

# 4

## Computer Systems

As we discussed in previous chapters, digital forensic investigators must control the environment they operate in. The diversity of computer hardware, operating systems, and filesystems requires the digital forensic investigator to have a firm understanding of all the different and potential configurations they may encounter. This requires the digital forensic investigator to have procedures or controls to protect the integrity of digital evidence and the processes used to examine it. If you do not understand the boot process and how the system reacts when it starts or which filesystem is used on storage devices, you could make a fatal mistake. In addition, you must understand how they work together. Failure to understand these essential components could lead you to alter the digital evidence. You will also find that you will be less effective when you testify in judicial or administrative proceedings.

In this chapter, we will cover the following topics:

- Understanding the boot process
- Understanding filesystems
- Understanding the NTFS filesystem

# Understanding the boot process

To control the environment as we start our investigation, we must understand the environment. Here, digital evidence is being stored, created, and accessed. In most cases, this will be a computer system. I use the term “computer system,” which comprises the operating system, the filesystem, and the hardware bundled together to create a computer. To be effective, you must understand the physical media the data is stored on, the filesystem used on the storage device, and how that data is tracked and accessed while on the storage device.


Once you understand the process, you can then implement controls to protect the integrity of the digital evidence.

So, what is the boot process? When you push the power button and electricity energizes the system, commands are issued. As it executes the commands, the system is taking steps (like on a ladder) to achieve the goal of a running operating system. If something breaks any of those steps, the system will not load.

The first step is the **Power-On Self-Test (POST)**; the CPU will access the **Read-Only Memory (ROM)** and the **Basic Input/Output System (BIOS)** and test essential motherboard functions. This is where you hear the beep sound when you turn the power on to the computer system. If there is an error, the system will notify you of the error through beep codes. If you do not have the motherboard manual, a Google search will help determine the meaning of the specific beep code.

Once the **POST** test has been successfully completed, the BIOS is activated and executed. Note that the system has not accessed the storage media. This

is because all the program executions occur at the motherboard level and not in the storage devices. The user can access the BIOS by using the correct key combination displayed on the screen.



**Note**

The time allowed for you to hit the correct key can sometimes be relatively short. If you are unsuccessful, the system will continue booting and accessing the storage device. If you are trying to access the suspect's computer system, disengage the storage devices if they are accessible before starting the process. This will ensure that you are not booting to the suspect's storage device and destroying evidence.

The BIOS will have the basic information of the system: the amount of RAM, the type of CPU, information about the attached drives, and the system date and time. The easiest way to document this information is to photograph it as it is displayed on the screen. This is also where you can change the boot sequence. Typically, the system checks the CD/DVD first and then the designated hard drive. This is where you would be able to change the setting of the boot device when we create the boot media later in the chapter. Changing the boot device tells the BIOS to access the device we provide and not the suspect's device.

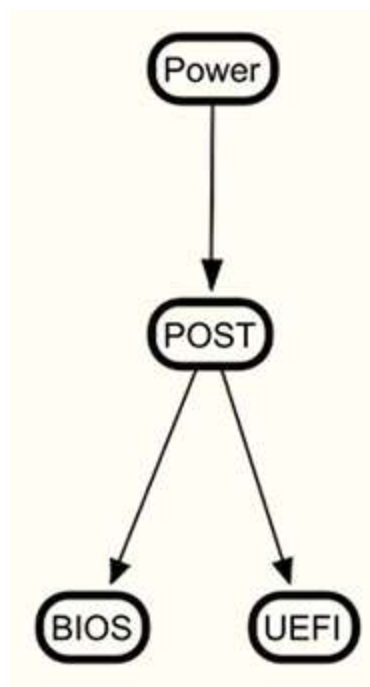
In 2010, the BIOS function was replaced by the **Unified Extensible Firmware Interface (UEFI)**. It provides the same service as the BIOS, but has been enhanced, as follows:

- By providing better security at the preboot process

- Faster startup
- Will support drives larger than 2 TB
- Support for 64-bit device drivers
- Support for the **GUID Partition Table (GPT)**

The Secure Boot feature allows us to use authenticated operating systems when booting the computer system. This can be an issue if you attempt to use an alternative booting device.

As you can see in the following diagram, once the power is turned on and it has completed the POST test, depending on the system, it may boot with the BIOS, or it may boot with the UEFI scheme:



*Figure 4.1: Boot process*

The BIOS will look for the **Master Boot Record (MBR)** of the boot device. The MBR is located at sector zero and holds information about the partitions, the filesystems, and the boot loader code for the installed

operating system. Once the MBR is found in the boot loader and activated, control is then passed over to the operating system to complete the booting process.

The UEFI will look for the GPT; the GPT will have a protective MBR to ensure legacy systems will not mistakenly read this as being unpartitioned and overwrite the data. It will also contain the partition entries and backup partition table header. A GPT disk can contain up to 128 partitions for a Windows operating system. Like in the BIOS scheme, once the active partition and boot loader have been found, the operating system will take over the booting process.

Since you now understand the boot process, we still want to control the boot environment by creating forensic boot media, which we will discuss next.

## **Forensic boot media**

It is a widespread practice to remove the hard drive from the system to create a forensic image. However, sometimes, the investigator cannot remove the storage device from the system, and they need to create a forensic image of the storage device. To accomplish this task, you need to use a bootable CD/DVD or USB device to create a forensic environment to create a forensic image.

Using boot media, you will want to ensure that it will create that sound forensic environment and not cause any changes to the source device. As we discussed during the boot process, we want to intercept any potential changes to that source device, and we want to have the system boot inside an environment we control. While it is still possible to boot using a CD/DVD, finding systems without an optical drive is becoming more

common. Without an optical drive, we must use a boot USB device to create a sound forensic environment to access the storage device.

Linux is a standard operating system that is used to create a USB-based (live) operating system to create the forensic environment needed to examine these devices. As discussed in *Chapter 3, Acquisition of Evidence*, PALADIN is one such tool. It is freely available to download and purchase if you wish to have it preinstalled on a USB device. Sumuri also provides some limited technical support in the operation of PALADIN.

There is also a Windows-based bootable environment known as **WinFE (Windows Forensic Environment)**. WinFE was developed by Troy Larson in 2008 and has spawned other tools such as Mini-WinFE, which was developed by Brett Shavers and Misty (<http://reboot.pro/files/file/375-mini-winfe/>). The benefit of using the Windows bootable environment is that you now have Windows-based forensic tools. It is possible to run X-Ways or FTK Imager from this secure environment. I would not recommend using a tool that is resource-heavy. What I mean is that some forensic suites such as EnCase Forensic or FTK require significant resources to run effectively. X-Ways can be run from a USB device, as can some artifact-specific tools like RegRipper.

As with any tool or procedure, you must validate it to ensure you are getting the expected results. This means that before you go out into the field and boot a suspect's computer utilizing a forensic USB device, you must test it in the laboratory environment to ensure no changes are made.

Some of the challenges that you, as the examiner, need to be concerned with when using a bootable USB device include the following:

- Ensuring the system will boot to the device and not the internal hard drive by changing the boot order in the BIOS
- In some systems, it's difficult to access the BIOS in the time provided during the boot process
- Ensuring the system can boot to a USB device – some older systems cannot
- Knowing which filesystems the bootable device can write-protect and which ones it cannot
- Dealing with the secure boot feature of the UEFI boot process

As mentioned earlier, secure boot is a security feature of the UEFI process that allows trusted operating systems to boot the system. Therefore, if we want to use a bootable forensic operating system, the secure boot feature must be disabled.

You must enter the UEFI environment by pressing the catch key, such as *F2* or *F12* (this will vary depending on the computer manufacturer). Once you have entered the setup utility, navigate to the **Security** menu (this might vary depending on the computer manufacturer) and disable the secure boot option. Some Linux distributions and WinFE have received signed status and will boot a secure boot-enabled system.

You must document your steps as you go through this process. For example, if you miss hitting the catch key and start the boot process in the host operating system, you must document that it occurred. Even beginning a partial boot will change the timestamps and make entries in various logs in the operating system.

Now that you understand what a bootable forensic device is let's go ahead and create one in the next section.

# Creating a bootable forensic device

To create a bootable forensic device, you will need a USB (I recommend using an 8 GB, or larger, device) and an ISO file for the operating system you wish to install. I will demonstrate using an ISO for PALADIN and free software called Rufus (<https://rufus.ie/>). Rufus is a utility used to create bootable USB devices.

Once you download Rufus, execute the executable and the program will run:



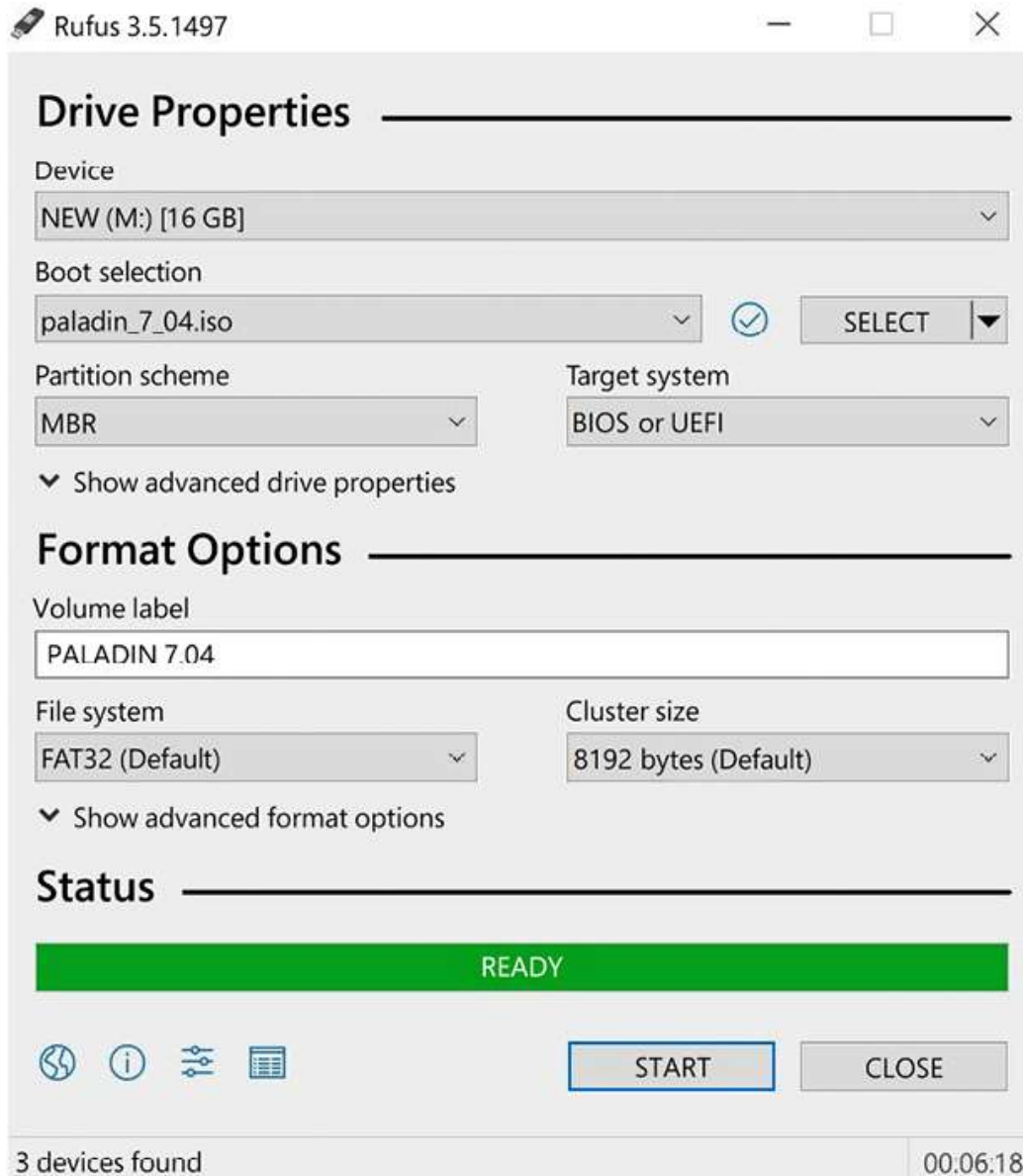


Figure 4.2: Rufus

Something similar to the preceding screenshot (Rufus) will appear, and you will have to select the appropriate choice from the drop-down menus:

- **Device:** This is the destination. It is the USB device you want to host the bootable operating system on.
- **Boot selection:** This will be the “live” operating system. Here, I am using an ISO file for PALADIN 7.04.

- **Partition scheme:** You have a choice of using MBR or GPT. Using MBR will give you greater flexibility in the devices you can boot.
- **Target system:** With the MBR selection for the partition scheme, you can use the device on either a BIOS or UEFI system. If you select GPT for the partition scheme, you can only target UEFI systems.

Under **Format Options**, accept the default values and then click on the **START** button. Once the program completes, you will have a fully functioning, bootable forensic environment.

We have created a forensic boot environment; let's discuss the storage media you will encounter. We will now discuss hard drives.

## Hard drives

The term “physical drive storage device” refers to the hard disk drive itself. That is a physical device that contains platters or solid-state storage that holds data. The term “logical device/volume/partition” refers to the formatting of the physical device. A physical device can contain one or more logical devices/volumes/partitions. It is a common misconception that the term “C drive” refers to the physical device when, in actuality, it refers to a logical partition on the physical device.

Several components make up the interior of the hard drive (as shown in the following figure). If you were to open the case, you would find the hard drive comprised of one or more platters. One or more platters could be stacked together with a spindle in the center. The platters, made of a metal alloy or glass, are coated with a magnetic substance in which the heads magnetically encode information on the platters. The heads can write data on both sides of the platter. The spindles of the hard disk cause the disks to

rotate at thousands of revolutions per minute; the faster the spindle causes the platters to spin, the higher the efficiency of accessing the data encoded on the platters. To read or write data to the platters, the heads are positioned less than **.1** microns from the platter's surface. Additionally, the actuator controls the heads; it swings across the platter, placing the head in the correct position to read/write the data.

The storage devices are manufactured with tight tolerances and can be damaged by sudden sharp movement or a mechanical shock:



*Figure 4.3: Hard drive*

A hard drive can have different interfaces, for example, you may run into some of the following:

- **Small Computer System Interface (SCSI):** An older standard that is typically seen in the corporate environment. Limited to 16 chained devices and will have a terminator at the end of the chain.
- **Integrated Drive Electronics (IDE/EIDE):** An old standard but may still be found in older consumer computer systems.
- **Serial Advanced Technology Attachment (SATA):** A current standard found in many consumer and commercial environments.
- **Serial Attached SCSI (SAS):** A current standard that is typically found in commercial environments.

**Solid state drives (SSDs)** are storage devices that contain no moving parts. Instead, they are made up of memory chips. As we discussed earlier, a traditional hard drive has several moving parts in which to read/write data to the spinning platters. With an SSD storage device, all the data is stored in memory chips, allowing for the following:

- Less weight
- Increased reliability
- Improved data access speed
- Reduced power consumption

For an SSD to function reliably, there are several operations controlled by the firmware of the device. These functions are as follows:

- **Wear leveling:** This spreads the writes across the different chips so that it uses the chips at the same rate.

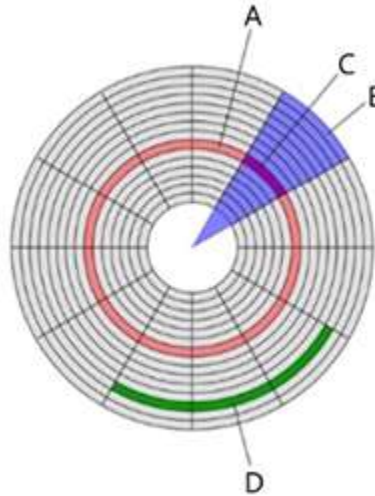
- **Trim:** This will wipe the unallocated space of the device.
- **Garbage collection:** As the firmware scans the memory modules, it may identify pages within the data blocks that have been deleted. The firmware will move the allocated pages to a new block and will wipe the data block so that it can reuse the blocks. The firmware can only delete data in blocks.

The real-world effect on forensics is that we can no longer recover data that is or was in unallocated space. Since these operations are conducted at the firmware layer, these operations start automatically as soon as power is given to the device.

## Drive geometry

The drive geometry of a platter drive details how data is stored on the device; the drive geometry defines the number of heads, the number of tracks, the cylinders, and the sectors per track. The manufacturer performs what it refers to as a low-level format, which creates the basic structure of the disk by defining the sectors and tracks. A track is a circular path on the platter's surface, as indicated in the following diagram. The red circle (**A**) is a single track, and each side of the platter will have its own set of tracks. They then subdivide the track into sectors. A sector (**B**) is the smallest storage unit on the device.

Initially, a sector used to be 512 bytes in size; however, newer disks are being formatted with a sector size of 4,096 bytes:



*Figure 4.4: Drive diagram*

The platters have an addressing scheme to locate the data; originally, **Cylinder, Head, Sector (CHS)** was used. **Cylinder** refers to the vertical axis of the same sectors on all the platters. **Head** refers to the read/write heads; each platter has two heads. Finally, **sector** refers to the number of sectors per track. This addressing scheme worked for large-capacity hard drives; however, as the storage capacity increased, the CHS scheme could not scale because of file size limitations, so **Logical Block Addressing (LBA)** was created. With the LBA scheme, you can address the sectors with a sector number starting from zero.

So, we have discussed the physical components of the device. We will now dive deeper and examine some of the internal aspects.

## **MBR (Master Boot Record) partitions**

Three steps are required before the computer system can use the storage device. First, we have discussed the low-level format conducted by the

manufacturer, but now we will discuss partitioning.

Partitioning occurs when we divide the physical device into logical segments called “volumes.” With the MBR partitioning scheme, we are restricted to four primary partitions. For example, with one physical device, you can have a primary partition used to host the Windows operating system. You can also have a second primary partition that hosts a Linux operating system. Note that you must have a primary partition to boot into an operating system. When a user selects the booted operating system, this is known as the **active partition**.

To get around the partition limit, developers created the extended partition. One of the four partition records is designated as an extended partition, which can then be divided into logical volumes.

As we discussed previously, we can find the MBR at sector zero. The MBR contains the information needed by the system to boot. The MBR will be in sector zero, so it will be no longer than 512 bytes. The partition table will show us which partition is the active partition. Once the starting sector of the active partition is located, the boot process will continue:

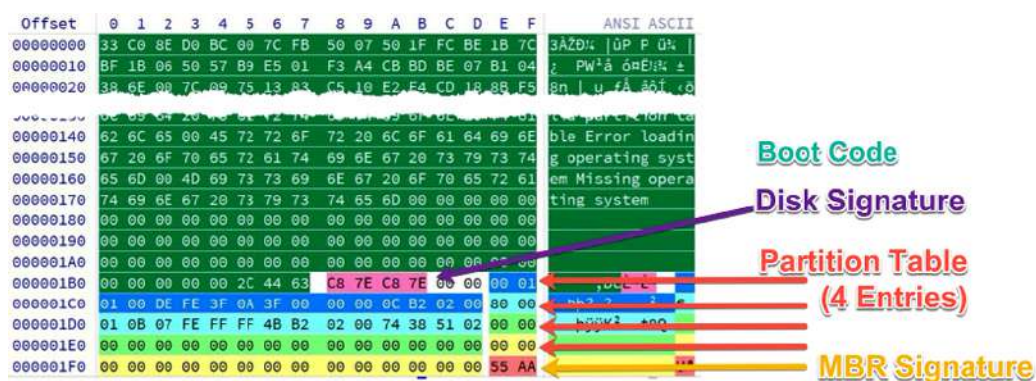


Figure 4.5: MBR map

The preceding MBR map depicts sector zero of a hard disk. This is the MBR for the physical disk. The first 440 bytes are highlighted; this is the boot code. The next 4 bytes are the disk signature and identify the disk to the operating system. The following 64 bytes comprise the partition table. Each 16-byte entry refers to a specific partition. Remember, it restricts us to 4 primary partitions utilizing the MBR partitioning scheme. The final 2 bytes is the signature for the MBR. It identifies the ending of the MBR and will be the last 2 bytes of the sector.

In the following table, I have extracted the four partition tables and reformatted the hex values for easier reading. The first byte will designate which partition is the active partition. A value of `x/80` identifies the active bootable partition.

A value of `x/00` shows the non-active (bootable) partition:

00	01	01	00	DE	FE	3F	0A	3F	00	00	00	0C	B2	02	00
80	00	01	0B	07	FE	FF	FF	4B	B2	02	00	74	38	51	02
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00


Figure 4.6: Partition tables

Typically, you would see the first partition marked as the active partition; in this case, it is the second partition, which is bootable. The next 3 bytes represent a starting sector for the CHS calculation. So, when we examine the partition table, we can see that the physical device has partition 0 and partition 1. The entries for partitions 2 and 3 are zeroed out. This tells us that there are only two partitions on this physical device.



The fifth byte represents the filesystem on the partition. For partition 0, we can see the hex value of **DE**, which tells us that it is part of the Dell PowerEdge Server utilities. Partition 1 has a hex value of **07**, which shows the NTFS filesystem.

If I found the hexadecimal values of **05** or **0f**, that would show an extended partition. We would then have to look into the extended boot records of the extended partitions.



**Note**

You can find a full list of partition identifiers at [https://www.win.tue.nl/~aeb/partitions/partition\\_types-1.html](https://www.win.tue.nl/~aeb/partitions/partition_types-1.html).

The next 3 bytes are the values for the ending sector of the CHS calculation. The next 4 bytes show the starting sector of the partition, and the last 4 bytes show the size of the partition.

The sector values used in the CHS calculation are legacy values for older storage devices. The values showing the start sector and the total number of sectors (partition size) are being used for the current drives using LBA.

Each partition will have a **Volume Boot Record (VBR)** at sector zero of the partition. The system uses the VBR to boot the operating system in that volume. It is an operating system-specific artifact and is created when the partition is formatted.

It will also appear on unpartitioned devices, such as removable media, for example, a USB or floppy disk.

Primary partitions are not the only partitions that you may encounter; you can also encounter an extended partition, which is the subject of the next section.

## Extended partitions

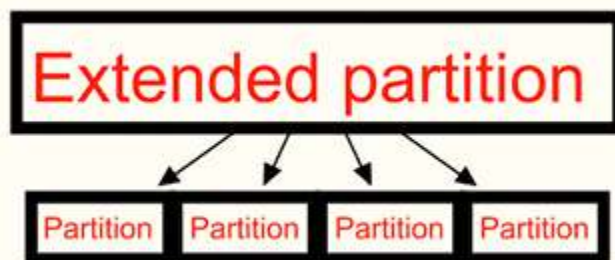
The limitation of the MBR of only allowing four primary partitions resulted in the creation of the extended primary partition. Here, it takes the place of one (and only one) primary partition and enables the user to create additional logical partitions over the four primary partitions.

The following partition map illustrates the replacement of a primary partition with an extended partition:



*Figure 4.7: Partition map*

The following diagram shows the extended partition. Here, the user has created multiple logical partitions within the extended partition boundary:



*Figure 4.8: Extended partition map*

The extended partition will not have a VBR. It will have an **extended boot record (EBR)**, which will point to the first extended logical partition. The first extended logical partition will contain information about itself and a

pointer to the next extended logical partition. In effect, this will create a daisy chain of pointers from one extended logical partition to the next.

We have now covered the aspects relating to the MBR; let's now go over the GPT-formatted aspects.

## GPT partitions

A GUID is a **globally unique identifier** and uses a 128-bit hexadecimal value to identify different aspects of the computer system. A GUID comprises five groups and is formatted as `00112233-4455-6677-8899-aabbccddeeff`, and, while there is no central authority to ensure uniqueness, it is doubtful that you would get a repeating GUID.

RFC 4122 defines the five different GUIDs as follows:

- **Version 1:** Date-time and MAC address: The system generates this version using both the current time and client MAC address. This means that if you have a version 1 GUID, you can figure out when it was created by inspecting the timestamp value.
- **Version 2:** DCE security: This version isn't explicitly defined in RFC 4122, so it doesn't have to be generated by compliant generators. It is like a version 1 GUID except that the first four bytes of the timestamp are replaced by the user's POSIX UID or GID, and the upper byte of the clock sequence is replaced by either the POSIX UID or GID domain. (**UID** stands for **User Identifier**. **POSIX** stands for **Portable Operating System Interface**, which is a set of standards to ensure compatibility between operating systems.)
- **Version 3:** MD5 hash and namespace: This GUID is generated by taking a namespace (for example, a fully qualified domain name) and a

name, converting it into bytes, concatenating it, and hashing it. Once it has specified the special bits such as version and variant, it then converts the resulting bytes into hexadecimal form. The special property regarding this version is that the GUIDs generated from the same name in the same namespace will be identical even if they were generated at different times.

- **Version 4:** Random: The system creates this GUID using random numbers. Of the 128 bits in a GUID, it reserves 6 for special use (version + variant bits) giving us 122 bits that can be filled at random.
- **Version 5:** SHA-1 hash and namespace: This version is identical to version 3 except that SHA-1 is used in the hashing step in place of MD5.

The GPT is a partitioning scheme that is used for newer storage devices and is part of the new UEFI standard. The UEFI standard replaces the BIOS, while the GPT replaces the MBR partitioning scheme.

The GPT petitioning scheme uses LBA, and a protective MBR is found in the physical sector zero. The protective MBR allows for some backward compatibility and helps to remove any issues when dealing with legacy utilities that do not recognize the GPT partitioning scheme. There is no boot code available in the protective MBR. As you can see in the following diagram, this is the first partition entry of the partition table of the protective MBR. The partition is identified by hex value **EE**, which shows it is a GPT partition disk, as shown in the following GPT hex:

[illegible]

Figure 4.9: GPT hex

While the MBR contains the partition table within physical sector 0, GPT houses the partition table header at physical sector 1. The GPT header can be identified by the **EFI** signature of hexadecimal values 45 46 49 20 50 41 52 54, as shown in the following diagram:

00000000200	45 46 49 20 50 41 52 54	00 00 01 00 5C 00 00 00	EFI PART \
00000000210	6C D3 30 12 00 00 00 00	01 00 00 00 00 00 00 00	100
00000000220	80 00 00 00 80 00 00 00	04 00 00 00 00 00 00 00	0- 00
00000000260	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	

Figure 4.10: EFI PART

The following table shows the layout of the GPT header, which you can use to identify the layout of the disk:

### GPT header format


Offset	Length	Contents	
0 (0x00)	8 bytes	Signature ("EFI PART", 45h 46h 49h 20h 50h 41h 52h 54h)	
8 (0x08)	4 bytes	Revision (for GPT version 1.0 (through at least UEFI version 2.7 (May 2017)), the value is 00h 00h 01h 00h)	
12 (0x0C)	4 bytes	Header size	
16 (0x10)	4 bytes	CRC32 checksum of the GPT header	
20 (0x14)	4 bytes	Reserved; must be zero	
24 (0x18)	8 bytes	Current LBA (location of this header copy)	
32 (0x20)	8 bytes	Backup LBA (location of the other header copy)	
40 (0x28)	8 bytes	First usable LBA for partitions (primary partition table last LBA + 1)	
48 (0x30)	8 bytes	Last usable LBA (secondary partition table first LBA – 1)	
56 (0x38)	16 bytes	Disk GUID in mixed endian	
72 (0x48)	8 bytes	Starting LBA of array of partition entries (always 2 in primary copy)	
80 (0x50)	4 bytes	Number of partition entries in array	
84 (0x54)	4 bytes	Size of a single partition entry (usually 80h or 128)	
88 (0x58)	4 bytes	CRC32 checksum of the of the partition table	
92 (0x5C)	*	Reserved; must be zeroes for the rest of the block (420 bytes for a sector size of 512 bytes; but can be more with larger sector sizes)	



Figure 4.11: GPT header format

The GPT partition entries are typically found in physical sector 2. The following diagram shows the GPT partition table entries:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000000400	A4	BB	94	DE	D1	06	40	4D	A1	6A	BF	D5	01	79	D6	AC	»"bN @M;jzO yO-
00000000410	C4	04	7F	C0	41	4E	2D	46	9C	B1	AA	A1	9A	A8	07	FC	Ä ÅAN-Fœ±"i;š" ü
00000000420	00	08	00	00	00	00	00	00	FF	9F	0F	00	00	00	00	00	ÿÿ
00000000430	01	00	00	00	00	00	00	80	42	00	61	00	73	00	69	00	€Basi
00000000440	63	00	20	00	64	00	61	00	74	00	61	00	20	00	70	00	c data p
00000000450	61	00	72	00	74	00	69	00	74	00	69	00	6F	00	6E	00	artition
00000000460	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000000470	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000000480	28	73	2A	C1	1F	F8	D2	11	BA	4B	00	A0	C9	3E	C9	3B	(s*Á ø0 °K É>É;
00000000490	4A	0C	5D	1C	1C	51	E1	4F	94	D5	FC	6D	48	0F	27	86	J ] Qáo"ÖümH '†
000000004A0	00	A0	0F	00	00	00	00	00	FF	B7	12	00	00	00	00	00	ÿ•
000000004B0	00	00	00	00	00	00	00	80	45	00	46	00	49	00	20	00	€EFI
000000004C0	73	00	79	00	73	00	74	00	65	00	6D	00	20	00	70	00	system p
000000004D0	61	00	72	00	74	00	69	00	74	00	69	00	6F	00	6E	00	artition
000000004E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000004F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000000500	16	E3	C9	E3	5C	0B	B8	4D	81	7D	F9	2D	F0	02	15	AE	ãÉä\ ,M }ù-ð •
00000000510	C2	6D	C0	11	34	28	79	4E	87	FA	CD	56	0B	1D	F1	C3	ÄmÄ 4(yN†úÍV ñÄ
00000000520	00	B8	12	00	00	00	00	00	FF	37	13	00	00	00	00	00	, ÿ7
00000000530	00	00	00	00	00	00	00	80	4D	00	69	00	63	00	72	00	€M i c r
00000000540	6F	00	73	00	6F	00	66	00	74	00	20	00	72	00	65	00	o s o f t r e
00000000550	73	00	65	00	72	00	76	00	65	00	64	00	20	00	70	00	s e r v e d p
00000000560	61	00	72	00	74	00	69	00	74	00	69	00	6F	00	6E	00	a r t i t i o n
00000000570	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000000580	A2	A0	D0	EB	E5	B9	33	44	87	C0	68	B6	B7	26	99	C7	¢ Ðëä¹3D†Àh¶•&™Ç
00000000590	21	1F	93	09	AF	7F	A9	44	81	D8	1E	73	C1	4B	9E	AF	! “ - ©D Ø sÁKŽ-
000000005A0	00	38	13	00	00	00	00	00	FF	0F	9E	3B	00	00	00	00	8 ÿ ž;
000000005B0	00	00	00	00	00	00	00	00	42	00	61	00	73	00	69	00	Basi
000000005C0	63	00	20	00	64	00	61	00	74	00	61	00	20	00	70	00	c data p
000000005D0	61	00	72	00	74	00	69	00	74	00	69	00	6F	00	6E	00	artition
000000005E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000005F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Figure 4.12: GPT sector 2

Each partition entry is 128 bytes and provides information about the partitions. The following table shows the contents of the partition entries, which include the partition type GUID, the GUID that is unique to that specific partition, the starting and ending sectors, and the partition name in Unicode:

GUID partition entry format		
Offset	Length	Contents
0 (0x00)	16 bytes	Partition type GUID
16 (0x10)	16 bytes	Unique partition GUID
32 (0x20)	8 bytes	Starting LBA
40 (0x28)	8 bytes	Ending LBA
48 (0x30)	8 bytes	Attribute flags
56 (0x38)	72 bytes	Partition name

*Figure 4.13: GUID*

A partition should hold all the data on the disk within the partition's boundaries; however, there are spaces on the disk outside of the normal partition boundaries where a technical user may hide data. We will discuss those areas next.

## Host Protected Area (HPA) and Device Configuration Overlay (DCO)

HPA and DCO are hidden areas on the hard drive created by the manufacturers. The manufacturer uses the HPA to store recovery and diagnostics tools and it cannot be changed or accessed by the user. The DCO allows the manufacturer to use standard parts to build different products. It will enable the creation of a standard set of sectors on a component to achieve uniformity. For example, the manufacturer might use



one set of parts to create a 500 GB hard drive, and while using the same components, it can also create a 600 GB hard drive. Once again, the user would usually not have access to this location. However, some utilities are freely available and could be used by a user to access these locations and store data.

The following screenshot shows you how an HPA may appear in X-Ways:

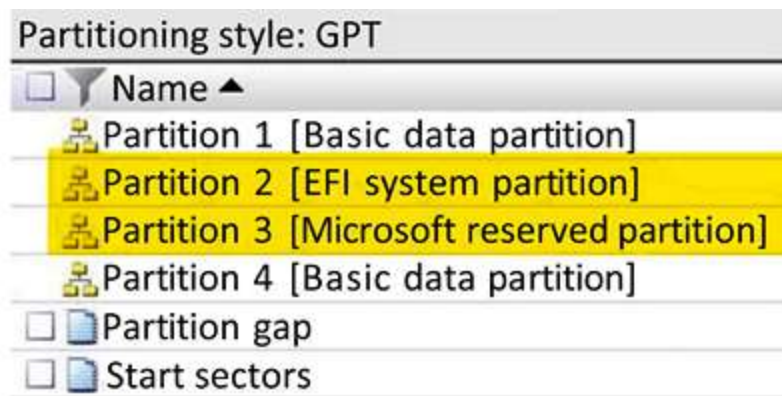


Figure 4.14: HPA 1

The following screenshot shows you how an HPA may appear in FTK Imager:

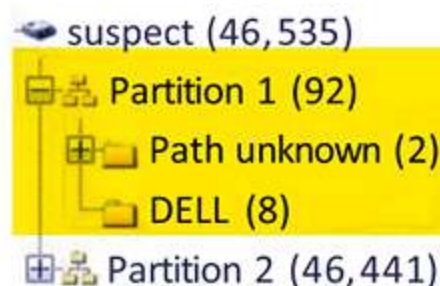


Figure 4.15: HPA 2

Let's move on and discuss some potential filesystems that you may encounter.

# Understanding filesystems

A hard drive can have multiple partitions on it, and, in each partition, there will be (in most cases) a filesystem. There might be hundreds of thousands to millions of files within a partition. The filesystem tracks where every file is and how much space is available within the partition boundaries.

We discussed sectors earlier in the *Hard drives* section; they are the smallest units available to store data. The filesystem stores data based on clusters. Clusters are comprised of one or more sectors. A cluster is the smallest allocation unit the filesystem can write to. There are many filesystems available, and some are restricted to specific operating systems unless the user enables drivers that will allow the operating system to read the filesystem.

We will now look at some of the common filesystems you may encounter.

## The FAT filesystem

The **File Allocation Table (FAT)** filesystem has been around since the early days of home computing, and it is one of the few filesystems that nearly all operating systems can read. It is the de facto standard filesystem for removable devices.

As time has gone by, the FAT filesystem has gone through numerous changes:

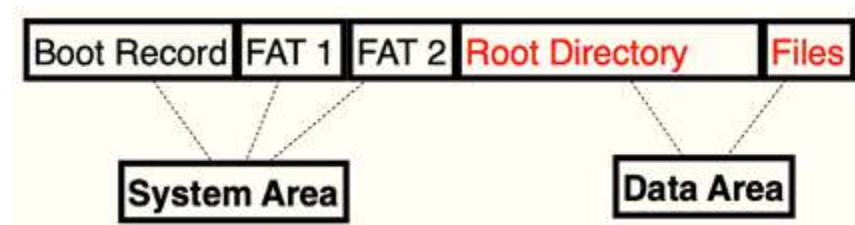
- **FAT12:** The first version was created in 1977 and used 12 bits (hence, the FAT12 designation) to address available clusters. This limited its use to only storage devices that could contain 4,096 clusters. It is rarely seen nowadays, but you might find it on a floppy diskette.

- **FAT16:** This was created in 1984 and used 16 bits (I see a pattern) to address the available clusters. It had the same issues as the FAT12, as it could not be scaled to be used with larger-capacity devices.
- **VFAT:** This was introduced with Windows 95 and added the Virtual File Allocation Table. It added the use of the **long filename (LFN)** and additional timestamps.
- **FAT32:** This uses 28 bits to address available clusters, theoretically allowing for a maximum volume size of 2.2 TB. Microsoft implemented restrictions that limited the volume size to 32 GB with a maximum file size of 4 GB. It is still in use today and can be found on most removable devices.

We will discuss the FAT32 filesystem for the remainder of this section on the FAT filesystem.

The FAT filesystem is laid out in two areas (as shown in the following diagram, Figure 4.16 – FAT areas):

- **System Area:** This stores the volume boot record and FAT tables
- **Data Area:** This stores the root directory and files:



*Figure 4.16: FAT areas*

Next, we will discuss what falls under **System Area**.

## Boot record

We have the **VBR** in the system area. We can find it in logical sector 0 (LS 0), the first sector within the partition boundaries. The boot process creates the VBR when the partition is formatted and contains information about the volume and boot code to continue the boot process for the operating system. If it is a primary partition, the VBR will consist of several sectors, typically sectors 0, 1, and 2, with a backup in sectors 6, 7, and 8. The VBR and backups are stored in a “reserve area,” which is typically 32 sectors before the first file allocation table begins:

EB 58 90	4D 53 44 4F 53	35 2E 30	00 02 08 2A 20
02 00 00 00 00 F8 00 00	3F 00 FF 00 80 00 00 00		
00 E8 3F 00 EB 0F 00 00	00 00 00 00 02 00 00 00		
01 00 06 00 00 00 00 00	00 00 00 00 00 00 00 00		
80 00 29 D9 7C BE FC 4E	4F 20 4E 41 4D 45 20 20		
20 20 46 41 54 33 32 20	20 20 33 C9 8E D1 BC F4		
7B 8E C1 8E D9 BD 00 7C	88 56 40 88 4E 02 8A 56		
61 72 74 0D 0A 00 00 00	00 00 00 00 00 00 00 00		
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00		
00 00 00 00 00 00 00 00	AC 01 B9 01 00 00 55 AA		

Figure 4.17: VBR

In the preceding diagram, we can see a volume boot sector, which helps to decipher the following information:

- **x00**: We will find the jump instructions for the system to continue booting
- **x03**: The OEM ID shows which operating system was used to format the device
- **x0B**: Bytes per sector
- **x0E**: Number of reserve sectors

- `x10`: Number of FATs (this should be 2)
- `x11`: Unused root entries (for FAT32, this should be 0 because the root directory is in the data area)
- `x13`: Number of sectors (this will be 0 if the number of sectors exceeds 65,536)
- `x15`: Media descriptor (`xF8` will show a hard disk, while `xF0` will show a removable device)
- `x16`: Number of sectors per FAT (for FAT32, this should be 0)
- `x18`: Number of sectors per track (this should be 63 for hard disks)
- `x1A`: Number of heads (this should be 255 for hard disks)
- `x1C`: Number of hidden sectors (the number of hidden sectors before the start of the FAT volume)
- `x20`: Number of total sectors (that is, the total sectors for the volume)
- `x24`: Logical sectors per FAT
- `x28`: Extended flags
- `x2A`: FAT version
- `x2C`: The starting root directory cluster (usually, cluster 2)
- `x30`: Location of the filesystem information sector (typically, this is set to 1)
- `x32`: Location of the backup sector(s) (usually, this is set to 6)
- `x34`: Reserved (set to 0)
- `x40`: Physical drive number (`x80` for hard drives)
- `x41`: Reserved
- `x42`: Extended boot signature (this should be `x29`)
- `x43`: Volume serial number (a 32-bit value is usually generated from the date and time; this can track removable devices)

- `x47`: Volume label (this might not be accurate; different OSes may not use this field)
- `x52`: Filesystem type

Next, we will take a look at the file allocation table.

## File allocation table

The next component of the FAT filesystem is the file allocation table, which immediately follows the VBR. There are two file allocation tables (FAT1 and FAT2) by default. FAT2 is a duplicate of FAT1.

The purpose of the file allocation table is to track the clusters and track which files occupy which clusters. Each cluster is represented within the file allocation table starting with cluster 0. The file allocation table uses 4 bytes (32 bits) per cluster entry. The file allocation table will use the following entries to represent the cluster's current status:

- **Unallocated:** `x0000 0000`
- **Allocated:** The next cluster that is used by the file (for example, it represents cluster 7 as `x0700 0000`)
- **Allocated:** The last cluster that is used by the file (`xFFFF FFF8`)
- **Bad cluster:** Not available for use (`xFFFF FFF7`)

A cluster is the smallest allocation unit the filesystem can address. A sector is the smallest allocation unit on the disk. A cluster is made up of one or more sectors. It is very easy to get confused if you are co-mingling those terms. Consider the following cluster example:

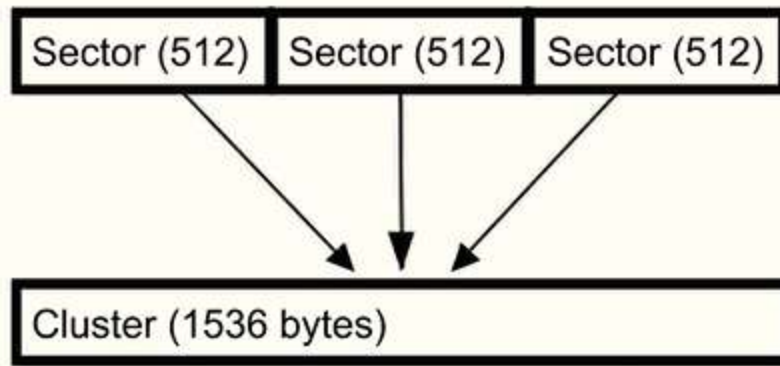


Figure 4.18: Cluster example

As users add files to the data area, the system will update the file allocation table. A file may occupy one or more clusters. Additionally, the clusters may not be sequential, so you could have the data of a file spread in different physical locations on the disk; we typically refer to this as fragmentation.

In the following diagram, we can see a representation of the file allocation table; in this scenario, we have a single file occupying three clusters: **Cluster 4**, **Cluster 5**, and **Cluster 6**. You can see that **Cluster 4** is pointing to **Cluster 5** and **Cluster 5** is pointing to **Cluster 6**. **Cluster 6** has the hexadecimal value for **end of file (EOF)**:

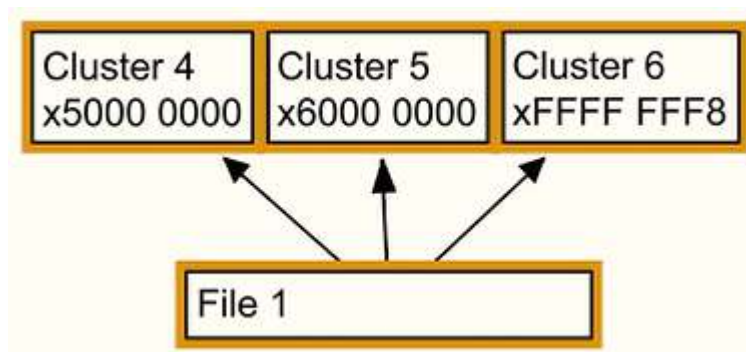
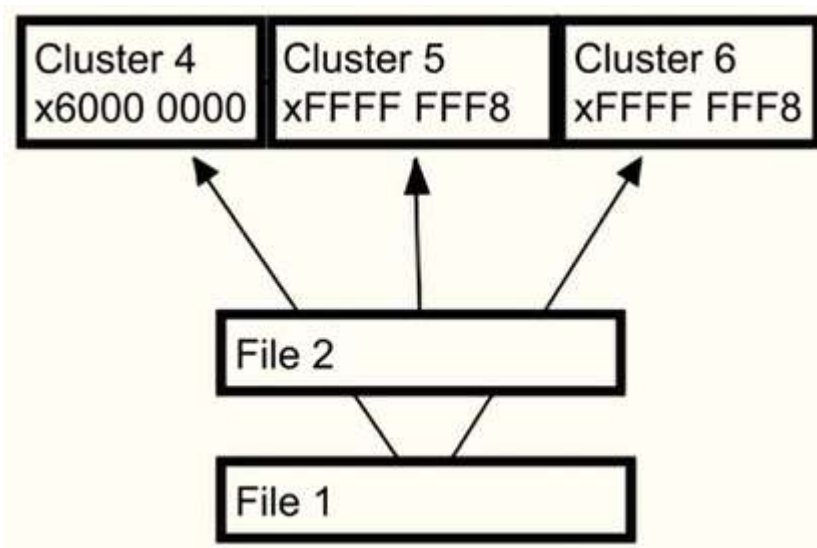


Figure 4.19: Non-fragmented file entry

In the following diagram, we can see a similar representation of the file allocation table with some changes. We now have two files, with File 1 occupying clusters **4** and **6**. We can see that **Cluster 4** is pointing to the next cluster containing the file data, which is **Cluster 6**. This is an example of file fragmentation. File 2 is wholly contained within the cluster boundaries of **Cluster 5**.

**Cluster 5** will not point to a subsequent cluster; instead, it has the EOF hexadecimal value:



*Figure 4.20: Fragmented file entry*

We have covered the system area of the FAT; we will now discuss the data area of the FAT filesystem.

## Data area

The root directory is housed in the data area because, when the system stored it in the system area, it could not grow enough to work with larger-capacity devices. The critical component of the root directory is the



directory entry. If there is a file, directory, or subdirectory, there will be a corresponding directory entry.

Each directory entry is 32 bytes in length and helps track the file's name, starting cluster, and file size in bytes.

In the following diagram, we can see a FAT32 directory with multiple file entries. The filesystem will stop looking for file entries when it runs into a hexadecimal `00`, and all values following the hexadecimal `00` will be ignored:

E5 6C 00 6F 00 6E 00 67	00 66 00 0F 00 D4 69 00	ãl.o.n.g.f...ôì.
6C 00 65 00 6E 00 61 00	6D 00 00 00 65 00 2E 00	l.e.n.a.m...e...
E5 4F 4E 47 46 49 7E 31	54 58 54 20 00 6B B0 6D	ãONGFI~lTXT .k°m
D3 4E D3 4E 00 00 B1 6D	D3 4E 00 00 00 00 00 00	ÓNÓN...±mÓN.....
53 48 4F 52 54 20 20 20	54 58 54 20 18 6B B0 6D	SHORT TXT .k°m
D3 4E D3 4E 00 00 93 6D	D3 4E 00 00 00 00 00 00	ÓNÓN...mÓN.....
24 52 45 43 59 43 4C 45	42 49 4E 16 00 30 B5 6D	\$RECYCLEBIN..0µm
D3 4E D3 4E 00 00 B6 6D	D3 4E 06 00 00 00 00 00	ÓNÓN...†mÓN.....
Unused Entry 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....

Figure 4.21: FAT directory entry

In the following FAT directory map, we can see the layout of the directory entry and a **short filename (SFN)** directory entry with the specific offsets highlighted:

Offset (hex)	Size (Bytes)	Description
x00	1	The first character of the file name or status byte
x01	7	Filename (padded with spaces if required)
x08	3	Three characters of the file extension
x0B	1	Attributes
x0C	1	Reserved
x0D	1	Created time and date of the file
x0E	2	File creation time
x10	2	File creation date
x12	2	Last accessed date
x14	2	Two high bytes of FAT32 starting cluster
x16	2	Time of the Last Write to File (last modified or when created)
x18	2	Date of the Last Write to File (last modified or when created)
0x1A	2	Two low bytes of the starting cluster for FAT32
0X1C	4	File size (zero for a directory)

Figure 4.22: FAT directory map

If the first byte is `xE5`, then the filesystem will consider that entry as deleted. The remaining bytes of the file or directory name will remain, as will the other metadata.

The short filename must conform to the specifications as follows:

- Eight characters are allowed; if there are less than eight characters, then the name will be padded with `x20`.
- Three characters are allocated for the file extension (if there are less than three characters, then the name will be padded with `x20`).
- Spaces and the following characters are not permitted: “+ \* , . / : ; < = > ? [ \ ]”.

The directory entry will always be stored in uppercase. The attribute byte (offset `x0B`) is considered a packed byte, which means the different values

have different meanings.

The following diagram shows that bit values in the `Attribute` flag can be combined, and the resulting hex value will reflect the combinations. If a file had the **READ ONLY** flag and the **HIDDEN** flag, then that would give us a value of `0000 0011`, and, when converted to hexadecimal, we get the value of `x03`:

0000 0001	READ ONLY
0000 0010	HIDDEN FILE
0000 0100	SYSTEM FILE
0000 1000	VOLUME LABEL
0000 1111	LONG FILENAME
0001 0000	DIRECTORY
0010 0000	ARCHIVE

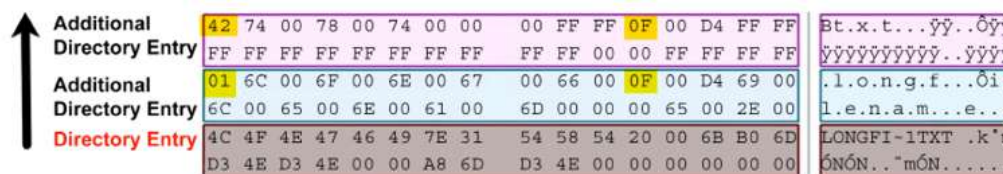
*Figure 4.23: Packed byte*

When we look at the example at the bottom of the preceding FAT directory map, we find the hexadecimal value of 20 at the offset `x0B`; when we convert the hexadecimal into binary, we get `0010 0000`. This tells us that the file is an archive.

We can also encounter an **LFN**; the technique for handling an LFN is a little bit more complicated. We will discuss LFNs in the next section.

## Long filenames

When a user creates an LFN, the system will generate an alias that conforms to the SFN standard. It will format the alias so that the first three characters will become the extension after the file extension dot. The first six characters will be converted to uppercase and used for the alias. The alias will then add a ~ character with a number following it. It will start with the number 1 and increase incrementally if additional files have the same alias name.



Since this is an LFN, the filesystem will create additional directory entries. In this specific case, there will be two additional directory entries to facilitate the use of the LFN. The first byte of each additional directory entry is the sequence byte. The right nibble is the sequence number. As we look at the directory entry depicted in the preceding diagram, the directory entry above the SFN entry has a hexadecimal value of `x01`. Here, the value of `1` tells us that this is the first value in the sequence. When we move up to the second directory entry, we can see that it has a hexadecimal value of `x42`, the right nibble informs us this is the second directory entry for this LFN file. The left nibble of the value, `4`, tells us this is the last directory entry for the file. In each of the LFN directory entries, you will find that the attribute byte is `x0F`.

But what happens when a file is deleted? Well, you may be able to recover the file and its associated metadata. In the next section, we will discuss recovering deleted files.

## Recovering deleted files

When a file is deleted in the FAT filesystem, the data itself does not get changed. Instead, the first character of the directory entry will change to `xE5`, and the file allocation table entries are reset to `x00`. When the filesystem reads the directory entries and encounters `xE5`, it will skip that entry and read from the subsequent entries.

To recover deleted files, we need to reverse the filesystem's process to delete the files. Remember, it has not changed the file contents, and they still physically reside in their assigned clusters. So we now need to reverse engineer the deletion and recreate the file entry and the entries in the file allocation table. To do this, we need to find the first cluster of the file, the file size, and the size of the clusters in the volume.

In the following diagram, we have a directory entry showing that a file has been deleted. We can see `xE5` at the start of the directory entry. (Note that this will require the use of a hex editor to make the changes.)

Then, we must determine the starting cluster, `x00 x08` (but shown as `x08 x00` in the diagram). This value refers to cluster number 8. Then, to determine the file size, take a look at the last 4 bytes, `x27 x00 x00 x00` (remember that the FAT filesystem stores data in little-endian, which means the least-significant byte is on the left, so we would read that value as `x00 x00 x00 x27`, and when we convert it into a decimal, we have a value of 39 bytes for the file size):

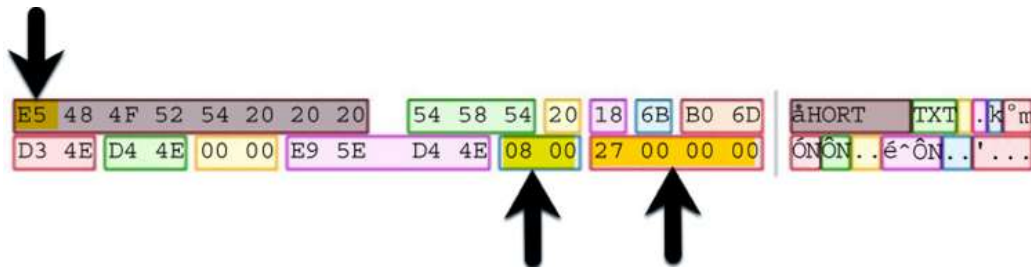


Figure 4.25: Deleted entry

Now we must determine how many sectors make up a cluster and what the sector size is. You will need to go to the boot record to get that information. The boot record shows us that there are 512 bytes per sector, and there are 8 sectors per cluster, which gives us a cluster size of 4,096 bytes (as shown in the following diagram):

Bytes per sector	011	512	15
Sectors per cluster	013	8	114

Figure 4.26: Boot record

This means that our file will only occupy a single cluster. We then go to the file allocation table and look at the entry for cluster 8 and see that it is zeroed out:

```
00 00 00 00 FF FF FF 0F 0B 00 00 00 0C 00 00 00
0D 00 00 00 0E 00 00 00 0F 00 00 00 10 00 00 00
```

Figure 4.27: Deleted FAT

To recover the deleted file, perform the following steps:

1. You need to change the entry in the file allocation table from `x0000 0000` to `xFFFF FFF8` or `xFFFF FF0F`. If this were a larger file, you would need to change the file allocation table entry to point to the next

cluster until you reach the last cluster and the end of the file size. As you are rechainning the entries, if you come to an entry marked as allocated when expecting to find the entry unallocated, you may be dealing with a fragmented file. Another alternative is when the clusters were made available to the filesystem. The system placed a new file in the now-available sectors, which would cause the data to be overwritten. There are not many options available if you run into either one of these situations. If the data is overwritten, then you are stuck. If it is fragmented, you must guess where the next cluster will be, which is not very likely with a large-capacity device.

2. The next step is to return to the directory entry and replace `xE5` with another character. When replacing the `xE5` character of the filename in the directory entry, be careful not to guess what the character is. If you select the incorrect character, you could change the meaning or create a bias with the new filename, which would be improper. When recovering a deleted file, it is recommended that you replace that first character with an underscore or a dash, so there is no misunderstanding about the filename.

When recovering a file with an LFN, it is essential to relink the LFN to the SFN. This is because when the additional directories are created to accommodate the LFN, the system creates a checksum based on the data of the SFN. Therefore, when you change the `xE5` value on the SFN entry, you also want to use the same replacement character for the subsequent `xE5` entries for the LFN directory entries. You link the LFN to the SFN because the SFN directory entry contains information such as the date and time, the starting cluster, and the file size.

It is still possible to recover scraps of data that existed on the disk but no longer have any artifacts in the filesystem. This information will be stored

in slack space, discussed in the next section.

## Slack space

Now is the time to bring up slack space. Remember that the smallest unit the filesystem can write to is a cluster and that clusters are made up of one or more sectors. I keep repeating this because I have seen people who are new to the field get confused about the difference between the two. This is important because files come in a variety of sizes; almost no files will conveniently fit within the cluster boundaries. So, you will have files that spill over into the next cluster. The space between the end of the logical file and the cluster boundary is called “file slack.” This slack space can contain data from the previous file. Until it is overwritten, that data will remain for you to examine.

You might find evidence of documents, digital images, chat history, or emails; any data that has been stored on the device, you may find remnants in slack space after the user has deleted the file.

This concludes the *FAT filesystems* section; next up is NTFS.

## Understanding the NTFS filesystem

The **New Technology File System (NTFS)** is the default filesystem for Microsoft Windows operating systems. FAT32 had some significant shortcomings, which required a more reliable and efficient filesystem, along with additional administrative improvements to help Microsoft remain viable in a corporate environment. They initially designed NTFS for a server environment; however, as the hard drive capacity has increased, it is



now the default filesystem in the commercial and consumer market for the Windows operating system.

NTFS is far more complicated than the FAT filesystem; however, the overall purpose remains the same:

- To record the metadata of a file, that is, the filename, the date timestamps, and the file size
- To mark the clusters the file occupies
- To record which clusters are allocated and which clusters are unallocated

The NTFS filesystem comprises the following system files:

<b>\$MFT</b>	<b>Describes all files on the volume, including file names, timestamps, stream names, and lists of cluster numbers where data streams reside, indexes, security identifiers, and file attributes.</b>
<b>\$MFTMirr</b>	<b>Duplicate of the first vital entries of \$MFT, usually 4 entries (4 kb).</b>
<b>\$LogFile</b>	<b>Contains transaction log of file system metadata changes.</b>
<b>\$Volume</b>	<b>Contains information about the volume, namely the volume object identifier, volume label, file system version, and volume flags.</b>
<b>\$AttrDef</b>	<b>A table of MFT attributes that associates numeric identifiers with names.</b>
<b>\$ (Root file name index )</b>	<b>The root folder.</b>
<b>\$Bitmap</b>	<b>Tracks the allocation status of all clusters in the partition.</b>
<b>\$Boot</b>	<b>Volume boot record.</b>
<b>\$BadClus</b>	<b>A file that contains all the clusters marked as having bad sectors.</b>
<b>\$Secure</b>	<b>Access control list database.</b>
<b>\$UpCase</b>	<b>Converts lowercase characters to matching Unicode uppercase characters.</b>
<b>\$Extend</b>	<b>A file system directory containing various optional extensions, such as \$Quota, \$ObjId, \$Reparse, or \$UsnJrnl.</b>

*Figure 4.28: NTFS table*

To identify a partition with NTFS, we need to look at the MBR or the GPT, depending on which formatting scheme was used. In the following diagram, we can see the MBR for the hard drive and the partition table highlighted after the boot code:

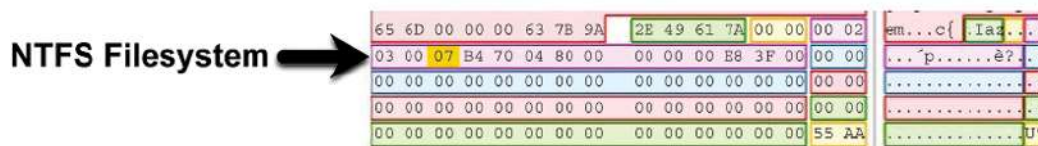


Figure 4.29: NTFS MBR

Looking at the partition table, we can see that there is a single partition, and, at offset decimal 11 from the start of the partition table, we can see the hexadecimal value of 07. As we discussed earlier in this chapter, this is the filesystem identification for NTFS.

With an NTFS-formatted partition, there is no system or data area like we saw with a FAT-formatted partition. Everything in NTFS is considered a file to include the system data. When we look at the VBR, we can see that it contains information for the system to continue the boot process:

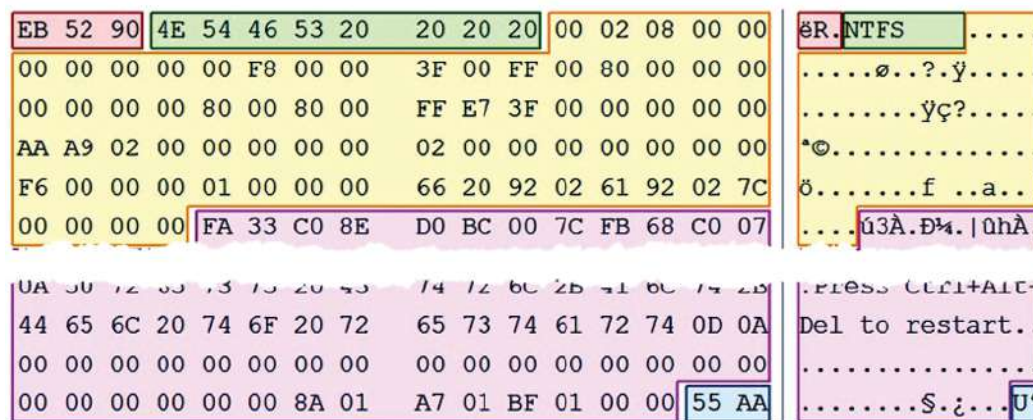


Figure 4.30: NTFS VBR

The information in the VBR is a file; the \$Boot record contains all the information that we would expect to find in the VBR. The following \$Boot diagram shows the data structure for the \$Boot file:

JMP instruction	000	EB 52 90
OEM ID	003	NTFS
▼ BIOS Parameter Block	00B	
Bytes per sector	00B	512
Sectors per cluster	00D	8
Reserved sectors	00E	0
(always zero)	010	00 00 00
(unused)	013	00 00
Media descriptor	015	248
(unused)	016	00 00
Sectors per track	018	63
Number of heads	01A	255
Hidden sectors	01C	128
(unused)	020	00 00 00 00
Signature	024	80 00 80 00
Total sectors	028	4,188,159
SMFT cluster number	030	<a href="#">174,506</a>
SMFTMirr cluster number	038	<a href="#">2</a>
Clusters per File Record Segment	040	246
Clusters per Index Block	044	1
Volume serial number	048	66 20 92 02 61 92 02 7C
Checksum	050	0
Bootstrap code	054	FA 33 C0 8E D0 BC 00 7C
Signature (55 AA)	1FE	55 AA

Figure 4.31: \$Boot record

Arguably, the most important system file in the NTFS filesystem is the **\$MFT** (master file table). The MFT tracks all the files in the volume to include itself. It tracks each file within the MFT through file entries called a file record. Each file record is uniquely numbered and is 1,024 bytes. Each file record starts with a header, with the ASCII text “**FILE**”, and has an EOF marker of hexadecimal **FF FF FF FF**. A new file record is created when files are added to the volume. If a file has been deleted, the record will zero out and make it available for reuse. The MFT will look for an

empty file record and use it before creating a new record. The file record can be reused rather quickly, which would overwrite the previous data in the file record.

As shown in the following NTFS file record example, we can see a file record and file header starting with the ASCII values of **FILE**. If the record were corrupted or had an error, you would see the ASCII value of **BAAD**.

The file header is 56 bytes:

46 49 4C 45	30 00	03 00	39 6B 20 00 00 00 00 00	FILE0...9k .....
01 00	01 00	38 00	01 00	...8...Ø.....
00 00 00 00 00 00 00 00	04 00	00 00	28 00 00 00	.....(...
03 00	00 00 00 00	00 00	10 00 00 00 60 00 00 00	.....`.....
00 00 00 00 00 00 00 00	48 00 00 00 18 00 00 00	00 00 00 00 00 00 00 00	30 00 00 00 80 00 00 00	.....H.....
00 00 00 00 08 01 00 00	00 00 00 00 00 00 00 00	62 00 00 00 18 00 01 00	67 00 66 00 69 00 6C 00	.....0.....
00 00 00 00 00 00 00 00	10 00 6C 00 6F 00 6E 00	65 00 2E 00 74 00 78 00	80 00 00 00 18 00 00 00	.....b.....
00 00 00 00 00 02 00	65 00 6E 00 61 00 6D 00	00 00 00 00 18 00 00 00	00 00 00 00 18 00 00 00	..l.o.n.g.f.i.l.
74 00 00 00 00 00 00 00	00 00 18 00 00 00 01 00	00 00 00 00 18 00 00 00	00 00 00 00 18 00 00 00	e.n.a.m.e...t.x.
80 00 00 00 A0 00 00 00	F3 D0 1C A7 50 97 F4 8A	E2 74 67 93 FC 80 74 47	B7 1C B0 00 00 00 00 00	t.....
FF FF FF FF	82 79 47 11			.....
				óÐ.SP.ô.â tg.íl.tG
				[ [¥ÚÚZ.E . ° .....
				yyyy

Figure 4.32: NTFS file record

In the following NTFS file record map, we can see the data structure of a file record header:



Signature (must be 'FILE')	000	FILE
Offset to the update sequence	004	0x30
Update sequence size in words	006	3
\$LogFile Sequence Number (LSN)	008	2,124,601
Sequence number	010	1
Hard link count	012	1
Offset to the first attribute	014	0x38
Flags	016	01 00
Real size of the FILE record	018	472
Allocated size of the FILE record	01C	1,024
Base FILE record	020	0
Next attribute ID	028	4
<b>ID of this record</b>	<b>02C</b>	<b>40</b>
Update sequence number	030	03 00
Update sequence array	032	00 00 00 00
<b>Attribute \$10</b>	<b>038</b>	
<b>Attribute \$30</b>	<b>098</b>	
<b>Attribute \$80</b>	<b>118</b>	
<b>Attribute \$80</b>	<b>130</b>	
End marker	1D0	0xFFFFFFFF

*Figure 4.33: NTFS file record map*

The file record also contains defined data blocks called file attributes. These store specific types of information about the file. The following file attributes table shows several common file attributes that you are likely to see in almost every record:



Figure 4.35: \$Standard\_Information Attribute

Here is a map of the values you will find in the attribute:

<b>Attribute \$10</b>	<b>038</b>	
Attribute type	038	0x10
Length (including header)	03C	96
Non-resident flag	040	0
Name length	041	0
Name offset	042	0x00
> Flags	044	00 00
Attribute ID	046	0
Length of the attribute	048	72
Offset to the attribute data	04C	0x18
Indexed flag	04E	0
Padding	04F	0
<b>✓ \$STANDARD_INFORMATION</b>	<b>050</b>	
File created (UTC)	050	6/20/2019 1:32 PM
File modified (UTC)	058	6/19/2019 8:45 PM
Record changed (UTC)	060	6/19/2019 8:45 PM
Last access time (UTC)	068	6/20/2019 1:32 PM
> File Permissions	070	20 00 00 00
Maximum number of versions	074	0
Version number	078	0
Class Id	07C	0
Owner Id	080	0
Security Id	084	264
Quota Charged	088	0
Update Sequence Number	090	0

Figure 4.36: File attribute map

**\$File\_Name Attribute (0x30):** The next attribute is the **\$30 File\_Name Attribute**. This attribute stores the name of the file attribute and is always



resident. The maximum filename length is 255 Unicode characters. It is identified by the hexadecimal header of `x/ 30 00 00 00`:

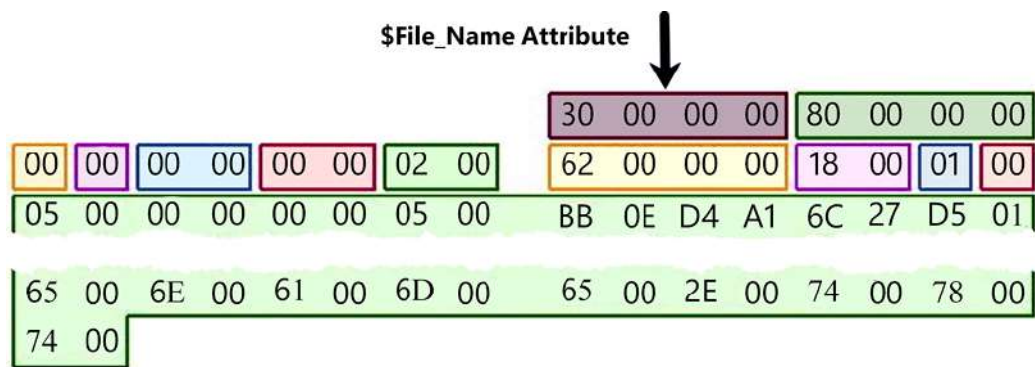


Figure 4.37: \$File\_Name Attribute

The following is a map of the values you will find in the attribute:

<b>Attribute \$30</b>	<b>098</b>	
Attribute type	098	0x30
Length (including header)	09C	128
Non-resident flag	0A0	0
Name length	0A1	0
Name offset	0A2	0x00
> Flags	0A4	00 00
Attribute ID	0A6	2
Length of the attribute	0A8	98
Offset to the attribute data	0AC	0x18
Indexed flag	0AE	1
Padding	0AF	0
<b>▼ \$FILE_NAME</b>	<b>0B0</b>	
Parent directory file record number	0B0	5
Parent directory sequence number	0B6	5
File created (UTC)	0B8	6/20/2019 1:32 PM
File modified (UTC)	0C0	6/20/2019 1:32 PM
Record changed (UTC)	0C8	6/20/2019 1:32 PM
Last access time (UTC)	0D0	6/20/2019 1:32 PM
Allocated size	0D8	0
Real size	0E0	0
> File attributes	0E8	20 00 00 00
(used by EAs and reparse)	0EC	0
File name length	0F0	16
File name namespace	0F1	0
File name	0F2	longfilename.txt

Figure 4.38: Filename attribute map

**\$Data Attribute (0x80):** The following attribute for this entry is the **\$80** Data Attribute. The data attribute contains the file's contents or points to where the contents are in the volume. This attribute is the file data itself.

If the data attribute is resident, we only see the attribute header and the resident content header. The resident content of the attribute is the file's data. Only tiny files have a resident data attribute. We will discuss resident versus non-resident data later on in this chapter.

You may find multiple data attributes per file. In this record, the second \$80 Data Attribute, Dropbox, has added some information to the file:

**\$Data Attribute**

00	00	18	00	00	00	01	00	80	00	00	00	18	00	00	00
80	00	00	00	A0	00	00	00	00	16	18	00	00	00	03	00
14	81	05	52	12	43	12	81	0C	23	41	03	82	82	AC	0A
F3	D0	1C	A7	50	97	F4	8A	E2	74	67	93	FC	80	74	47
5B	5B	A5	DA	DA	5A	00	CB	B7	C1	B0	00	00	00	00	00

Figure 4.39: \$Data Attribute

The following is a map of the values you will find in the attribute:

<b>Attribute \$80</b>	<b>130</b>	
Attribute type	130	0x80
Length (including header)	134	160
Non-resident flag	138	0
Name length	139	22
Name offset	13A	0x18
> Flags	13C	00 00
Attribute ID	13E	3
Length of the attribute	140	83
Offset to the attribute data	144	0x48
Indexed flag	146	0
Padding	147	0
Attribute Name	148	com.dropbox.attributes
▼ <b>\$DATA</b>	<b>178</b>	
Data	178	78 9C AB 56 4A 29 CA 2F 48
End marker	1D0	0xFFFFFFFF

Figure 4.40: Data attribute map

When examining the \$Data Attribute 0x80, the filesystem may store the file's contents within the MFT file record itself. Since the file record is 1,024 bytes long, it would have to be a tiny file. When the data content of the file fits within the file record, it is called “resident data”:



Figure 4.41: Resident data file

In the current example, we have a file named `resident.txt` that is 23 bytes. This is smaller than the 1,024 bytes of the file record. To look at the data of the file, we need to look at the `$Data Attribute 0x80` of the file record, as follows:

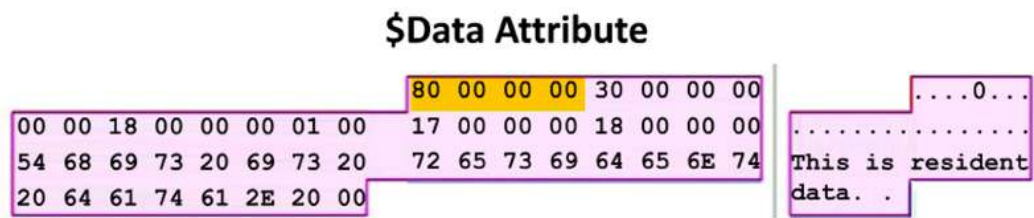


Figure 4.42: Resident data example

On examining the attribute, we can see the ASCII and hex representation of the file content we observed in the preceding resident data example. When dealing with a non-resident file, such as the one depicted in the following diagram, we can see that the `nonresident.txt` file, which is 145 KB in size, is larger than the 1,024-byte file record:

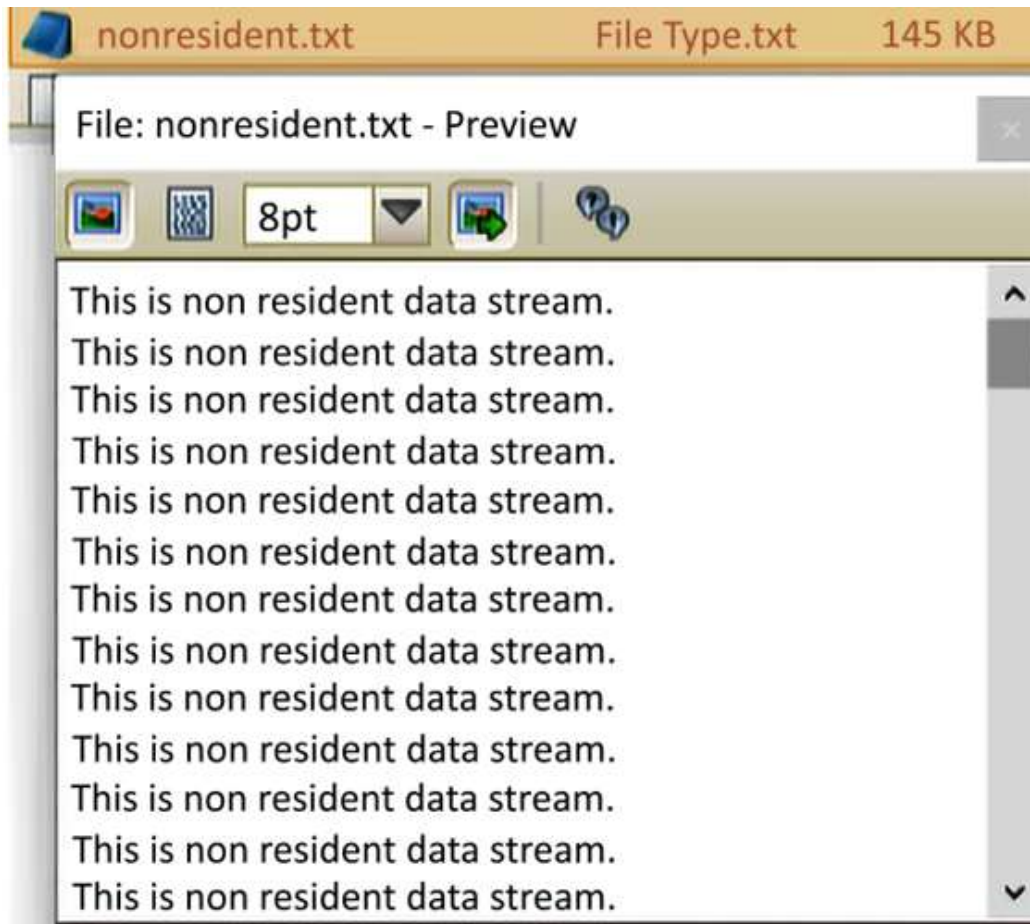


Figure 4.43: Non-resident data

When you look at the `$Data Attribute 0x80` of the file, as shown in the preceding diagram, we do not see the contents of the file, but we have pointers to the location of the file within the volume boundaries. We consider this to be non-resident content. Once the content of the attribute becomes non-resident, it can never become resident again. We commonly refer to the pointers in the file record of the attribute as a “run list” for the data runs of the non-resident data:

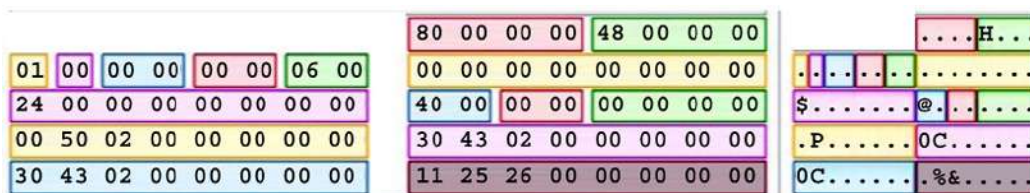


Figure 4.44: Non-resident data example

You can have a single data run, or multiple data runs, within the `$Data` Attribute `0x80`. Deciphering the run list for the data runs can be tricky. In the following run list, we have the `$Data` Attribute `0x80` with two run lists:

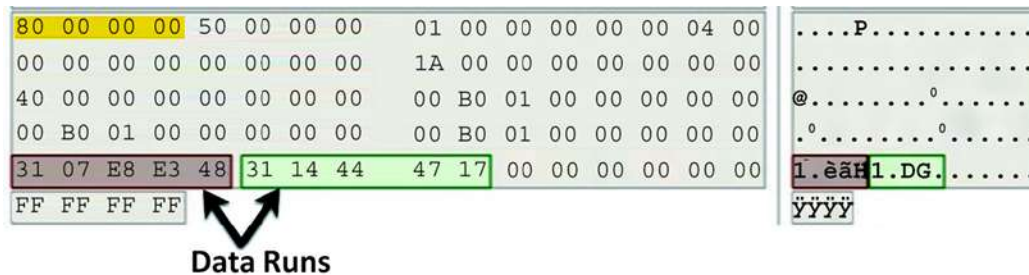


Figure 4.45: Run list

If the file is not fragmented, then you will have one run list pointing to the data run in the volume. If the file is fragmented (which is very common), then you will have multiple run lists providing information about the starting cluster for each fragment. I have taken the two run lists highlighted in the preceding list and created the following chart:

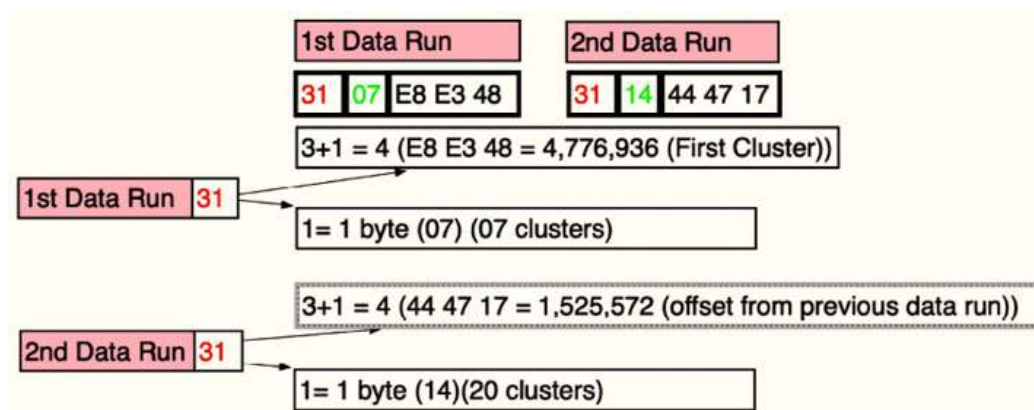


Figure 4.46: Run list map

The first run list comprises the hexadecimal values of `31 07 E8 E3 48`. Take the first byte of the header (`x/31`) and add the left and right nibbles ( $3+1 = 4$ ). 4 is the number of bytes in the run list entry (this is `x/07 E8 E3 48`).

The right nibble (`x/1`) tells us that 1 byte represents the number of clusters being used for this fragment. We find a value of `x/07` in the length field, which represents 7 clusters for this fragment. The left nibble (`x/3`) informs us that 3 bytes (`x/E8 E3 48`) will represent the logical starter cluster of the fragment. At the end of the first run, we have a second run list of `x/31 14 44 47 17`.

Like the prior run list, we take the first byte of the header (`x/31`) and add the left and right nibbles ( $3+1 = 4$ ). 4 is the number of bytes in the run list entry (which is `x/14 44 47 17`). The right nibble (`x/1`) tells us that 1 byte represents the number of clusters being used for this fragment. We find a value of `x/14` in the length field, which represents 20 clusters for this fragment.

The left nibble (`x/3`) informs us that 3 bytes (`x/44 47 17`) will represent the offset from the previous run list cluster. This process will keep going until the system hits `x/ 00 00 00 00`, which shows the end of the run lists.

That concludes our adventure into the world of NTFS. If you find yourself with a headache, you are not alone! This is just the basics of the filesystem. You can find entire books that have been written about NTFS, if you want to go into much greater detail.

## Summary



In this chapter, we looked at how physical disks are constructed and prepared in order to store data. We discussed different partition schemes and how they address the creation of logical partitions. We also learned how filesystems differ and how data is organized.

In the next chapter, we will learn about the computer investigative process and how to analyze timelines, analyze media, and perform string searching for data.

## Questions

1. Newer computer systems utilize the BIOS booting method.
  - a. True
  - b. False
2. A UEFI-based computer system will utilize \_\_\_\_\_ to boot from.
  - a. MBR
  - b. VBR
  - c. GPT
  - d. LSD
3. A cluster is the smallest storage unit on a hard drive.
  - a. True
  - b. False
4. An MBR-formatted disk can have more than four primary partitions.
  - a. True
  - b. False

5. A FAT32-formatted partition is laid out in two areas: a system area and a \_\_\_\_\_ area.
- a. Disk
  - b. Doughnut
  - c. Data
  - d. Designer
6. In a FAT32-formatted partition, the root directory is in the system area.
- a. True
  - b. False
7. In a NTFS-formatted partition, the filename is stored in the \_\_\_\_\_ attribute.
- a. Standard information
  - b. Filename
  - c. Data
  - d. Security descriptor

The answers can be found at the end of the book under *Assessment*.

## Further reading

Carrier, B. *File System Forensic Analysis*. Addison-Wesley, Reading, PA., Mar. 2005 (available at <https://www.kobo.com/us/en/ebook/file-system-forensic-analysis-1>).

# Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/CyberSec>

