore, there may be totally unused FAT sectors at the end of each FAT in the FAT region of the volume. Each implementation must determine the value for the last valid sector in the FAT using *`CountOfClusters`* (the last valid sector in the FAT is the one containing the FAT entry numbered *`CountOfClusters + 1`*).

  All sectors reserved for the FAT beyond the last valid sector (defined as the one containing the FAT entry for the last cluster) must be set to 0x0 during volume initialization/format.

Section 5: File System Information (`FSInfo`) Structure

The File System Information (`FSInfo`) structure helps optimize file system driver implementations by containing a count of the number of free clusters on the volume as well as the cluster number for the first available (free) cluster.

NOTE: The information contained in the structure must be considered as advisory only and file system driver implementations must validate the values at volume mount. File system driver implementations are not required to ensure that information within the structure is kept consistent although this is recommended.

The `FSInfo` structure is only present on volumes formatted FAT32. The structure must be persisted on the media during volume initialization (format). The structure must be located at sector #1 – immediately following the sector containing the BPB.

A copy of the structure is maintained at sector #7.

The below table describes the fields comprising the `FSInfo` structure:

Descriptive name of field	Offset (byte)	Size (bytes)	Description
FSI_LeadSig	0	4	Value = 0x41615252. This lead signature is used to validate the beginning of the `FSInfo` structure in the sector.
FSI_Reserved1	4	480	Reserved. Must be set to 0.
FSI_StrucSig	484	4	Value = 0x61417272. An additional signature validating the integrity of the `FSInfo` structure.
FSI_Free_Count	488	4	Contains the last known free cluster count on the volume. The value 0xFFFFFFFF indicates the free count is not known. The contents of this field must be validated at volume mount (and subsequently maintained in memory by the file system driver implementation). It is recommended that this field contain an accurate count of the number of free clusters at volume dismount (during controlled dismount/shutdown).

FSI_Nxt_Free	492	4	Contains the cluster number of the first available (free) cluster on the volume. The value 0xFFFFFFFF indicates that there exists no information about the first available (free) cluster. The contents of this field must be validated at volume mount. It is recommended that this field be appropriately updated at volume dismount (during controlled dismount/shutdown).
FSI_Reserved2	496	12	Reserved. Must be set to 0.
FSI_TrailSig	508	4	Value = 0xAA550000. . This trail signature is used to validate the integrity of the data in the sector containing the `FSInfo` structure.

Section 6: Directory Structure

A FAT directory is a special file type. The directory serves as a container for other files and sub-directories. Directory contents (data) are a series of 32 byte *directory entries*. Each directory entry in turn, typically represents a contained file or sub-directory directory.

The below table describes fields comprising each directory entry:

Descriptive name of field	Offset (byte)	Size (bytes)	Description
DIR_Name	0	11	"Short" file name limited to 11 characters (*8.3 format*).
DIR_Attr	11	1	Legal file attribute types are as defined below: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME is defined as follows: (ATTR_READ_ONLY **|** ATTR_HIDDEN **|** ATTR_SYSTEM **|** ATTR_VOLUME_ID) The upper two bits of the attribute byte are reserved and must always be set to 0 when a file is created. These bits are not interpreted.
DIR_NTRes	12	1	Reserved. Must be set to 0.
DIR_CrtTimeTenth	13	1	Component of the file creation time. Count of tenths of a second. Valid range is: 0 <= `DIR_CrtTimeTenth` <= 199
DIR_CrtTime	14	2	Creation time. Granularity is 2 seconds.
DIR_CrtDate	16	2	Creation date.
DIR_LstAccDate	18	2	Last access date. Last access is defined as a read or write operation performed on the file/directory described by this entry. This field must be updated on file modification (write operation) and the date value must be equal to DIR_WrtDate.
DIR_FstClusHI	20	2	High word of first data cluster number for file/directory described by this entry. Only valid for volumes formatted FAT32. Must be set to 0 on volumes formatted FAT12/FAT16.

DIR_WrtTime	22	2	Last modification (write) time. Value must be equal to DIR_CrtTime at file creation.
DIR_WrtDate	24	2	Last modification (write) date. Value must be equal to DIR_CrtDate at file creation.
DIR_FstClusLO	26	2	Low word of first data cluster number for file/directory described by this entry.
DIR_FileSize	28	4	32-bit quantity containing size in bytes of file/directory described by this entry.

6.1 File/Directory Name (field DIR_Name)

The DIR_Name field contains the 11 character ("*short*") name of the file or sub-directory described by the corresponding entry containing the field. The DIR_Name field is logically comprised of two components:

- The 8-character main part of the name
- The 3-character extension

Each of the above two components are "trailing space padded" if required (using value: 0x20).

Note the following:

1. An implied '.' character separates the main part of the name from the extension. The "." separator character is never stored in the DIR_Name field.
2. DIR_Name[0][3] = 0xE5 indicates the directory entry is free (available). However, 0xE5 is a valid KANJI lead byte value for the character set used in Japan. For KANJI character set based names, the value 0x05 is stored in DIR_Name[0] - if required - to represent 0xE5. If the FAT file system implementation reads DIR_NAME[0] = 0x05 and if the character set used is KANJI, it must perform the appropriate substitution in memory prior to returning the name to the application.
3. DIR_Name[0] = 0x00 also indicates the directory entry is free (available). However, DIR_Name[0] = 0x00 is an additional indicator that *all directory entries following the current free entry are also free*.
4. DIR_Name[0] cannot equal 0x20 (in other words, names cannot start with a space character).
5. All names in the directory must be unique.

Restrictions on characters comprising the name

- Lower case characters are not allowed
- Illegal values for characters in the name are as follows:
    - Values less than 0x20 (except for the special case of 0x05 in DIR_Name[0] described earlier)
    - 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C

The below examples illustrate how names are handled and persisted:

[3] This indicates the first byte in the field.

User entered file/directory name	Stored contents of DIR_Name field
“foo.bar”	“FOO BAR”
“FOO.BAR”	“FOO BAR”
“Foo.Bar”	“FOO BAR”
“foo”	“FOO “
“foo.”	“FOO “
“PICKLE.A”	“PICKLE A “
“prettybg.big”	“PRETTYBGBIG”
“.big”	“ BIG”

Name Limits and Character Set for Short File Names

Short file names are limited to 8 characters followed by an optional period (“.”) and extension of up to 3 characters. The characters comprising a short file name may be any combination of letters, digits, or characters with code point values greater than 127. The following special characters are also allowed:

$ % ' - _ @ ~ ` ! ( ) { } ^ # &

Names are stored in a short name directory entry in the OEM code page that the system is configured for at the time the directory entry is created. Short name directory entries remain in OEM character set for compatibility. OEM characters are single 8-bit characters or can be DBCS character pairs for certain code pages.

Short file names passed to the file system are always converted to upper case and their original case value is lost. A problem that is generally true of most OEM code pages is that they map lower to upper case extended characters in a non-unique fashion i.e. they map multiple extended characters to a single upper case character. This creates problems because it does not preserve the information that the extended character provides. This mapping also prevents the creation of some file names that would normally differ, but because of the mapping to upper case they become the same file name.

6.2 File/Directory Attributes

Attribute values associated with a file or sub-directory impact behavior of the file system driver implementation. Certain attribute values can be legally combined. The table below describes the possible attribute values for a file or directory:

Attribute (value)	Description
ATTR_READ_ONLY (0x01)	The file cannot be modified – all modification requests must fail with an appropriate error code value.
ATTR_HIDDEN (0x02)	The corresponding file or sub-directory must not be listed unless a request is issued by the user/application explicitly requesting inclusion of “hidden files”.
ATTR_SYSTEM (0x04)	The corresponding file is tagged as a component of the operating system. It must not be listed unless a request is issued by the

	user/application explicitly requesting inclusion of "system files".
ATTR_VOLUME_ID (0x08)	The corresponding entry contains the volume label. `DIR_FstClusHI` and `DIR_FstClusLO` must always be 0 for the corresponding entry (representing the volume label) since no clusters can be allocated for this entry. Only the root directory (see *Section 6.x* below) can contain one entry with this attribute. No sub-directory must contain an entry of this type. Entries representing long file names (see *Section 7*) are exceptions to these rules.
ATTR_DIRECTORY (0x10)	The corresponding entry represents a directory (a child or sub-directory to the containing directory). `DIR_FileSize` for the corresponding entry must always be 0 (even though clusters may have been allocated for the directory).
ATTR_ARCHIVE (0x20)	This attribute must be set when the file is created, renamed, or modified. The presence of this attribute indicates that properties of the associated file have been modified. Backup utilities can utilize this information to determine the set of files that need to be backed up to ensure protection in case of media and other failure conditions.

6.3 Date / Time[4]

The following date/time fields in a directory entry are optional (i.e. the file system implementation may choose not to maintain/update the contents of these fields):

- `DIR_CrtTime`
- `DIR_CrtTimeTenth`
- `DIR_CrtDate`
- `DIR_LstAccDate`

Any of the fields above not supported by a file system implementation must be set to 0. Field values set to 0 should be ignored by a file system implementation.

The following date/time fields must be appropriately updated by the file system driver implementation:

- `DIR_WrtTime`

[4] Bit position 0 is the least significant bit; bit position 15 is the most significant bit of a 16-bit word

- DIR_WrtDate

Date format

Contents of the DIR_CrtDate, DIR_LstAccDate, and DIR_WrtDate fields must be in the following format:

- Bit positions 0 through 4 represent the day of the month (valid range: 1..31 inclusive)
- Bit positions 5 through 8 represent the month of the year (1 = January, 12 = December, valid range: 1..12 inclusive)
- Bit positions 9 through 15 are the count of years from 1980 (valid range: 0..127 inclusive allowing representation of years 1980 through 2107)

Time format

Directory entry timestamps are 16-bit values with a granularity of 2 seconds. Contents of the DIR_CrtTime and DIR_WrtTime must be in the following format:

- Bit positions 0 through 4 contain elapsed seconds – as a count of 2-second increments (valid range: 0..29 inclusive allowing representation of 0 through 58 seconds)
- Bit positions 5 through 10 represent number of minutes (valid range: 0..59 inclusive)
- Bit positions 11 through 15 represent hours (valid range: 0..23 inclusive)

The valid time range is from Midnight 00:00:00 to 23:59:58.

6.4 File/Directory Size

The maximum file size is 0xFFFFFFFF bytes. Any chain of clusters allocated to a file must be <= 0x100000000 bytes. The last byte in such a chain of clusters cannot be part of the file.

The maximum valid directory size is 2^{21} bytes.

6.5 Directory creation

When a new directory is created, the file system implementation must ensure the following:

- The ATTR_DIRECTORY bit must set in its DIR_Attr field
- The DIR_FileSize must be set to 0
- At least one cluster must be allocated – the contents of the DIR_FstClusLO and DIR_FstClusHI fields must refer to the first allocated cluster number
- If only a single cluster is allocated, the corresponding file allocation table entry must indicate end-of-file condition
- The contents of the allocated cluster(s) must be initialized to 0
- Except for the root directory, each directory must contain the following two entries at the beginning of the directory:
    - The first directory entry must have a directory name set to ".."

      This *dot entry* refers to the current directory. Rules listed above for the DIR_Attr field and DIR_FileSize field must be followed. Since the *dot entry* refers to the current directory (the one containing the *dot entry*), the contents of the DIR_FstClusLO and DIR_FstClusHI fields must be the same as that of the

current directory. All date and time fields must be set to the same value as that for the containing directory.

- The second directory entry must have the directory name set to "`..`"

  This *dotdot entry* refers to the parent of the current directory. Rules listed above for the `DIR_Attr` field and `DIR_FileSize` field must be followed. Since the *dotdot entry* refers to the parent of the current directory (the one containing the *dotdot entry*), the contents of the `DIR_FstClusLO` and `DIR_FstClusHI` fields must be the same as that of the parent of the current directory. If the parent of the current directory is the *root directory* (see below), the `DIR_FstClusLO` and `DIR_FstClusHI` contents must be set to 0. All date and time fields must be set to the same value as that for the containing directory.

6.6 Root Directory

The root directory is a special container file created during volume initialization (format) and, therefore, always present on the formatted volume.

On FAT12 and FAT16 volumes, the root directory must immediately follow the last file allocation table. The location of the first sector of the root directory is computed as below:

```
FirstRootDirSecNum = BPB_ResvdSecCnt + (BPB_NumFATs * BPB_FATSz16)
```

The size of the root directory on FAT12 and FAT16 volumes is computed using the contents of the `BPB_RootEntCnt` field.

On volumes formatted FAT32, the root directory can be of variable size. The location of the first cluster of the root directory on the FAT32 volume is stored in the `BPB_RootClus` field.

Only the root directory can contain an entry with the `DIR_Attr` field contents equal to `ATTR_VOLUME_ID`.

There is no name for the root directory (on most operating system implementations, the implied name "`\`" is used). Further, the root directory does not have any associated date/time stamps. Lastly, the root directory does not contain either the *dot* or *dotdot* entries.

6.7 File allocation

The cluster number of the first cluster of the file is recorded in the directory entry associated with the file. For zero-length files, the first cluster number in the associated directory entry is set to 0.

The file allocation table entry corresponding to the first allocated cluster for a file either indicates that this is the sole/last allocated cluster for the file or contains the cluster number for the next allocated cluster.

Determination of the first sector of an allocated cluster

Section 3.5 describes the computation of `RootDirSectors`, and `FATSz`.

The first sector of the beginning of the data region (cluster #2) is computed as given below:

```
FirstDataSector = BPB_ResvdSecCnt + (BPB_NumFATs * FATSz)
                  + RootDirSectors
```

Given any valid data cluster number **N**, the sector number of the first sector of that cluster is computed as follows:

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

Section 7: Long File Name Implementation (optional)

The DIR_Name field in the directory entry only allows for a 11 character file / sub-directory name comprised of a main part (maximum length of 8 characters) and an extension (maximum length of 3 characters). The contents of this field are also known as the "*short name*" and the corresponding directory entry is also known as the *short name directory entry*. Applications and users typically prefer creating longer (more descriptive) names for files/sub-directories. This section describes how such long file names can be stored on media.

Support for long file names is optional for file system implementations. However, such support is very strongly recommended.

NOTE: Long file names can be supported on any FAT variant i.e. on media formatted FAT12, FAT16 or FAT32.

A long file name for a target file or sub-directory is stored in a set (one or more) of additional directory entries associated with the short name directory entry describing the target file or sub-directory. This set of additional directory entries (also known as the *long name directory entries*) must immediately precede the corresponding short name directory entry and is, therefore, physically contiguous with the short name directory entry.

NOTE: The sequence of long name directory entries is stored in reverse order (last entry in the set is stored first, followed by entry n-1, followed by entry n-2, and so on, until entry 1).

A long name directory entry structure is described in the table below:

Descriptive Name	Offset (byte)	Size (bytes)	Description
LDIR_Ord	0	1	The order of this entry in the sequence of long name directory entries (each containing components of the long file name) associated with the corresponding short name directory entry. The contents of this field must be masked with 0x40 (LAST_LONG_ENTRY) for the last long directory name entry in the set. Therefore, each sequence of long name directory entries begins with the contents of this field masked with LAST_LONG_ENTRY.
LDIR_Name1	1	10	Contains characters 1 through 5 constituting a portion of the long name.

LDIR_Attr	11	1	Attributes – must be set to `ATTR_LONG_NAME` defined as below:
			
			`ATTR_LONG_NAME = (ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID)`
			
			NOTE: A mask to determine whether a directory entry is part of the set of a long name directory entries is defined below:
			
			`#define ATTR_LONG_NAME_MASK = (ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID | ATTR_DIRECTORY | ATTR_ARCHIVE)`

LDIR_Type	12	1	Must be set to 0.
LDIR_Chksum	13	1	Checksum of name in the associated short name directory entry at the end of the long name directory entry set.
LDIR_Name2	14	12	Contains characters 6 through 11 constituting a portion of the long name.
LDIR_FstClusLO	26	2	Must be set to 0.
LDIR_Name3	28	4	Contains characters 12 and 13 constituting a portion of the long name.

The below illustrates how long name directory entries are stored:

Entry	Ordinal
Nth long name directory entry	LAST_LONG_ENTRY (0x40) | N
… Additional long name directory entries	…
1st long name directory entry	1
Short name directory entry associated with preceding long name directory entry set	N/A

7.1 Ordinal Number Generation

The below rules must be followed in storing ordinal numbers for each long name directory entry in a set of such entries (associated with a short name directory entry):

1. The first member of a set has an `LDIR_Ord` value of 1.
2. The `LDIR_Ord` value for each subsequent entry must contain a monotonically increasing value.
3. The Nth (last) member of the set must contain a value of (`N | LAST_LONG_ENTRY`)

If any member of the set of long name directory entries does not follow the rules above, the set is considered corrupt.

7.2 Checksum Generation

An 8-bit checksum is computed on the name contained in the short name directory entry at the time the short and long name directory entries are created. All 11 characters of the name in the short name entry are used in the checksum calculation. The check sum is placed in every long name directory entry in the `LDIR_Chksum` field. If any of the check sums in the set of long name entries associated with the appropriate short name directory entry, do not agree with the computed checksum of the name contained in the short name directory entry, the set of long name directory entries is considered corrupt.

The below algorithm describes the logic used to generate the check sum value:

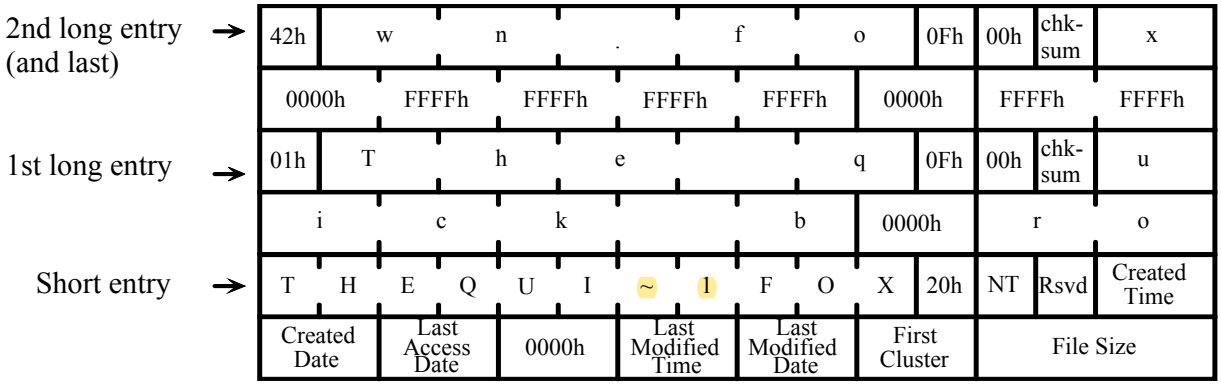
```
//-----------------------------------------------------------------------------
//      ChkSum()
//      Returns an unsigned byte checksum computed on an unsigned byte
//      array.  The array must be 11 bytes long and is assumed to contain
//      a name stored in the format of a MS-DOS directory entry.
//      Passed:  pFcbName    Pointer to an unsigned byte array assumed to be
//                           11 bytes long.
//      Returns: Sum         An 8-bit unsigned checksum of the array pointed
//                           to by pFcbName.
//-----------------------------------------------------------------------------
unsigned char ChkSum (unsigned char *pFcbName)
{
        short FcbNameLen;
        unsigned char Sum;

        Sum = 0;
        for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {
                // NOTE: The operation is an unsigned char rotate right
                Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + *pFcbName++;
        }
        return (Sum);
}
```

7.3 Example illustrating persistence of a long name

The following example is provided to illustrate how a long name is stored across several long name directory entries. Names are also NULL terminated and padded with 0xFFFF characters in order to detect corruption of long name fields. A name that fits exactly in a set of long name directory entries (i.e. is an integer multiple of 13) is not NULL terminated and not padded with 0xFFFF.

Name: `"The quick brown.fox"`



Name Limits and Character Set for Long File Names

Long file names are limited to 255 characters, not including the trailing NULL. The characters may be any combination of those defined for short file names (see *Section 6.1*) with the addition of the period ("**.**") character used multiple times within the long name. A space is also a valid character in a long file name as it always has been for a short file name. The following six special characters are allowed in a long file name (they are not legal in a short file name):

+ , ; = [ ]

Embedded spaces within a long file name are allowed. Leading and trailing spaces in a long file name are ignored.

Leading and embedded periods are allowed in a name and are stored in the long file name. Trailing periods are ignored.

Long file names are stored in long directory entries in UNICODE. UNICODE characters are 16-bit characters. It is not be possible to store UNICODE in short name directory entries since the names stored there are 8-bit characters or DBCS characters.

Long file names passed to the file system are not converted to upper case and their original case value is preserved. UNICODE solves the case mapping problem prevalent in some OEM code pages by always providing a translation for lower case characters to a single, unique upper case character

7.4 Rules governing name creation and matching

The union of all long and short file names is defined as the namespace of objects contained in the volume.

For such a namespace, the following rules must be observed:

- Any name within a specific directory, whether it is a short name or a long name, can occur only once (difference in case is ignored and such names must be considered as conflicting)
- When a character on the media (whether stored in the OEM character set or in UNICODE) cannot be translated into the appropriate character in the OEM or ANSI code page, it is always translated to the "_" (underscore) character – the character is not modified on the media. The "_" character is the same in all OEM code pages and ANSI.

Section 8: EA Support (optional)

This section to be expanded.

Appendix 1: Example BPB for 512 byte sector-size 128 MB HDD media formatted FAT16

Byte Position	Length	Description	Contents
0	3	Intel jump instruction to boot code	0xeb903c
3	8	OEM Name String	MSDOS5.0
11	2	Bytes per sector	0x200
13	1	Sectors per cluster	0x8
14	2	Reserved sectors	0x4
16	1	Number of FATs	0x2
17	2	Root Entry Count	0x200
29	2	16-bit Count of sectors	0
21	1	Media Type ID	0xf8
22	2	Sectors per FAT	0x86
24	2	Sectors per track	0x3f
26	2	Number of heads	0xff
28	4	Number of hidden sectors	0x3f
32	4	32-bit Count of sectors	0x42a92
36	1	Physical drive Number	0x80
37	1	Reserved	0
38	1	Extended boot signature	0x29
39	4	Volume serial number	0xa0309f1e
43	11	Volume label	NO NAME
54	8	Informational FS Type	FAT16
62	448	Reserved	0x0
510	2	Signature	0x55AA

Appendix 2: Example BPB for 512 byte sector-size 128 MB HDD media formatted FAT32

Byte Position	Length	Description	Contents
0	3	Intel jump instruction to boot code	0xeb9058
3	8	OEM Name String	MSDOS5.0
11	2	Bytes per sector	0x200
13	1	Sectors per cluster	0x4
14	2	Reserved sectors	0x20
16	1	Number of FATs	0x2
17	2	Root Entry Count	0
29	2	16-bit Count of sectors	0
21	1	Media Type ID	0xf8
22	2	Sectors per FAT	0
24	2	Sectors per track	0x3f
26	2	Number of heads	0xff
28	4	Number of hidden sectors	0x3f
32	4	32-bit Count of sectors	0x42a92
36	4	32-bit count of sectors per FAT	0x214
40	2	Extended flags	0
42	2	File system version	0
44	4	Root directory start cluster	0x2
48	2	File system information sector	0x1
50	2	Backup boot sector	0x6
52	12	Reserved	0
64	1	Physical drive Number	0x80
65	1	Reserved	0
66	1	Extended boot signature	0x29
67	4	Volume serial number	0x20bccd50
71	11	Volume label	NO NAME
82	8	Informational FS Type	FAT32
90	420	Reserved	0x0
510	2	Signature	0x55AA