# FAT32 Data Layout Specification

Thank you OSDev & Wikipedia

| component | offset | size | description |
|---|---|---|---|
| BIOS Parameter Block | 0 | 3 | Literal: EB 3C 90<br>Comment: used to prevent data execution in boot environment |
| | 3 | 8 | Literal: MSWIN4.1<br>Comment: ignored, but keep it for consistency with standard |
| | 11 | 2 | Comment: bytes per sector<br>Default value: 512 bytes |
| | 13 | 1 | Comment: Sectors per cluster<br>Default: 64; this gives 32Kb clusters |
| | 14 | 2 | Comment: # of reserved sectors; presumably 2? Not sure if FAT is included |
| | 16 | 1 | Comment: # of FATs<br>Default value: 2 (duplicate FAT for redundancy) |
| | 17 | 2 | Comment: # of FAT12 or FAT16 root directory entries<br>Default: 0; this is FAT32 |
| | 19 | 2 | Comment: Sectors on the volume; 0 if > 65535 |
| | 21 | 1 | Comment: Media Descriptor Type<br>Default: F8 |
| | 22 | 2 | Comment: Sectors Per FAT (for FAT16 and 12)<br>Default value: 0; always ignored |
| | 24 | 2 | Physical sectors per track<br>Default value: 0; always ignored |
| | 26 | 2 | Number of heads per disk<br>Default value: 0 |
| | 28 | 4 | Count of hidden sectors preceding volume; used when partitioning is used<br>Default value: 0; we’re not partitioning |
| | 32 | 4 | Comment: large amount of sectors on media; used if sector count > 65535<br>See offset 19 |
| FAT32 extended fields | 36 | 4 | Clusters per FAT |
| | 40 | 2 | Flags<br>Default: 0; ignored |

| | | | |
|---|---|---|---|
| | 42 | 2 | FAT version<br>Default: ??? |
| | 44 | 4 | Cluster # of root directory<br>Default: 2, becuase OSDev says so |
| | 48 | 2 | Sector # for FSInfo structure |
| | 50 | 2 | Sector # for backup boot sector |
| | 52 | 12 | Reserved<br>Default: 0 |
| | 64 | 1 | Driver number<br>Default: 80 (generally ignored otherwise) |
| | 65 | 1 | Flags for Windows NT<br>Default: 0; ignored generally |
| | 66 | 1 | Signature<br>Default: 0x28 |
| | 67 | 4 | VolumeID serial #<br>Default: 0; generally ignored |
| | 71 | 11 | Volume Label String; padded with spaces (0x20)<br>Default: “AlphaGoS” |
| | 82 | 8 | System identifier string<br>Default: “FAT32   “; ignore this |
| | 90 | 420 | Boot code; default: empty |
| | 510 | 2 | Bootable partition signature. Default: 0xAA55 |
| FS INFO Sector | 0 (starting in sector 1) | 4 | Compatibility guard for earlier versions of FAT<br>Default: 0x52 0x52 0x61 0x41 |
| | 4 | 480 | Reserved; default to 0 |
| | 484 | 4 | Info sector signature; Default 0x72 0x72 0x41 0x61 |
| | 488 | 4 | Last known number of free data clusters; should not rely on this; defaults to 0xFFFFFFFF; should be ignored for this implementation |
| | 492 | 4 | Number of most recently known allocated data cluster; set to 0xFFFFFFFF. Ignore |
| | 496 | 12 | Reserved; set to 0 |
| | 508 | 4 | Sector signature. Set to 0x00 0x00 0x55 0xAA |

## FAT Table

Entries are 32 bit values. #of entries is determined by value at offset 36 of above. Note that there are two FATs back to back.

| Value | Meaning | Comment |
|---|---|---|
| 0x?0000000 | Free Cluster | |
| 0x?0000001 | Reserved Cluster | |
| 0x?0000002 - 0x?FFFFFEF | Used cluster; value points to next cluster | |
| 0x?FFFFFF0 - 0x?FFFFFF6 | Reserved | |
| 0x?FFFFFF7 | Bad Cluster (do not use) | |
| 0x?FFFFFF8-0xFFFFFF | Last cluster in file | Once this is hit, stop reading the file |

The first two entries in the FAT are special:

- Cluster zero: 0x011111F0
- Cluster 1: 0x0FFFFFFF

Data begins immediately after last FAT table
Per specification above (see offset 44) the root directory is at cluster 2

## Directory Entries

32 byte entries
1024 entries per cluster
Just Data

| Offset | Length | Comment | Values Details |
|---|---|---|---|
| 0 | 11 | 8.3 file name; first 8 are name; last 3 are ext | First byte 0 ->no more entries<br>First byte 0xE5 -> unused entry<br>First byte 0x1 -> used entry<br>All other |

|  |  |  |  |
|---|---|---|---|
|  |  |  | values: undefined; field unused for compliance |
| 11 | 1 | Attributes;<br>0x01: readonly; 0x02: hidden; 0x04: system; 0x08: volume_id<br>Above flags indicate this is associated with a long-file-name entry<br>0x10: directory; 0x20=archive |  |
| 12 | 1 | Reserved for use by Windows NT |  |
| 13 | 1 | Creation time in tenths of a second; 0-199 inclusive |  |
| 14 | 2 | Creation time: Hour: 5bits, Minutes:6bits: seconds: 5bits (multiply by 2) |  |
| 16 | 2 | Creation date: Year:7bits;Month:4bits;Day:5bits |  |
| 18 | 2 | Last accessed date. See format above |  |
| 20 | 2 | High 16 bits of cluster number |  |
| 22 | 2 | Last modification time |  |
| 24 | 2 | Last modification date |  |
| 26 | 2 | Low 16 bits of cluster number |  |
| 28 | 4 | Size of file in bytes |  |

## Long file names:

- Long file names always precede the short-file-name version
- Any number of long file name entries may exist for a file or directory
- Because multiple name blocks may exist, they contain an ordering field; they are not guaranteed to appear in correct display order on disk

| Offset | Length | Comment |
|---|---|---|
| 0 | 1 | Order (index) in the overall long file name |

| | | |
|---|---|---|
| 1 | 10 | First 5 2-byte (think unicode) characters |
| 11 | 1 | Always 0x0F (used to identify that this is a name block) |
| 12 | 1 | Zero (for name entry) |
| 13 | 1 | Checksum of the short file name; ignore |
| 14 | 12 | Next 6 2-byte characters |
| 26 | 2 | Always Zero |
| 28 | 4 | Final 2 2-byte characters of the name |

### FAT32/VFAT Patent Compliance

The process of using directory entries to store extended names in addition to the 8.3 filenames is patented. To avoid infringing on the 100% completely valid and sensible patent, this implementation will not consider the 8.3 filename

# Implementations for Common Processes on FAT32 FS

## Lookup File By Name - Long File Names

Assumes starting from root

- Go to root directory at cluster 2
- Iterate through 32-byte entries
    - If this is a long-name entry
        - Read up to short name entry
        - Assemble name; don’t forget to account for 2-byte vs 1-byte chars
        - Compare both long and short names
    - If end of cluster reached AND cluster is marked as having a next cluster
        - Jump to that cluster
        - Continue search
        - Note: you could jump clusters while assembling a long-file-name

## Calculating Raw-Byte offsets for clusters

- Simple driver specification says that data is accessed by byte offset from 0
- Note: on 32-bit systems, this limits us to 4Gb

Notes:

- Sectors are 512 bytes; not apparently relevant here
- FAT Table is offset by 2 sectors

- Clusters are 32Kb each
- FAT table consumes 32bits * Cluster-per-fat space
- Note that there are 2 FATs (most likely)
- Clusters 0 and 1 are reserved; data starts at cluster 2

Byte_for_cluster = 2*sector_size + 4*cluster_per_fat*2 + 32768*cluster
= 1024 + 8*cluster_count + 32768*cluster_number

## Calculating Raw-Byte offsets to get to the FAT

- FAT0: 1024
- FAT1: 1024 + sizeof(FAT0)

Size of a FAT is 4*clusters_per_fat

# Implementing Query on top of the Simple Driver Specification

Supported metadata properties:

- Created
- Last updated
- Is Directory
- Readonly
- Hidden
- System
- Last read
- Size in bytes

Side effects

- Updates last read

# Implementing Set on top of the Simple Driver Specification

Supported metadata properties:

- Created - not settable
- Last updated - not settable
- Is Directory - not settable
- Readonly
- Hidden
- System
- Last read - not settable
- Size in bytes - not settable
- Name

Side Effects

- Updates last updated

Cautions:

- Changing the name may have dramatic implications for directory table in case of long file names

General algorithm:

1. Recursively read filesystem until directory entry for file is found
2. If it is a regular property, update it.
3. Else if the name is being updated:
    a. Determine # of entries currently being used for name
    b. If # of entries needed for new name is same as old name, update in place
    c. If the # of entries differs, construct new entries in memory
        i. Then shift table up
        ii. Append entry at end of directory
        iii. Note: may need to jump to next cluster for any part of this operation
4. Update last-updated field

# Implementing Read on top of the Simple Driver Specification

Cautions:

- Be careful about managing # of bytes read when jumping between clusters

Side Effects

- Updates last read

General algorithm:

1. Recursively read filesystem until directory entry for file is found
2. Read the cluster and the size of the file
    a. Use the file offset to determine start point in file and jump clusters as needed
    b. Read into a buffer until end of cluster is reached, then jump
    c. Continue reading until bytes read count is consistent with request or end of file is reached (offset + bytes_read == file_size)

Note: reading is finished when the current position is cluster_start

# Implementing Create on top of the Simple Driver Specification

Cautions:

- Don’t forget to update duplicate FAT

Side Effects

- Creates entry in directory table
- Updates FAT table

General Algorithm:

1. Recursively find the directory entry to which this will be added
2. Construct long file name entry
3. Construct short file name entry
4. Grab a free cluster from FAT, update entries

# Implementing Delete on top of the Simple Driver Specification

Cautions:

- Make sure to delete associated long file name entries
- Don't delete the root directory
- When updating FAT, don't forget to update duplicate FAT

Side effects:

- Removes entries (potentially many) from FAT table
- Removes entries (potentially many) from Directory entry

General Algorithm:

1. Recursively find the directory entry for this
2. Recursively (accounting for jumps) free associated cluster entries in the FAT
3. Mark directory entry as unused

# Implementing Write on top of the Simple Driver Specification

Cautions:

- When updating FAT, don't forget to update duplicate FAT

Side effects:

- Updates last updated in directory entry
- Updates size in bytes in directory table
- May require pulling more clusters from the FAT table and updating the existing ones

General Algorithm:

1. Recursively read filesystem until directory entry for file is found
2. Read the cluster and the size of the file
    a. Use the file offset to determine start point in file and jump clusters as needed
    b. Write from buffer until end of cluster is reached or end of buffer is reached
        i. If end of cluster is reached, grab a new cluster, update FATs, continue writing at next FAT

# Implementing the Formatter on top of the RAW File System

Because an unformatted device does not have the information needed by FAT, it must first be formatted using a separate utility. To do this from within the OS:

- Mount the device with RawFS
- Run the formatter utility on the mounted directory
- Unmount the directory
- Test and mount with FAT32FS

Issues: need to decide on number of clusters based on size of device

- Fixed overhead: 1024 bytes (1Kb)
- Cluster size: 32kbytes
- Cluster count: ((device_size_kbytes - 1Kb) / 32kbytes) + 2 -1
    - -1 above accounts for size of FAT table ( it is at most 4kb)