EE6012/ET4027 – Laboratory Assignment #0

Updated 13/Jan/2021

J. Walker

No work is to be handed in based on this exercise. Actual exercises will be assigned separately. This is a *pre-assignment exercise* to allow the student to get familiar with some basic concepts and some basic tools. It is very helpful to do the analysis 'by hand' first to become familiar with file system and disk structures.

Students should work through this document before starting the formal assignments.

Objectives:

- Learn how to use a hex editor /disk editor.
- Get familiar with using a disk image file.
- Learn some low-level disk structures, in particular the MBR, to find partition information, including hidden partitions.
- Learn about a volume layout on a sector-by-sector basis.
- Learn how to discover files on the volume including deleted files.
- Do a 'by hand' analysis of the various structures to discover partitions and files, as a step towards learning how to write a forensics analysis software tool.
- Learn how to write a simple forensic tool.

Information required from the Master Boot Record (MBR)

The MBR is a commonly used boot sector on a disk drive that is used to describe the organisation of logical partitions on the disk. The MBR has a limit on its maximum addressable storage space for a disk, so the scheme is being superseded by the GUID Partition Table (GPT) scheme. However, the MBR provides us with an example scheme that we can use to learn how to analyse a disk structure in the context of developing a simple forensic tool.

A first task of this exercise is to get partition information from the MBR of a disk drive. We will examine the MBR of a disk drive to discover the following information:

- How many partitions
- Information on each partition
- Information on any hidden partitions (unallocated space)

The diagram in figure P1 shows the style of information we are looking for. This example disk drive has four partitions of various types. There is also a 'hidden partition' on the disk. The partition table within the Master Boot Record (MBR) lists all of this information. The MBR is the first sector on the disk drive.

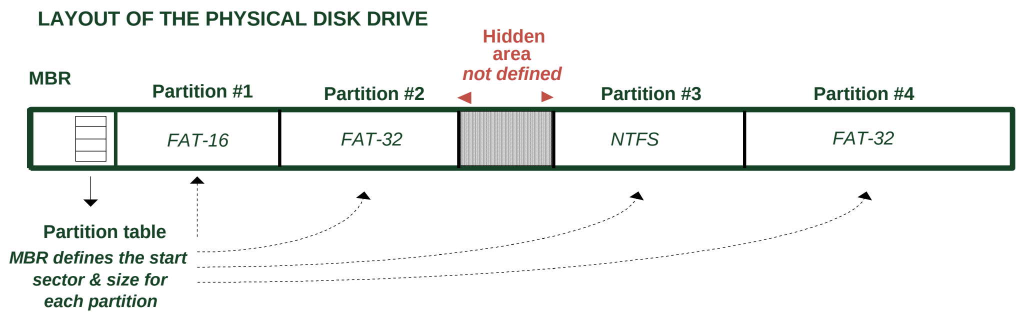


Figure P1 Layout of the physical disk drive

Where to find the required partition information

We now know that the MBR is located on the first sector of the disk drive. It contains the boot code and it also contains the partition table, as seen in the figure P2. The partition table has four entries. A disk drive might have one or more partitions.

Byte offset	Description	Size
000h to 1BDh	Boot Code	446 Bytes
1BEh	1st Partition Table Entry #1	16 Bytes
1CEh	2nd. Partition Table Entry #2	16 Bytes
1DEh	3rd Partition Table Entry #3	16 Bytes
1EEh	4th Partition Table Entry #4	16 Bytes
1FEh	Boot Record Signature (0xAA55)	2 Bytes

Figure P2 Layout of the MBR showing the fields

Let’s look at the 16-byte Partition Table Entry within the MBR in a little more detail, as shown in the figure P3.

Byte Offset	Description	Bytes
00h	Flag – bootable partition (00h=Inactive, 80h=Active)	1
01h	Beginning CHS address	3
04h	Type of partition (See list below)	1
05h	Ending CHS address	3
08h	Starting LBA address (sector)	4
0Ch	Size of partition in sectors	4

Figure P3 The 16-byte Partition Table Entry

Note, the two fields containing the CHS (cylinder, head, sector) addresses will be ignored in these discussions; since this is a legacy addressing scheme for older systems. Today LBA (Logical Block Addressing) is used to identify disk data blocks. The ‘type of partition’ field indicates the type of file system that will reside within the partition. The table of figure P4 lists some of the types. This is not a complete list.

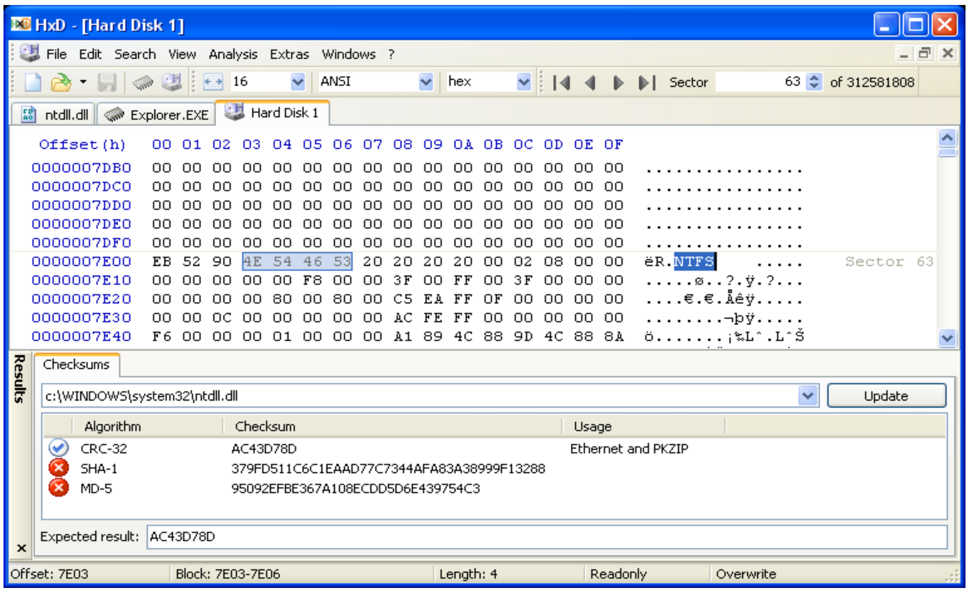
Code	Description
00h	Unknown or empty
01h	12-bit FAT
04h	16-bit FAT (< 32MB)
05h	Extended MS-DOS Partition
06h	FAT-16 (32MB to 2GB)
07h	NTFS
0Bh	FAT-32 (CHS)
0Ch	FAT-32 (LBA)
0Eh	FAT-16 (LBA)
etc.	

Figure P4 Partition Type codes

Inspecting the partition table by hand

We will go through an exercise where we will inspect a partition table ‘by hand’ (i.e. not using an automated tool) so as to discover the partition information. Later we (you?) will write a small program to automate this process.

First we need a simple tool to explore the information on the disk. We will use a hex editor. A hex editor is a useful tool that allows a user to edit or examine the raw content of computer files and disk drive structures at a fundamental binary level. Typically, a hex editor will access sectors from the physical disk drive. If the structure of a disk drive can be examined also, then the hex editor is often referred to as a disk editor. The HxD tool is a hex editor and disk editor which was developed by Mael Horz. It works under Windows. The tool can also edit the memory used by the running processes. HxD is a freeware distribution. Figure 5 shows what the HxD interface looks like. This is just one example hex editor tool, the student can choose any other relevant hex editor.


HxD - [Hard Disk 1]
File Edit Search View Analysis Extras Windows ?
16 ANSI hex Sector 63 of 312581808
ntdll.dll Explorer.EXE Hard Disk 1
Results

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000007DB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000007DC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000007DD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000007DE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000007DF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000007E00 EB 52 90 4E 54 46 53 20 20 20 20 00 02 08 00 00  ëR.NTFS    .....   Sector 63
00000007E10 00 00 00 00 00 F8 00 00 3F 00 FF 00 3F 00 00 00  .....ø..?.ÿ.?...
00000007E20 00 00 00 00 80 00 80 00 C5 EA FF 0F 00 00 00 00  ....€.€.Åêÿ.....
00000007E30 00 00 0C 00 00 00 00 00 AC FE FF 00 00 00 00 00  ........¬þÿ.....
00000007E40 F6 00 00 00 01 00 00 00 A1 89 4C 88 9D 4C 88 8A  ö.......¡‰Lˆ.LˆŠ
```

Checksums

c:\WINDOWS\system32\ntdll.dll Update

	Algorithm	Checksum	Usage
✓	CRC-32	AC43D78D	Ethernet and PKZIP
✗	SHA-1	379FD511C6C1EAAD77C7344AFA83A38999F13288	
✗	MD-5	95092EFBE367A108ECDD5D6E439754C3	

Expected result: AC43D78D

Offset: 7E03 | Block: 7E03-7E06 | Length: 4 | Readonly | Overwrite

Figure P5 HxD screen example

Here are the 3 steps required to inspect the partition table:

1) Use a disk image file as the sample disk (we can work on a live disk also).
2) Run the HxD hex editor tool to inspect the low-level binary information on the disk.
3) Inspect each 16-byte Partition Table Entry in turn and write the relevant information into a table.

Detail on each step:

1) Use a disk image file as the sample disk

- Find a file called ‘Sample_1.dd’ in the location given in the Assignment/Laboratory project document.
- Copy this file to your PC.
- This file is a bit level representation of a real disk and we will use this as our sample disk.

2) Run the HxD hex editor tool to inspect the low-level binary information on the disk

- Install the HxD tool on your PC (HxD is the suggestion – use another one if you wish) http://mh-nexus.de/en/downloads.php?product=HxD
- Run the HxD and get familiar with its interface.
- On the ‘Extras’ tab click ‘Open disk image’ and open the ‘Sample_1.dd’ file. Accept dialogue box question on using 512 byte sectors.
- You will see information similar to below on your screen.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  FA 33 C0 8E D8 8E C0 8E D0 BC FC 7B FB FC BE 00  ú3ÀŽØŽÀŽÐ¼ü{ûü¾.
00000010  7C BF 00 80 B9 80 00 F3 66 A5 EA 1F 80 00 00 80  |¿.€¹€.óf¥ê.€..€
00000020  26 CE 81 80 0F 85 07 01 80 26 DE 81 80 0F 85 FE  &Î.€.….€&Þ.€.…þ
00000030  00 80 26 EE 81 80 0F 85 F5 00 BE BE 81 E8 04 00  .€&î.€.…õ.¾¾.è..
```

Note, you can select **hex** or **dec** addresses from the toolbar. You are looking at raw sector information with 16 bytes in a line. When working on a disk image you can navigate to specific sector numbers. If you have patience you can use HxD to find any information you want by exploring the disk raw data.

3) Inspect each 16-byte Partition Table Entry in turn and note the relevant information by hand in a table.

From the image file, the first sector is the MBR for the disk as shown in figure P6.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  FA 33 C0 8E D8 8E C0 8E D0 BC FC 7B FB FC BE 00  ú3ÀŽØŽÀŽÐ¼ü{ûü¾.
00000010  7C BF 00 80 B9 80 00 F3 66 A5 EA 1F 80 00 00 80  |¿.€¹€.óf¥ê.€..€
00000020  26 CE 81 80 0F 85 07 01 80 26 DE 81 80 0F 85 FE  &Î.€.…..€&Þ.€.…þ
00000030  00 80 26 EE 81 80 0F 85 F5 00 BE BE 81 E8 04 00  .€&î.€.…õ.¾¾.è..
00000040  CD 18 EB FE 8A 04 24 80 75 01 C3 66 8B 44 08 E8  Í.ëþŠ.$€u.Ãf‹D.è
00000050  8D 00 88 44 01 89 5C 02 60 BB 00 7C B8 00 00 8E  ..ˆD.‰\.`».|¸..Ž
00000060  C0 8B 4C 02 8A 74 01 B8 01 02 CD 13 0F 82 C4 00  À‹L.Št.¸..Í..‚Ä.
00000070  81 3E FE 7D 55 AA 0F 85 BF 00 61 60 33 FF 8E C7  .>þ}Uª.…¿.a`3ÿŽÇ
00000080  B4 08 CD 13 83 E1 3F 8A D6 B6 00 42 66 C1 E2 10  ´.Í.ƒá?ŠÖ¶.BfÁâ.
00000090  8B D1 66 39 16 18 7C 66 89 16 18 7C 61 0F 95 C0  ‹Ñf9..|f‰..|a.•À
000000A0  BF 24 7C 66 81 3E 03 7C 4E 54 46 53 74 0B 66 83  ¿$|f.>.|NTFSt.fƒ
000000B0  3E 11 7C 00 75 03 BF 40 7C 8A E2 86 25 38 D4 74  >.|.u.¿@|Šâ†%8Ôt
000000C0  04 3C 01 75 15 52 BB 00 7C B8 00 00 8E C0 8B 4C  .<.u.R».|¸..ŽÀ‹L
000000D0  02 8A 74 01 B8 01 03 CD 13 5A EA 00 7C 00 00 1E  .Št.¸..Í.Zê.|...
000000E0  52 56 66 50 33 FF 8E C7 B4 08 CD 13 83 E1 3F 0F  RVfP3ÿŽÇ´.Í.ƒá?.
000000F0  B6 C6 40 F7 E1 66 0F B7 D8 66 58 66 33 D2 66 F7  ¶Æ@÷áf.·ØfXf3Òf÷
00000100  F3 8B E8 8B C2 33 D2 F7 F1 FE C2 8B CD 8A DA 8A  ó‹è‹Â3Ò÷ñþÂ‹ÍŠÚŠ
00000110  FD C0 E3 02 C1 EB 02 8A F9 5E 5A 1F C3 AC 3C 00  ýÀã.Áë.Šù^Z.Ã¬<.
00000120  74 0B 56 BB 07 00 B4 0E CD 10 5E EB F0 EB FE BE  t.V»..´.Í.^ëðëþ¾
00000130  3E 81 EB E9 BE 56 81 EB E4 BE 75 81 EB DF 49 6E  >.ëé¾V.ëä¾u.ëßIn
00000140  76 61 6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20  valid partition 
00000150  74 61 62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64  table.Error load
00000160  69 6E 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79  ing operating sy
00000170  73 74 65 6D 00 4D 69 73 73 69 6E 67 20 6F 70 65  stem.Missing ope
00000180  72 61 74 69 6E 67 20 73 79 73 74 65 6D 00 00 00  rating system...
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001B0  00 00 00 00 00 2C 44 63 75 44 4F 43 00 00 00 01  .....,DcuDOC....
000001C0  01 00 06 FE 3F 1F 3F 00 00 00 E1 D7 07 00 00 00  ...þ?.?...á×....
000001D0  01 24 0B FE 3F 63 24 D3 08 00 40 B0 0F 00 00 00  .$.þ?c$Ó..@°....
000001E0  01 64 07 FE 3F 7A 64 83 18 00 57 A3 05 00 00 00  .d.þ?zdƒ..W£....
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............Uª
```

Figure P6 The MBR sector detail

From figure P2, earlier, we know that the first Partition Table Entry starts at byte 0x1BE and is 16 bytes long. These 16 bytes are shown in bold text in the figure P6. We can look at these raw16 bytes as follows:

00 01 01 00 06 FE 3F 1F 3F 00 00 00 E1 D7 07 00

We can map these bytes to the Partition Table Entry of figure P3 above to the table as shown in figure P7. Note, the 00h byte offset in the table of figure P7 represents 0x1BE location from figure P6.

Byte Offset	**Description**	**Bytes**	**Mapped in values**
00h	Flag – bootable partition (00h=Inactive, 80h=Active)	1	00
01h	Beginning CHS address	3	01 01 00
04h	Type of Partition (see list)	1	06
05h	Ending CHS address	3	FE 3F 1F
08h	Starting LBA address (sector)	4	3F 00 00 00
0Ch	Size of partition in sectors	4	E1 D7 07 00

Figure P7 Partition Table Entry mapping in actual values

Since we are not interested in the older CHS fields (cylinder, head, sector) we will list only the fields of interest in the table below. Note the little-endian format for the stored bytes on the disk, where the least significant byte is stored at the lowest address. This has to be converted to big-endian format, by hand for now, and converted to a decimal number as shown in brackets.

#	flag	type	start sector	size
1	0x00	0x06	0x0000003f (63)	0x0007D7E1 (514017)

Now we have the information we want for the first partition. To examine the second partition we know from figure P2 that the second Partition Table Entry starts at byte 0x1CE and is 16 bytes long. We extract these 16 bytes and iterate again. We do the same for the third and fourth partitions and we can finally draw the table as shown in figure P8.

#	flag	type	start sector	size
1	0x00	0x06 (fat-16)	0x0000003f (63)	0x0007D7E1 (514017)
2	0x00	0x0B (fat-32)	0x0008D324 (578340)	0x000FB040 (1028160)
3	0x00	0x07 (ntfs)	0x00188364 (1606500)	0x0005A357 (369495)
4	0x00	0x00 unassigned	0x0	0x0

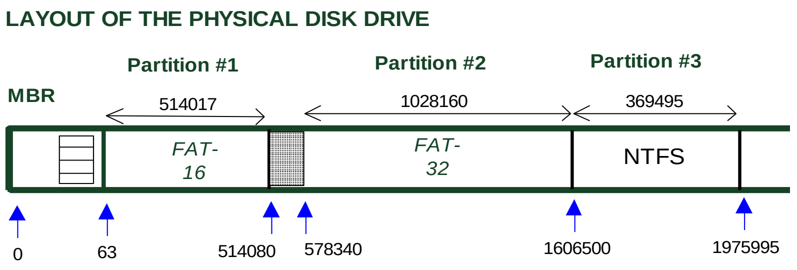


Figure P8 The required partition information

We can summarise the above table by saying that there are three partitions defined which are of various types. There is a gap between the end of Partition #1 and the start of Partition #2. This unused space might be referred to as a ‘hidden’ partition. There may be other unused space on the disk. Such a hidden partition might be used to hold ‘secret’ data but we have now discovered that such a hidden partition does exist.

Volume analysis

Where is the start of a volume?

The MBR's Partition Table Entries will define the starting cluster for each volume. For example, refer to figure P8 and we see that sector 0x3f (63 decimal) is the first sector of the first partition. We also note from figure P8 that this partition is formatted as a FAT-16 partition type, so we say this is a FAT-16 volume.

What is the layout of a FAT volume?

The layout of a FAT volume is shown is figure P9. The volume's boot sector is the first sector of the volume and is part of the Reserved Area. In the early days a disk drive represented a single volume so hence we still have a 'boot sector' within a volume.

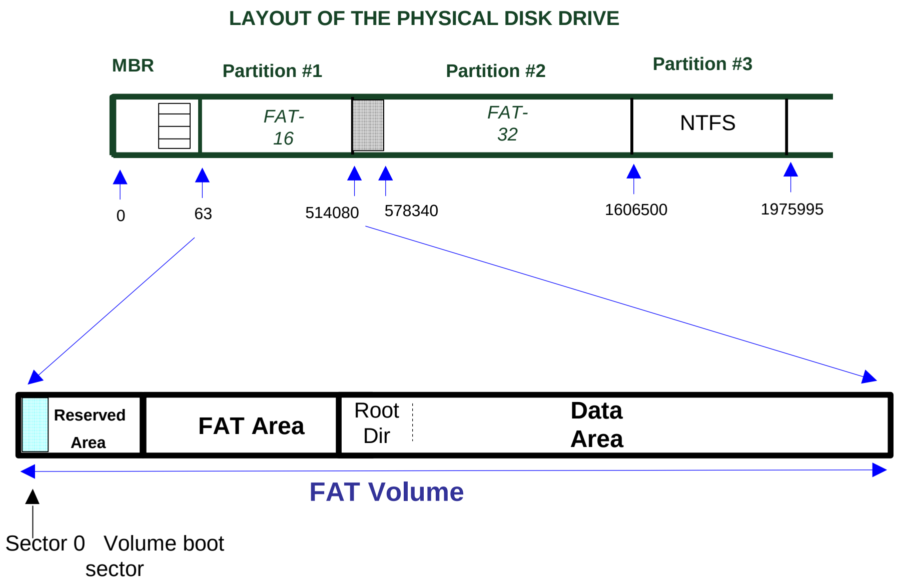


Figure P9 Layout of a FAT volume

We can look in detail at the boot sector (sector 63 of the disk, sector 0 of the volume) for our example image file (Sample_1.dd). The sector detail is shown in the table of figure P10. Note this table represents the FAT-16 structure. The FAT-32 is slightly different in the byte range 24h to 3Dh, but we will focus on the FAT-16 for this tutorial work.

Offset	**Description**	**Size**
00h	Jump to boot code instruction	3 Bytes
03h	OEM Name	8 Bytes
0Bh	No. of bytes per Sector	1 Word
0Dh	No. of sectors per Cluster	1 Byte
0Eh	Size of reserved area in sectors	1 Word (2)
10h	Number of copies of FAT	1 Byte
11h	Maximum no. of Root Directory entries	1 Word (2)
13h	Number of sectors in Partition (16 bit number)	1 Word (2)
15h	Media type (0xf8 fixed, 0xf0 removable).	1 Byte
16h	Size of each FAT in sectors	1 Word (2)
18h	Sectors per Track	1 Word (2)
1Ah	Number of Heads	1 Word (2)
1Ch	Number of sectors before start of partition	1 Double Word (4)
20h	Number of sectors in Partition (32-bit number) Either this field or value in 13h above is used.	1 Double Word (4)
24h	BIOS 1nt 13h drive number	1 Byte
25h	Reserved	1 Byte
26h	Extended Boot Record signature = 29h	1 Byte
27h	Volume Serial Number (based on date/time)	1 Double Word (4)
2Bh	Volume label in ASCII	11 Bytes
36h	System Identifier (FAT12, FAT16, or FAT32)	8 Bytes
3Eh to 1FDh	Not used in these exercises	
1FEh	Boot Record Signature (0x55AA)	2 Bytes

Figure P10   Layout of a FAT-16 volume's boot sector

Now we can use the hex editor to display the actual contents of the boot sector shown in figure P10. We are only interested in the first 64 entries (approx) as this is all that is required to examine the volume. (Microsoft refers to parts of the boot sector as the BPB, the bios parameter block.)

The hex editor printout of figure P11 corresponds to the table of figure P10, for the relevant part of the boot sector.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00007E00  EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 08 02 00  ë<.MSDOS5.0.....
00007E10  02 00 02 00 00 F8 FB 00 3F 00 FF 00 3F 00 00 00  .....øû.?.ÿ.?...
00007E20  E1 D7 07 00 80 00 29 CD 31 52 F4 4E 4F 20 4E 41  á×..€.)Í1RôNO NA
00007E30  4D 45 20 20 20 20 46 41 54 31 36 20 20 20 33 C9  ME    FAT16   3É
00007E40  8E D1 BC F0 7B 8E D9 B8 00 20 8E C0 FC BD 00 7C  ŽÑ¼ð{ŽÙ¸. ŽÀü½.|


NB: address 07E00 represents the start of sector 63 i.e. 63 x 512 = 32256d or 07E00h
```

Figure P11   Hex editor display of part of FAT-16 volume's boot sector

How to find the first cluster of the data area

Note, Microsoft defines the first cluster address of a volume as **Cluster # 2**, i.e. there is no addressable cluster with address 0 or 1. Cluster #1 stores the 'dirty status' for the volume. For a FAT-32 file system the starting cluster (i.e. Cluster #2) begins at the first sector of the data area. However, for the FAT-16 file system the first cluster starts at the first sector after the root directory area. The root directory is within the data area but the starting cluster (Cluster #2) is located at the first sector beyond the root directory. Figure P12 illustrates this.

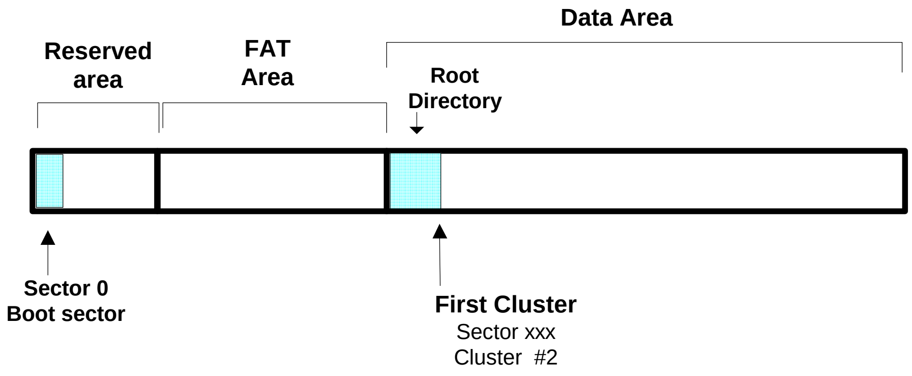


Figure P12 FAT-16 Volume's layout showing location of first cluster (Cluster #2)

In order to find the first cluster (Cluster #2) we need to calculate the size in sectors of the reserved area, the size of the FAT area and the size of the root directory. The boot sector as listed in figure P10 will provide much of this information.

Size of the reserved area in sectors
As shown in the table of figure P10 the size of the reserved area in sectors is given in bytes 0x0E and 0x0F. We note from figure P11 that this entry is 0200 (which is 2 sectors).

Size of the FAT area in sectors
From figure P10 the size of a FAT in sectors is given in bytes 0x16 and 0x17. We note from figure P11 that this entry is FB00 (which is 00FB, which is 251 sectors). We also note from figure P10 that the number of FAT copies is given in byte 0x10, and we see from figure P11 that this entry is 02. The calculated FAT Area size in sectors is:

FAT Area size = (size of FAT in sectors) * (number of FAT copies)

= 251 x 2 = 502 sectors

Size of root directory in sectors
As shown in figure P10 the maximum number of root directories is given in bytes 0x11 and 0x12. We note from figure P11 that this entry is 0002 (which is 0200, which is 512 directory entries). The calculated root directory size in sectors is:

Root dir size = (max no. of dir entries) * (dir entry size in bytes) / sector size

= 512 x 32 / 512 = 32 sectors

NB: the Directory Entry size for a FAT volume is always 32 bytes.

Number of sectors per cluster
As shows in figure P10 the number of sectors per cluster is given in byte 0x0D. We note from figure P11 that this entry is 08. This means that the cluster size is 4kB (8 sectors).

The various information on sizes above is summarised in figure P13.

Information	**Little endian**	**Big endian**	**Decimal**
Reserved area size in sectors	0200	0002	2
Fat size in sectors	FB00	00FB	251
No. of FATs	02	02	2
FAT Area (2 x 251 sectors)			502
No. of root dir entries	0002	0200	512
Root dir size in sectors			32
No. of sectors per cluster	08	08	8

Figure P13 Information to define a volume's layout

What is the sector address of the start of the data area (DA)?

Sector address (DA)

= (first sector of volume) + (size of reserved area) + (size of FAT Area)
= (63) + (2) + (251 * 2)
= 567

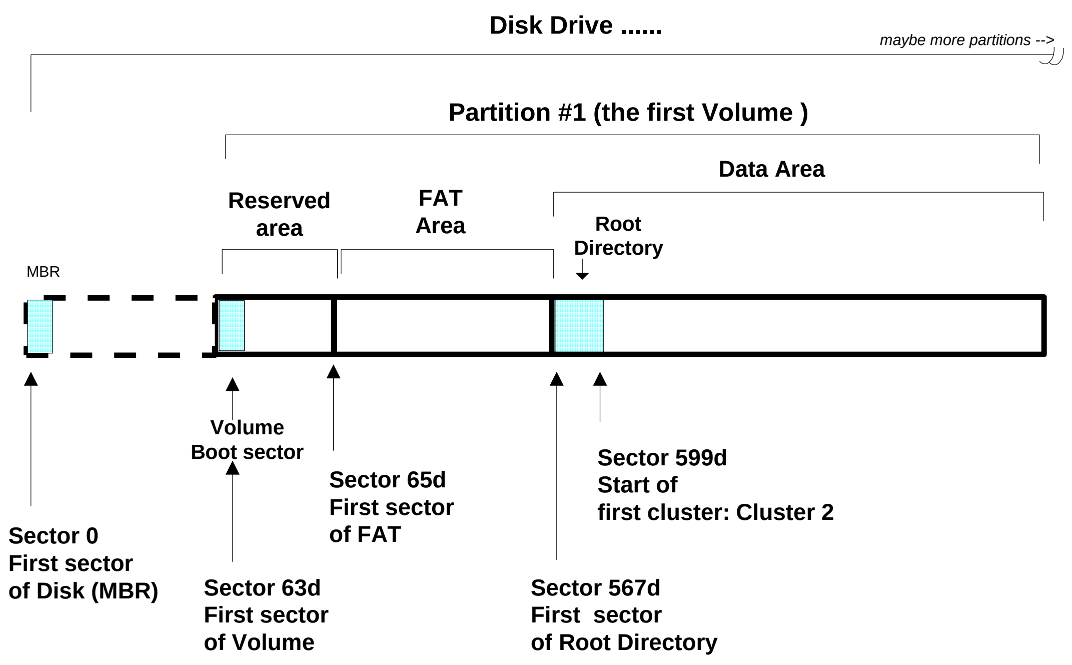
Now we can calculate the Cluster #2 sector address in the disk as follows:

Cluster #2 address = (first sector of data area) + (size of root directory)
= 567 + 32
= 599

In summary the key sector addresses to define the disk layout are listed here in decimal values:

First sector of Disk	**0**
First sector of Reserved Area	**63**
First sector of FAT Area	**65**
First sector of Data Area	**567**
Cluster #2 location	**599 (599 to 607 i.e. 8 sectors)**

Figure P14 illustrates the disk layout indicating the key sector numbers.



Disk Drive ......
maybe more partitions -->
Partition #1 (the first Volume)
Data Area
Reserved area
FAT Area
Root Directory
MBR
Volume Boot sector
Sector 65d First sector of FAT
Sector 599d Start of first cluster: Cluster 2
Sector 0 First sector of Disk (MBR)
Sector 63d First sector of Volume
Sector 567d First sector of Root Directory

Figure P14 Disk layout showing the key sector numbers

Files in the root directory

As already stated, for the FAT-16 file system the root directory is always in a fixed location, i.e. at the start of the data area. For the FAT-32 the root directory is usually located here also, but it can be located elsewhere within the data area. For this exercise we will assume the FAT-16 file system will be used.

For our ‘Sample_1.dd’ disk image file we already know, from the above, that the root directory starts at sector 567d. Using the hex editor we can examine that sector. Figure P15 shows the first few lines of the hex editor display for the root directory, which shows a series of 32 byte directory entries.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00046E00  4D 59 50 41 52 54 49 54 49 4F 4E 08 00 00 00 00  MYPARTITION.....
00046E10  00 00 00 00 00 00 21 81 53 3E 00 00 00 00 00 00  ......!.S>......
00046E20  44 41 54 41 20 20 20 20 54 58 54 20 18 2A 2B 81  DATA    TXT .*+.
00046E30  53 3E 53 3E 00 00 19 5A 52 3E 02 00 68 0E 01 00  S>S>...ZR>..h...
00046E40  E5 4F 4F 4B 20 20 20 20 54 58 54 20 18 B2 2E 81  åOOK    TXT .².. 
00046E50  53 3E 53 3E 00 00 9B 54 53 3E 13 00 CE EE 00 00  S>S>..›TS>..Îî..
00046E60  45 41 53 59 20 20 20 20 54 58 54 20 10 57 33 81  EASY    TXT .W3.
00046E70  53 3E 53 3E 00 00 D7 55 53 3E 22 00 68 0E 01 00  S>S>..×US>".h...
00046E80  41 42 00 6F 00 6F 00 6B 00 54 00 0F 00 AE 77 00  AB.o.o.k.T...®w.
00046E90  6F 00 2E 00 74 00 78 00 74 00 00 00 00 00 FF FF  o...t.x.t.....ÿÿ
00046EA0  42 4F 4F 4B 54 57 4F 20 54 58 54 20 00 98 35 81  BOOKTWO TXT .˜5.
00046EB0  53 3E 53 3E 00 00 EA 54 53 3E 33 00 56 0F 01 00  S>S>..êTS>3.V...
00046EC0  52 45 50 4F 52 54 20 20 54 58 54 20 18 8D 62 86  REPORT  TXT ..b†
00046ED0  53 3E 53 3E 00 00 9B 54 53 3E 13 00 CE EE 00 00  S>S>..›TS>..Îî..
00046EE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure P15 Hex representation of the root directory

A directory entry

A standard 32 byte directory entry is shown in figure P16. The various fields are somewhat self explanatory. It is worth highlighting some details. If the first byte of the file name has the value 0xE5 then this file has been deleted. In the archive field, the archive bit (bit 5) of 0x0B is set by default when any file is created or modified. This bit is cleared when the file is archived.

Bytes	**Meaning**
0x00 to 0x0A	11 byte file name (8 + 3 bytes) (If first byte = 0xE5 then this file is deleted!)
0x0B	Attribute bits: (0x0f means long filename entry) Bit 0: read only Bit 1: hidden file Bit 2: system file Bit 3: volume label Bit 4: directory. Bit 5: archive Bits 6-7: unused.
0x0C	Reserved
0x0D	Creation time (tenths of second)
0x0E to 0x0F	Creation time (hours, mins, secs)
0x10 to 0x11	Creation day
0x12 to 0x13	Accessed day
0x14 to 0x15	FAT32 only – high 2 bytes of starting cluster address
0x16 to 0x17	Written time (hours, mins, secs)
0x18 to 0x19	Written day
0x1A to 0x1B	Starting cluster address (0 for empty file)
0x1C to 0x1F	File size in bytes

Figure P16   Directory entry structure (32 bytes)

Exploring the root directory entries

The first directory entry

We can now look the first directory entry in the root directory, which is the first 32 bytes of the hex dump of figure P15. These 32 bytes are shown again below:

```
00046E00  4D 59 50 41 52 54 49 54 49 4F 4E 08 00 00 00 00  MYPARTITION.....
00046E10  00 00 00 00 00 00 21 81 53 3E 00 00 00 00 00 00  ......!.S>......
```

We can now map these 32 bytes to the directory entry structure of figure P16 as shown in figure P17. We note that this entry represents the volume label which is ‘MYPARTITION’. Since this is not an actual file, it is only a volume label, both the starting cluster field (0x1A .. 0x1B) and the file size field (0x1C .. 0x1F) are set to zero.

Bytes	Meaning	Mapped bytes
0x00 to 0x0A	11 byte file name (8 + 3 bytes)	4D 59 50 41 52 54 49 54 49 4F 4E (MYPARTITION)
0x0B	Attribute bits: (0x0f means long filename entry) Bit 0: read only Bit 1: hidden file Bit 2: system file Bit 3: volume label Bit 4: directory. Bit 5: archive Bits 6-7: unused.	08 (0000**1**000b)
etc.		
etc.		
etc.		
0x1A to 0x1B	Starting cluster (0 for empty file)	0000
0x1C to 0x1F	File size in bytes	00000000

Figure P17 Example directory entry structure for first entry

The second directory entry

We can now map the next 32 bytes of the hex dump to the directory entry structure as follows:

```
00046E20   44 41 54 41 20 20 20 20 54 58 54 20 18 2A 2B 81  DATA    TXT .*+.
00046E30   53 3E 53 3E 00 00 19 5A 52 3E 02 00 68 0E 01 00  S>S>...ZR>..h...
```

Figure P18 shows the directory entry structure for this file, which is seen to be a file called ‘DATA.TXT’.

Bytes	Meaning	Mapped bytes
0x00 to 0x0A	11 byte file name (8 + 3 bytes)	4441544120202020545854 (DATA.TXT)
0x0B	Attribute bits: (0x0f means long filename entry) Bit 0: read only Bit 1: hidden file Bit 2: system file Bit 3: volume label Bit 4: directory Bit 5: archive Bits 6-7: unused	20h (00**1**00000b)
etc.		
etc.		
etc.		
0x1A to 0x1B	Starting cluster (0 for empty file)	0200
0x1C to 0x1F	File size in bytes	680E 0100

Figure P18 Directory entry structure for the second entry

The third directory entry

We can now map another 32 bytes of the hex dump to the directory entry structure as follows, which is the third directory entry:

```
00046E40  E5 4F 4F 4B 20 20 20 20 54 58 54 20 18 B2 2E 81  åOOK    TXT .².. 
00046E50  53 3E 53 3E 00 00 9B 54 53 3E 13 00 CE EE 00 00  S>S>..›TS>..Îî..
```

We note the first byte has the value 0xE5 so this file has been deleted. We will explore this more later on.

The fourth directory entry

We can now map the next 32 bytes of the hex dump to the directory entry structure as follows, which is the fourth directory entry:

```
00046E60  45 41 53 59 20 20 20 20 54 58 54 20 10 57 33 81  EASY    TXT .W3.
00046E70  53 3E 53 3E 00 00 D7 55 53 3E 22 00 68 0E 01 00  S>S>..×US>".h...
```

This seems to be a normal file. It is called EASY.TXT. There is nothing special to note about this file.

The fifth directory entry

We can now map the next 32 bytes of the hex dump to the directory entry structure as follows, which is the fifth directory entry:

```
00046E80  41 42 00 6F 00 6F 00 6B 00 54 00 0F 00 AE 77 00  AB.o.o.k.T...®w.
00046E90  6F 00 2E 00 74 00 78 00 74 00 00 00 00 00 FF FF  o...t.x.t.....ÿÿ
```

This entry is interesting as it actually has a long filename and thus it will receive special treatment. We know it is a long filename entry because the attribute value in 0x0B is **0x0f**. See the discussion later on the 'VFAT and long filenames'.

Finding the clusters for a file

We now know how to find the directory entries (at least for the root directory) on a volume, so how do we find the clusters that belong to a given file? We take the second directory entry as an example. This is the DATA.TXT file, and we will explore this file a little.

We saw above that the DATA.TXT file has the following hex content:

```
00046E20  44 41 54 41 20 20 20 20 54 58 54 20 18 2A 2B 81  DATA    TXT .*+.
00046E30  53 3E 53 3E 00 00 19 5A 52 3E 02 00 68 0E 01 00  S>S>...ZR>..h...
```

We can now map these 32 bytes to the directory entry structure to find the starting cluster and the file size in bytes, as follows:

	Little endian	**Big endian (decimal)**
Starting cluster	0200	0002
File size (bytes)	68 0E 01 00	00010E68 (69,224d)

Find the first sector of the file
From above we know that the starting cluster of the file is Cluster #2. This is not surprising as this was the first file to be created on a fresh disk, so the very first cluster was allocated to this file. From figure P14 above we know that the sector address of the first cluster, Cluster #2, is 599d. We know from figure P13 that cluster size is 8 sectors. So the first cluster of the file resides on the sectors 599d to 606d.

Using the hex editor we can examine these sectors to see what information is contained in the sectors. We can do a search for a specified string etc. Figure P19 shows the hex dump content of part of the first sector for the DATA.TXT file.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

0004AE00  0D 0A 0D 0A 4E 54 46 53 20 54 55 54 4F 52 49 41  ....NTFS TUTORIA
0004AE10  4C 0D 0A 0D 0A 44 2E 20 48 65 66 66 65 72 6E 61  L....D. Hefferna
0004AE20  6E 20 20 55 4E 69 76 65 72 73 69 74 79 20 6F 66  n  UNiversity of
0004AE30  20 4C 49 6D 65 72 69 63 6B 0D 0A 0D 0A 4E 54 46   LImerick....NTF
0004AE40  53 20 69 73 20 74 68 65 20 63 75 72 72 65 6E 74  S is the current
0004AE50  20 73 74 61 6E 64 61 72 64 20 66 69 6C 65 20 73   standard file s
0004AE60  79 73 74 65 6D 20 66 6F 72 20 4D 69 63 72 6F 73  ystem for Micros
0004AE70  6F 66 74 20 6F 70 65 72 61 74 69 6E 67 20 73 79  oft operating sy
0004AE80  73 74 65 6D 73 2E 20 4E 54 46 53 20 77 61 73 20  stems. NTFS was 
0004AE90  66 69 72 73 74 20 69 6E 74 72 6F 64 75 63 65 64  first introduced
0004AEA0  20 69 6E 20 31 39 39 33 20 61 73 20 74 68 65      in 1993 as the
```

Figure P19 Part of first sector for the DATA.TXT file

Later we will see how to find subsequent clusters for the file by using the FAT entries. First we will take a quick look at the FAT.

A quick look at the FAT

We know the starting cluster of the file. The FAT is used to discover the remaining clusters. The FAT entry representing a given cluster contains the address of the next cluster in the file. In this fashion the set of cluster addresses for a file are chained using the FAT. Within the FAT the first addressable cluster is always Cluster #2 , so entries 0 and 1 in the FAT are not needed. The FAT entries can contain special codes as follows:

Cluster is not allocated	**0x0000**
Last cluster in file	**0xFFFF** (can be any value 0xFFF8 to 0xFFFF)
Damaged cluster	**0xFFF7**

We know from figure P14 that sector 65d is the start sector for the FAT in our example disk. Figure P20 below shows the hex editor dump for part of this sector. In other words this is a list of the content for the start of the actual FAT.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00008200  F8 FF FF FF 03 00 04 00 05 00 06 00 07 00 08 00  øÿÿÿ............
00008210  09 00 0A 00 0B 00 0C 00 0D 00 0E 00 0F 00 10 00  ................
00008220  11 00 12 00 FF FF 14 00 15 00 16 00 17 00 18 00  ....ÿÿ..........
00008230  19 00 1A 00 1B 00 1C 00 1D 00 1E 00 1F 00 20 00  .............. .
00008240  21 00 FF FF 23 00 24 00 25 00 26 00 27 00 28 00  !.ÿÿ#.$.%.&.'.(.
00008250  29 00 2A 00 2B 00 2C 00 2D 00 2E 00 2F 00 30 00  ).*.+.,.-.../.0.
00008260  31 00 32 00 FF FF 34 00 35 00 36 00 37 00 38 00  1.2.ÿÿ4.5.6.7.8.
00008270  39 00 3A 00 3B 00 3C 00 3D 00 3E 00 3F 00 40 00  9.:.;.<.=.>.?.@.
00008280  41 00 42 00 43 00 FF FF 00 00 00 00 00 00 00 00  A.B.C.ÿÿ........
```

Figure P20   A hex display for part of the FAT

Since this is a FAT-16 file system we should look upon the FAT as a list of 16-bit (2 bytes) entries. The table below shows the first few entries from figure P20 in table form:

Cluster	**Byte offset into FAT**	**Actual FAT entry**	**Value hex Cluster no.**
#0	0x0	F8 FF	FFF8
#1	0x2	FF FF	FFFF
#2	0x4	03 00	**0003**
#3	0x6	04 00	**0004**
#4	0x8	05 00	**0005**
#5	0xA	06 00	**0006**
etc… etc….			

We see from this table that Cluster #2 points to Cluster #3, and Cluster #3 points to Cluster #4 etc. In this fashion we can follow the chain of clusters that makes up this file DATA.TXT. Eventually we would find a cluster that contains the code 0xFFFF and we would know that this is the final cluster of the file. It so happens that this DATA.TXT file was the first file to be written to a fresh disk and its clusters happen to be contiguously located. The new disk is not fragmented.

Exploring a deleted file

Say we want to investigate if there are any deleted files in the root directory. We could inspect the first bye of every directory (i.e. every 32nd byte: 0x0, 0x20, 0x40, 0x60 etc…) to find if any of these contains the 0xE5 code. For example, we saw above that the third entry in our root directory is a deleted file. We noted this because the first byte has the value 0xE5, so this file has been deleted. We will now explore this deleted file. Here is the hex dump of the file's directory again:

```
00046E40  E5 4F 4F 4B 20 20 20 20 54 58 54 20 18 B2 2E 81  åOOK    TXT .².. 
00046E50  53 3E 53 3E 00 00 9B 54 53 3E 13 00 CE EE 00 00  S>S>..›TS>..Îî..
```

We can possibly discover some or all of the clusters that existed for this deleted file as the relevant clusters may not yet have been reassigned to other files. So, we can attempt to recover the deleted file's contents. This will teach us that when a file is deleted by a user, that file is not necessarily really deleted by the file system. Simply, the first byte of the file name is changed to the code 0xE5 and the file's clusters become available for reassignment to other files later on.

From the hex dump of the directory entry we note the file name is **åOOK.TXT**. The **'å'** character is simply the ASCII character represented by the 0xE5 code. We could guess that the actual file was called BOOK.TXT but cannot know this for certain now.

Bytes	**Meaning**	**Mapped bytes**
0x00 to 0x0A	11 byte file name (8 + 3 bytes)	E54F4F4B20202020545854 åOOK.XT
0x0B	Attribute bits: (0x0f means long filename entry) Bit 0: read only Bit 1: hidden file Bit 2: system file Bit 3: volume label Bit 4: directory. Bit 5: archive Bits 6-7: unused.	20 (00**1**00000b)
etc.		
etc.		
etc.		
0x1A to 0x1B	Starting cluster (0 for empty file)	1300
0x1C to 0x1F	File size in bytes	CEEE0000

Figure P21 Directory entry structure for the third entry, a deleted file

From the figure P21 we can now highlight the starting cluster and the file size as follows:

	Little endian	**Big endian (decimal)**
Starting cluster	1300	0013 (19)
File size	CEEE0000	0000EECE (61134)

We see that the starting cluster is Cluster #13h (19d). So, how do we know the sector address of Cluster #13h. A cluster sector address (CSA) can be calculated for a given cluster number as follows (in decimal) assuming 8 sectors per cluster:

CSA = (sector address for Cluster #2) + ((cluster number – 2) * 8)

= 599 + ((19 – 2) * 8)
= 599 + 136
= 735

So, sector no. 735 is the first sector of Cluster #13h, and it is the first sector of the file. Using the hex editor we can look at sector 735. Here is a list of part of this sector:

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

0005BE00   0D 0A 0D 0A 53 65 63 74 69 6F 6E 20 41 3A 20 20  ....Section A:
0005BE10   50 72 6F 63 65 73 73 65 73 20 20 2D 20 20 20 62  Processes  -   b
0005BE20   79 20 44 2E 20 48 65 66 66 65 72 6E 61 6E 20 55  y D. Heffernan U
0005BE30   6E 69 76 65 72 73 69 74 79 20 6F 66 20 4C 69 6D  niversity of Lim
0005BE40   65 72 69 63 6B 0D 0A 0D 0A 0D 0A 54 68 65 20 70  erick......The p
0005BE50   75 72 70 6F 73 65 20 6F 66 20 74 68 69 73 20 73  urpose of this s
0005BE60   65 63 74 69 6F 6E 20 69 73 20 74 6F 20 69 6E 74  ection is to int
0005BE70   72 6F 64 75 63 65 20 6F 70 65 72 61 74 69 6E 67  roduce operating
0005BE80   20 73 79 73 74 65 6D 73 2C 20 77 69 74 68 20 61   systems, with a
0005BE90   6E 20 65 6D 70 68 61 73 69 73 20 6F 6E 20 70 72  n emphasis on pr
```

So, we are looking at the contents of a deleted file. The user might think that he/she has deleted the file, but in this case the investigator can access the file’s data. Using the FAT entries the subsequent clusters of the file can be discovered.

The VFAT and long filenames

The fifth directory entry on our disk volume is actually a long filename. Let's explore what this means. Microsoft, starting with Windows NT 3.51 and Windows 95, introduced the VFAT (Virtual FAT). This provides long filename (LFN) support, where the filename can be up to 255 characters long. Each character is a 2-byte Unicode character. The long filenames are stored in special directory entries. A special directory entry is a 32-byte structure that looks like this:

Bytes	**Meaning**
0x00	Sequence number (OR'ed with 01000000)
0x01 to 0x0A	Unicode characters 1 .. 5
0x0B	Attribute 0x0F
0x0C	Reserved
0x0D	Checksum
0x0E to 0x19	Unicode characters 6 .. 11
0x1A to 0x1B	Reserved
0x1C to 0x1F	Unicode characters 12 .. 13

The special 0xF attribute in the 0x0B location ensures that this directory entry is recognised as a special entry for VFAT operation. The long name is stored in three fields of the structure to give 13 Unicode character locations (1..5 + 6..11 + 12..13), starting with the tail end of the file name. The Unicode characters are of course stored in little endian fashion. Byte 0x00 stores a sequence number. The sequence numbers start with 1. So, for every 13 characters of a single filename, there is a 32-byte special directory entry. An ordinary directory entry follows these special entries, and it contains a short version of the long file name.

We saw earlier that the fifth entry in our example was a long filename, as the attribute value is **0x0f** as seen here:

```
00046E80   41 42 00 6F 00 6F 00 6B 00 54 00 0F 00 AE 77 00   AB.o.o.k.T...®w.
00046E90   6F 00 2E 00 74 00 78 00 74 00 00 00 00 00 FF FF   o...t.x.t.....ÿÿ
```

In the first byte the sequence number 0x41 tells us there is only one special directory entry (i.e. 41 before OR'ing with 40 is 01). The next entry must then be an ordinary directory entry for this same file with a standard MSDOS file name. This can be seen from the hex dump as follows:

```
00046EA0   42 4F 4F 4B 54 57 4F 20 54 58 54 20 00 98 35 81   BOOKTWO TXT .˜5.
00046EB0   53 3E 53 3E 00 00 EA 54 53 3E 33 00 56 0F 01 00   S>S>..êTS>3.V...
```

If the file had a name which was longer than 13 characters then another special directory entry would have been required. This is a very crude way of implementing long filenames but Microsoft needed to retrofit this feature to an existing file system design.

How about directory files?

We looked at the root directory only. A subdirectory (a folder) is just another file but it is of type: directory. A directory entry which has attribute bit 4 (directory bit) set means that the file in question is a directory file.

END …….. for now!