FILE SYSTEM FORENSICS

Assignment Two

Dara OSullivan
20080494

Table of contents

1. Decode the partition table of your laptop from the hex manually and verify your results using The Sleuth Kit.

2. I began this process my using the FAU tool to get the hex of my physical drive with the command below.

```
dd --localwrt if=//./physicaldrive0 of=PhysicalDrvieFull.txt bs=512
.
```

3. This file was then opened in a hex editor so it could be decoded.

+Offset-	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000C0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000190h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001B0h	00	00	00	00	00	00	00	00	39	82	AA	26	00	00	00	00
000001C0h	02	00	EE	FE	FF	33	01	00	00	00	FF	FF	FF	FF	00	00
000001D0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001F0h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA
00000200h	45	46	49	20	50	41	52	54	00	00	01	00	5C	00	00	00
00000210h	C8	F5	CA	78	00	00	00	00	01	00	00	00	00	00	00	00
00000220h	AF	C2	E7	0E	00	00	00	00	22	00	00	00	00	00	00	00

4. From this point we can see the boot sector begins at byte offset 1B0 in hex which is 432 in decimal. From here I took the four 16-byte lines above the 55 AA marker and decoded them as the following.

Bootable Flag	Starting CHS	Partition Type	Ending CHS	Starting LBA	Length of LBA
Bytes 0	Bytes 1,2,3	Bytes 4	Bytes 5,6,7	Bytes 8,9,10,11	Bytes 12,13,14,15
00	00 02 00	EE	FE FF 33	01 00 00 00	FF FF FF FF

From here I saw that my partition type was EE. Partition type- EE indicates a GPT Protective MBR followed by a GPT/EFI Header. Due to this we need to use the GPT Header sector to decode further.

5. Locating of GPT Header

The header will be decoded according to the break down below.

1) A GPT Header

Byte Range	Description
0-7	Signature value EFIPART
8-11	Version
12-15	Size of GPT header in bytes
16-19	CRC32 of Partition Table
20-23	Reserved
24-31	LBA of the current GPT partition structure
32-39	LBA of the other GPT partition structure
40-47	LBA of start of partition area
48-55	LBA of end of partition area
56-71	Disk GUID
72-79	LBA of the start of the partition table
80-83	Number of entires in partition table
84-87	Size of each entry in partition table
88-91	CRC32 of Partition Table
92-End	Reserved

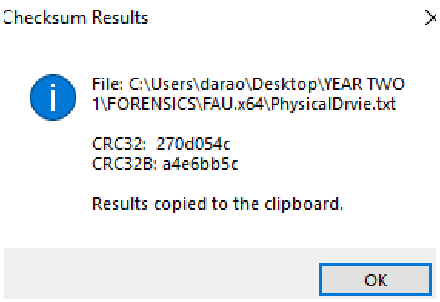
From our hex editor we get the hex of the GPT Header which begins after the indicator 55 AA.

```
45 46 49 20 50 41 52 54 00 00 01 00 5C 00 00 00
C8 F5 CA 78 00 00 00 00 01 00 00 00 00 00 00 00
AF C2 E7 0E 00 00 00 00 22 00 00 00 00 00 00 00
8E C2 E7 0E 00 00 00 00 AF 89 8B A6 4C EC F2 4A
B8 8A 0F 0E 28 90 D6 57 02 00 00 00 00 00 00 00
80 00 00 00 80 00 00 00 56 A3 97 B4 00 00 00 00
```

6. Decoding of Partition

Bytes	Description	Value
45 46 49 20 50 41 52 54	EFI signature	
00 00 01 00	Version	
5C 00 00 00	Size of GPT header in bytes	92
C8 F5 CA 78	CRC32 of partition table	78 CA F5 C8
00 00 00 00	Reserved	0
01 00 00 00 00 00 00 00	LBA of current GPT partition structure	01
AF C2 E7 0E 00 00 00 00	LBA of other GPT partition structure	250,069,679
22 00 00 00 00 00 00 00	LBA of start of partition structure	sector 34
8E C2 E7 0E 00 00 00 00	LBA of end of partition structure	sector 250 069 646
AF 89 8B A6 4C EC F2 4A B8 8A 0F 0E 28 90 D6 57	Disk GUID	57D690280E0F8AB84AF2EC4CA68B89AF
02 00 00 00 00 00 00 00	LBS of the start of the partition table	sector 2
80 00 00 00	Number of entries in partition tables	80 which is 128
56 A3 97 B4	CRC32 of partition table	B4 97 A3 56

7. Comparing the results of the Checksum



2. Explain the concept of the MFT in an NTFS File System. Locate and manually decode the hex of the $MFT entry.

NTFS Boot Sector	Master File Table	File System Data	Master File Table Copy

The Master File Table (MFT) is the most important part of any NTFS File System because it contains information on all files and directories within the system. Every file and directory has at least one entry on the table, which is always 1KB in size. The first 42 bytes of each entry contains the Header while the remaining bytes store attributes, with each having a specific purpose, such as storing a file name or file content. Each entry is given an address based on its location in the table, and the exact size of each entry is defined in the Boot Sector.

Since a file can have one or more entries in the MFT, if said file cannot contain all of its attributes in one entry, it can simply create another entry to store the rest. The MFT is a file itself, meaning it must also have an entry for itself. This is called $MFT and it shows the location of the MFT on the disk. Its starting location can be seen in the Boot Sector, which is always located in the first sector of the filesystem. $MFT will always be the first entry in the MFT.

1. Locating

   I began by looking at byte offset 30 and counting 4 bytes in the boot sector which holds the logical cluster number for the file $MFT. The value is 40 00 in hex which is 16,384 in decimal. We know that 16384 bytes is 4 clusters as there are 8 sectors of 512 bytes in each cluser. From this we can loacte the MFT.

NTFS_Copy.001

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000000h	EB	52	90	4E	54	46	53	20	20	20	20	00	02	08	00	00
000010h	00	00	00	00	00	F8	00	00	3D	00	20	00	24	F4	00	00
000020h	00	00	00	00	80	00	80	00	08	3D	00	00	00	00	00	00
000030h	04	00	00	00	00	00	00	00	D0	03	00	00	00	00	00	00
000040h	F6	00	00	00	01	00	00	00	B2	CF	CC	39	42	3D	B1	41
000050h	00	00	00	00	FA	33	C0	8E	D0	BC	00	7C	FB	68	C0	07
000060h	1F	1E	68	66	00	CB	88	16	0E	00	66	81	3E	03	00	4E
000070h	54	46	53	75	15	B4	41	BB	AA	55	CD	13	72	0C	81	FB
000080h	55	AA	75	06	F7	C1	01	00	75	03	E9	D2	00	1E	83	EC
000090h	18	68	1A	00	B4	48	8A	16	0E	00	8B	F4	16	1F	CD	13
0000A0h	9F	83	C4	18	9E	58	1F	72	E1	3B	06	0B	00	75	DB	A3
0000B0h	0F	00	C1	2E	0F	00	04	1E	5A	33	DB	B9	00	20	2B	C8
0000C0h	66	FF	06	11	00	03	16	0F	00	8E	C2	FF	06	16	00	E8
0000D0h	40	00	2B	C8	77	EF	B8	00	BB	CD	1A	66	23	C0	75	2D
0000E0h	66	81	FB	54	43	50	41	75	24	81	F9	02	01	72	1E	16
0000F0h	68	07	BB	16	68	70	0E	16	68	09	00	66	53	66	53	66
000100h	55	16	16	16	68	B8	01	66	61	0E	07	CD	1A	E9	6A	01
000110h	90	90	66	60	1E	06	66	A1	11	00	66	03	06	1C	00	1E
000120h	66	68	00	00	00	00	66	50	06	53	68	01	00	68	10	00
000130h	B4	42	8A	16	0E	00	16	1F	8B	F4	CD	13	66	59	5B	5A
000140h	66	59	66	59	1F	0F	82	16	00	66	FF	06	11	00	03	16
000150h	0F	00	8E	C2	FF	0E	16	00	75	BC	07	1F	66	61	C3	A0
000160h	F8	01	E8	08	00	A0	FB	01	E8	02	00	EB	FE	B4	01	8B
000170h	F0	AC	3C	00	74	09	B4	0E	BB	07	00	CD	10	EB	F2	C3
000180h	0D	0A	41	20	64	69	73	6B	20	72	65	61	64	20	65	72
000190h	72	6F	72	20	6F	63	63	75	72	72	65	64	00	0D	0A	42
0001A0h	4F	4F	54	4D	47	52	20	69	73	20	6D	69	73	73	69	6E
0001B0h	67	00	0D	0A	42	4F	4F	54	4D	47	52	20	69	73	20	63
0001C0h	6F	6D	70	72	65	73	73	65	64	00	0D	0A	50	72	65	73
0001D0h	73	20	43	74	72	6C	2B	41	6C	74	2B	44	65	6C	20	74
0001E0h	6F	20	72	65	73	74	61	72	74	0D	0A	00	00	00	00	00
0001F0h	00	00	00	00	00	00	00	00	80	9D	B2	CA	00	00	55	AA
000200h	07	00	42	00	4F	00	4F	00	54	00	4D	00	47	00	52	00
000210h	04	00	24	00	49	00	33	00	30	00	00	E0	00	00	00	30
000220h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

MFT Hex :

46	49	4C	45	30	00	03	00	00	00	00	00	00	00	00	00
01	00	01	00	38	00	01	00	98	01	00	00	00	04	00	00
00	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00

2. Decoding

Offset in Record	Size in Bytes	Description
0x00	**4**	Always: **FILE** (or *Magic Number* = 0x**454C4946**; or as they appear in a disk editor: "46 49 4C 45")
0x04	**2**	Offset to 'Update Sequence...Array' (from start of FILE header); often called the "Fix-up Array" in forensic circles. (Prior to NTFS v. 3.1, this was 0x00**2A**. For NTFS v. 3.1, it became 0x00**30**.)
0x06	**2**	*Size* in WORDs of the 'Update Sequence Number *and* Array' (USA)
0x08	**8**	$LogFile Sequence Number (or **LSN**)
0x10	**2**	Sequence Number
0x12	**2**	Hard Link Count
0x14	**2**	Offset to first Attribute
0x16	**2**	Flags
0x18	**4**	*Actual Size* of this FILE Record.
0x1C	**4**	*Allocated Size* of this FILE Record; normally 0x400 (1 KiB; 2 sectors).
0x20	**8**	File Reference to *Base FILE* Record
0x28	**2**	Next Attribute ID
0x2A	**2**	*Used to* align to 4-byte boundary (only since NTFS version 3.1)
0x2C	**4**	MFT Record Number (only since NTFS version 3.1)

Table 4.

Byte offset 0x00 = 46 49 4C 45 = FILE in Ascii

Byte offset 0x04 = 30 00 = Fixup Array

Byte offset 0x06 = 03 00 = Update sequence number and array

Byte offset 0x08= 00 00 00 00 00 00 00 00 = $LogFile Sequence number and array

Byte offset 0x10 = 01 00 = Sequence number

Byte offset 0x12 = 01 00 = Hard Link Count

Byte offset 0x14 = 38 00 = Offset to first attribute

Byte offset 0x16 = 01 00 = Flags

Byte offset 0x18 = 98 01 00 00 = Actual size file

Byte offset 0xIC = 00 04 00 00 = Allocated size of this file record (I kb 2 sectors)

Byte offset 0x20= 0 = File reference to base file record

Byte offset 0x28= 04 00 = Next attribute id

3. Using TSK decode another two MFT Entries of your choice. How big is the file and what data units does it occupy? Explain the attribute types and attribute values associated with these entries.

We know that each entry in the MFT table is 1024 bytes and we know the MFT table starts at cluster four. We also know the $LogFile is the third entry in the MFT table so from this information we can workout the byte offset of the $LogFile and then decode from there.

0. Master file table $MFT.
1. Master file table mirror $MftMirr.
2. Log file $LogFile.
3. Volume $Volume contains information such as the volume label and the volume version.
4. Attribute definitions $AttrDef.
5. The root folder "."
6. Cluster bitmap $Bitmap which represents the volume by showing free and unused clusters.
7. Bot sector $Boot, includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable.
8. Bad cluster file $BadClus, which contains a list of bad clusters for the volume.
9. Security file $Secure which contains unique security descriptors for all files within a volume.
10. Upcase table $Upcase which converts lowercase characters to matching Unicode uppercase characters.
11. NTFS extension file $Extend, that is used for future use.
12. Reserved
13. Reserved
14. Reserved
15. Reserved

Cluster four starts at byte offset 16384 as 4 x 8 (sectors in each cluster) x 512 (bytes in each sector). 16384 + 2048 (accounting for the first two entries both of size 1024 bytes) = 18432 which is out byte offset for the $LogFile. Then from sluethkit I used the blkcat command, blkcat -h NTFS_Copy.001 4, to see the hex of the Master file table. From here I found the $LogFile at the byte offset 2048 from the beginning of the data unit.

These two entries could have also been located from the inodes. The two entries will be decoded according to the following table.

Offset	Size	Description
0x00	4	Attribute Type Identifier (see Table 4)
0x04	4	Length of Attribute (determines the location of next attribute)
0x08	1	Non-resident flag
0x09	1	Lenght of name
0x0a	2	Offset to name
0x0c	2	Flags
0x0e	2	Attribute Identifier
0x10	4	Size of content.
0x15	2	Offset to content

$LogFile

4 6 4 9 4 c 4 5	3 0 0 0 0 3 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 2 0 0 0 1 0 0	3 8 0 0 0 1 0 0	5 8 0 1 0 0 0 0	0 0 0 4 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 3 0 0 0 0 0 0	0 2 0 0 0 0 0 0
0 2 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	6 0 0 0 0 0 0 0
0 0 0 0 1 8 0 0	0 0 0 0 0 0 0 0	4 8 0 0 0 0 0 0	1 8 0 0 0 0 0 0
0 0 4 0 f 7 1 9	e 2 d a c c 0 1	0 0 4 0 f 7 1 9	e 2 d a c c 0 1
0 0 4 0 f 7 1 9	e 2 d a c c 0 1	0 0 4 0 f 7 1 9	e 2 d a c c 0 1
0 6 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	3 0 0 0 0 0 0 0	7 0 0 0 0 0 0 0
0 0 0 0 1 8 0 0	0 0 0 0 0 2 0 0	5 2 0 0 0 0 0 0	1 8 0 0 0 1 0 0
0 5 0 0 0 0 0 0	0 0 0 0 0 5 0 0	0 0 4 0 f 7 1 9	e 2 d a c c 0 1
0 0 4 0 f 7 1 9	e 2 d a c c 0 1	0 0 4 0 f 7 1 9	e 2 d a c c 0 1
0 0 4 0 f 7 1 9	e 2 d a c c 0 1	0 0 0 0 2 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0	0 0 0 0 0 0 0 0	0 6 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 8 0 3 2 4 0 0	4 c 0 0 6 f 0 0	6 7 0 0 4 6 0 0	6 9 0 0 6 c 0 0
6 5 0 0 0 0 0 0	0 0 0 0 0 0 0 0	8 0 0 0 0 0 0 0	4 8 0 0 0 0 0 0
0 1 0 0 4 0 0 0	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
f f 0 1 0 0 0 0	0 0 0 0 0 0 0 0	4 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 2 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0	0 0 0 0 0 0 0 0	2 2 0 0 0 2 d 1	0 3 0 0 0 0 0 0
f f f f f f f f	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

Byte offset 0x00 = Attribute Type identifier = 46 49 4C 45 = FILE in ascii

Byte offset 0x04 = Length of Attribute = 30 00 30 00 =

Byte offset 0x08 = Non-resident flag = 00

Byte offset 0x09 = Length of name= 00

Byte offset 0x0a = Offset to name = 00

Byte offset 0x0c = Flags = 00

Byte offset 0x0e = Attribute identifier = 00

Byte offset 0x10 = size of content = 02 00 01 00

Byte offset 0x15 = offset to content = 38 00

$BitMap

To find the $Bitmap we need to go to the second cluster which holds the MFT file because the $BitMap is the 7th entry in the MFT table.

```
4 6 4 9 4 c 4 5   3 0 0 0 0 3 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 6 0 0 0 1 0 0   3 8 0 0 0 1 0 0   5 0 0 1 0 0 0 0   0 0 0 4 0 0 0 0
0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 3 0 0 0 0 0 0   0 6 0 0 0 0 0 0
0 2 0 0 0 0 0 0   0 0 0 0 0 0 0 0   1 0 0 0 0 0 0 0   6 0 0 0 0 0 0 0
0 0 0 0 1 8 0 0   0 0 0 0 0 0 0 0   4 8 0 0 0 0 0 0   1 8 0 0 0 0 0 0
0 0 4 0 f 7 1 9   e 2 d a c c 0 1   0 0 4 0 f 7 1 9   e 2 d a c c 0 1
0 0 4 0 f 7 1 9   e 2 d a c c 0 1   0 0 4 0 f 7 1 9   e 2 d a c c 0 1
0 6 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0   0 0 0 1 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   3 0 0 0 0 0 0 0   6 8 0 0 0 0 0 0
0 0 0 0 1 8 0 0   0 0 0 0 0 2 0 0   5 0 0 0 0 0 0 0   1 8 0 0 0 1 0 0
0 5 0 0 0 0 0 0   0 0 0 0 0 5 0 0   0 0 4 0 f 7 1 9   e 2 d a c c 0 1
0 0 4 0 f 7 1 9   e 2 d a c c 0 1   0 0 4 0 f 7 1 9   e 2 d a c c 0 1
0 0 4 0 f 7 1 9   e 2 d a c c 0 1   0 0 1 0 0 0 0 0   0 0 0 0 0 0 0 0
f 8 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 6 0 0 0 0 0 0   0 0 0 0 0 0 0 0
0 7 0 3 2 4 0 0   4 2 0 0 6 9 0 0   7 4 0 0 6 d 0 0   6 1 0 0 7 0 0 0
8 0 0 0 0 0 0 0   4 8 0 0 0 0 0 0   0 1 0 0 4 0 0 0   0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 1 0 0 0 0 0   0 0 0 0 0 0 0 0
f 8 0 0 0 0 0 0   0 0 0 0 0 0 0 0   f 8 0 0 0 0 0 0   0 0 0 0 0 0 0 0
2 1 0 1 f c 0 0   0 0 0 0 0 0 0 0   f f f f f f f f   0 0 0 0 0 0 0 0
```

Byte offset 0x00 = Attribute Type identifier = 46 49 4C 45 = FILE in ascii

Byte offset 0x04 = Length of Attribute = 30 00 30 00 =

Byte offset 0x08 = Non-resident flag = 00

Byte offset 0x09 = Length of name= 00

Byte offset 0x0a = Offset to name = 00

Byte offset 0x0c = Flags = 00

Byte offset 0x0e = Attribute identifier = 00

Byte offset 0x10 = size of content = 06 00 01 00

Byte offset 0x15 = offset to content = 38 00

4. Explain the five categories of data associated with a file system.

The five categories of data associated with a file system are: Filesystem, Filename, Metadata, Content, Application.

- File System: The general layout of the file is held in the data here. It will tell us the size of the data units and how many there are. We can be aware of the general layout of a particular file system but the file system category data gives us huge insight into where to find the data in this particular system. From this category we find out the sector and cluster sizes, the type of file system being used, the location of the first MFT, and the size of the entries. To view the actual data structure locations, you need to use a hex editor to see the layout. This category is essential for finding being able to find files in the file system and it helps hugely to find hidden data as you can clearly see the empty spaces etc in the system. This category of data holds the $MFT, $MFTMirr, $Boot, $Volume, $Attrdef file information.

  We can see these attributes in our NTFS system:

```
r/r 4-128-1:    $AttrDef
r/r 8-128-2:    $BadClus
r/r 8-128-1:    $BadClus:$Bad
r/r 6-128-1:    $Bitmap
r/r 7-128-1:    $Boot
d/d 11-144-2:   $Extend
r/r 2-128-1:    $LogFile
r/r 0-128-1:    $MFT
r/r 1-128-1:    $MFTMirr
r/r 9-128-2:    $Secure:$SDS
r/r 9-144-3:    $Secure:$SDH
r/r 9-144-4:    $Secure:$SII
r/r 10-128-1:   $UpCase
r/r 3-128-3:    $Volume
r/r 64-128-2:   picture10.JPG
r/r 65-128-2:   picture9.jpg
V/V 66: $OrphanFiles
```

  From this we now know $MFT is entry 0, $MFTMirr is entry 1, $Boot is entry 7, $Volume is entry 3 and $Affrdef is entry 4 in our filesystem.

- File Name Category: This category of data holds the data structures that store the name of each file and directory. We know where the data is from the $Index_Root and $Index_Allocation and it instructs us on how to analyse them.

- Data Unit Category: This category contains the data units (i.e. blocks and clusters) in the file system that store the file contents. Data units are a fixed size and most file systems require it to be a power of 2, 1024- or 4096-bytes for example. Most of the data on a file system will reside in this category.

Below is the content information from our given NTFS. From this we can see 512 bytes in a sector and 4096 bytes in a sector so we can work out that there are 8 sectors in each cluster.

```
CONTENT INFORMATION
--------------------------------------------
Sector Size: 512
Cluster Size: 4096
Total Cluster Range: 0 - 1952
Total Sector Range: 0 - 15623
```

- Metadata Category: This is where the descriptive data about files and directories are stored. This layer holds MFT entries of the NTFS. This category holds information such as access times, file size, permissions, and pointers to the data units that were allocated by the file or directory.

```
METADATA INFORMATION
--------------------------------------------
First Cluster of MFT: 4
First Cluster of MFT Mirror: 976
Size of MFT Entries: 1024 bytes
Size of Index Records: 4096 bytes
Range: 0 - 66
Root Directory: 5
```

- Application Category: Here content such as disk quotas and journaling which record changes are stored. This content is non-essential but can be very useful in the running of the file system and operating system. These are features that make life easier for the file system and operating system.

5. In terms of NTFS identify the major data structures associated with each category and their function. How do these compare with a FAT file system and an EXT filesystem?

NTFS File system:

$MFT

The Master File Table (MFT) is the most important part of any NTFS File System because it contains information on all files and directories within the system. Every file and directory has at least one entry on the table, which is always 1KB in size. The first 42 bytes of each entry contains the header while the remaining bytes store attributes, with each having a specific purpose, such as storing a file name or file content. Each entry is given an address based on its location in the table, and the exact size of each entry is defined in the Boot Sector. Since a file can have one or more entries in the MFT, if said file cannot contain all of its attributes in one entry, it can simply create another entry to store the rest. The MFT is a file itself, meaning it must also have an entry for itself. This is called $MFT and it shows the location of the MFT on the disk. Its starting location can be seen in the Boot Sector, which is always located in the first sector of the filesystem. $MFT will always be the first entry in the MFT.

$MFTMirr

The $MFTMirr is a system file that duplicates at least the first four FILE records of the MFT ($MFT, $MFTMirr, $LogFile and $Volume) for recovery purposes. $MFTMirr is entry one of the MFT file.

$Boot

The $Boot structure file allows the system to boot. This metadata file points at the boot sector of the volume. It contains information about the size of the volume, clusters and the MFT. It is the only metadata file to have a static location. It is found at MFT entry 7. Its $DATA attribute always exists within the first sector of the file system because it is required to actually boot up the system.

$Volume

$Volume resides at entry 3 in the MFT,and it stores information about the version and the volume label.

Comparison with FAT AND EXT filesystem.

In a FAT system, a file allocated table (FAT) would be the equivalent of the master file table (MFT) and in EXT file system they use a super block which is a record of the characteristics of the filesystem, including its size, the block size, the empty and the filled blocks and their respective counts, the size and location of the inode tables, the disk block map and usage information, and the size of the block groups. The FAT 1 could be compared similarly to MFTMirr whilst in EXT file systems backup copies of the superblock are stored in block groups throughout the filesystem. In NTFS the boot sector is located in the first and last sectors. In FAT file system it is located in the first sector. Both NTFS and FAT do not use inodes when allocating data but EXT does.

NTFS Metadata:

$STANDARD_INFORMATION

This exists for all files and holds details such as timestamps, version and security IDs.

$FILE_NAME

This Attribute stores the name of the file attribute and is always resident. Every file and directory in the system has a File_Name attribute.

$DATA

$DATA stores the contents of each file. This attribute has a no minimum or maximum size.

$ATTRIBUTE_LIST

When MFT is full, the attributes which can be made non-resident are moved out of the MFT. If this is done and the MFT space is still not holding all the necessary information an $ATTRIBUTE_LIST attribute can be used. The remaining attributes are placed in a new MFT record and the $ATTRIBUTE_LIST holds the information of where to find them.

$SECURITY_DISCRIPTOR

The function of the security descriptor is to prevent unauthorised access to files. It does this by storing: The owner of the file, permissions granted to other users by the owner and what actions are being logged. It has an MFT entry of 9.

Comparison with FAT AND EXT filesystem.

In FAT metadata descriptive data such as last access time is held. FAT metadata can tend to hold more non-essential data in comparison to NTFS and EXT. An address for each metadata entry is held in a table, sometimes fixed in size sometimes not.

EXT file systems give data's inodes along with the block pointer to which will point you directly to the associated block in the superblock. In comparison to NTFS and FAT, EXT has a greater attribute field and holds more additional data.

NTFS Application:

Disk Quotas

The function of Disk Quotas is to limit the amount of space a user can take up with certain files, these quotas are set by the administrator.

Journaling

Journaling is used to help drives recover from power failures. Logging of activities before they are preformed allows the file system to be recovered. A journal entry is made every time the metadata is changed, noting the changes made. The journal file is found at MFT entry 2.

Comparison with FAT AND EXT filesystem.

In FAT systems journaling works on the application category. It works in the same way as in NTFS, saving and noting the changes of metadata.

In EXT the aim of journaling is for records to updates to the file system to enable faster crash recovery. In contrast to the other two file systems with EXT it records what block updates will occur and when the update process is complete.

NTFS Content:

Clusters

A NTFS file is made up of clusters, some are resident some are non-resident. A cluster is the smallest logical amount of disk space that can be allocated to hold a file.

$BitMap

$BitMap lists the allocation status of a cluster. Every bit in the $DATA attribute is associated to one logical cluster address. A bit set to 1 indicates the cluster is allocated, a bit set to 0 indicates the cluster is unallocated. It is located at entry 6 in the MFT.

$BadClus

$BadClus's function is to track sectors that are damaged on the drive. Just as the $BitMap, it holds a list of available and not available sectors in its $DATA attribute. This exists at entry 8 in the MFT.

Comparison with FAT AND EXT filesystem.

On FAT32 small files can waste lots of space as cluster sizes become large to make the limited number of usable clusters fit across a larger drive. NTFS can have many more clusters, thus they can each be smaller, allowing less wastage on small files. Data units exist in allocated or unallocated states.

EXT files and directory contents block are equivalent to cluster block sizes. All blocks belong to a block group.