DMA Controller

DMA is a device which can acquire complete control of the buses and hence can be used to transfer data directly from port to memory or vice versa. Transferring data like this can prove faster because a transfer will consume 2 bus cycles if it is performed using the processor. So in this approach the processor is bypasses and its cycles are stolen and are used by the DMA controller.

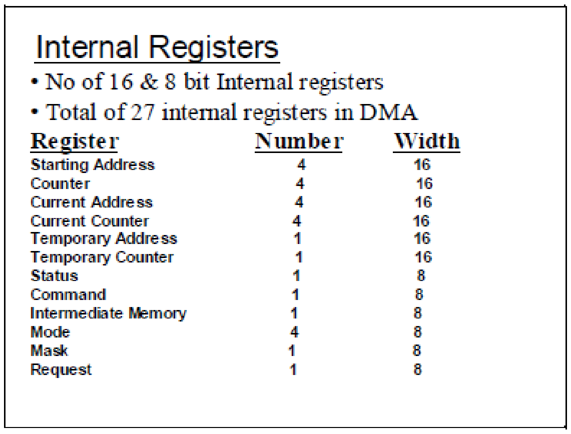
Direct Memory Access (DMA)

The latch B of the DMA interface is used to hold the higher 4 or 8 bits of the 20 or 24 bit absolute address respectively. The lower 16bits are loaded in the base address register and the number of bytes to be loaded are placed in the count register. The DMA requests to acquire buses through the HOLD signal, it receives a HLDA (Hold Acknowledge) signal if no higher priority signal is available. On acknowledgment the DMA acquires control of the buses and can issue signals for read and write operations to memory and I/O ports simultaneously. The DREQ signals are used by various devices to request a DMA operation. And if the DMA controller is successful in acquiring the bus it sends back the DACK signal to signify that the request is being serviced. For the request to be serviced properly the DMA channel must the programmed accurately before the request.

A single DMA can transfer 8bit operands to and from memory in a single a bus cycle. If 16bit values are to be transmitted then two DMA controllers are required and should be Cascaded

DMA Programming Model

• DMA has 4 – Channels
• Each Channel can be programmed to transfer a block of maximum size of 64k.
• For each Channel there is a
• **Base Register**
• **Count Register**
• **Higher Address Nibble/Byte is placed in Latch B.**
• The Mode register is conveyed which Channel is to be programmed and for what purpose i.e. Read Cycle, Write Cycle, Memory to memory transfer.
• A request to DMA is made to start it's transfer.



Internal Registers

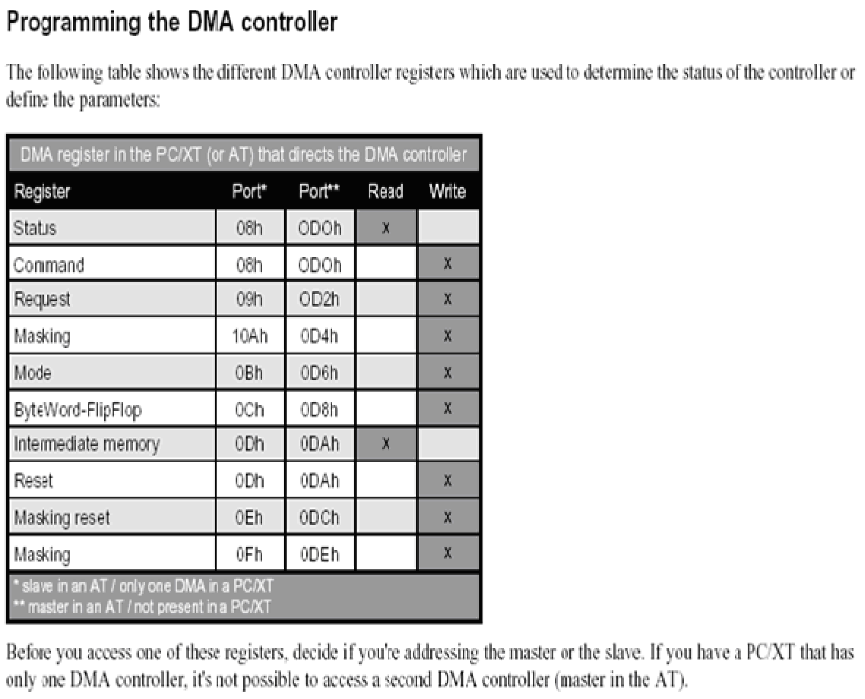
• No of 16 & 8 bit Internal registers
• Total of 27 internal registers in DMA

Register	Number	Width
Starting Address	4	16
Counter	4	16
Current Address	4	16
Current Counter	4	16
Temporary Address	1	16
Temporary Counter	1	16
Status	1	8
Command	1	8
Intermediate Memory	1	8
Mode	4	8
Mask	1	8
Request	1	8

DMA Modes

- Block Transfer
- Single Transfer
- Demand Transfer

In block transfer mode the DMA is
programmed to transfer a block and does not pause or halt until the whole block is transferred irrespective of the requests received meanwhile.
In Single transfer mode the DMA transfers a single byte on each request and updates the counter registers on each transfer and the registers need not be programmed again. On the next request the DMA will again transfer a single byte beginning from the location it last ended.
Demand transfer is same as block transfer, only difference is that the DREQ signal remains active throughout the transfer and as soon as the signal deactivates the transfer stops and on reactivation of the DREQ signal the transfer may start from the point it left.



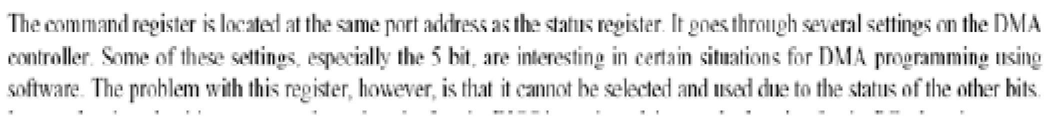
Programming the DMA controller

The following table shows the different DMA controller registers which are used to determine the status of the controller or define the parameters:

DMA register in the PC/XT (or AT) that directs the DMA controller				
Register	Port*	Port**	Read	Write
Status	08h	0D0h	x	
Command	08h	0D0h		x
Request	09h	0D2h		x
Masking	10Ah	0D4h		x
Mode	0Bh	0D6h		x
ByteWord-FlipFlop	0Ch	0D8h		x
Intermediate memory	0Dh	0DAh	x	
Reset	0Dh	0DAh		x
Masking reset	0Eh	0DCh		x
Masking	0Fh	0DEh		x

* slave in an AT / only one DMA in a PC/XT
** master in an AT / not present in a PC/XT

Before you access one of these registers, decide if you're addressing the master or the slave. If you have a PC/XT that has only one DMA controller, it's not possible to access a second DMA controller (master in the AT).



The command register is located at the same port address as the status register. It goes through several settings on the DMA controller. Some of these settings, especially the 5 bit, are interesting in certain situations for DMA programming using software. The problem with this register, however, is that it cannot be selected and used due to the status of the other bits.

Terminal count if reached signifies that the whole of the block as requested through some DMA channel has been transferred.

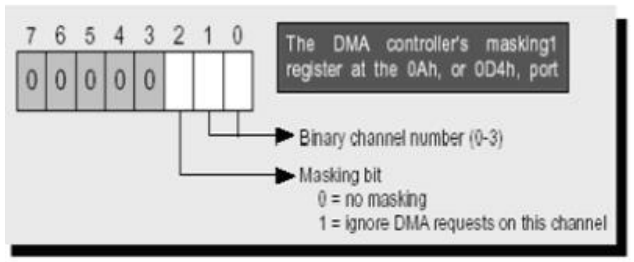
This is the command register. It is used to program various common parameters of transfer for all the channels.

DMA Request Register

The request register is used to initiate a DMA transfer under software control. This is done by simulating the activation or clearing of one of the DREQx lines The request register is also used to initiate a memory to memory transfer, since a peripheral device is not involved and therefore cannot send a signal over a DREQx line.

This register can be used to simulate a DMA request through software (in case of memory to memory transfer). The lower 2 bits contains the channel number to be requested and the bit # 2 is set to indicate a request.

DMA Mask–1 Register


7 6 5 4 3 2 1 0
0 0 0 0 0
The DMA controller's masking1 register at the 0Ah, or 0D4h, port
Binary channel number (0-3)
Masking bit
0 = no masking
1 = ignore DMA requests on this channel

Another way to mask or make a channel receptive to DMA requests is provided by mask register 2. In contrast to the mask register 1, all four channels are affected. Use this register only to change the status for all four channels simultaneously.

As its name suggests, the mode register determines a channel's operating mode. You can specify if the next DMA transfer will happen as a single transfer, a block transfer, or a demand transfer. It also specifies if the channel is to cascade two DMA controllers. In most cases you won't have to change this later setting since this happens when the computer is booted.

This register can be used to mask/unmask requests from a device for a certain DMA channel. The lower 2 bits contains the channel number and the bit #2 is set if the channel is to be masked.

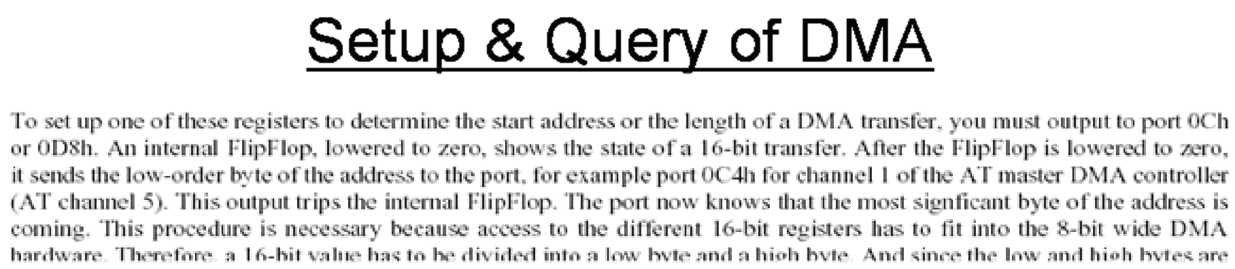
DMA Mask–2 Register

This register can also be used to mask the DMA channels. It contains a single bit for each channel. The corresponding bit is set to mask the requests for that channel.

DMA Mode Register

Bit 5 of the mode register, determine the "direction" of a transfer. This "direction" isn't to or from a peripheral, rather it's forward or backward direction in memory. So you can decrement instead of increment the memory address during a DMA transfer. In his case, a data block is read backwards to forwards by the peripheral. Also the ending address of the buffer is loaded into the proper register before starting the transfer.

This register can be used to set the mode on each. The slide shows the detail of the values and bits which should be placed in the register in order to program a required mode.



Setup & Query of DMA

To set up one of these registers to determine the start address or the length of a DMA transfer, you must output to port 0Ch or 0D8h. An internal FlipFlop, lowered to zero, shows the state of a 16-bit transfer. After the FlipFlop is lowered to zero, it sends the low-order byte of the address to the port, for example port 0C4h for channel 1 of the AT master DMA controller (AT channel 5). This output trips the internal FlipFlop. The port now knows that the most signficant byte of the address is coming. This procedure is necessary because access to the different 16-bit registers has to fit into the 8-bit wide DMA hardware. Therefore, a 16-bit value has to be divided into a low byte and a high byte. And since the low and high bytes are

A channel is programmed for a start address and the count of bytes to be transferred before the transfer can take place. Both these values are placed in various registers according to the channel number as shown by the slide above. Once the transfer starts these values start changing. The start address is updated in to the current address and the count is also updates as bytes are transferred. During the transfer the status of the transfer can be analyzed by getting the values of these registers listed In the slide above for the channel(s) involved in the transfer.

File System

File System

•Disk Architecture
•Disk Partitioning
•File systems

Disk Architecture

•Disk is a circular which is hollow from the center
•This shape is inherently useful for random access.

Tracks are the circular division of the disk and the sectors are the longitudinal division of the disk

Addressable unit Parameters

• Heads
•Sectors
•Tracks

An addressable unit on disk can be addressed by three parameters i.e. head #, sector # and track #. The disk rotates and changing sectors and a head can move to and fro changing tracks. Each addressable unit has a unique combination of sec#, head# and track# as its physical address.

Blocks

• Blocks are the sectors per track
•Smallest addressable unit in memory
•Address of block is specified as a unique combination of three parameters (i.e. track, head, sec)

Density of Magnetic media

• Density of magnetic media is the determinant of

the amount of data that can reside stably on the disk for example floppy disk come with different densities.
•Double Density
•High Density

Effect of surface area on disk size

• Increasing the surface area clearly increases the amount of data that can reside on the disk as more magnetic media no resides on disk but it might have some drawbacks like increased seek time in case only one disk platter is being used

Effect of surface area on disk size

• Increasing the surface area clearly increases the amount of data that can reside on the disk as more magnetic media no resides on disk but it might have some drawbacks like increased seek time in case only one disk platter is being used

Cylinders

• In case of hard disk where there are number of platters the term track is replaced by cylinder
•Cylinder is a collection of corresponding tracks if track on platter changes so will the tracks on rest of the platters as all the heads move simultaneously

Rotational Delay

• While accessing a selected block the time required by the disk to rotate to the specified sector is called rotational delay

Seek Time

• While accessing a selected block Time required by the head to reach the particular track/cylinder is called seek time

Access Time

• The accumulative time that is required to access the selected block is called access time
•Access time includes the delay required by disk rotation as well as head movement.

Head is like Electric Coil

• Disk follow the basic principle of magnetism of dynamo.
•When ever a magnetized portion of disk runs along the coil like head electricity is produced in the head which is interpreted as a logic 1
•And whenever a demagnetized portion on the

disk runs through the head no electricity is
produced in head which is interpreted as logic 0

Head position and precautions

• The head is touching the surface of floppy disk which rotates at a low speed of 300 RPM
•The head is not touching the surface of hard disk which run at high speeds up to 9600 RPM but is at a few microns distance from the surface
•All the magnetic disk are made out of magnetic media and hence data may be lost by exposing them to sunlight, heat, radiation, magnetic or electric fields.
•Dust is harmful and even fatal in case of head disk by the virtue of its speed and its distance of head from the surface

Hard Disk

Reading/Writing a physical Block

• biosdisk(int cmd, int drive, int head, int track, int sector, int nsects, void * buffer);
Cmd 0 = disk reset
1 = disk status (status of last disk operation)
2 = disk read
3 = disk write
4 = disk verify
5 = disk format

The function
biosdisk() can be used to read or write a physical block. The slide shows its parameter. It takes the command (cmd), drive number, head number, track number, number of sectors to be read or written and the memory from where the data is to read from or written to. Command signifies the operation that is to be performed.

Reading Writing a physical Block

• Drive 0x80 = first fixed disk (physcial drive)
0x81 = second fixed disk
0x82 = third fixed disks
…. ….
0x00 = first removable disk
0x01 = second removable disk
…. ….
Drive number is described in the slide below it starts from 0 for first removable disk and starts from 0x80 for first fixed disk.

Limitation of biosdisk

• Biosdisk() calls int 13H/0/1/2/3/4/5
•Details of 13H services used by Biosdisk()
•On Entry
AH = service #

AL=No. of sectors
BX = offset address of data buffer
CH = track #
CL = sector #
DH = head/side #
DL = Drive #
ES = Segment Address of buffer.
However there are some limitation of this biosdisk() while using large disks. This function uses the int 13H services listed in the slide above.

Limitation of biosdisk()

• Large sized disk are available now with thousands of tracks
•But this BIOS routine only is capable of accessing a max. of 1024 tracks.
•Hence if a large disk is being used not whole of the disk can be accessed using this routine.
The parameter sizes provided by these services may not be sufficient to hold the track number of block to be accessed.

Extended BIOS functions

• Extended BIOS functions of int 13h can be used for operations on higher tracks
•As discussed later usual BIOS functions can access a maximum of 504MB of disk approx.

Highest biosdisk() capacity

• Hence the highest capacity of disk can be accessed using bios functions is
•63x16x1024x512= 504 MB approx.
But IDE disk interface can support disks with memory space larger than 504MB

Highest IDE capacity

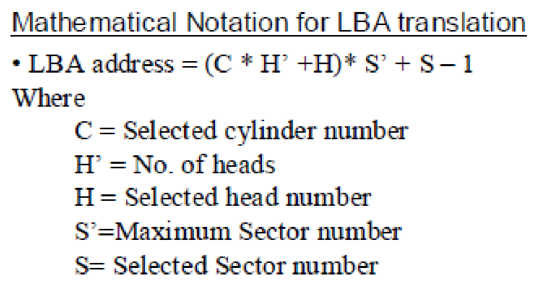
• Hence highest physical capacity of the disk according to the IDE interface is 255x16x65536x512 = 127GB
•Extended BIOS functions allow to access disk with sizes greater than 504 MB through LBA translation.
Extended services require that the address of the block is specified as a LBA address.

LBA Translation Method

• Each unique combination of three parameters is assigned a unique index as shown below
•Firstly all the sectors of a cylinder are indexed for head=0, then when whole track has been indexed the sector in the track of same cylinder with head =1 are indexed and so on up till the end of all heads
When done with one cylinder the same is repeated for the next cylinder till the end of cylinders

LBA translation is done by numbering the blocks with a single index. The indexes are assigned to blocks as shown in the slide below. In terms of the disk geometry firstly all the sectors of a track will be indexed sequentially, then the track exhausts the next track is chosen on the other side of the disk and so on all the tracks in a cylinder are indexed. When all the blocks within a cylinder has been indexed the same is done with the next cylinder.

if the CHS (cylinder, head , sector) address of a disk is known it can be translated in to the LBA address and vice versa. For this purpose the total number of cylinders, heads and sectors must also be known.



Mathematical Notation for LBA translation

- LBA address = $(C * H' + H) * S' + S - 1$

Where

C = Selected cylinder number

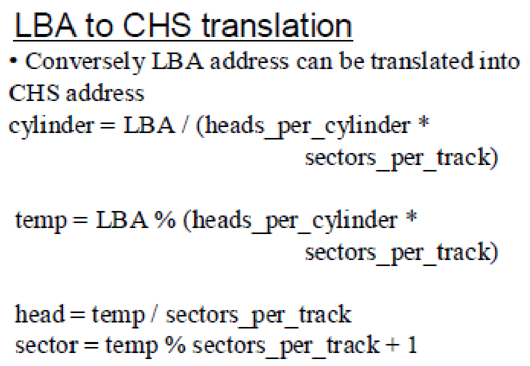
H' = No. of heads

H = Selected head number

S'=Maximum Sector number

S= Selected Sector number

Also conversely LBA to CHS translation can also be done using the formulae discussed in the following slide but for this the total number of cylinders, heads and sectors within the disk geometry should be known



LBA to CHS translation

- Conversely LBA address can be translated into CHS address

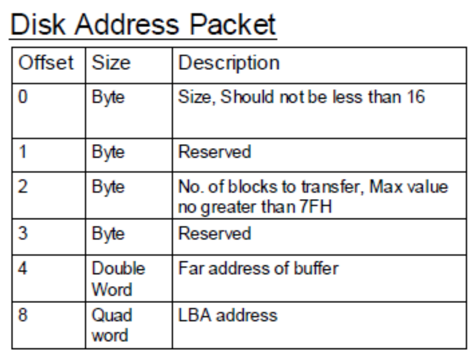
cylinder = LBA / (heads_per_cylinder * sectors_per_track)

temp = LBA % (heads_per_cylinder * sectors_per_track)

head = temp / sectors_per_track

sector = temp % sectors_per_track + 1

Disk Address Packet is a data structure used by extended int 13H services to address a block and other information for accessing the block. Its structure is defined in the slide below.



Disk Address Packet

Offset	Size	Description
0	Byte	Size, Should not be less than 16
1	Byte	Reserved
2	Byte	No. of blocks to transfer, Max value no greater than 7FH
3	Byte	Reserved
4	Double Word	Far address of buffer
8	Quad word	LBA address

Hard Disk, Partition Table

Extended Read

• Service used for extended read is int 13h/42h
On Entry
AH=42H
DL=drive #
DS:SI= far address of Disk address packet
On Exit
If CF=0
AH=0= Success
If CF=1
AH= Error code

Interrupt 13H/42H can be used to read a LBA addressed block whose LBA address is placed in the Disk Address packet as described in the slide above.

Extended Write

• Service used for extended write is int 13h/43h
On Entry
AH=43H
AL=0,1 write with verify off
2 write with verify on
DL=drive #
DS:SI= far address of Disk address packet
On Exit
If CF=0
AH=0= Success
If CF=1
AH= Error code
Similarly int 13H / 43H can be used to write onto to LBA addressed block as described in the slide above.

The above slides list a program that that performs a block read operation using the interrupt 13H/42H. A structure of type DAP is create an appropriate values are placed into it which includes its LBA address. The offset address of dap is placed in SI register and the DS already contains its segment address as it has been declared a global variable. The drive number is also specified and the interrupt is invoked. The interrupt service reads the contents of the block and places it in a buffer whose address was specified in dap.

Disk Partitioning

• Partition Table contains information pertaining to disk partitions.
• Partition Table is the first physical sector
Head = 0
Track/Cylinder = 0
Sec = 1 or LBA = 0
• Partition Table at CHS = 001 is also called MBR (Master Boot Record).

Structure of Partitioning Table

• Total size of Partition Table is 512 bytes.

• First 446 bytes contains code which loads the boot block of active partition and is executed at Boot Time.
• Rest of the 66 bytes is the Data part.
• Last two bytes of the Data part is the Partition table signature.

File System for Each O.S.

• On a single disk there can be 4 different file systems and hence 4 different O.S.
• Each O.S. will have its individual partition on disk.
• Data related to each partition is stored in a 16-bytes chunk within the Data Part of Partition Table.

Structure of Data Part of P.T.

The data part can contain information about four different partitions for different Operating systems. Each partition information chunk is 16 bytes long and the last two bytes at the end of the partition table data part is the partition table signature whose value should be AA55 indicating that the code part contains valid executable code.

Primary Partition

• Partition defined in the MBR (Master Boot Record) are primary partition.
• Each Primary Partition contains information about its respective O.S.
• However if only one O.S. is to be installed then extended partitions.

Extended Partitions

However if a single operating system is to be kept for instance, then the disk can be divided into primary and extended partitions. Information about primary and extended partition is kept in the first physical block. The extended partition may again be divided into a number of partitions, information about further partitions will be kept in extended partition table which will be the first physical block within extended partition (i.e. it will not the first block of primary partition.). Moreover there can be extended partitions within extended partitions and such that in then end there are number of logical partitions this can go on till the last drive number in DOS.

Partition Table II

Here it can be seen that the first partition table maintains information about the primary and extended partitions. The second partition table similarly stores information about a logical and a extended partition within the previous extended partition. Similarly for each such extended partition there will be a partition table that stores information about the logical partition and may also contain information about any further extended partition. In this way the partition tables form a chain as depicted in the slide below. The last partition table within the chain contains just a single entry signifying the logical drive.

Reading Extended Partition

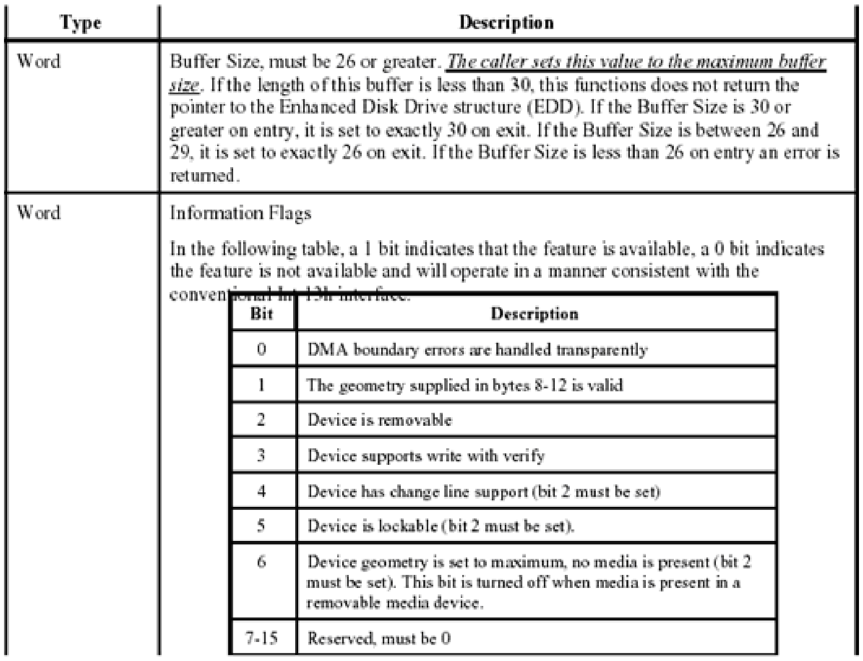
Above is a listing of a simple program that reads the partition table using the extended 13H services. It then displays the contents of the data part of the partition table read. For this purpose it uses various data structures designed in reflection of the partition table and 16 bytes data entries within. The program uses recursion and calls the getpart() function recursively whenever it finds an extended partition to read the data within the extended partition table.

Get Drive Parameters

On Entry:
AH – 48
DL – Drive number
DS:SI – Address of result buffer
On Exit:
Carry Clear
AH – 0
DS:SI – result buffer
Carry Set
AH – Error Code

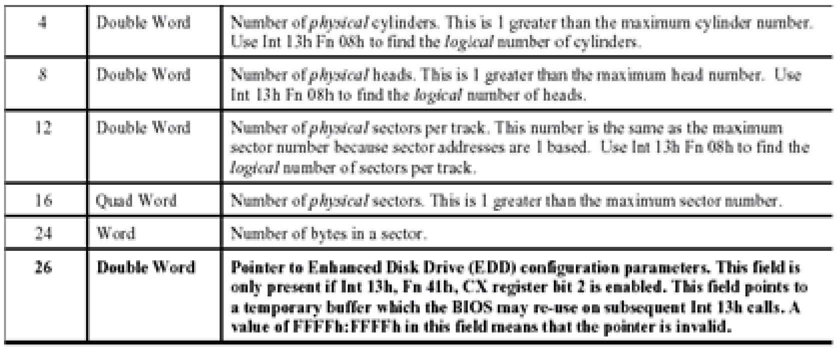
The partition table data entry also stores the CHS address of the starting block. But this address is left insignificant if a LBA enable disk is in question. However LBA address can be used in place of the CHS address, and in case CHS address is required it can be calculated if the total number of tracks, sectors and heads are known. To get the total number of tracks, sectors and head the above described service can be used.

File System Data Structures (LSN, BPB)



Type	Description
Word	Buffer Size, must be 26 or greater. *The caller sets this value to the maximum buffer size*. If the length of this buffer is less than 30, this functions does not return the pointer to the Enhanced Disk Drive structure (EDD). If the Buffer Size is 30 or greater on entry, it is set to exactly 30 on exit. If the Buffer Size is between 26 and 29, it is set to exactly 26 on exit. If the Buffer Size is less than 26 on entry an error is returned.
Word	Information Flags In the following table, a 1 bit indicates that the feature is available, a 0 bit indicates the feature is not available and will operate in a manner consistent with the conventional Int 13h interface.

Bit	Description
0	DMA boundary errors are handled transparently
1	The geometry supplied in bytes 8-12 is valid
2	Device is removable
3	Device supports write with verify
4	Device has change line support (bit 2 must be set)
5	Device is lockable (bit 2 must be set).
6	Device geometry is set to maximum, no media is present (bit 2 must be set). This bit is turned off when media is present in a removable media device.
7-15	Reserved, must be 0



4	Double Word	Number of *physical* cylinders. This is 1 greater than the maximum cylinder number. Use Int 13h Fn 08h to find the *logical* number of cylinders.
8	Double Word	Number of *physical* heads. This is 1 greater than the maximum head number. Use Int 13h Fn 08h to find the *logical* number of heads.
12	Double Word	Number of *physical* sectors per track. This number is the same as the maximum sector number because sector addresses are 1 based. Use Int 13h Fn 08h to find the *logical* number of sectors per track.
16	Quad Word	Number of *physical* sectors. This is 1 greater than the maximum sector number.
24	Word	Number of bytes in a sector.
26	**Double Word**	**Pointer to Enhanced Disk Drive (EDD) configuration parameters. This field is only present if Int 13h, Fn 41h, CX register bit 2 is enabled. This field points to a temporary buffer which the BIOS may re-use on subsequent Int 13h calls. A value of FFFFh:FFFFh in this field means that the pointer is invalid.**

LSN (Logical Sector Number)

Boot Block has LSN = 0
• If the blocks are indexed from the boot block such that the boot block has index = 0,Then this index is called LSN.
• LSN is relative index from the start of logical drive, not the physical drive.

For fixed disk
Hidden Blocks =
No. of Sec/Track

LSN is also indexed like LBA the only difference is that LBA is the address relative to the start of physical drive (i.e. absolute), whereas LSN address is the address from the start of logical partition i.e relative.

As in the above example it can be noticed that the LBA = 0 is not the same as LSN=0. The LBA=0 block is the first block on disk. Whereas each logical partition has LSN=0 block which is the first block in logical drive and is not necessarily the first block on physical drive. Also notice the hidden blocks between the first physical block on each partition and its first LSN block. These hidden blocks are not used by the operating system for storing any kind of data.

Conclusion

• LBA is physical or absolute address.
• LSN is relative address with respect to the start of Logical Drive.

File System Data Structures

• BIOS Parameter Block (BPB)
• Drive Parameter Block (DPB)
• File Control Block (FCB)
• FAT 12, FAT 16, FAT 32
• Master File Table (MFT)

To understand the file systems of DOS and Windows the above given data structure should be understood which are used by the operating system for file management.

Anatomy of a FAT based file system

Starting block(s) is /are
the boot block(s), immediately after which the FAT (File allocation table) starts. A typical volume will contain two copies of FAT. After FAT the root directory is situated which contain information about the files and folders in the root directory. Whole of this area constitutes the systems area rest of the area is used to store user data and folders

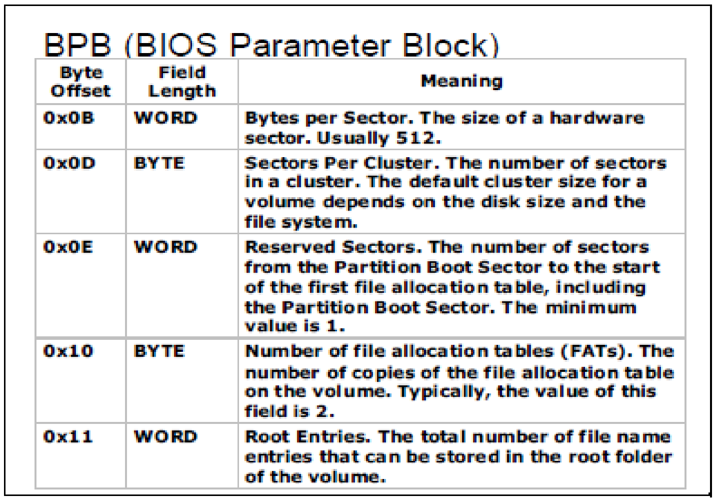
Clusters

• A cluster is a collection of contiguous blocks.
• User Data is divided into clusters
• Number of blocks within a cluster is in power of 2.
• Cluster size can vary depending upon the size of the disk.
• DOS has a built in limit of 128 blocks per cluster.
• But practically limit of 64 blocks per cluster has been established.
• We will learn more about the size of clusters, later.

BPB (BIOS Parameter Block)

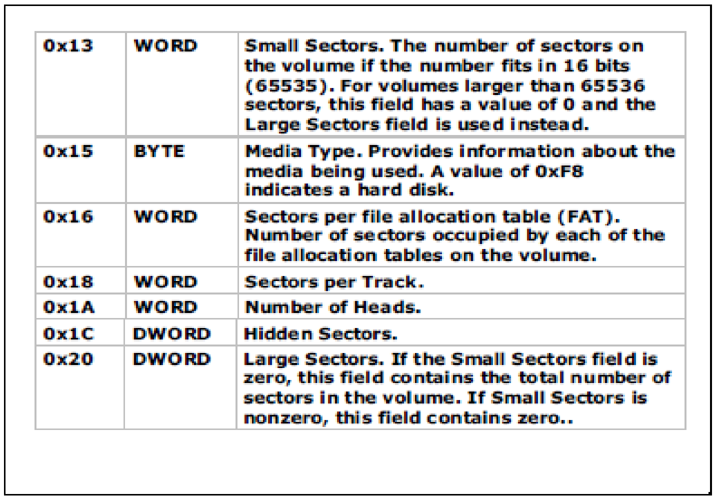
• Situated within the Boot Block.
• Contains vital information about the file system.

BIOS parameter block is a data structure maintained by DOS in the boot block for each drive. The boot block is typically a 512 byte block which as seen the previous slides is the first logical block i.e. LSN = 0. It contains some code and data. The data part constitutes the BPB.

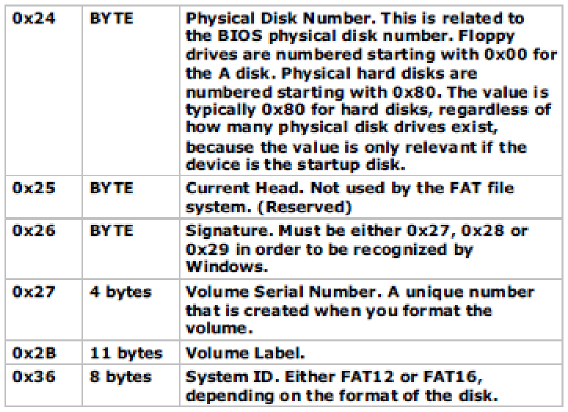


BPB (BIOS Parameter Block)

Byte Offset	Field Length	Meaning
0x0B	WORD	Bytes per Sector. The size of a hardware sector. Usually 512.
0x0D	BYTE	Sectors Per Cluster. The number of sectors in a cluster. The default cluster size for a volume depends on the disk size and the file system.
0x0E	WORD	Reserved Sectors. The number of sectors from the Partition Boot Sector to the start of the first file allocation table, including the Partition Boot Sector. The minimum value is 1.
0x10	BYTE	Number of file allocation tables (FATs). The number of copies of the file allocation table on the volume. Typically, the value of this field is 2.
0x11	WORD	Root Entries. The total number of file name entries that can be stored in the root folder of the volume.



0x13	WORD	Small Sectors. The number of sectors on the volume if the number fits in 16 bits (65535). For volumes larger than 65536 sectors, this field has a value of 0 and the Large Sectors field is used instead.
0x15	BYTE	Media Type. Provides information about the media being used. A value of 0xF8 indicates a hard disk.
0x16	WORD	Sectors per file allocation table (FAT). Number of sectors occupied by each of the file allocation tables on the volume.
0x18	WORD	Sectors per Track.
0x1A	WORD	Number of Heads.
0x1C	DWORD	Hidden Sectors.
0x20	DWORD	Large Sectors. If the Small Sectors field is zero, this field contains the total number of sectors in the volume. If Small Sectors is nonzero, this field contains zero..



0x24	BYTE	Physical Disk Number. This is related to the BIOS physical disk number. Floppy drives are numbered starting with 0x00 for the A disk. Physical hard disks are numbered starting with 0x80. The value is typically 0x80 for hard disks, regardless of how many physical disk drives exist, because the value is only relevant if the device is the startup disk.
0x25	BYTE	Current Head. Not used by the FAT file system. (Reserved)
0x26	BYTE	Signature. Must be either 0x27, 0x28 or 0x29 in order to be recognized by Windows.
0x27	4 bytes	Volume Serial Number. A unique number that is created when you format the volume.
0x2B	11 bytes	Volume Label.
0x36	8 bytes	System ID. Either FAT12 or FAT16, depending on the format of the disk.

File System Data Structures II (Boot block)

The LSN of the boot block is 0. The information contained within the BPB in boot block can be used to calculate the LSN of the block from where the user data starts. It can be simply calculated by adding the number of reserved sector, sectors occupied by FAT copies * number of FAT copies and the the number of blocks reserved for root dir.

Inside a Boot Block

• Contains Code and Data
jmp codepart
OSName
BIOS
Parameter Block
codepart:
• Boot Block executes at Booting time.

Besides the LBA address a LSN address can also be used to address a block. If the LSN address is known the absread() function can be used to read a block and abswrite() can be used to write on it as described in the slide below where nsect is the number of sector to be read/written.

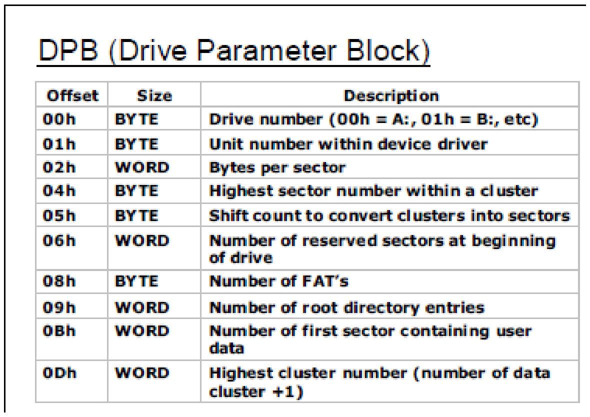
Reading/ Writing a Block

• absread( )
is used to read a block given its LSN
• abswrite( )
is used to write a block given its LSN
absread(int drive, int nsects, long lsec, void *buffer);
abswrite(int drive, int nsects, long lsec, void *buffer);

File System Data Structures III (DPB)

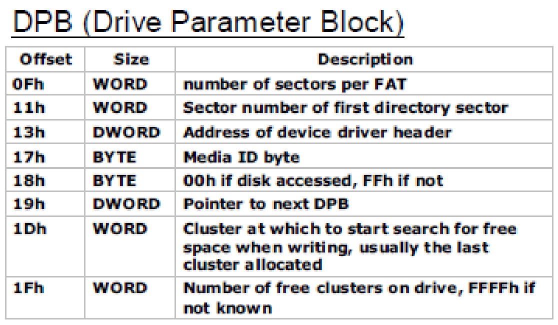
Besides the BPB another data structure can be used equivalently called the DPB (Drive parameter block). The operating system translates the information in BPB on disk into the DPB which is maintained main memory. This data structure can be accessed using the undocumented service 21H/32H. Its detail is shown in the slide below.

The DPB contains the information shown in the table below. This information can be derived from the BPB but is placed in memory in the form of DPB.



DPB (Drive Parameter Block)

Offset	Size	Description
00h	BYTE	Drive number (00h = A:, 01h = B:, etc)
01h	BYTE	Unit number within device driver
02h	WORD	Bytes per sector
04h	BYTE	Highest sector number within a cluster
05h	BYTE	Shift count to convert clusters into sectors
06h	WORD	Number of reserved sectors at beginning of drive
08h	BYTE	Number of FAT's
09h	WORD	Number of root directory entries
0Bh	WORD	Number of first sector containing user data
0Dh	WORD	Highest cluster number (number of data cluster +1)



DPB (Drive Parameter Block)

Offset	Size	Description
0Fh	WORD	number of sectors per FAT
11h	WORD	Sector number of first directory sector
13h	DWORD	Address of device driver header
17h	BYTE	Media ID byte
18h	BYTE	00h if disk accessed, FFh if not
19h	DWORD	Pointer to next DPB
1Dh	WORD	Cluster at which to start search for free space when writing, usually the last cluster allocated
1Fh	WORD	Number of free clusters on drive, FFFFh if not known

Root Directory, FAT12 File System

The DOS directory structure is a Tree like structure. The top most level of the tree being the root directory. The root directory contains files and folders. Each folder can contains more files and folders and so on it continues recursively.

File

• Is logically viewed as an organization of Data.
• Physically it can be a collection of clusters or blocks.
• O.S. needs to maintain information about the cluster numbers of which a file may be comprised of.

Control information about files are maintained in a data structure called the File control block (FCB). The FCB for each file is created and stored in the disk.

The root directory consists of FCBs for all the files and folders stored on the root directory. To obtain these FCBs, the portion on disk reserved for root directory can be read.

In the above two slides first the contents of DPB are read to find the start of the root directory. Using this block number the contents of root directory are read, as it can be seen they contain number of FCBs each containing information about a file within the directory.
The user data area is divided into clusters. The first cluster in user data area is numbered 2 in a FAT based systems. A cluster is not the same as block and also there are no system calls available which use the cluster number. All the system calls use the LSN address. If

the cluster number is known it should be converted into LSN to access the blocks within the cluster. Moreover all the information about file management uses the cluster number rather than the LSN for simplicity and for the purpose of managing large disk space. So here we devise a formula to convert the cluster number into LSN.

FAT12 File System II, FAT16 File System

The root directory contains a collection of FCBs. The FCB for the file in question is searched from where the first cluster of the file can be get.

After calculating the sector number for the cluster the contents of the file can be accessed by reading all the blocks within the cluster. In this way only the starting cluster will be read. If the file contains a number of cluster the subsequent clusters numbers within the file chain can be accessed from the FAT.

Larger File Contents

• Larger files would be comprised of numerous clusters.
• The first Cluster # can be read from FCB for rest of the Cluster, a chain is maintained within the FAT.

FAT12

• FAT is a simple table which contains cluster number of each file.
• FAT12 will have 12-bit wide entries and can have 2^12 entries maximum.
• Although some of these entries may be reserved.

Above slides show how a cluster chain for a file is maintained in the FAT. The first cluster number is in the FCB. Subsequent clusters will be accessed from the FAT using the previous cluster number as index to look up into the FAT for the next cluster number. A FAT theoretically will contain 2_n entries where n is 12 for FAT 12 and 16 for FAT16. But all the entries are not used some of the entries are reserved following slide shows its detail.

Unused FAT Entries

• Reserved Entries = FF0H ~ FF6H
• EOF value = FF7H ~ FFFH
• First Two Clusters = 0,1
• Free Cluster = 0
• Max. range of Cluster # = 2 ~ FEFH
• Total # of Clusters of FAT12 = FEEH

There can various volume with various sizes with FAT12 or FAT16. The number of entries for FAT 12 or FAT16 are limited then the question arises how can a certain volume with moderate space and another volume with large space can be managed by the same FAT system. The answer is that the number of entries might be same but the size of cluster may be different. The cluster size can vary from 512 bytes to 32K in powers of 2 depending upon the volume size.

FAT12 File System (Selecting a 12-bit entry within FAT12 System)

There is no primitive data type of 12 bits. But the entries in 12 bit FAT are 12 bits wide. Three consecutive bytes in FAT 12 will contain two entries.

File Organization

Deleted Files

• 0xE5 at the start of file entry is used to mark the file as deleted.
• The contents of file still remain on disk.
• The contents can be recovered by placing a valid file name, character in place of E5 and then recovering the chain of file in FAT.
• If somehow the clusters used by deleted file has been overwritten by some other file, it cannot be recovered.

FAT32 File System

Let's now perform few more experiments to see how long file names are managed. Windows can have long file names up to 255 characters. For This purpose a file is created with a long file name

Long FileName

Volume in drive H is NEW VOLUME
Volume Serial Number is 8033-3F79

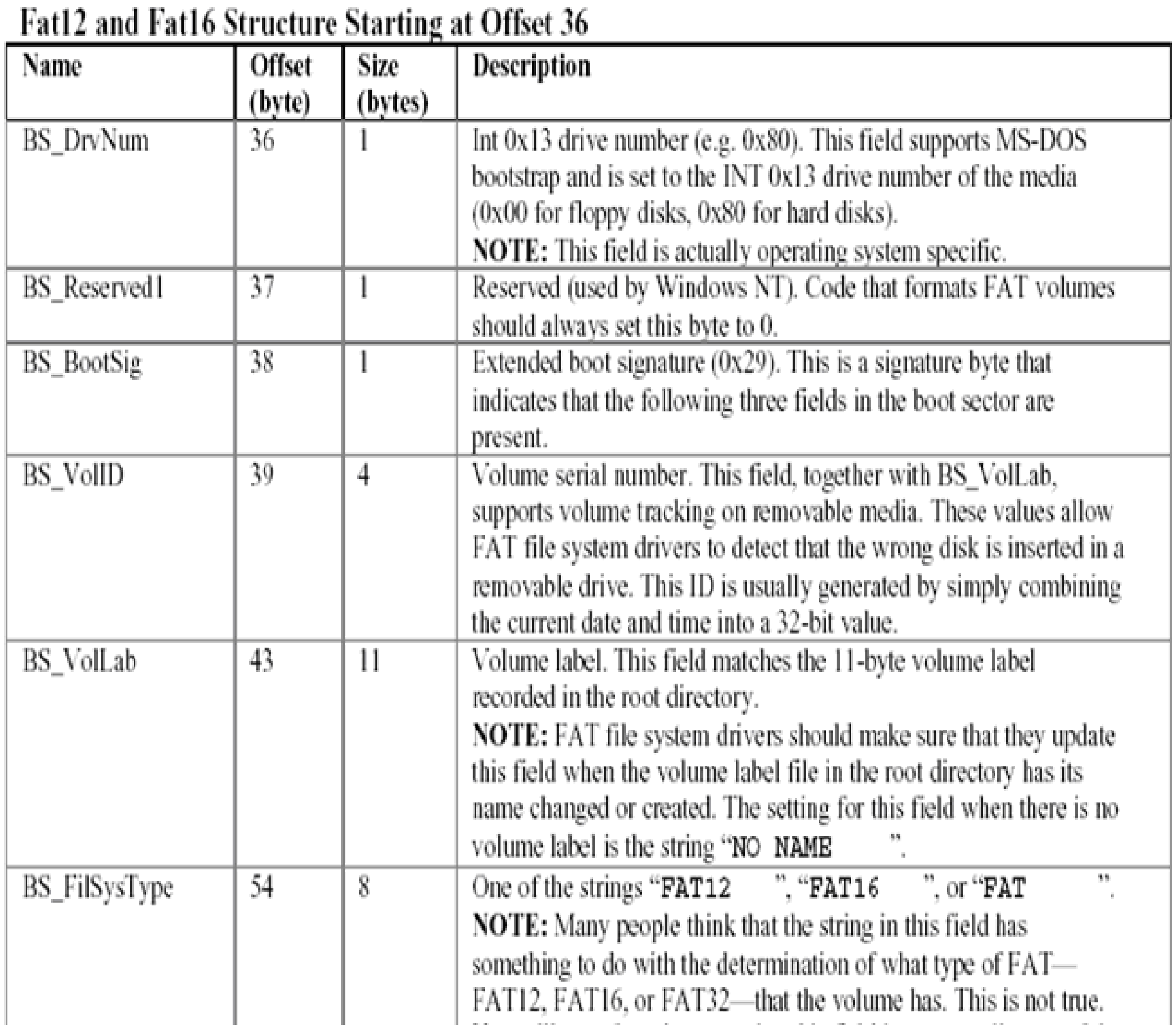
I n the above slide it can be noticed that the long file name is also stored with the FCBs. Also the fragments of Unicode strings in the long file name forms a chain. The first byte in the chain will be 0x01, the first byte of the next fragment will be 0x02 and so on till the last fragment. If the last fragment is the nth fragment starting from 0 such that n is between 0 and 25 the first byte of the last fragment will be given as ASCII of 'A' plus n.

Now lets move our discussion to FAT32. In theory the major difference between FAT 16 and FAT 32 is of course the FAT size. FAT32 evidently will contain more entries and can hence manage a very large disk whereas FAT16 can manage a space of 2 GB maximum practically.
Following slide shows the structure of the BPB for FAT32. Clearly there would be some additional information required in FAT32 so the structure for BPB used in FAT32 is different.

Fat32 BPB Structure

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).



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.

There can be different volumes with different volume sizes. The device driver for file handling would require knowing the FAT size. The following slide illustrates an algorithm that can be used to determine the FAT size in use after reading the BPB.

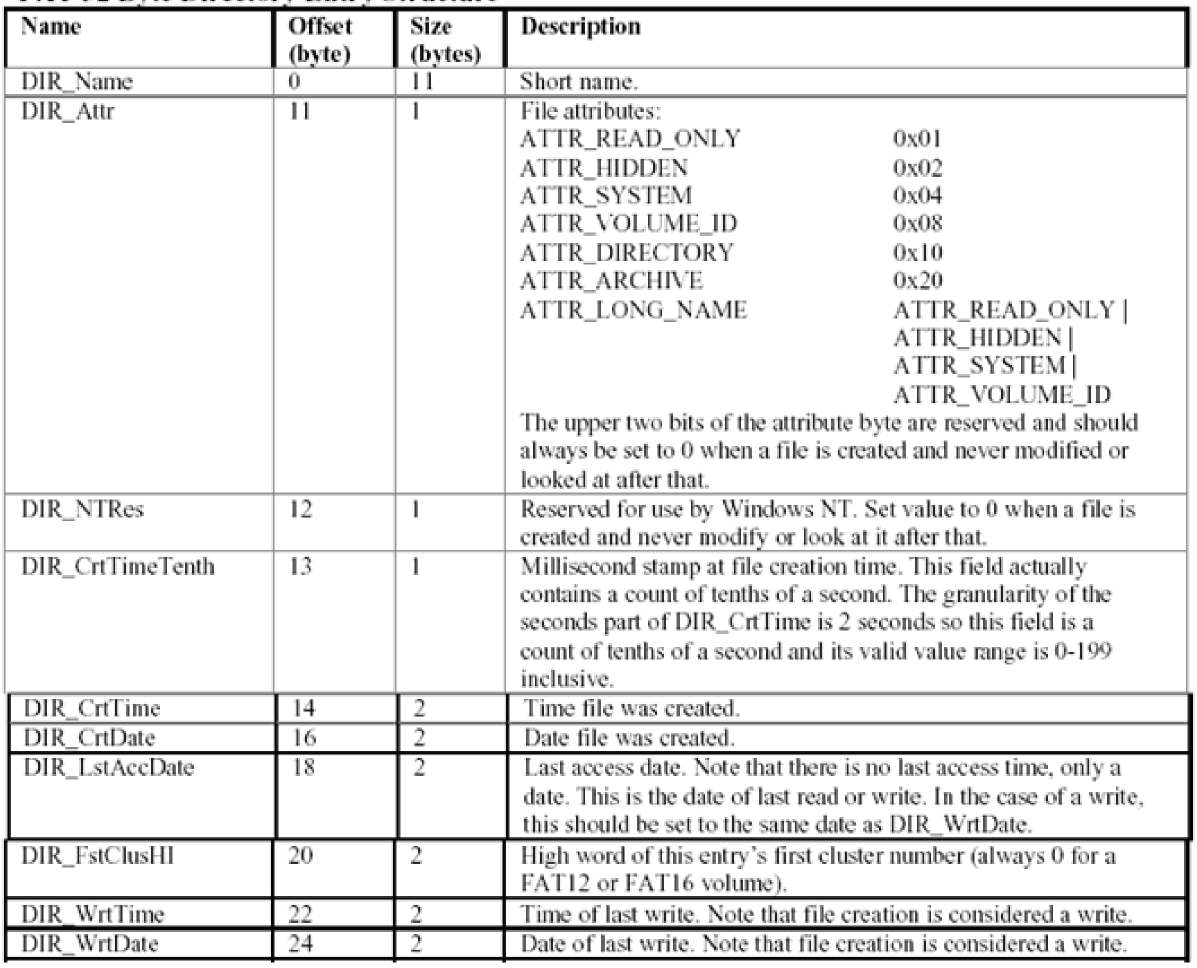
FAT32 Structure Starting at Offset 36

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.

FAT32 File System II

Fat32 Entry

• Each entry is of 32-bits size but only lower 28-bits are used.
• Higher 4-bits are not tempered.
• While reading higher 4-bits are ignored.
• While writing higher 4-bits are not changed.


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.

Anatomy of FAT32 based system differs from FAT16 based systems significantly as explained by the slide below.

Fat32 File System

Reserved Blocks

• No fixed space reserved for root

directory.
• FCB of root directory are saved in a cluster and the cluster # for root directory is saved in BPB as discussed earlier.

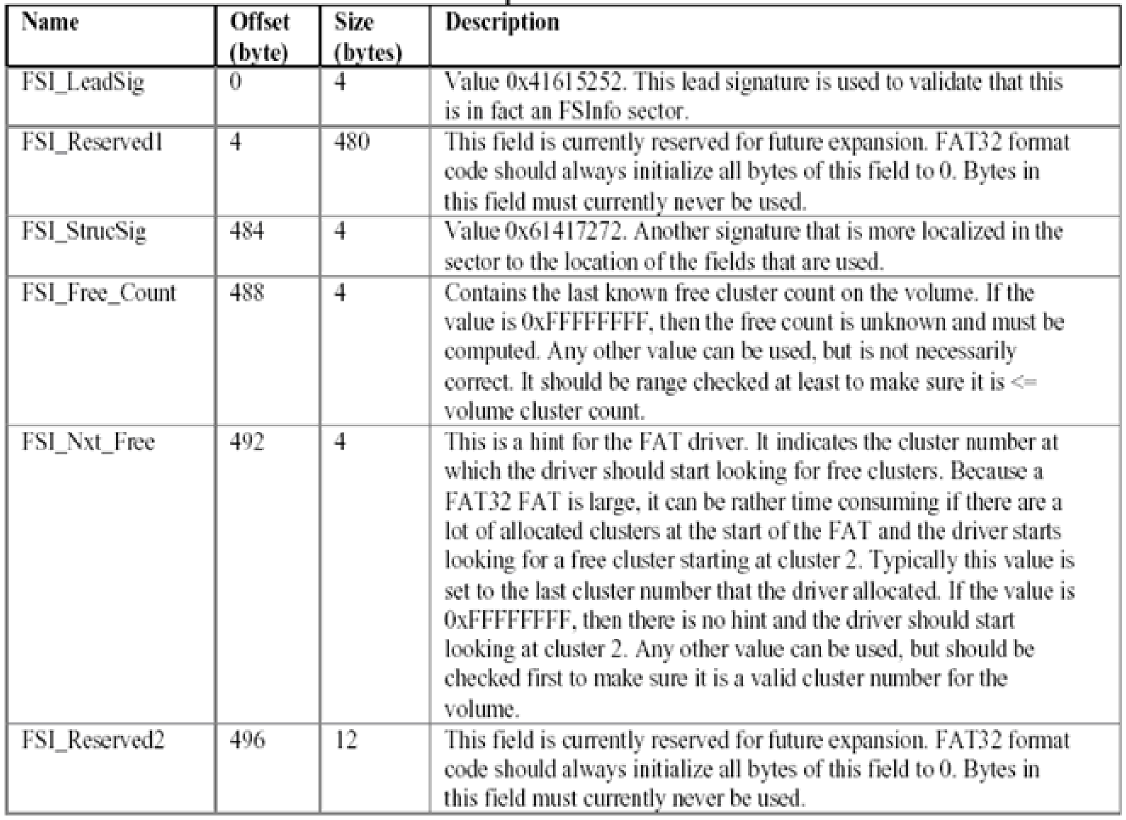
In reflection of the anatomy of FAT32 based system the method used to translate the cluster # into LSN also varies. The following formula is used for this purpose.

Starting Sector # for a Cluster

Starting Sector = Reserved Sect. + FatSize * FatCopies + (cluster # - 2) * size of cluster

In the FAT32 there is another special reserved block called FSInfo sector. The block contains some information required by the operating system while cluster allocation/deallocation to files. This information is also critical for FAT16 based systems. But in FAT12 and 16 this information is calculated when ever required. This calculation at the time of allocation is not feasible in FAT32 as the size of FAT32 is very large and such calculations will consume a lots of time, so to save time this information is stored in the FSInfo block and is updated at the time of allocation/deallocation.

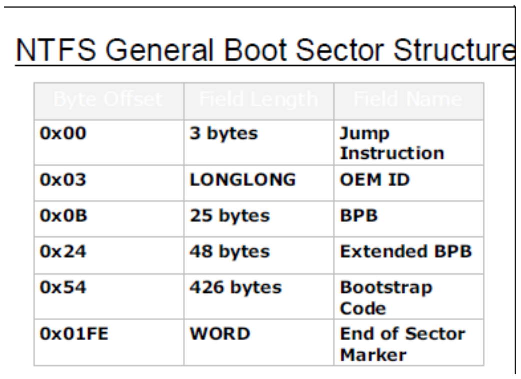
FAT32 FSInfo Sector Structure and Backup Boot Sector



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.

New Technology File System (NTFS)

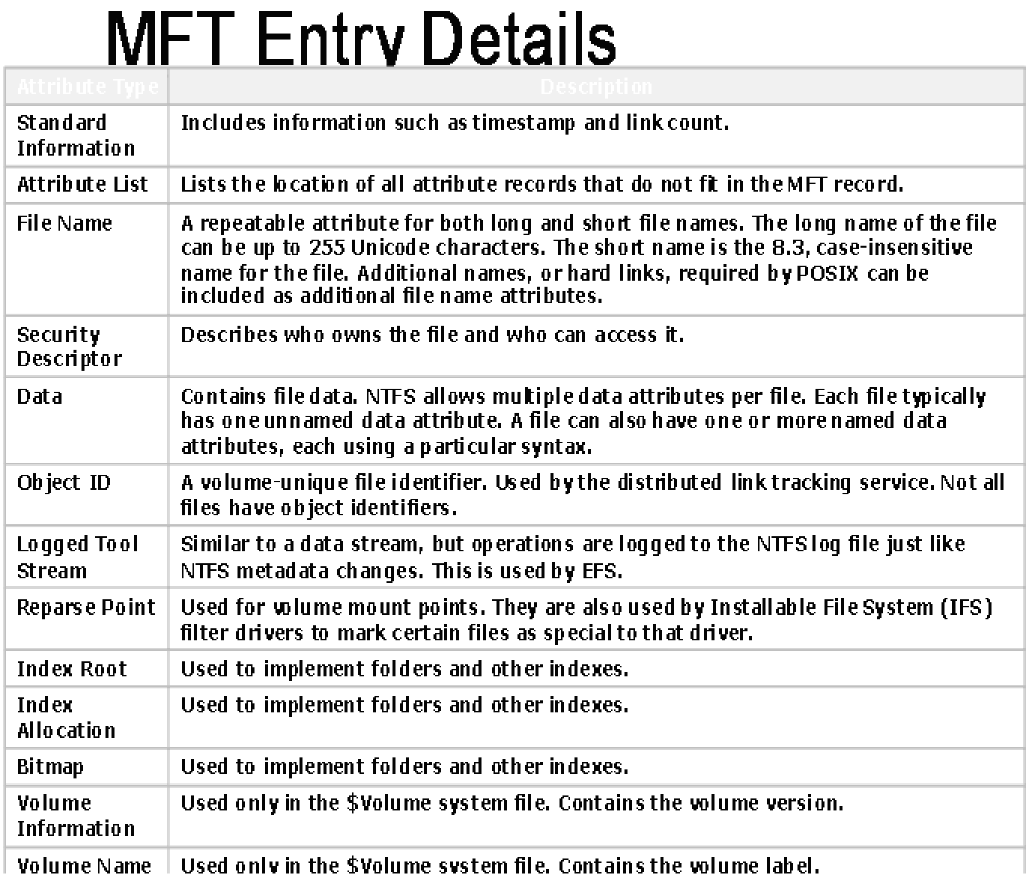
The following slide shows the anatomy of an NTFS based system. The FAT and root directory has been replaced by the MFT. It will generally have two copies the other copy will be a mirror image of the original. Rests of the blocks are reserved for user data. In the middle of the volume is a copy of the first 16 MTF record which are very important to the system.



NTFS General Boot Sector Structure

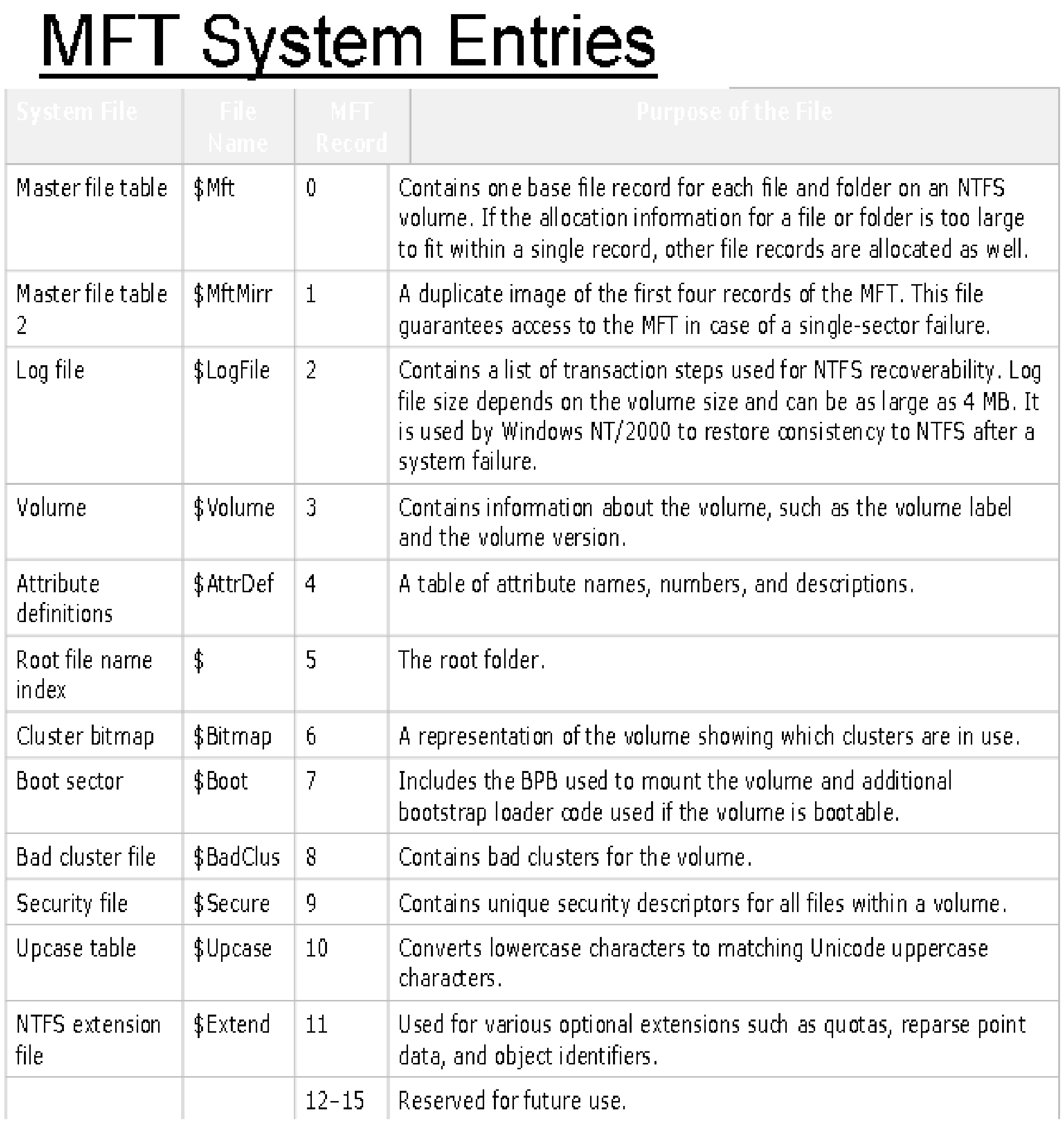
Byte Offset	Field Length	Field Name
0x00	3 bytes	Jump Instruction
0x03	LONGLONG	OEM ID
0x0B	25 bytes	BPB
0x24	48 bytes	Extended BPB
0x54	426 bytes	Bootstrap Code
0x01FE	WORD	End of Sector Marker

The first 16 entries of the MFT are reserved. Rests of the entries are used for user files. There is an entry for each file in the MFT. There can be difference in the way a file is managed depending upon the size of the file.



MFT Entry Details

Attribute Type	Description
Standard Information	Includes information such as timestamp and link count.
Attribute List	Lists the location of all attribute records that do not fit in the MFT record.
File Name	A repeatable attribute for both long and short file names. The long name of the file can be up to 255 Unicode characters. The short name is the 8.3, case-insensitive name for the file. Additional names, or hard links, required by POSIX can be included as additional file name attributes.
Security Descriptor	Describes who owns the file and who can access it.
Data	Contains file data. NTFS allows multiple data attributes per file. Each file typically has one unnamed data attribute. A file can also have one or more named data attributes, each using a particular syntax.
Object ID	A volume-unique file identifier. Used by the distributed link tracking service. Not all files have object identifiers.
Logged Tool Stream	Similar to a data stream, but operations are logged to the NTFS log file just like NTFS metadata changes. This is used by EFS.
Reparse Point	Used for volume mount points. They are also used by Installable File System (IFS) filter drivers to mark certain files as special to that driver.
Index Root	Used to implement folders and other indexes.
Index Allocation	Used to implement folders and other indexes.
Bitmap	Used to implement folders and other indexes.
Volume Information	Used only in the $Volume system file. Contains the volume version.
Volume Name	Used only in the $Volume system file. Contains the volume label.



MFT System Entries

System File	File Name	MFT Record	Purpose of the File
Master file table	$Mft	0	Contains one base file record for each file and folder on an NTFS volume. If the allocation information for a file or folder is too large to fit within a single record, other file records are allocated as well.
Master file table 2	$MftMirr	1	A duplicate image of the first four records of the MFT. This file guarantees access to the MFT in case of a single-sector failure.
Log file	$LogFile	2	Contains a list of transaction steps used for NTFS recoverability. Log file size depends on the volume size and can be as large as 4 MB. It is used by Windows NT/2000 to restore consistency to NTFS after a system failure.
Volume	$Volume	3	Contains information about the volume, such as the volume label and the volume version.
Attribute definitions	$AttrDef	4	A table of attribute names, numbers, and descriptions.
Root file name index	$	5	The root folder.
Cluster bitmap	$Bitmap	6	A representation of the volume showing which clusters are in use.
Boot sector	$Boot	7	Includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable.
Bad cluster file	$BadClus	8	Contains bad clusters for the volume.
Security file	$Secure	9	Contains unique security descriptors for all files within a volume.
Upcase table	$Upcase	10	Converts lowercase characters to matching Unicode uppercase characters.
NTFS extension file	$Extend	11	Used for various optional extensions such as quotas, reparse point data, and object identifiers.
		12-15	Reserved for future use.

For NTFS simply the following formula will be used to translate the sector number into cluster number.

Determining the Sector # from Cluster

Sector # = Cluster # * Sector Per Cluster

The following slides explain how the NTFS volume can be accessed in DOS. Normally it can not be accessed if the system has booted in DOS as the DOS device drivers do not understand NTFS.7

Accessing NTFS volume in DOS

• NTFS volume can not be accessed in DOS using DOS based function like absread( ) etc.

• DOS device drivers does not understand

the NTFS data structures like MFT etc.
• If NTFS volume is accessed in DOS, it will fire the error of Invalid Media.

How to Access NTFS volume using BIOS Functions

• If the system has booted in DOS then a NTFS volume can be accessed by an Indirect Method, using BIOS functions..
• This technique makes use of physical addresses.
• Sector can be accessed by converting their LSN into LBA address and then using the LBA address in extended BIOS functions to access the disk sectors.

The above program uses the DPB to reach the clusters of a file. The getDPB() function gets the far address of the DPB. Using this address the drive parameters are used to determine the location of FAT and root directory. The file is firstly searched in the root directory through sequential search. If the file name and extension is found the first cluster number is used to look up into the FAT for subsequent clusters. The particular block containing the next cluster within the FAT is loaded and the entry is read, similarly the whole chain is traversed till the end of file is encountered.

Disk Utilities

Format
• Low Level Format
-- sets the block size.
-- sets the Initial values in the block.
-- indexes the block for optimal usage.
-- can be accomplished using BIOS routines for small disks or extended BIOS services for larger disks.
• Quick Format
-- initializes the data structures for file management.
-- initializes and sets the size of FAT, root directory etc, according to the drive size.
-- initializes the data in boot block and places appropriate boot strap code for the boot block.

Disk Partitioning Software

•Write the code part of partition table to appropriately load the Boot Block of active partition in primary partition table.
• Places data in the partition table regarding primary and extended partitions.

• As per specification of the user assigns a appropriate size to primary and extended partition by modifying their data part.

Scan Disk

Surface Scan for Bad Sectors

• It attempts to write a block.
• After write it reads back the block contents.
• Performs the CRC test on data read back.
• If there is an error then the data on that block is not stable the cluster of that block should be marked bad.
• The cluster is marked bad by placing the appropriate code for bad cluster so that they may not be allocated to any file.

Lost Chains

• The disk scanning software may also look for lost chains.
• Lost chains are chains in FAT which apparently don't belong to any file.
• They may occur due to some error in the system like power failure during deletion process.

Looking for Lost Chains

• For each file entry in the directory structure its chain in FAT is traversed.
• All the cluster in the file are marked.
• When done with all the files and folders, if some non-zero and non-reserved clusters are left then they belong to some lost chains.
• The lost chains are firstly discretely identified and then each chain can either be restored to newly named files or can be deleted.

Cross References

• If a cluster lie in more than one file chain, then its said to be Cross Referenced.
• Cross references can pose great problems.
• Cross references can be detected easily by traversing through the chain of all files and marking the cluster # during traversal.
• If a cluster is referenced more than once then it indicates a cross reference.
• To solve the problem only one reference should be maintained.

Defragmenter

• Disk fragmentation is unwanted.
• Fragmentation means that clusters of a same file are not contiguously placed, rather they are far apart, increasing seek time hence access time.
• So its desirable that files clusters may be placed contiguously, this can be done by compaction or defragme ntation.
• Defragmentation Software reserves space for each file in contiguous block by moving the data in clusters and readjusting.
• As a result of defragmentation the FAT entries will change and data will move from one cluster to other localized cluster to reduce seek time.
• Defragmentation has high computation cost and thus cannot be performe d frequently.

File Restoration

• FAT structure provides the possibility of recovering a file after deletion, if its clusters were contiguous and have not been over-written.
• DOS perform file deletion by placing 0xE5 at the first byte of it FCB entry and placing 0's (meaning available) in the entries for the file clusters in the FAT.
• Two task should be performed successfully to undelete a file
-- Replacing the 0xE5 entry in FCB by a valid file name character.
-- placing the appropriate values in FAT for representation of file cluster chain.
• If any one of the above cannot be done then the file cannot be fully recovered.

Memory Management

Memory Management

• Understanding of the data structures and techniques used for memory management.
• Study of the overall memory areas used by operating system and applications.

The following slide shows the memory map of the first 1MB of RAM. The first 640KB is called conventional RAM and the higher 384KB is called system memory. Some of the memory areas are reserved for special purposes as described by the slide rest is user area where user application can reside.

In higher processors, the main memory may be greater than 1MB. In this slide it shows that the memory portion higher than 1MB is called extended memory and some unused portion in system memory is called the expanded memory.

Expanded Memory

• also called EMS
• can be accessed using a driver called EMM386.EXE
• this driver allows the use of unused memory within system memory.

Extended Memory

• also called XMS
• can be accessed by installing the driver HIMEM.SYS
• this driver enable the extended memory by shifting from Real to Protected Mode.

Dual Modes in Higher PCs

Higher PCs can operate in two modes
• REAL MODE

• PROTECTED MODE

Real Mode

• PCs initially boots up in Real Mode. It may be shifted to protected mode during the booting process using drivers like HIMEM.SYS
• Only first 1 MB of RAM can be accessed in Real Mode.
• The Real Mode address is a 20-bit address, stored and represented in the form of Segment : Offset
• OS like DOS has a memory management system in reflection of the Real Mode.

Protected Mode

• PC has to be shifted to Protected Mode if originally boots in Real Mode.
• In Protected Mode whole of the RAM is accessible that includes the Conventional, Expanded and Extended Memories.
• OS like Windows has a memory management system for Protected Mode.
• A privilege level can be assigned to a memory area restricting its access.

Memory Management in DOS

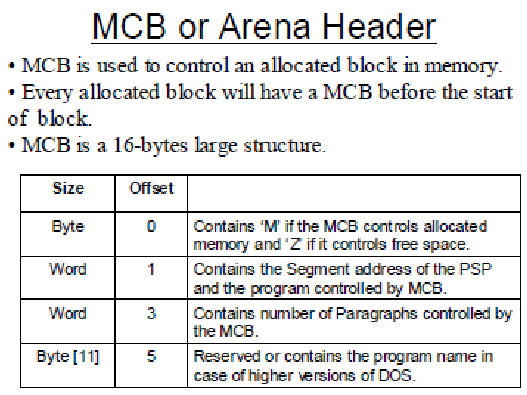
• DOS uses the conventional memory first 640 KB for its memory management.
• Additional 64 KB can be utilized by installing EMM386.EXE and additional 64 KB in the start of extended memory by installing HIMEM.SYS
• Smallest allocatable unit in DOS is a Paragraph, not a Byte.

Paragraph

• Whenever memory is to be allocated DOS allocates memory in form of Paragraph.
• A Paragraph can be understood from the following example consider two Physical Addresses
1234 H : 0000 H
1235 H : 0000 H
• Note there is a difference of 1 between the Segment address.
• Now lets calculate the Physical address
12340 H
12350 H
Difference = 10 H
• A difference of 1 H in Segment address cause a difference of 10 H in Physical address.
• DOS loader assign a segment address whenever memory area is allocated, hence a change of 1 in Segment address will impart a difference of 16 D | 10 H in physical address.

Data Structures for Memory Management

• DOS makes use of various Data Structures for Memory Management:
• MCB (Memory Control Block)
• EB (Environment Block)
• PSP (Program Segment Prefix)



MCB or Arena Header

- MCB is used to control an allocated block in memory.
- Every allocated block will have a MCB before the start of block.
- MCB is a 16-bytes large structure.

Size	Offset	
Byte	0	Contains 'M' if the MCB controls allocated memory and 'Z' if it controls free space.
Word	1	Contains the Segment address of the PSP and the program controlled by MCB.
Word	3	Contains number of Paragraphs controlled by the MCB.
Byte [11]	5	Reserved or contains the program name in case of higher versions of DOS.

Environment Block

• Contains Environment information like Environment variables and file paths for that program

PSP

• is situated before the start of a process.
• contains control information like DTA (Disk Transfer Area) and command line parameters.

The following slide shows that two MCBs are allocated for each program typically. The first MCB controls the Environment Block the next MCB controls the PSP and the program. If this is the last program in memory then the MCB after the program has 'Z' in its first byte indicating that it is the last MCB in the chain.

All the MCB forms a chain. If the address of first MCB is known the segment address of next MCB can be determined by adding the number of paragraph controlled by MCB + ! into the segment address of the MCB. Same is true for all MCBs and hence the whole chain can be traversed.

How to Access the Start of Chain

• An documented service can be used to obtain the address of the first MCB.
• Service 21H/52H is used for this purpose.
• This service returns
The address of DOS internal data structures in ES : BX
• 4-bytes behind the address returned lies the far address of the first MCB in memory.
• Using this address and hence traversing through the chain of MCBs using the information within MCBs.

The above slide shows how service 21H/52H is used to get the address of first MCB in memory.

In the following slide the dump of the first MCB is taken. 'M' in the first byte at the location read indicates the placement of MCB at this location. The address of next MCB can be calculated by adding the number of paragraphs controlled by MCB + 1 into the segment address. Using this method all the MCBs in memory are traversed till the last MCB with first byte 'Z' is encountered.

Non-Contiguous memory allocation

Non-Contiguous Allocation

• Earlier Operating System like DOS has contiguous memory management system i.e. a program cannot be loaded in memory if a contiguous block of memory is not available to accommodate it.
• 80286 and higher processors support non-contiguous allocation.

• 80286 support Segmentation in Protected Mode, i.e. a process is subdivided into segment of variable size and each segment or few segments of the process can be placed anywhere in memory
• 80386 and higher processors also support Paging, i.e. a Process may be divided into fixed size Pages and then only few pages may be loaded any where in memory for Process Execution.
• The key to such non-contiguous allocation systems is the addressing technique.

Address Translation

• In Protected Mode the direct method of seg * 10H + offset for Logical to Physical address translation is discarded and an indirect method is adopted.

Selectors

• In Protected Mode the Segment Registers are used as Selector.
• As the name suggest they are used to select a descriptor entry from some Descriptor Table.

Descriptor

• A Descriptor describes a Memory Segment by storing attributes related to a Memory Segment.
• Significant attributes of a Memory Segment can be its base (starting) address, it length or limit and its access rights.

Descriptor Table

• GDT: Global Descriptor Table
• LDT: Local Descriptor Table
• IDT: Interrupt Descriptor Table
• GDT and LDT can have up to 8192 entries, each of 8-bytes width.
• IDT can have up to 256 entries

Address translation in Protected mode

Selector

• A Selector is called a Selector because it acts as an index into the Descriptor Table to select a GDT or LDT entry.

Address Translation in Protected Mode

• All the tables are maintained in Main Memory.
• Segment Registers are used as Selectors.
• The Descriptor Entry selected from the Descriptor Table is placed in a hidden cache to optimize address translation.

Address Translation in Protected Mode

•Whenever a Selector is assigned a new value, the hardware looks up into the Descriptor Table and loads the Base Address, Limit and Access Rights into the hidden cache.
•Whenever an instruction is issued the address referred is translating into Physical address using the effective Offset within the instruction and the Base Address in the corresponding Segment Cache, e.g.
movAX, [1234H]
effective offset = 1234H
base = base within the cache of DS
abs. address = base +1234H
Or in instruction
movDL, [EBP]

effective offset address = EBP
base address = base address in cache of SS register
abs. address = base address + EBP
• Hence the absolute address cannot be calculated directly from the Segment address value.

Control Register

• 80386 and above have 4 Control Registers CR0 ~ CR3.
• These Control Registers are used for conveying certain control information for Protected Mode Addressing and Co-Processors.
• Here we will illustrate only the least significant bit of CR0.
• The least significant bit of CR0 is PE-bit which can be set to enable Protected Mode Addressing and can be cleared to enter Real Mode.

Moving to Protected Mode

• Protected Mode can be entered by setting the PE bit of CR0, but before this some other initialization must be done. The following steps accomplish the switching from Real to Protected Mode correctly.
1. Initialize the Interrupt Descriptor Table, so it contains valid Interrupt gates for at least the first 32 Interrupt type numbers. The IDT may contain up to 256, 8-byte interrupt gates defining all 256 interrupt types.
2. Initialize the GDT, so it contains a NULL Descriptor, at Descriptor 0 and valid Descriptor for at least one Data and one Stack.
3. Switch to Protected by setting the PE-bit in CR0.
4. Perform a IntraSegment (near) JMP to flush the Internal Pre-fetch Queue.
5. Load all the Data Selectors (Segment Registers) with their initial Selectors Values.
6. The 80386 is now in Protected Mode.

Viruses

• Viruses are special program having ability to embed themselves in a system resources and there on propagate themselves.

State of Viruses

• Dormant State: A Virus in dormant state has embedded itself within and is observing system activities.
• Activation State: A Virus when activated would typically perform some unwanted tasks causing data loss. This state may triggered as result of some event.
• Infection State: A Virus is triggered into this state typically as a result of some disk operation. In this state, the Virus will infect some media or file in order to propagate itself.

Viruses

Types of Viruses

• Partition Table Virus
• Boot Sector Virus
• File Viruses

How Partition Table Virus Works

• The Partition Table Code is executed at boot time to choose the Active Partition.
• Partition Table Viruses embed themselves in the Partition Table of the disk.
• If the Virus Code is large and cannot be accommodated in the Code Part of 512-bytes Partition Table block then it may also use other Physically Addressed Blocks to reside itself.
• Hence at Boot time when Partition Table is to be executed to select the Active Partition, the virus executes. The Virus when executed loads itself in the Memory, where it can not be reached by the OS and then executes the original Partition Table Code (stored in some other blocks after infection) so that the system may be booted

properly.
•When the system boots the Virus will be resident in memory and will typically intercept 13H (the disk interrupt).
•Whenever a disk operation occurs int 13H occurs. The Virus on occurrence of 13H checks if removable media has been accessed through int 13H. If so then it will copy its code properly to the disk first Physical Block (and other blocks depending upon size of Virus Code). The removable disk is now infected.
• If the disk is now removed and is then used in some other system, the Hard Drive of this system will not be infected unless the system is booted from this disk. Because only on booting from this removable disk its first physical block will get the chance to be executed.

How Partition Table Virus Loads itself

• The transient part of Command.Com loads itself such that its last byte is loaded in the last byte of Conventional Memory. If somehow there is some Memory beyond Command.Com's transient part it will not be accessible by DOS.
• At 40:13H a word contains the amount of KBs in Conventional Memory which is typically 640.
• If the value at 40:13H is somehow reduced to 638 the transient part of Command.Com will load itself such that its last byte is loaded at the last byte of 638KB mark in Conventional RAM.
• In this way last 2KB will be left unused by DOS. This amount of memory is used by the Virus according to its own size.

How Boot Sector Virus Works

• Boot Sector also works in almost the same pattern, the only difference is that it will embed itself within the Boot Block Code.

File Viruses

• Various Viruses embeds themselves in different executable files.
• Theoretically any file that can contain any executable code, a Virus can be embedded into it. i.e. .COM, .EXE are executable files so Viruses can be embedded into them, Plain Text Files, Plain Bitmap Files are pure data and cannot be executed so Viruses cannot be actively embedded into them, and even if they are somehow embedded they will never get a chance to execute itself.

COM File

• COM File is a mirror image of the program code. Its image on disk is as it is loaded into the memory.
• COM Files are single segment files in which both Code and Data resides.
• COM File will typically have a Three Bytes Near Jump Instruction as the first instruction in program which will transfer the execution to the Code Part of the Program.

How COM File Virus Infects Files

• A COM File Virus if resident may infect COM Files on execution.
• Typically COM File Virus will Interrupt 21H Service 4B. This Service is used to load a Program.
• Whenever a Program is to be Loaded int 21H Service # 4BH is used to Load a Program. The Virus if resident will check the parameters of this Service to get the file path. If the File is .COM File then the Virus appends itself to the

file and tempers with the first 3-bytes of .COM File so that the execution branches to the Virus Code when the program is executed.

How COM Virus Loads Itself

• When a file is Loaded in Memory it will occupy a number of Paragraphs controlled by some MCB.
• If the file is infected the Virus is also loaded within the Memory Area allocated to the Program.
• In this case the Virus does not exist as an Independent Program as it does not have its own PSP. If the Program is terminated the Virus Code will also be unloaded with the program. The Virus will try to attain an Independent Status for which it needs to relocate itself and create its own PSP and MCB in Memory.
• When the program runs the Virus Code executes first. The Virus creates an MCB, defines a new PSP initializes the PSP and relocates itself, updates the last MCB, so that it can exist as an Individual Program, and then transfers the execution back to the Original Program Code.
• Now if the Original Program Terminates the Virus will still remain resident.

EXE File Viruses

• The EXE File Viruses also works the same way in relocating themselves.
• The main difference in COM File and DOS EXE File is that the COM File starts its execution from the first instruction, whereas the entry point of execution in EXE File can be anywhere in the Program.
• The entry point in case of EXE File is tempered by the Virus which is stored in a 27-byte header in EXE File.

Detection

• Viruses can be detected by searching for their Signature in Memory or Executable Files.
• Signature is a binary subset of Virus Code. It is a part of Virus Code that is unique for that particular Virus only and hence can be used to identify the Virus
• Signature for a Virus is selected by choosing a unique part of its Code. To find a Virus this Code should be searched in memory and in files. If a match is found then the system is infected.

Removal

Partition Table & Boot Sector Viruses

• Partition Table and Boot Sector Viruses can be removed by re-writing the Partition Table or Boot Sector Code.
• If the Virus is resident it may exhibit stealth i.e. prevent other programs from writing on Partition Table or Boot Sector by intercepting int 13H
• In case it's a stealth Virus the system should be booted from a clean disk will not give the Virus any chance to execute or load itself.

File Viruses

• If the Virus size is known Viruses can be removed easily from file.
• Firstly, the original value of first 3-bytes in case of COM File or the entry point in case of EXE should be restored.
• The appended portion of Virus can be removed by coping the contents of original file into a temporary file.
•The Virus Code is not copied.
• The original file is then deleted and the temporary file is renamed as the original file.