Microsoft

Hardware White Paper

Designing Hardware for Microsoft ®Operating Systems

FAT: General Overview of On-Disk Format

Version 1.02, May 5, 1999
Microsoft Corporation

The FAT (File Allocation Table) file system has its origins in the late 1970s and early1980s and was the file system supported by the Microsoft® MS-DOS® operating system. It was originally developed as a simple file system suitable for floppy disk drives less than 500K in size. Over time it has been enhanced to support larger and larger media. Currently there are three FAT file system types: FAT12, FAT16 and FAT32. The basic difference in these FAT sub types, and the reason for the names, is the size, in bits, of the entries in the actual FAT structure on the disk. There are 12 bits in a FAT12 FAT entry, 16 bits in a FAT16 FAT entry and 32 bits in a FAT32 FAT entry.

Contents

Disclaimer

IMPORTANT—READ CAREFULLY: This Microsoft Agreement ("Agreement") is a legal agreement between you (either an individual or a single entity) and Microsoft Corporation ("Microsoft") for this version of the Microsoft specification identified above ("Specification"). **BY DOWNLOADING, COPYING OR OTHERWISE USING THE SPECIFICATION, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT DOWNLOAD, COPY, OR USE THE SPECIFICATION.**

The Specification is owned by Microsoft or its suppliers and is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties.

1. **LIMITED COVENANT NOT TO SUE**.

(a) Provided that you comply with all terms and conditions of this Agreement and subject to the limitations in Sections 1(b) – (e) below, Microsoft grants to you the following non-exclusive, worldwide, royalty-free, non-transferable, non-sublicenseable, reciprocal limited covenant not to sue:

(i) under any copyrights owned or licensable by Microsoft without payment of consideration to unaffiliated third parties, to reproduce the Specification solely for the purposes of creating portions of products which comply with the Specification in unmodified form; and

(ii) under its Necessary Claims solely to make, have made, use, import, and directly and indirectly, offer to sell, sell and otherwise distribute and dispose of portions of products which comply with the Specification in unmodified form.

For purposes of the foregoing, the Specification is **"unmodified"** if there are no changes, additions or extensions to the Specification, and **"Necessary Claims"** means claims of a patent or patent application which are (1) owned or licenseable by Microsoft without payment of consideration to an unaffiliated third party; and (2) have an effective filing date on or before December 31, 2010, that must be infringed in order to make a portion(s) of a product that complies with the Specification. Necessary Claims does not include claims relating to semiconductor manufacturing technology or microprocessor circuits or claims not required to be infringed in complying with the Specification (even if in the same patent as Necessary Claims).

(b) The foregoing covenant not to sue shall not extend to any part or function of a product which (i) is not required to comply with the Specification in unmodified form, or (ii) to which there was a commercially reasonable alternative to infringing a Necessary Claim.

(c) The covenant not to sue described above shall be unavailable to you and shall terminate immediately if you or any of your Affiliates (collectively "Covenantee Party") "Initiates" any action for patent infringement against: (x) Microsoft or any of its Affiliates (collectively "Granting Party"), (y) any customers or distributors of the Granting Party, or other recipients of a covenant not to sue with respect to the Specification from the Granting Party ("Covenantees"); or (z) any customers or distributors of Covenantees (all parties identified in (y) and (z) collectively referred to as "Customers"), which action is based on a conformant implementation of the Specification. As used herein, "Affiliate" means any entity which directly or indirectly controls, is controlled by, or is under common control with a party; and control shall mean the power, whether direct or indirect, to direct or cause the direct of the management or policies of any entity whether through the ownership of voting securities, by contract or otherwise. "Initiates" means that a Covenantee Party is the first (as between the Granting Party and the Covenantee Party) to file or institute any legal or administrative claim or action for patent infringement against the Granting Party or any of the Customers. "Initiates" includes any situation in which a Covenantee Party files or initiates a legal or administrative claim or action for patent infringement solely as a counterclaim or equivalent in response to a Granting Party first filing or instituting a legal or administrative patent infringement claim against such Covenantee Party.

(d) The covenant not to sue described above shall not extend to your use of any portion of the Specification for any purpose other than (a) to create portions of an operating system (i) only as necessary to adapt such operating system so that it can directly interact with a firmware implementation of the Extensible Firmware Initiative Specification v. 1.0 ("EFI Specification"); (ii) only as necessary to emulate an implementation of the EFI Specification; and (b) to create firmware, applications, utilities and/or drivers that will be used and/or licensed for only the following purposes: (i) to install, repair and maintain hardware, firmware and portions of operating system software which are utilized in the boot process; (ii) to provide to an operating system runtime services that are specified in the EFI Specification; (iii) to diagnose and correct failures in the hardware, firmware or operating system software; (iv) to query for identification of a computer system (whether by serial numbers, asset tags, user

or otherwise); (v) to perform inventory of a computer system; and (vi) to manufacture, install and setup any hardware, firmware or operating system software.

(e) Microsoft reserves all other rights it may have in the Specification and any intellectual property therein. The furnishing of this document does not give you any covenant not to sue with respect to any other Microsoft patents, trademarks, copyrights or other intellectual property rights; or any license with respect to any Microsoft intellectual property rights.

2. **ADDITIONAL LIMITATIONS AND OBLIGATIONS**.
   (a) The foregoing covenant not to sue is applicable only to the version of the Specification which you are about to download. It does not apply to any additional versions of or extensions to the Specification.
   (b) Without prejudice to any other rights, Microsoft may terminate this Agreement if you fail to comply with the terms and conditions of this Agreement. In such event you must destroy all copies of the Specification.
3. **INTELLECTUAL PROPERTY RIGHTS**. All ownership, title and intellectual property rights in and to the Specification are owned by Microsoft or its suppliers.
4. **U.S. GOVERNMENT RIGHTS**. Any Specification provided to the U.S. Government pursuant to solicitations issued on or after December 1, 1995 is provided with the commercial rights and restrictions described elsewhere herein. Any Specification provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 is provided with RESTRICTED RIGHTS as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFAR, 48 CFR 252.227-7013 (OCT 1988), as applicable.
5. **EXPORT RESTRICTIONS**. Export of the Specification, any part thereof, or any process or service that is the direct product of the Specification (the foregoing collectively referred to as the "Restricted Components") from the United States is regulated by the Export Administration Regulations (EAR, 15 CFR 730-744) of the U.S. Commerce Department, Bureau of Export Administration ("BXA"). You agree to comply with the EAR in the export or re-export of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan, Syria and the Federal Republic of Yugoslavia (including Serbia, but not Montenegro), or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any person or entity who you know or have reason to know will utilize the Restricted Components in the design, development or production of nuclear, chemical or biological weapons; or (iii) to any person or entity who has been prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked or denied your export privileges. For additional information see http://www.microsoft.com/exporting.
6. **DISCLAIMER OF WARRANTIES. To the maximum extent permitted by applicable law, Microsoft and its suppliers provide the Specification (and all intellectual property therein) and any (if any) support services related to the Specification ("Support Services")** ***AS IS AND WITH ALL FAULTS*****, and hereby disclaim all warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties or conditions of merchantability, of fitness for a particular purpose, of lack of viruses, of accuracy or completeness of responses, of results, and of lack of negligence or lack of workmanlike effort, all with regard to the Specification, any intellectual property therein and the provision of or failure to provide Support Services. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT, WITH REGARD TO THE SPECIFICATION AND ANY INTELLECTUAL PROPERTY THEREIN. THE ENTIRE RISK AS TO THE QUALITY OF OR ARISING OUT OF USE OR PERFORMANCE OF THE SPECIFICATION, ANY INTELLECTUAL PROPERTY THEREIN, AND SUPPORT SERVICES, IF ANY, REMAINS WITH YOU.**
7. **EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SPECIFICATION, ANY INTELLECTUAL PROPERTY THEREIN, THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS AGREEMENT, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**
8. **LIMITATION OF LIABILITY AND REMEDIES. Notwithstanding any damages that you might incur for any reason whatsoever (including, without limitation, all damages referenced above and all direct or general damages), the entire liability of Microsoft and any of its suppliers under any provision of this Agreement and your exclusive remedy for all of the foregoing shall be limited to the greater**

of the amount actually paid by you for the Specification or U.S.$5.00. The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails its essential purpose.

9. **APPLICABLE LAW**. If you acquired this Specification in the United States, this Agreement is governed by the laws of the State of Washington. If you acquired this Specification in Canada, unless expressly prohibited by local law, this Agreement is governed by the laws in force in the Province of Ontario, Canada; and, in respect of any dispute which may arise hereunder, you consent to the jurisdiction of the federal and provincial courts sitting in Toronto, Ontario. If this Specification was acquired outside the United States, then local law may apply.
10. **QUESTIONS**. Should you have any questions concerning this Agreement, or if you desire to contact Microsoft for any reason, please contact the Microsoft subsidiary serving your country, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.
11. **ENTIRE AGREEMENT. This Agreement is the entire agreement between you and Microsoft relating to the Specification and the Support Services (if any) and they supersede all prior or contemporaneous oral or written communications, proposals and representations with respect to the Specification or any other subject matter covered by this Agreement. To the extent the terms of any Microsoft policies or programs for Support Services conflict with the terms of this Agreement, the terms of this Agreement shall control.**

Si vous avez acquis votre produit Microsoft au CANADA, la garantie limitée suivante vous concerne :

RENONCIATION AUX GARANTIES. Dans toute la mesure permise par la législation en vigueur, Microsoft et ses fournisseurs fournissent la Specification (et à toute propriété intellectuelle dans celle-ci) et tous (selon le cas) les services d'assistance liés à la Specification ("Services d'assistance") TELS QUELS ET AVEC TOUS LEURS DÉFAUTS, et par les présentes excluent toute garantie ou condition, expresse ou implicite, légale ou conventionnelle, écrite ou verbale, y compris, mais sans limitation, toute (selon le cas) garantie ou condition implicite ou légale de qualité marchande, de conformité à un usage particulier, d'absence de virus, d'exactitude et d'intégralité des réponses, de résultats, d'efforts techniques et professionnels et d'absence de négligence, le tout relativement à la Specification, à toute propriété intellectuelle dans celle-ci et à la prestation ou à la non-prestation des Services d'assistance. DE PLUS, IL N'Y A AUCUNE GARANTIE ET CONDITION DE TITRE, DE JOUISSANCE PAISIBLE, DE POSSESSION PAISIBLE, DE SIMILARITÉ À LA DESCRIPTION ET D'ABSENCE DE CONTREFAÇON RELATIVEMENT À LA SPÉCIFICATION ET À TOUTE PROPRIÉTÉ INTELLECTUELLE DANS CELLE-CI. VOUS SUPPORTEZ TOUS LES RISQUES DÉCOULANT DE L'UTILISATION ET DE LA PERFORMANCE DE LA SPÉCIFICATION ET DE TOUTE PROPRIÉTÉ INTELLECTUELLE DANS CELLE-CI ET CEUX DÉCOULANT DES SERVICES D'ASSISTANCE (S'IL Y A LIEU).

EXCLUSION DES DOMMAGES INDIRECTS, ACCESSOIRES ET AUTRES. Dans toute la mesure permise par la législation en vigueur, Microsoft et ses fournisseurs ne sont en aucun cas responsables de tout dommage spécial, indirect, accessoire, moral ou exemplaire quel qu'il soit (y compris, mais sans limitation, les dommages entraînés par la perte de bénéfices ou la perte d'information confidentielle ou autre, l'interruption des affaires, les préjudices corporels, la perte de confidentialité, le défaut de remplir toute obligation y compris les obligations de bonne foi et de diligence raisonnable, la négligence et toute autre perte pécuniaire ou autre perte de quelque nature que ce soit) découlant de, ou de toute autre manière lié à, l'utilisation ou l'impossibilité d'utiliser la Spécification, toute propriété intellectuelle dans celle-ci, la prestation ou la non-prestation des Services d'assistance ou autrement en vertu de ou relativement à toute disposition de cette convention, que ce soit en cas de faute, de délit (y compris la négligence), de responsabilité stricte, de manquement à un contrat ou de manquement à une garantie de Microsoft ou de l'un de ses fournisseurs, et ce, même si Microsoft ou l'un de ses fournisseurs a été avisé de la possibilité de tels dommages.

LIMITATION DE RESPONSABILITÉ ET RECOURS. Malgré tout dommage que vous pourriez encourir pour quelque raison que ce soit (y compris, mais sans limitation, tous les dommages mentionnés ci-dessus et tous les dommages directs et généraux), la seule responsabilité de Microsoft et de ses fournisseurs en vertu de toute disposition de cette convention et votre unique recours en regard de tout ce qui précède sont limités au plus élevé des montants suivants: soit (a) le montant que vous avez payé pour la Spécification, soit (b) un montant équivalant à cinq dollars U.S. (5,00 $ U.S.). Les limitations, exclusions et renonciations ci-dessus s'appliquent dans toute la mesure permise par la législation en vigueur, et ce même si leur application a pour effet de priver un recours de son essence.

DROITS LIMITÉS DU GOUVERNEMENT AMÉRICAIN

Tout Produit Logiciel fourni au gouvernement américain conformément à des demandes émises le ou après le 1er décembre 1995 est offert avec les restrictions et droits commerciaux décrits ailleurs dans la présente convention. Tout Produit Logiciel fourni au gouvernement américain conformément à des demandes émises avant le 1er décembre 1995 est offert avec des DROITS LIMITÉS tels que prévus dans le FAR, 48CFR 52.227-14 (juin 1987) ou dans le FAR, 48CFR 252.227-7013 (octobre 1988), tels qu'applicables.

Sauf lorsqu'expressément prohibé par la législation locale, la présente convention est régie par les lois en vigueur dans la province d'Ontario, Canada. Pour tout différend qui pourrait découler des présentes, vous acceptez la compétence des tribunaux fédéraux et provinciaux siégeant à Toronto, Ontario.

Si vous avez des questions concernant cette convention ou si vous désirez communiquer avec Microsoft pour quelque raison que ce soit, veuillez contacter la succursale Microsoft desservant votre pays, ou écrire à: Microsoft Sales Information Center, One Microsoft Way, Redmond, Washington 98052-6399.

Notational Conventions in this Document

Numbers that have the characters “0x” at the beginning of them are hexadecimal (base 16) numbers.

Any numbers that do not have the characters “0x” at the beginning are decimal (base 10) numbers.

The code fragments in this document are written in the ‘C’ programming language. Strict typing and syntax are not adhered to.

There are several code fragments in this document that freely mix 32-bit and 16-bit data elements. It is assumed that you are a programmer who understands how to properly type such operations so that data is not lost due to truncation of 32-bit values to 16-bit values. Also take note that all data types are UNSIGNED. Do not do FAT computations with signed integer types, because the computations will be wrong on some FAT volumes.

General Comments (Applicable to FAT File System All Types)

All of the FAT file systems were originally developed for the IBM PC machine architecture. The importance of this is that FAT file system on disk data structure is all “little endian.” If we look at one 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[4]—here is where the 32 bits numbered 00 through 31 are (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
```

This is important if your machine is a “big endian” machine, because you will have to translate between big and little endian as you move data to and from the disk.

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

Boot Sector and BPB

The first important data structure on a FAT volume is called the BPB (BIOS Parameter Block), which is located in the first sector of the volume in the Reserved Region. This sector is sometimes called the “boot sector” or the “reserved sector” or the “0^{th} sector,” but the important fact is simply that it is the first sector of the volume.

This is the first thing about the FAT file system that sometimes causes confusion. In MS-DOS version 1.x, there was not a BPB in the boot sector. In this first version of the FAT file system, there were only two different formats, the one for single-sided and the one for double-sided 360K 5.25-inch

floppy disks. The determination of which type was on the disk was done by looking at the first byte of the FAT (the low 8 bits of FAT[0]).

This type of media determination was superseded in MS-DOS version 2.x by putting a BPB in the boot sector, and the old style of media determination (done by looking at the first byte of the FAT) was no longer supported. All FAT volumes must have a BPB in the boot sector.

This brings us to the second point of confusion relating to FAT volume determination: What exactly does a BPB look like? The BPB in the boot sector defined for MS-DOS 2.x only allowed for a FAT volume with strictly less than 65,536 sectors (32 MB worth of 512-byte sectors). This limitation was due to the fact that the "total sectors" field was only a 16-bit field. This limitation was addressed by MS-DOS 3.x, where the BPB was modified to include a new 32-bit field for the total sectors value.

The next BPB change occurred with the Microsoft Windows 95 operating system, where the FAT32 type was introduced. FAT16 was limited by the maximum size of the FAT and the maximum valid cluster size to no more than a 2 GB volume if the disk had 512-byte sectors. FAT32 addressed this limitation on the amount of disk space that one FAT volume could occupy so that disks larger than 2 GB only had to have one partition defined.

The FAT32 BPB exactly matches the FAT12/FAT16 BPB up to and including the BPB_TotSec32 field. They differ starting at offset 36, depending on whether the media type is FAT12/FAT16 or FAT32 (see discussion below for determining FAT type). The relevant point here is that the BPB in the boot sector of a FAT volume should always be one that has all of the new BPB fields for either the FAT12/FAT16 or FAT32 BPB type. Doing it this way ensures the maximum compatibility of the FAT volume and ensures that all FAT file system drivers will understand and support the volume properly, because it always contains all of the currently defined fields.

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 following shows the start of sector 0 of a FAT volume, which contains the BPB:

Boot Sector and BPB Structure

Name	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	"MSWIN4.1" There are many misconceptions about this field. It is only a name string. Microsoft operating systems don't pay any attention to this field. Some FAT drivers do. This is the reason that the indicated string, "MSWIN4.1", is the recommended setting, because it is the setting least likely to cause compatibility problems. If you want to put something else in here, that is your option, but the result may be that some FAT drivers might not recognize the volume. 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. If maximum compatibility is desired, only the value 512 should be used. There is a lot of FAT code in the world that is basically "hard wired" to 512 bytes per sector and doesn't bother to check this field to make sure it is 512. Microsoft operating systems will properly support 1024, 2048, and 4096, but these values are not recommended.
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. Note however, that a value should never be used that results in a "bytes per cluster" value (BPB_BytsPerSec * BPB_SecPerClus) greater than 32K (32 * 1024). There is a misconception that values greater than this are OK. Values that cause a cluster size greater than 32K bytes do not work properly; do not try to define one. Some versions of some systems allow 64K bytes per cluster value. Many application setup programs will not work correctly on such a FAT volume.
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 must not be 0. For FAT12 and FAT16 volumes, this value should never be anything other than 1. For FAT32 volumes, this value is typically 32. There is a lot of FAT code in the world "hard wired" to 1 reserved sector for FAT12 and FAT16 volumes and that doesn't bother to check this field to make sure it is 1. Microsoft operating systems will properly support any non-zero value in this field.

BPB_NumFATs	16	1	The count of FAT data structures on the volume. This field should always contain the value 2 for any FAT volume of any type. Although any value greater than or equal to 1 is perfectly valid, many software programs and a few operating systems' FAT file system drivers may not function properly if the value is something other than 2. All Microsoft file system drivers will support a value other than 2, but it is still highly recommended that no value other than 2 be used in this field. The reason the standard value for this field is 2 is to provide redundancy for the FAT data structure so that if a sector goes bad in one of the FATs, that data is not lost because it is duplicated in the other FAT. On non-disk-based media, such as FLASH memory cards, where such redundancy is a useless feature, a value of 1 may be used to save the space that a second copy of the FAT uses, but some FAT file system drivers might not recognize such a volume properly.
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	0xF8 is the standard value for "fixed" (non-removable) media. For removable media, 0xF0 is frequently used. The legal values for this field are 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, and 0xFF. The only other important point is that whatever value is put in here must also be put in the low byte of the FAT[0] entry. This dates back to the old MS-DOS 1.x media determination noted earlier and is no longer usually used for anything.
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. This field contains the "sectors per track" geometry value.
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 should always be zero on media that are not partitioned. Exactly what value is appropriate is operating system specific.
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 FAT32 volumes, this field 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).

At this point, the BPB/boot sector for FAT12 and FAT16 differs from the BPB/boot sector for FAT32. The first table shows the structure for FAT12 and FAT16 starting at offset 36 of the boot sector.

Fat12 and Fat16 Structure Starting at Offset 36

Name	Offset (byte)	Size (bytes)	Description
BS_DrvNum	36	1	Int 0x13 drive number (e.g. 0x80). This field supports MS-DOS bootstrap and is set to the INT 0x13 drive number of the media (0x00 for floppy disks, 0x80 for hard disks). **NOTE:** This field is actually operating system specific.
BS_Reserved1	37	1	Reserved (used by Windows NT). Code that formats FAT volumes should always set this byte to 0.
BS_BootSig	38	1	Extended boot signature (0x29). 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 is usually 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 should make sure 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:** Many people think that the string in this field has something to do with the determination of what type of FAT—FAT12, FAT16, or FAT32—that the volume has. This is not true. You will note from its name that this field is not actually part of the BPB. This string is informational only and is not used by Microsoft file system drivers to determine FAT typ,e because it is frequently not set correctly or is not present. See the FAT Type Determination section of this document. This string should be set based on the FAT type though, because some non-Microsoft FAT file system drivers do look at it.

Here is the structure for FAT32 starting at offset 36 of the boot sector.

FAT32 Structure Starting at Offset 36

Name	Offset (byte)	Size (bytes)	Description
BPB_FATSz32	36	4	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. This field is the FAT32 32-bit count of sectors occupied by ONE FAT. BPB_FATSz16 must be 0.
BPB_ExtFlags	40	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. 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	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. High byte is major revision number. Low byte is minor revision number. This is the version number of the FAT32 volume. This supports the ability to extend the FAT32 media type in the future without worrying about old FAT32 drivers mounting the volume. This document defines the version to 0:0. If this field is non-zero, back-level Windows versions will not mount the volume. **NOTE:** Disk utilities should respect this field and not operate on volumes with a higher major or minor version number than that for which they were designed. FAT32 file system drivers must check this field and not mount the volume if it does not contain a version number that was defined at the time the driver was written.
BPB_RootClus	44	4	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. This is set to the cluster number of the first cluster of the root directory, usually 2 but not required to be 2. **NOTE:** Disk utilities that change the location of the root directory should make every effort to place the first cluster of the root directory in the first non-bad cluster on the drive (i.e., in cluster 2, unless it's marked bad). This is specified so that disk repair utilities can easily find the root directory if this field accidentally gets zeroed.
BPB_FSInfo	48	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Sector number of FSINFO structure in the reserved area of the FAT32 volume. Usually 1. **NOTE:** There will be a copy of the FSINFO structure in BackupBoot, but only the copy pointed to by this field will be kept up to date (i.e., both the primary and backup boot record will point to the same FSINFO sector).
BPB_BkBootSec	50	2	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. If non-zero, indicates the sector number in the reserved area of the volume of a copy of the boot record. Usually 6. No value other than 6 is recommended.
BPB_Reserved	52	12	This field is only defined for FAT32 media and does not exist on FAT12 and FAT16 media. Reserved for future expansion. Code that formats FAT32 volumes should always set all of the bytes of this field to 0.
BS_DrvNum	64	1	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_Reserved1	65	1	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.

BS_BootSig	66	1	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_VolID	67	4	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_VolLab	71	11	This field has the same definition as it does for FAT12 and FAT16 media. The only difference for FAT32 media is that the field is at a different offset in the boot sector.
BS_FilSysType	82	8	Always set to the string "**FAT32** ". Please see the note for this field in the FAT12/FAT16 section earlier. This field has nothing to do with FAT type determination.

There is one other important note about Sector 0 of a FAT volume. If we consider the contents of the sector as a byte array, it must be true that sector[510] equals 0x55, and sector[511] equals 0xAA.

NOTE: Many FAT documents mistakenly say that this 0xAA55 signature occupies the "last 2 bytes of the boot sector". This statement is correct if — and only if — BPB_BytsPerSec is 512. If BPB_BytsPerSec is greater than 512, the offsets of these signature bytes do not change (although it is perfectly OK for the last two bytes at the end of the boot sector to also contain this signature).

Check your assumptions about the value in the BPB_TotSec16/32 field. Assume we have a disk or partition of size in sectors DskSz. If the BPB TotSec field (either BPB_TotSec16 or BPB_TotSec32 — whichever is non-zero) is *less than or equal to* DskSz, there is nothing whatsoever wrong with the FAT volume. In fact, it is not at all unusual to have a BPB_TotSec16/32 value that is slightly smaller than DskSz. It is also perfectly OK for the BPB_TotSec16/32 value to be considerably smaller than DskSz.

All this means is that disk space is being wasted. It does not by itself mean that the FAT volume is damaged in some way. However, if BPB_TotSec16/32 is *larger* than DskSz, the volume is seriously damaged or malformed because it extends past the end of the media or overlaps data that follows it on the disk. Treating a volume for which the BPB_TotSec16/32 value is "too large" for the media or partition as valid can lead to catastrophic data loss.

FAT Data Structure

The next data structure that is important is the FAT itself. What this data structure does is define a singly linked list of the "extents" (clusters) of a file. Note at this point that a FAT directory or file container is nothing but a regular file that has a special attribute indicating it is a directory. The only other special thing about a directory is that the data or contents of the "file" is a series of 32=byte FAT directory entries (see discussion below). In all other respects, a directory is just like a file. The FAT maps the data region of the volume by cluster number. The first data cluster is cluster 2.

The first sector of cluster 2 (the data region of the disk) is computed using the BPB fields for the volume as follows. First, we 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, so on a FAT32 volume RootDirSectors is always 0. The 32 in the above is the size of one FAT directory entry in bytes. Note also that this computation rounds *up*.

The start of the data region, the first sector of cluster 2, is computed as follows:

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

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

NOTE: This sector number is relative to the first sector of the volume that contains the BPB (the sector that contains the BPB is sector number 0). This does not necessarily map directly onto the drive, because sector 0 of the volume is not necessarily sector 0 of the drive due to partitioning.

Given any valid data cluster number **N**, the sector number of the first sector of that cluster (again relative to sector 0 of the FAT volume) is computed as follows:

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

NOTE: Because BPB_SecPerClus is restricted to powers of 2 (1,2,4,8,16,32….), this means that division and multiplication by BPB_SecPerClus can actually be performed via SHIFT operations on 2s complement architectures that are usually faster instructions than MULT and DIV instructions. On current Intel X86 processors, this is largely irrelevant though because the MULT and DIV machine instructions are heavily optimized for multiplication and division by powers of 2.

FAT Type Determination

There is considerable confusion over exactly how this works, which leads to many “off by 1”, “off by 2”, “off by 10”, and “massively off” errors. It is really quite simple how this works. The FAT type—one of FAT12, FAT16, or FAT32—is determined by the count of clusters on the volume and *nothing* else.

Please read everything in this section carefully, all of the words are important. For example, note that the statement was “count of clusters.” This is not the same thing as “maximum valid cluster number,” because the first data cluster is 2 and not 0 or 1.

To begin, let’s discuss exactly how the “count of clusters” value is determined. This is all done using the BPB fields for the volume. First, we determine the count of sectors occupied by the root directory as noted earlier.

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

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

Next, we 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);
```

Now we determine the count of clusters:

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

Please note that this computation rounds *down*.

Now we can determine the FAT type. *Please note carefully or you will commit an off-by-one error!*

In the following example, when it says <, it does not mean <=. Note also that the numbers are correct. The first number for FAT12 is 4085; the second number for FAT16 is 65525. These numbers and the '<' signs are not wrong.

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

This is the one and only way that FAT type is determined. There is no such thing as a FAT12 volume that has more than 4084 clusters. There is no such thing as a FAT16 volume that has less than 4085 clusters or more than 65,524 clusters. There is no such thing as a FAT32 volume that has less than 65,525 clusters. If you try to make a FAT volume that violates this rule, Microsoft operating systems will not handle them correctly because they will think the volume has a different type of FAT than what you think it does.

NOTE: As is noted numerous times earlier, the world is full of FAT code that is wrong. There is a lot of FAT type code that is off by 1 or 2 or 8 or 10 or 16. For this reason, it is highly recommended that if you are formatting a FAT volume which has maximum compatibility with all existing FAT code, then you should you avoid making volumes of any type that have close to 4,085 or 65,525 clusters. Stay at least 16 clusters on each side away from these cut-over cluster counts.

Note also that the CountofClusters value is exactly that—the *count* of data clusters starting at cluster 2. The maximum valid cluster number for the volume is CountofClusters + 1, and the "count of clusters including the two reserved clusters" is CountofClusters + 2.

There is one more important computation related to the FAT. Given any valid cluster number **N**, where in the FAT(s) is the entry for that cluster number? The only FAT type for which this is complex is FAT12. For FAT16 and FAT32, the computation is simple:

```
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);
```

FATOffset by BPB_BytsPerSec. ThisFATSecNum is the sector number of the FAT sector that contains the entry for cluster N in the first FAT. If you want the sector number in the second FAT, you add FATSz to ThisFATSecNum; for the third FAT, you add 2*FATSz, and so on.

You now read sector number ThisFATSecNum (remember this is a sector number relative to sector 0 of the FAT volume). Assume this is read into an 8-bit byte array named SecBuff. Also assume that the type WORD is a 16-bit unsigned and that the type DWORD is a 32-bit unsigned.

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

Fetches the contents of that cluster. To set the contents of this same cluster you do the following:

```
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 how the FAT32 code above works. A FAT32 FAT entry is actually only a 28-bit entry. The high 4 bits of a FAT32 FAT entry are reserved. The only time that the high 4 bits of FAT32 FAT entries should ever be changed is when the volume is formatted, at which time the whole 32-bit FAT entry should be zeroed, including the high 4 bits.

A bit more explanation is in order here, because this point about FAT32 FAT entries seems to cause a great deal of confusion. Basically 32-bit FAT entries are not really 32-bit values; they are only 28-bit values. For example, all of these 32-bit cluster entry values: 0x10000000, 0xF0000000, and 0x00000000 all indicate that the cluster is FREE, because you ignore the high 4 bits when you read the cluster entry value. If the 32-bit free cluster value is currently 0x30000000 and you want to mark this cluster as bad by storing the value 0x0FFFFFF7 in it. Then the 32-bit entry will contain the value 0x3FFFFFF7 when you are done, because you must preserve the high 4 bits when you write in the 0x0FFFFFF7 bad cluster mark.

Take note that because the BPB_BytsPerSec value is always divisible by 2 and 4, you never have to worry about a FAT16 or FAT32 FAT entry spanning over a sector boundary (this is not true of FAT12).

The code for FAT12 is more complicated because there are 1.5 bytes (12-bits) per FAT entry.

```
    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);
```

We now have to check for the sector boundary 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.                                                */
}
```

We now access the FAT entry as a WORD just as we do for FAT16, but if the cluster number is EVEN, we only want the low 12-bits of the 16-bits we fetch; and if the cluster number is ODD, we only want the high 12-bits of the 16-bits we fetch.

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

Fetches the contents of that cluster. To set the contents of this same cluster you do the following:

```
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.

The way the data of a file is associated with the file is as follows. In the directory entry, the cluster number of the first cluster of the file is recorded. The first cluster (extent) of the file is the data associated with this first cluster number, and the location of that data on the volume is computed from the cluster number as described earlier (computation of FirstSectorofCluster).

Note that a zero-length file—a file that has no data allocated to it—has a first cluster number of 0 placed in its directory entry. This cluster location in the FAT (see earlier computation of ThisFATSecNum and ThisFATEntOffset) contains either an EOC mark (End Of Clusterchain) or the cluster number of the next cluster of the file. The EOC value is FAT type dependant (assume FATContent is the contents of the cluster entry in the FAT being checked to see whether it is an EOC mark):

```
IsEOF = FALSE;
If(FATType == FAT12) {
    If(FATContent >= 0x0FF8)
        IsEOF = TRUE;
} else if(FATType == FAT16) {
    If(FATContent >= 0xFFF8)
        IsEOF = TRUE;
} else if (FATType == FAT32) {
    If(FATContent >= 0x0FFFFFF8)
        IsEOF = TRUE;
}
```

Note that the cluster number whose cluster entry in the FAT contains the EOC mark is allocated to the file and is also the last cluster allocated to the file. Microsoft operating system FAT drivers use the EOC value 0x0FFF for FAT12, 0xFFFF for FAT16, and 0x0FFFFFFF for FAT32 when they set the contents of a cluster to the EOC mark. There are various disk utilities for Microsoft operating systems that use a different value, however.

There is also a special “BAD CLUSTER” mark. Any cluster that contains the “BAD CLUSTER” value in its FAT entry is a cluster that should not be placed on the free list because it is prone to disk errors. The “BAD CLUSTER” value is 0x0FF7 for FAT12, 0xFFF7 for FAT16, and 0x0FFFFFF7 for FAT32. The other relevant note here is that these bad clusters are also lost clusters—clusters that appear to be allocated because they contain a non-zero value but which are not part of any files allocation chain. Disk repair utilities must recognize lost clusters that contain this special value as bad clusters and not change the content of the cluster entry.

NOTE: It is not possible for the bad cluster mark to be an allocatable cluster number on FAT12 and FAT16 volumes, but it is feasible for 0x0FFFFFF7 to be an allocatable cluster number on FAT32 volumes. To avoid possible confusion by disk utilities, no FAT32 volume should ever be configured such that 0x0FFFFFF7 is an allocatable cluster number.

The list of free clusters in the FAT is nothing more than the list of all clusters that contain the value 0 in their FAT cluster entry. Note that this value must be fetched as described earlier as for any other FAT entry that is not free. This list of free clusters is not stored anywhere on the volume; it must be computed when the volume is mounted by scanning the FAT for entries that contain the value 0. On FAT32 volumes, the BPB_FSInfo sector *may* contain a valid count of free clusters on the volume. See the documentation of the FAT32 FSInfo sector.

What are the two reserved clusters at the start of the FAT for? The first reserved cluster, 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, for FAT12 FAT[0] = 0x0FF8, for FAT16 FAT[0] = 0xFFF8, and for FAT32 FAT[0] = 0x0FFFFFF8. The second reserved cluster, FAT[1], is set by FORMAT to the EOC mark. On FAT12 volumes, it is not used and is simply always contains an EOC mark. For FAT16 and FAT32, the file system driver may use the high two bits of the FAT[1] entry for dirty volume flags (all other bits, are always left set to 1). 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, volume is “clean”.
If bit is 0, volume is “dirty”. This indicates that the file system driver did not Dismount the volume properly the last time it had the volume mounted. It would be a good idea to run a Chkdsk/Scandisk disk repair utility on it, because it may be damaged.

Bit HrdErrBitMask – If this bit is 1, no disk read/write errors were encountered.
If this bit is 0, the file system driver 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 on the volume. It would be a good idea to run a Chkdsk/Scandisk disk repair utility that does surface analysis on it to look for new bad sectors.

Here are two more important notes about the FAT region of a FAT volume:

1. 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 (see the CountofClusters computation earlier), and this entry is not necessarily at the end of the last FAT sector. FAT code should not make any assumptions about what the contents of the last FAT sector are after the CountofClusters + 1 entry. FAT format code should zero the bytes after this entry though.
2. The BPB_FATSz16 (BPB_FATSz32 for FAT32 volumes) value *may* be bigger than it needs to be. In other words, there may be totally unused FAT sectors at the end of each FAT in the FAT region of the volume. For this reason, the last sector of the FAT is always computed using the CountofClusters + 1 value, never from the BPB_FATSz16/32 value. FAT code should not make any assumptions about what the contents of these "extra" FAT sectors are. FAT format code should zero the contents of these extra FAT sectors though.

FAT Volume Initialization

At this point, the careful reader should have one very interesting question. Given that the FAT type (FAT12, FAT16, or FAT32) is dependant on the number of clusters—and that the sectors available in the data area of a FAT volume is dependant on the size of the FAT—when handed an unformatted volume that does not yet have a BPB, how do you determine all this and compute the proper values to put in BPB_SecPerClus and either BPB_FATSz16 or BPB_FATSz32? The way Microsoft operating systems do this is with a fixed value, several tables, and a clever piece of arithmetic.

Microsoft operating systems only do FAT12 on floppy disks. Because there is a limited number of floppy formats that all have a fixed size, this is done with a simple table:

```
"If it is a floppy of this type, then the BPB looks like this."
```

There is no dynamic computation for FAT12. For the FAT12 formats, all the computation for BPB_SecPerClus and BPB_FATSz16 was worked out by hand on a piece of paper and recorded in the table (being careful of course that the resultant cluster count was always less than 4085). If your media is larger than 4 MB, do not bother with FAT12. Use smaller BPB_SecPerClus values so that the volume will be FAT16.

The rest of this section is totally specific to drives that have 512 bytes per sector. You cannot use these tables, or the clever arithmetic, with drives that have a different sector size. The "fixed value" is simply a volume size that is the "FAT16 to FAT32 cutover value". Any volume size smaller than this is FAT16 and any volume of this size or larger is FAT32. For Windows, this value is 512 MB. Any FAT volume smaller than 512 MB is FAT16, and any FAT volume of 512 MB or larger is FAT32.

Please don't draw an incorrect conclusion here.

There are many FAT16 volumes out there that are larger than 512 MB. There are various ways to force the format to be FAT16 rather than the default of FAT32, and there is a great deal of code that implements different limits. All we are talking about here is the *default* cutover value for MS-DOS and Windows on volumes that have not yet been formatted. There are two tables—one is for FAT16 and the other is for FAT32. 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 */
    };
```

So given a disk size and a FAT type of FAT16 or FAT32, we now have a BPB_SecPerClus value. The only thing we have left is do is to compute how many sectors the FAT takes up so that we can set BPB_FATSz16 or BPB_FATSz32. Note that at this point we assume that BPB_RootEntCnt, BPB_RsvdSecCnt, and BPB_NumFATs are appropriately set. We also assume that DskSize is the size of the volume that we are either going to put 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 */
}
```

Do not spend too much time trying to figure out why this math works. The basis for the computation is complicated; the important point is that this is how Microsoft operating systems do it, and it works. Note, however, that this 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.

FAT32 FSInfo Sector Structure and Backup Boot Sector

On a FAT32 volume, the FAT can be a large data structure, unlike on FAT16 where it is limited to a maximum of 128K worth of sectors and FAT12 where it is limited to a maximum of 6K worth of sectors. For this reason, a provision is made to store the “last known” free cluster count on the FAT32 volume so that it does not have to be computed as soon as an API call is made to ask how much free space there is on the volume (like at the end of a directory listing). The FSInfo sector number is the value in the BPB_FSInfo field; for Microsoft operating systems it is always set to 1. Here is the structure of the FSInfo sector:

FAT32 FSInfo Sector Structure and Backup Boot Sector

Name	Offset (byte)	Size (bytes)	Description
FSI_LeadSig	0	4	Value 0x41615252. This lead signature is used to validate that this is in fact an FSInfo sector.
FSI_Reserved1	4	480	This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used.
FSI_StrucSig	484	4	Value 0x61417272. Another signature that is more localized in the sector to the location of the fields that are used.
FSI_Free_Count	488	4	Contains the last known free cluster count on the volume. If the value is 0xFFFFFFFF, then the free count is unknown and must be computed. Any other value can be used, but is not necessarily correct. It should be range checked at least to make sure it is <= volume cluster count.
FSI_Nxt_Free	492	4	This is a hint for the FAT driver. It indicates 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 cluster 2. Typically this value is set to the last cluster number that the driver allocated. If the value is 0xFFFFFFFF, then there is no hint and the driver should start looking at cluster 2. Any other value can be used, but should be checked first to make sure it is a valid cluster number for the volume.
FSI_Reserved2	496	12	This field is currently reserved for future expansion. FAT32 format code should always initialize all bytes of this field to 0. Bytes in this field must currently never be used.

FSI_TrailSig	508	4	Value 0xAA550000. This trail signature is used to validate that this is in fact an FSInfo sector. Note that the high 2 bytes of this value—which go into the bytes at offsets 510 and 511—match the signature bytes used at the same offsets in sector 0.

Another feature on FAT32 volumes that is not present on FAT16/FAT12 is the BPB_BkBootSec field. FAT16/FAT12 volumes can be totally lost if the contents of sector 0 of the volume are overwritten or sector 0 goes bad and cannot be read. This is a "single point of failure" for FAT16 and FAT12 volumes. The BPB_BkBootSec field reduces the severity of this problem for FAT32 volumes, because starting at that sector number on the volume—6—there is a backup copy of the boot sector information including the volume's BPB.

In the case where the sector 0 information has been accidentally overwritten, all a disk repair utility has to do is restore the boot sector(s) from the backup copy. In the case where sector 0 goes bad, this allows the volume to be mounted so that the user can access data before replacing the disk.

This second case—sector 0 goes bad—is the reason why no value other than 6 should ever be placed in the BPB_BkBootSec field. If sector 0 is unreadable, various operating systems are "hard wired" to check for backup boot sector(s) starting at sector 6 of the FAT32 volume. Note that starting at the BPB_BkBootSec sector is a *complete* boot record. The Microsoft FAT32 "boot sector" is actually three 512-byte sectors long. There is a copy of all three of these sectors starting at the BPB_BkBootSec sector. 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 is stored in the sector 0 BPB.

NOTE: All 3 of these sectors have the 0xAA55 signature in sector offsets 510 and 511, just like the first boot sector does (see the earlier discussion at the end of the BPB structure description).

FAT Directory Structure

This is the most simple explanation of FAT directory entries. This document totally ignores the Long File Name architecture and only talks about short directory entries. For a more complete description of FAT directory structure, see the document "FAT: Long Name On-Media Format Specification".

A FAT directory is nothing but a "file" composed of a linear list of 32-byte structures. The only special directory, which must always be present, is the root directory. For FAT12 and FAT16 media, the root directory is located in a fixed location on the disk immediately following the last FAT and is of a fixed size in sectors computed from the BPB_RootEntCnt value (see computations for RootDirSectors earlier in this document). For FAT12 and FAT16 media, the first sector of the root directory is sector number relative to the first sector of the FAT volume:

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

For FAT32, the root directory can be of variable size and is a cluster chain, just like any other directory is. The first cluster of the root directory on a FAT32 volume is stored in BPB_RootClus. Unlike other directories, the root directory itself on any FAT type does not have any date or time stamps, does not have a file name (other than the implied file name "\"), and does not contain "." and ".." files as the first two directory entries in the directory. The only other special aspect of the root directory is that it is the only directory on the FAT volume for which it is valid to have a file that has only the ATTR_VOLUME_ID attribute bit set (see below).

FAT 32 Byte Directory Entry Structure

Name	Offset (byte)	Size (bytes)	Description
DIR_Name	0	11	Short name.
DIR_Attr	11	1	File attributes: ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID The upper two bits of the attribute byte are reserved and should always be set to 0 when a file is created and never modified or looked at after that.
DIR_NTRes	12	1	Reserved for use by Windows NT. Set value to 0 when a file is created and never modify or look at it after that.
DIR_CrtTimeTenth	13	1	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second. The granularity of the seconds part of DIR_CrtTime is 2 seconds so this field is a count of tenths of a second and its valid value range is 0-199 inclusive.
DIR_CrtTime	14	2	Time file was created.
DIR_CrtDate	16	2	Date file was created.
DIR_LstAccDate	18	2	Last access date. Note that there is no last access time, only a date. This is the date of last read or write. In the case of a write, this should be set to the same date as DIR_WrtDate.
DIR_FstClusHI	20	2	High word of this entry's first cluster number (always 0 for a FAT12 or FAT16 volume).
DIR_WrtTime	22	2	Time of last write. Note that file creation is considered a write.
DIR_WrtDate	24	2	Date of last write. Note that file creation is considered a write.
DIR_FstClusLO	26	2	Low word of this entry's first cluster number.
DIR_FileSize	28	4	32-bit DWORD holding this file's size in bytes.

DIR_Name[0]

Special notes about the first byte (DIR_Name[0]) of a FAT directory entry:

- If DIR_Name[0] == 0xE5, then the directory entry is free (there is no file or directory name in this entry).

- If DIR_Name[0] == 0x00, then the directory entry is free (same as for 0xE5), and there are no allocated directory entries after this one (all of the DIR_Name[0] bytes in all of the entries after this one are also set to 0).

  The special 0 value, rather than the 0xE5 value, indicates to FAT file system driver code that the rest of the entries in this directory do not need to be examined because they are all free.

- If DIR_Name[0] == 0x05, then the actual file name character for this byte is 0xE5. 0xE5 is actually a valid KANJI lead byte value for the character set used in Japan. The special 0x05 value is used so that this special file name case for Japan can be handled properly and not cause FAT file system code to think that the entry is free.

The DIR_Name field is actually broken into two parts+ the 8-character main part of the name, and the 3-character extension. These two parts are “trailing space padded” with bytes of 0x20.

DIR_Name[0] may not equal 0x20. There is an implied ‘.’ character between the main part of the name and the extension part of the name that is not present in DIR_Name. Lower case characters are not allowed in DIR_Name (what these characters are is country specific).

The following characters are not legal in any bytes of DIR_Name:

- Values less than 0x20 except for the special case of 0x05 in DIR_Name[0] described above.
- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, and 0x7C.

Here are some examples of how a user-entered name maps into DIR_Name:

```
“foo.bar”            -> “FOO     BAR”
“FOO.BAR”            -> “FOO     BAR”
“Foo.Bar”            -> “FOO     BAR”
“foo”                -> “FOO        “
“foo.”               -> “FOO        “
“PICKLE.A”           -> “PICKLE  A  “
“prettybg.big”       -> “PRETTYBGBIG”
“.big”               -> illegal, DIR_Name[0] cannot be 0x20
```

In FAT directories all names are unique. Look at the first three examples earlier. Those different names all refer to the same file, and there can only be one file with DIR_Name set to “FOO   BAR” in any directory.

DIR_Attr specifies attributes of the file:

ATTR_READ_ONLY	Indicates that writes to the file should fail.
ATTR_HIDDEN	Indicates that normal directory listings should not show this file.
ATTR_SYSTEM	Indicates that this is an operating system file.
ATTR_VOLUME_ID	There should only be one “file” on the volume that has this attribute set, and that file must be in the root directory. This name of this file is actually the label for the volume. DIR_FstClusHI and DIR_FstClusLO must always be 0 for the volume label (no data clusters are allocated to the volume label file).
ATTR_DIRECTORY	Indicates that this file is actually a container for other files.
ATTR_ARCHIVE	This attribute supports backup utilities. This bit is set by the FAT file system driver when a file is created, renamed, or written to. Backup utilities may use this attribute to indicate which files on the volume have been modified since the last time that a backup was performed.

Note that the ATTR_LONG_NAME attribute bit combination indicates that the “file” is actually part of the long name entry for some other file. See the FAT Long Filename specification for more information on this attribute combination.

When a directory is created, a file with the ATTR_DIRECTORY bit set in its DIR_Attr field, you set its DIR_FileSize to 0. DIR_FileSize is not used and is always 0 on a file with the ATTR_DIRECTORY attribute (directories are sized by simply following their cluster chains to the EOC mark). One cluster is allocated to the directory (unless it is the root directory on a FAT16/FAT12 volume), and you set DIR_FstClusLO and DIR_FstClusHI to that cluster number and place an EOC mark in that clusters entry in the FAT. Next, you initialize all bytes of that cluster to 0. If the directory is the root directory, you are done (there are no dot or *dotdot* entries in the root directory). If the directory is not the root directory, you need to create two special entries in the first two 32-byte

directory entries of the directory (the first two 32 byte entries in the data region of the cluster you just allocated).

The first directory entry has DIR_Name set to:
“. ”

The second has DIR_Name set to:
“.. ”

These are called the *dot* and *dotdot* entries. The DIR_FileSize field on both entries is set to 0, and all of the date and time fields in both of these entries are set to the same values as they were in the directory entry for the directory that you just created. You now set DIR_FstClusLO and DIR_FstClusHI for the *dot* entry (the first entry) to the same values you put in those fields for the directories directory entry (the cluster number of the cluster that contains the *dot* and *dotdot* entries).

Finally, you set DIR_FstClusLO and DIR_FstClusHI for the *dotdot* entry (the second entry) to the first cluster number of the directory in which you just created the directory (value is 0 if this directory is the root directory even for FAT32 volumes).

Here is the summary for the *dot* and *dotdot* entries:
- The *dot* entry is a directory that points to itself.
- The *dotdot* entry points to the starting cluster of the parent of this directory (which is 0 if this directories parent is the root directory).

Date and Time Formats

Many FAT file systems do not support Date/Time other than DIR_WrtTime and DIR_WrtDate. For this reason, DIR_CrtTimeMil, DIR_CrtTime, DIR_CrtDate, and DIR_LstAccDate are actually optional fields. DIR_WrtTime and DIR_WrtDate *must* be supported, however. If the other date and time fields are not supported, they should be set to 0 on file create and ignored on other file operations.

Date Format. A FAT directory entry date stamp is a 16-bit field that is basically a date relative to the MS-DOS epoch of 01/01/1980. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word):

Bits 0–4: Day of month, valid value range 1-31 inclusive.
Bits 5–8: Month of year, 1 = January, valid value range 1–12 inclusive.
Bits 9–15: Count of years from 1980, valid value range 0–127 inclusive (1980–2107).

Time Format. A FAT directory entry time stamp is a 16-bit field that has a granularity of 2 seconds. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word).

Bits 0–4: 2-second count, valid value range 0–29 inclusive (0 – 58 seconds).
Bits 5–10: Minutes, valid value range 0–59 inclusive.
Bits 11–15: Hours, valid value range 0–23 inclusive.

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

Other Notes Relating to FAT Directories

- Long File Name directory entries are identical on all FAT types. See the FAT Long File Name Specification for details.

- DIR_FileSize is a 32-bit field. For FAT32 volumes, your FAT file system driver must not allow a cluster chain to be created that is longer than 0x100000000 bytes, and the last byte of the last cluster in a chain that long cannot be allocated to the file. This must be done so that no file has a file size > 0xFFFFFFFF bytes. This is a fundamental limit of all FAT file systems. The maximum allowed file size on a FAT volume is 0xFFFFFFFF (4,294,967,295) bytes.

- Similarly, a FAT file system driver must not allow a directory (a file that is actually a container for other files) to be larger than 65,536 * 32 (2,097,152) bytes.

  NOTE: This limit does *not* apply to the number of files in the directory. This limit is on the size of the directory itself and has nothing to do with the content of the directory. There are two reasons for this limit:

  1. Because FAT directories are not sorted or indexed, it is a bad idea to create huge directories; otherwise, operations like creating a new entry (which requires every allocated directory entry to be checked to verify that the name doesn't already exist in the directory) become very slow.
  2. There are many FAT file system drivers and disk utilities, including Microsoft's, that expect to be able to count the entries in a directory using a 16-bit WORD variable. For this reason, directories cannot have more than 16-bits worth of entries.

Specification Compliance

Compliance with this specification is defined by testing on the FAT reference operating system(s). The reference operating systems for FAT are Microsoft Windows 98 and Microsoft Windows 2000 (based on NT Technology).

Your FAT volume is in compliance with this specification if and only if both of the reference operating systems will mount the volume, check it for errors using the operating system supplied disk tools (Chkdsk.exe for Windows 2000 and Scandisk.exe for Windows 98) and fail to find any errors. The basic procedure is to manufacture a FAT volume using your system and tools and then move the disk, or disk media for a removable drive, to a computer running the reference operating systems and test it.