Everything I know about NTFS

NTFS, the MFT, File Deletion and Recovery

Introduction:

These musings were born out of a curiosity about just how Windows, and its NTFS file system, manages live and deleted files. NTFS is an immense subject, and this page only scratches at a few aspects of it - in particular the MFT. Although much of the detailed information at a technical level is available in some form or other, not a lot that I've found is in a coherent and complete form, and a great deal is quite often old, confusing, incorrect, contradictory or a mixture of some or all of these. Of course I can't say whether what I've constructed isn't also confusing or incorrect, and in a few years it will be old too.

This isn't a novel, so I didn't make it up as I went along. It is taken from numerous corporate and private technical articles, forums and other works (including as much as I can from Microsoft, errors and all) and many hours trying to grasp what I was reading, and investigating, correcting, testing and verifying what I was writing. I am obliged to those I have borrowed from, and will also be obliged to those who point out any errors without any reward apart from that of contribution. I've tried to explain what happens inside NTFS: why it does will have to remain with Microsoft.

Software and hardware:

Although this article was written in early 2011 onwards, it was produced on, and pertains to, a desktop PC built in 2006 with what is now rather modest processing power, memory and disk capacity (3.0 ghz, 1.0 gb and 160 gb). As with all PCs it is based on Intel x86 architecture, which knowledge can save hours of frustration when looking inside system and other files. Most of the detail was produced whilst the PC was running Windows XP SP3 Home Edition, a still widely used operating system. However this been superseded by first Vista, then Win7, and now Win8. In early 2013

the PC's operating system was upgraded to Windows 8 Pro, which surprisingly runs rather well on such an old pc.

There are, of course, some changes made to NTFS since XP: Vista introduced Transactional NTFS, NTFS symbolic links and self-healing functionality, though - according to Wikipedia - those owe more to additional functionality of the operating system than to NTFS itself. Notwithstanding these changes, and the radical overhaul that came with Windows 8, the NTFS version number in Vista, Win7 and Win8 remains as it is in XP at 3.1 - the NTFS v3.1 on-disk format is unchanged. This article deals only with NTFS v3.1.

The only additional software applications I have used in my examination of NTFS are Piriform's excellent Recuva, which can list both live and deleted files and their cluster allocations, and very easily recover (i.e. copy) either live or deleted files, including system files, to a safe place for later examination; and WinHex, which displays a fairly accessible and readable hex view of system or user files along with other relevant information.

Recuva is free from www.piriform.com, and WinHex is also free in an evaluation version from www.winhex.com, although it does have an occasional and irritating pop-up reminding you of this. The evaluation version has no time limit and is fully functional except for the ability to write to the files, which is no bad thing.

Page specifics:

Data is held on storage in some form that represents bits, displayed by editors as hex, and discussed - in the main - in character or decimal form. On this page - unless obviously to the contrary - binary will rarely be shown, hex will be in the form 0x001234, and decimal as a simple number. Little-endian - of which more later - will be shown as 5C 01 00.

All the figures reproduced here are entirely my own work and the data is taken from my own pc. Whilst the Microsoft layouts are, I hope, accurate there is nothing to be gained from trying to interpret any user data - it is all mine.

NTFS Versions:

Since New Technology File System (NTFS) arrived in 1993 there have been five released versions (the alternative names are due to the NTFS version sometimes being aligned with the O/S version):

- v1.0 with NT 3.1, released mid-1993

- v1.1 with NT 3.5, released autumn 1994
- v1.2 with NT 3.51 (mid-1995) and NT 4 (mid-1996) (occasionally referred to as "NTFS 4.0", because O/S version is 4.0)
- v3.0 from Windows 2000 (occasionally "NTFS V5.0")
- v3.1 from Windows XP (autumn 2001; occasionally "NTFS V5.1"), Windows Server 2003 (spring 2003; occasionally "NTFS V5.2"),Windows Vista (mid-2005; occasionally "NTFS V6.0"), Windows Server 2008, Windows 7 (2009) and Windows 8 (2012)

How to establish what version is running on a particular system is explained later.

Getting started:

Despite what many may think to the contrary, a hard disk is physically formatted into tracks, clyinders and sectors at the factory, forever unalterable by man or beast. The overwhelming majority of disks in use today will have a fixed sector size of 512 bytes. I am not considering solid state drives, or RAID, or other esoteric devices.

To boot up an x86-based computer two areas on the disk are critical: they are:

- The Master Boot Record (MBR), which is located at sector 0 of cylinder 0, head 0, the first physical sector of a hard disk and is not part of any partition.
- The Volume Boot Record (VBR), which is located at sector 0 of each partition

These sectors contain both executable code and the data required to run the code. The boot sector of a non-partitioned disk is a VBR, there being no MBR.

The Master Boot Record:

There is often confusion between the Master Boot Record (MBR) and the Volume Boot Record (VBR). The MBR sits on the very first sector of the physical disk, and describes the partitions on the disk, and the VBR sits at the first sector of each partition, and describes that particular partition. All Windows operating systems require the disk to have at least one primary partition.

The Master Boot Record is created when the disk is partitioned. It contains a small amount of executable code called the master boot code, and the partition table for the disk. At the end of the MBR is a 2-byte structure called a signature word or end of

sector marker, which is always set to 55 AA. A disk signature, a unique number at offset 0x01B8, identifies the disk to the operating system.

The master boot code performs the following activities:

- Scans the partition table for the active partition
- Finds the starting sector of the active partition
- Loads a copy of the volume boot record from the active partition into memory
- Transfers control to the executable code in the volume boot record

The partition table at offset 0x1BE in the MBR is used to identify the type and location of partitions on a hard disk, and has a standard layout independent of any operating system. There can be four partition table entries, each 16 bytes long, which can hold either four primary or three primary and one extended partitions. The partition table holds information to identify the type, size and location of the partition, and whether it is active or not. In each entry in the partition table is:

- 0x0 - Boot Indicator: 0x00 inactive partition, 0x08 active partition
- 0x1 - Cylinder/Head/Sector address of first sector in the partition, 3 bytes
- 0x4 - System ID: 0x07 NTFS
- 0x5 - CHS address of last sector in the partition, 3 bytes

The Volume Boot Record:

The Volume Boot Record (VBR) is located at logical sector zero in the active partition: the VBR followed by the boostrap code occupy the first 16 sectors of the partition. The VBR occupies the first sector, and the operating system loader (NTLDR up to and including Windows XP, winload.exe and the Windows Boot Manager in Vista onwards) occupy subsequent sectors.

Booting the pc:

When a pc is booted up there is a defined process to go through to present a usable system. In a rather extreme simplification, on a powerup up or reset the CPU's registers will be initialised with default values and the Extended Instruction Pointer (EIP), which holds the address of the instruction being executed by the cpu, will be set. This initial instruction is a jump to the BIOS entry point. The BIOS code runs a

power-on self test (POST) and then locates the Master Boot Record at sector 0 on the disk.

The BIOS loads the MBR into RAM and transfers execution to the MBR boot code, which in turn checks the partition table within the MBR for an active partition. The MBR code loads the VBR code from the selected partition, which in turn loads the operating system kernel, completing the startup procedure.

System Files:

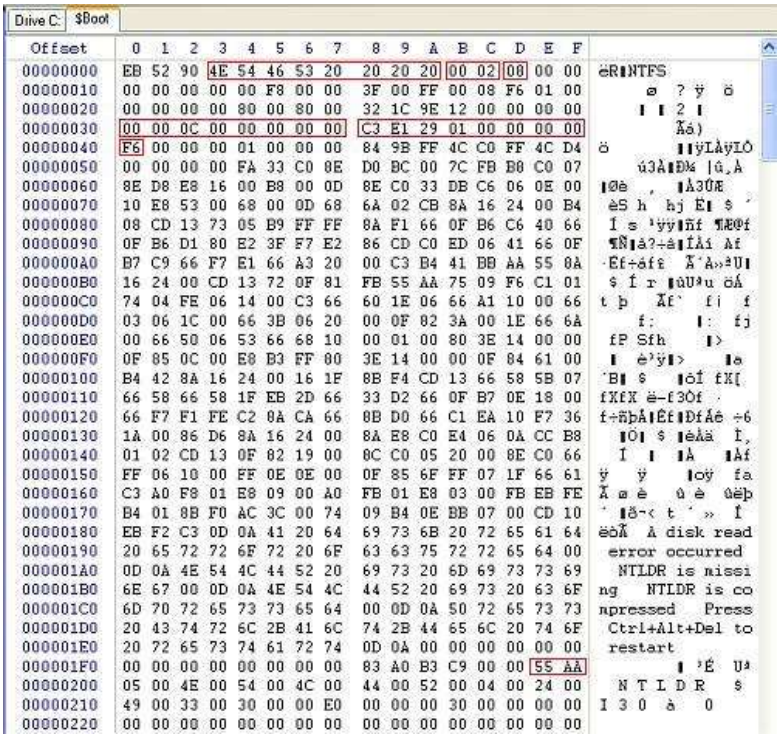
Now the pc has been booted into an NTFS partition we can see a number of system or meta files with names beginning with $ and a capital letter. The system files can be listed with WinHex or Recuva and include:

- $AttrDef
- $BadClus
- $Bitmap
- $Boot
- $LogFile
- $MFT
- $MFTMir
- $Secure
- $UpCase
- $Volume

$Boot file (VBR):

The 16 sectors (two clusters) at logical sector zero in the active partition, containing the VBR and the bootstrap code, are defined in NTFS as the $Boot file, and can be examined as such.

First sector of $Boot (VBR)



Volume Boot Record layout

Offset	Size	Description
00	Byte 3	Jump Instruction
03	LongLong	OEM ID
0B	Word	Bytes per Sector
0D	Byte	Sectors per Cluster
0E	Word	Reserved Sectors. Always 0 as NTFS places the boot sector at the beginning of the partition. If the value is not 0, NTFS fails to mount the volume.
10	Byte 3	Value must be 0 or NTFS fails to mount the volume.
13	Word	Value must be 0 or NTFS fails to mount the volume.
15	Byte	Media Descriptor. F8 Hard Disk, F0 HD Floppy. A legacy of MS-DOS FAT16 disks and not used in Windows XP onwards.
16	Word	Value must be 0 or NTFS fails to mount the volume
18	Word	Not used or checked by NTFS
1A	Word	Not used or checked by NTFS
1C	DWord	Not used or checked by NTFS
20	DWord	Value must be 0 or NTFS fails to mount the volume
24	DWord	Not used or checked by NTFS
28	LongLong	The total number of sectors on the hard disk
30	LongLong	Logical Cluster Number for the File $MFT
38	LongLong	Logical Cluster Number for the File $MFTMirr
40	Byte	Clusters Per MFT Record. The size of each record in the MFT. If this number is positive (up to 0x7F), it represents clusters per MFT record. If the number is negative (0x80 to 0xFF), the size of the file record is 2 raised to the absolute value of this number.
41	Byte 3	Not used by NTFS.
44	Byte	Clusters Per Index Buffer. The size of each index buffer used to allocate space for directories. If this number is positive it represents clusters per index buffer. If the number is negative the size of the buffer is 2 raised to the absolute value of this number.
45	Byte 3	Not used by NTFS
48	LongLong	Volume Serial Number
50	DWord	Not used by NTFS
54	Char 426	Bootstrap code
1FE	Word	End of sector marker

The VBR comprises the jump instruction and the file system identifier, followed by the Bios Partition Block (BPB, in green), the Extended Bios Partition Block (EBPB, in pink), the bootstrap code, and the end of sector marker. The EBPB facilitates additional NTFS

functions: truly DNA runs through Microsoft code, as the BPB carries traces of ancient operating systems.

By comparing the $Boot file with the VBR layout some interesting and some not quite so interesting parameters of the partition can be extracted, and also the location and size of NTFS's Master File Table.

Within the $Boot file it's easy to see the characters NTFS, but the hex numerical values in the main do not even remotely translate to any reasonable decimal values. The sector size is known to be 512, so why is it shown as 0x0002? This is where we meet the vitally important concept of little-endian.

Little-Endian:

A PC-compatible computer running x86 architecture holds numerical values in little-endian form, as opposed to the big-endian form used by some other architectures. Little-endian is a method of ordering the bytes of a numeric value, and applies when a field is two or more bytes long. To convert little-endian to a recognisable hex form, there are two rules to be followed: the value of each byte is not changed, and the bytes are reordered from right to left.

The sector size above, two bytes, is held as 00 02, which appears to be nonsense. But reordering the bytes by first taking the right-most byte, followed by the next right-most, produces the hex number 0x0200, which when converted to decimal is 512.

Similarly the MFT logical cluster number is held as 00 00 0C 00 00 00 00 00, an apparently huge number. Reordered it becomes 0x00000000000C0000, which in decimal is 786432. Recuva or Winhex will allow these values to be confirmed.

Negative numbers are also held in little-endian and they will appear later. A negative number is one where, after byte reordering, the left-most byte (after dropping the leading zeroes) has a value of 0x80 or higher. To determine the negative value the number is reordered as usual, the zero-value bytes on the left discarded, and the resulting number two's complemented (each bit is reversed and 1 is added).

The problem with little-endian is determining which fields are held in little-endian form and which are not.

Returing to the VBR:

The fields at offset 0x03, 0x0B and 0x0D show that this is an NTFS partition with a sector size of 512 bytes and a cluster size of 4096. Offsets 0x30 and 0x38 hold the MFT

logical cluster starting number at 786432, and that of the MFT Mirror at 19521987. Right at the end of the sector, at offset 0x01FE, is 0xAA55 - the boot sector security mark that Windows checks before running the boot code. At the start of the second sector, offset 0x200, can be seen the start of the Windows loader NTLDR.

At offset 0x40 we can see what is either a masterpiece of ingenuity or a ridiculous flourish of complexity, the clusters per MFT record. If this value is positive then the MFT record size is a multiple of the cluster size. If the value is negative however, as it most likely will be, then it is resolved by raising 2 to the power of the absolute value of this entry. The value is 0xF6, which is -10, so the MFT record size is 2 to the power of 10, which is 1024. Thankfully the value at offset 0x44 is 0x01, so the index buffer size is 4096 bytes.

The Master File Table:

All system files are important, but the MFT is undoubtedly the star of the party. In NTFS everything is a file, and the description of every file is held in the MFT.

The MFT is itself a file consisting of a forever expanding list of 1024-byte File Record Segments (records). There is a record, and sometimes many records, for every file and directory in the partition, including the MFT itself. Individual records are very easy to locate, as the first four bytes contain the file signature 0x46494C45, or the chars FILE, but finding a specific record can be a nightmare.

In XP there is an 'MFT Zone' allocated at the start of the MFT with a default size of 12.5% of the partition: this allows ample space for the MFT to expand in use in a contiguous manner. As disk sizes grew this became impracticable, so from Vista onwards the MFT zone is allocated in 200 mb blocks which can be in any position on the drive.

No MFT record is ever removed from the MFT, they are reused. When a file or directory is deleted its record is flagged as deleted and is then available for re-use in future file or directory allocations.

Although the address of the MFT can be extracted from the $Boot file, it is far easier just to open it up with WinHex, or take a copy with Recuva and study that. Opening the MFT shows that the first records are used by system files, with some space reserved for future expansion. The actual start position of user file records is record number 27.

System file records in the MFT

Record	File Name	Description
1	$MFT	Master File Table
2	$MFTMir	MFT mirror
3	$LogFile	Transaction log
4	$Volume	Volume information
5	$AttrDef	MFT attribute table
6	.	Root directory
7	$Bitmap	Logical cluster bitmap
8	$Boot	Volume boot record
9	$BadClus	Bad cluster table
10	$Secure	Access Control List database
11	$UpCase	Unicode uppercase table
12	$Extend	Optional extensions
13-16		Reserved for extensions
17	$Extend\$Quota	Disk quota information
18	$Extend\$objId	Distributed link tracking info
19	$Extend\$Reparse	Reparse point data
20	File.ext	Start of user files

Open the MFT with WinHex and start a search for the chars 'FILE'. It will be easy to page down through the first few records. The fourth record is the entry for the $Volume system file.

At offset 0x1C8 in the $Volume record are two bytes containing the NTFS version. Since XP was introduced the version has remained at 03 01. (As these two bytes are not an arithmetical field they are not in little-endian.)

By using the information found earlier at the start of the $Boot file and the contents of the $Volume record in the MFT, we can be finally establish what we already knew, that this is an NTFS partition running version 03.01.

MFT Records:

All MFT records follow the same structure. They are 1024 bytes in size, and start with a 48-byte header section and a 8-byte Fixup Array (the header length is variously described as 48 or 56 bytes, depending on whether the Fixup Array is classed as an attribute or not). At offset 0x38 is the start of a string of attributes (the header is not an attribute). The header describes the properties of the record, and the attributes describe all aspects of the file from its name to its data. The Fixup Array is described a little later.

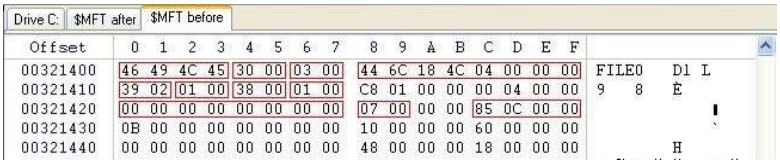
Each MFT record is allocated an ascending 48 bit relative record number, starting with the first record (describing the $MFT file itself) having record number 0. Each record also has a 16 bit sequence number that is incremented whenever the file represented by the record is deleted. The record number and sequence number combined produce a 64b record reference address.

The MFT reference address is used, amongst other things, to relate directory entries to file records, and to relate MFT records to each other. If a record were to be physically removed then the entire MFT, and all its internal references, would have to be restructured, a horrendous task. On deletion therefore a file's record is not removed from the MFT, but certain fields and flags in the record are set to denote that the record is available for reuse.

The attributes in an MFT record can occasionally be too long to fit into the available space in the record. In this case the attributes start in the base MFT record and continue in one or more extension records. The extension records have the base record's reference address held at offset 0x20: this field in the base record contains zeroes.

To pluck an MFT record (almost) at random we can see:

MFT record header



MFT Record header layout

Offset	Size	Description
00	Char 4	MFT record signature
04	Word	Offset to update sequence
06	Word	Number of entries in Fixup array
08	LongLong	$LogFile Sequence Number (LSN)
10	Word	Record usage number
12	Word	Hard link count
14	Word	Offset to first attribute
16	Word	Flags: 01 00 record in use, 02 00 directory
18	DWord	Actual size of MFT entry
1C	DWord	Allocated size of MFT entry
20	LongLong	File reference to the base FILE record
28	Word	Next attribute ID
2A	Word	
2C	DWord	MFT record number
30		Fixup values and Attributes

There's a wealth of information in the MFT record header. too much to describe it all. Looking at some of the more interestings fields we can see:

- 0x00 - record identifier 'FILE'. If the entry is unusable the record identifier would be 'BAAD'.
- 0x04 - offset to the fixup array (0x2A prior to XP, 0x30 from XP onwards)
- 0x06 - number of two-byte entries in the fixup array. The fixup array is used to validate sectors within the MFT record.

- 0x08 - holds the sequence number of the logfile entry that tracks every change to the file
- 0x10 - number of times this record has been used. In this example the value is 0x0239, 569. This number cycles in use. If it is zero it is left as zero
- 0x12 - link count, being the number of directories that reference this record: only used in base records
- 0x14 - Offset to the first attribute in the record
- 0x16 - flags: the low order (right-most) bit is is set to one if the record is in use, and the next right-most bit is set to one for an index, zero for a file. So 0x0001 is a live file, and 0x0003 a live folder: 0x0000 and 0x0002 being the deleted equivalent. There can be other values in the flags field but these do not affect the use of the live/delete and the file/folder bits.
- 0x20 - if this MFT record is the base entry for the file then this field is zero: if the record is an extension then this field holds the base record reference address
- 0x28 - sequence number ID for attributes starting from zero
- 0x2C - The relative number of this MFT record starting from zero. If this value - 0xC85 - is multiplied by 0x400 (the cluster size) we get 0x321400, the relative byte address in the MFT of this record.

The values at offsets 0x04 and 0x06 locate the Fixup Array at offset 0x30 with three two-byte entries. When the MFT record is updated the first array entry (the Update Sequence Number) is incremented by one, and the last two bytes of each sector of the record are first copied to the next two array entries and then overwritten with the USN. When the record is read the value of the USN is compared with the last two bytes of each sector: if successful then the values in the array are restored to the end of the sectors in memory for processing.

Having extracted more information from this MFT record header than we could ever want, we can now get a little closer to the file data by looking at the attributes.

Attributes:

Each attribute is a varying-length stream identified by a four-byte attribute type at offset 0x00. The first attribute is usually type 0x00000010, $Standard_Information. Attributes are stored in ascending type order and terminated with the End marker 0xFFFFFFFF, which is itself a special attribute consisting of the attribute type only. Attributes within an MFT record also have an ascending ID number starting from zero (which may not follow the physical order of the attributes).

A list of the attributes used on a particular system is held in the system file $AttrDef.

Attribute List

Attribute ID	Attribute Name
00 00 00 00	Unused
10 00 00 00	$Standard_Information
20 00 00 00	$Attribute_List
30 00 00 00	$File_Name
40 00 00 00	$Object_ID
50 00 00 00	$Security_Descriptor
60 00 00 00	$Volume_Name
70 00 00 00	$Volume_Information
80 00 00 00	$Data
90 00 00 00	$Index_Root
A0 00 00 00	$Index_Allocation
B0 00 00 00	$Bitmap
C0 00 00 00	$Reparse_Point
D0 00 00 00	$Ea_Information
E0 00 00 00	$EA
F0 00 00 00	$Property_Set
00 01 00 00	$Logged_Utility_Stream
00 10 00 00	First User Defined Attribute
FF FF FF FF	End of Attributes

MFT records may have many attributes or have attributes with a large content (such as a $Data attribute with many runlists) such that all the attributes can not be held within one base record. In this case extension MFT records are used to hold some of the attributes. The base MFT record is used for referencing the file, and the $Attribute_List attribute within it provides the references to all the extension records for the file. The $Attribute_List ID is 0x20 so it follows the $Standard_Information attribute: it is always contained in the base MFT record.

Attributes whose content is held entirely within the attribute (be it a base or extension MFT record) are known as Resident. Attributes whose content is not are known as Non-Resident. Resident attributes have the byte at offset 0x08 set to zero, non-resident to 0x01. The $Data attribute is most likely to be non-resident. Non-Residency is not to be confused with extension records. A file with a few hundred bytes of data may have that data held entirely in the $Data attribute: the attribute will be Resident. Most files are longer than this and will have their data held separately in clusters: the $Data attribute will hold external cluster information and will therefore be Non-Resident.

As a small moment of light relief, although a cluster is the minimum unit of data transfer in NTFS, it is possible to hold more files on a disk than there are clusters - if all the files are small enough to fit entirely within their 1k MFT records. This information is however only likely to be used to impress other geeks.

Attribute header:

The attribute header for resident and non-resident attributes differs from offset 0x10 onwards. The resident header is shorter at 0x18 bytes, with the non-resident header being 0x40 bytes.

Attribute Header - Resident

Offset	Size	Description
00	DWord	Attribute type
04	DWord	Length of attribute
08	Byte	Non-resident flag
09	Byte	Name length
0A	Word	Offset to name
0C	Word	Flags
0E	Word	Attribute ID
10	DWord	Length of content
14	Word	Offset to content
16	Byte	
17	Byte	

Attribute Header - Non-Resident

Offset	Size	Description
00	DWord	Attribute type
04	DWord	Length of attribute
08	Byte	Non-resident flag
09	Byte	Name length
0A	Word	Offset to name
0C	Word	Flags
0E	Word	Attribute ID
10	LongLong	Start VCN of runlist
18	LongLong	End VCN of runlist
20	Word	Offset to runlist
22	Word	Compression unit size
24	Byte 4	
28	LongLong	Allocated size of attribute content
30	LongLong	Actual size of attribute content
38	LongLong`	Initialised size of attribute content

When chosing a record to examine it's always better for it to have an unusual name so it can be searched for more easily, so meet aardvark.txt.

Aardvark.txt file record

Drive C: | $MFT after | $MFT before

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00321400	46	49	4C	45	30	00	03	00	44	6C	18	4C	04	00	00	00
00321410	39	02	01	00	38	00	01	00	C8	01	00	00	00	04	00	00
00321420	00	00	00	00	00	00	00	00	07	00	00	00	85	0C	00	00
00321430	0B	00	00	00	00	00	00	00	10	00	00	00	60	00	00	00
00321440	00	00	00	00	00	00	00	00	48	00	00	00	18	00	00	00
00321450	DE	3F	D5	5B	BF	A1	CB	01	DC	F0	7B	BC	6D	A1	CB	01
00321460	06	7C	01	55	81	A6	CB	01	AA	01	B1	4E	81	A6	CB	01
00321470	20	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00321480	00	00	00	00	5D	01	00	00	00	00	00	00	00	00	00	00
00321490	08	FC	52	58	00	00	00	00	30	00	00	00	78	00	00	00
003214A0	00	00	00	00	00	00	06	00	5A	00	00	00	18	00	01	00
003214B0	E7	25	00	00	00	00	01	00	DE	3F	D5	5B	BF	A1	CB	01
003214C0	DC	F0	7B	BC	6D	A1	CB	01	DC	F0	7B	BC	6D	A1	CB	01
003214D0	78	EE	4E	41	81	A6	CB	01	00	40	00	00	00	00	00	00
003214E0	9C	34	00	00	00	00	00	00	20	20	00	00	00	00	00	00
003214F0	0C	03	61	00	61	00	72	00	64	00	76	00	61	00	72	00
00321500	6B	00	2E	00	74	00	78	00	74	00	00	00	00	00	00	00
00321510	80	00	00	00	48	00	00	00	01	00	00	00	00	00	04	00
00321520	00	00	00	00	00	00	00	00	03	00	00	00	00	00	00	00
00321530	40	00	00	00	00	00	00	00	00	40	00	00	00	00	00	00
00321540	9C	34	00	00	00	00	00	00	9C	34	00	00	00	00	00	00
00321550	41	04	B4	7D	B9	00	00	00	80	00	00	00	68	00	00	00
00321560	01	0F	40	00	00	00	05	00	00	00	00	00	00	00	00	00
00321570	00	00	00	00	00	00	00	00	60	00	00	00	00	00	00	00
00321580	00	10	00	00	00	00	00	00	2E	00	00	00	00	00	00	00
00321590	2E	00	00	00	00	00	00	00	5A	00	6F	00	6E	00	65	00
003215A0	2E	00	49	00	64	00	65	00	6E	00	74	00	69	00	66	00
003215B0	69	00	65	00	72	00	00	00	41	01	B8	7D	B9	00	00	00
003215C0	FF	FF	FF	FF	82	79	47	11	9C	34	00	00	00	00	00	00

In aardvark's record we can follow the chain of attributes from the start of the MFT header.

- the offset to the first attribute in the record is 0x38. At 0x38 is 0x10, $Standard_Information
- at offset 0x04 in the attribute is the length of this attribute, 0x60, which leads to
- attribute 0x30, $File_Name, length 0x78, which leads to
- attribute 0x80, $Data, length 0x48, which leads to
- attribute 0x80, $Data, length 0x68, which leads to
- attribute 0xFFFFFFFF, the end of attributes attribute

Each attribute has a header, and following the header is the attribute data stream (the offsets do not include the header).

$Standard_Information attribute

Offset	Size	Description
00	LongLong	File creation time
08	LongLong	File altered time
10	LongLong	MFT record changed time
18	LongLong	File read time
20	DWord	DOS file permissions
24	DWord	Maximum number of versions
28	DWord	Version number
2C	DWord	Class ID
30	DWord	Owner ID
34	Dword	Security ID
38	LongLong	Quota charged
40	LongLong	Update sequence number

There's not a great deal in the $Standard_Information attribute that I want to bother with, mainly timestamps and ID info. You can plough through it or skip it. Fields in the $Standard_Information attribute are always up to date.

$File_Name attribute

Offset	Size	Description
00	LongLong	File reference to the parent directory
08	LongLong	File creation time
10	LongLong	File altered time
18	LongLong	MFT record changed time
20	LongLong	File read time
28	LongLong	Allocated size of file
30	LongLong	Real size of file
38	Dword	Flags
3C	Dword	Used by Extended Attributes and reparse
40	Byte	Filename length in chars
41	Byte	Filename namespace
42	Byte	Filename in Unicode

The $File_Name attribute is marginally more interesting, holding as we'd expect the file name in UTF-16 unicode. It also contains the allocted and real size of the file. The

allocated size is a multiple of the cluster size, and the real size is the actual size of the file. The allocated and real sizes are also held in MFT directory records, and these are the values displayed in Explorer folder listings.

Fields in the $File_Name attibute (except the reference to the parent directory) are not updated unless the filename is changed, instead just becoming outdated.

Just for posterity, the Filename Namespace values are:

- 0 - POSIX: case sensitive and allows all unicode chars
- 1 - Win32: case insensitive, subset of POSIX
- 2 - DOS: subset of Win32
- 3 - Win32 and DOS: both are identical

$Data attribute:

The $Data attribute has no specific layout following the attribute header. However if the attribute is resident it will contain the file's data in its entirety, if that data is small enough to fit within the MFT record, or if non-resident it will hold one or more cluster start and length fields.

At offset 0x08 in aardvark.txt's $Data attribute is the non-resident flag, which is set to 0x01. This attribute is non-resident. Using the non-resident attribute layout we can find the offset to the first datarun in the runlist, 0x40.

A datarun has three components, a length/offset byte, a cluster run number, and a cluster start number. The length/offset byte is further divided into two 4-bit parts, the *length* of the cluster run number component and the *length* of the cluster start number component.

The first byte of aardvark's first datarun is 0x41. Reading the four low-order bits shows that the cluster run number is one byte in length. Reading the four high-order bits shows that the cluster start number is four bytes in length. Adding the two together shows that this datarun occupies the five bytes following the length/offset byte.

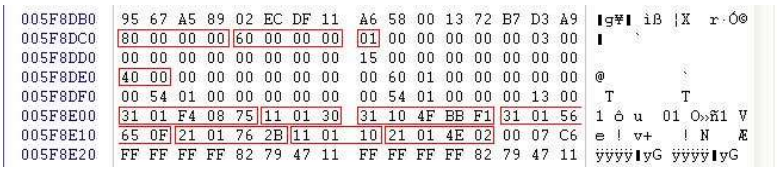
41 04 B4 7D B9 00

Looking at the values in the cluster run number and cluster start fields, it can be seen that aardvark's file data is held in four clusters starting at logical cluster number 0xB97DB4. Advancing to the next datarun gives a byte of 00, which indicates the end of the runlist. Aardvark has only one data extent on disk.

Following, and linked from, the $Data attribute that holds the runlists can be seen a second $Data attribute. This is non-resident and holds Zone Identifier information. This is an Alternate Data Stream created by IE when a file which can contain executable content is downloaded from the internet. It is normally unseen by the user and is used to check that the same security zone is used when the file is opened by IE. The aardvark file, by the way, was originally an html file which has been renamed so it could be used as an example. A Zone Identifier is not normally present in a txt file.

Of course interpreting a datalist isn't always as easy as it is in the example. There are multiple, cumulative and negative dataruns. The following example for instance has seven dataruns listed in the $Data attribute, representing seven extents on the disk:

Datalist with many dataruns



Here the first datarun is relatively straightforward. It has one byte for the number of clusters in the run, holding a value of one. Three bytes hold the cluster start Number of 0x7508F4, which is Logical Cluster Number 7670004. But the next datarun is peculiar: one cluster, starting at LCN 0x30? One cluster is correct, but the LCN in this datarun is *added* to the previous LCN, giving a cumulative LCN of 0x750924, LCN 7670052. Datarun LCN's are always relative to the cumulative LCN of all previous dataruns.

The third datarun has 0x10 clusters, but starts at 0xF1BB4F. This is a negative number. Its negative value is 0xE44B1, -935089. This number is *subtracted* from the cumulative LCN, giving us an LCN of 0x66C473, 6734963.

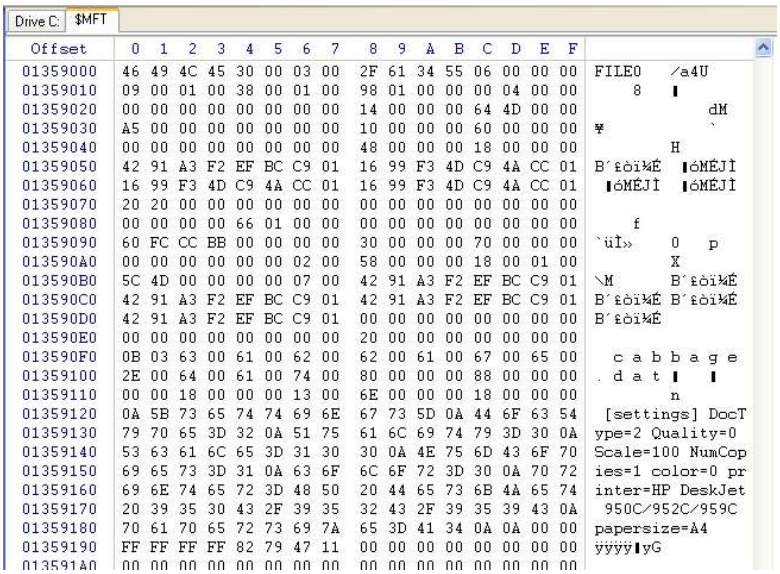
The order of the dataruns represents the logical order of the file data in the fragments, which is why occasionally negative values are used.

Very small files:

Most $Data attributes will, as in the above examples, have the non-resident flag set to indicate that the content that this attribute refers to - the file data - is held externally. Thus the attribute will have as its content dataruns describing the file data clusters.

Small files however, those around eight or nine hundred bytes or fewer, may have their data held entirely within the MFT record. In this case the $Data attribute is resident, and uses the internal attribute header. The Length of content and the Offset to content fields are used to specify the file data position and length of the data.

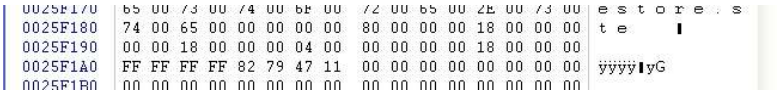
MFT record containing the entire file



The $Data attribute above has a content start position of 0x18, and a content length of 0x6E (110 bytes). The file data can quite easily be seen in the record. There are no cluster allocations for this file. The allocated and actual file size fields in the $File_Name attribute are not maintained when the $Data attribute is resident.

Not all small files are resident in the MFT. A larger file will allocate one or more data clusters: if the file size is subsequently reduced to a few bytes then one cluster holding the data will remain allocated, and the $Data attribute will remain non-resident.

$Data attribute for a zero length file

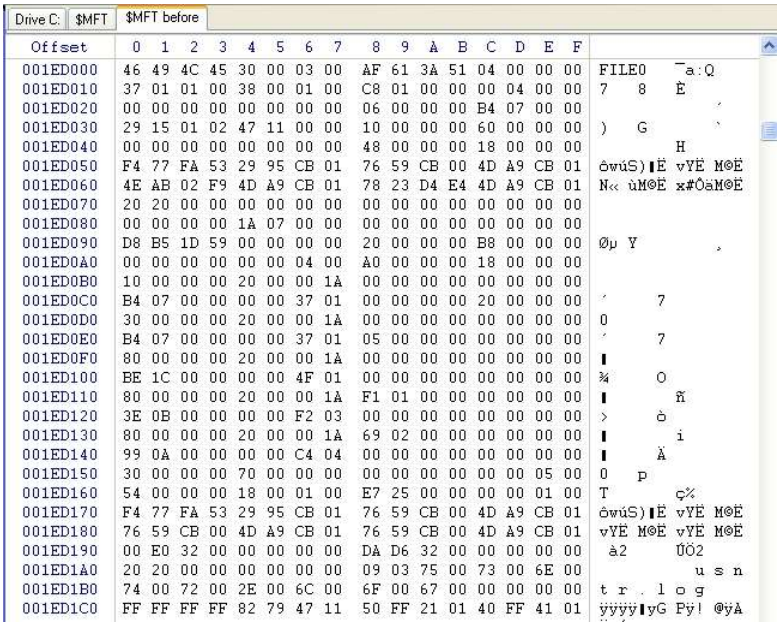


Zero length files still have a $Data attribute in their MFT record. The very short $Data attribute - header only - is of course resident and has the attribute length and the Offset to content values the same, 0x18. Thus the file is zero bytes in length.

MFT Extension records:

If an MFT record has many attributes, or an attribute with a large content, the attributes may not fit within the available space in a single MFT record. This will frequently happen if the file data is in many fragments, requiring the $Data attribute to hold many dataruns, In this case the attributes start in the base MFT record and continue in one or more extension records. File access is always through the base MFT record: the extension record addresses are held in the $Attribute_List attribute, which has an ID of 0x20 ensuring that it will always be present in the base record.

MFT base record with $Attribute_List attribute



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
001ED000	46	49	4C	45	30	00	03	00	AF	61	3A	51	04	00	00	00
001ED010	37	01	01	00	38	00	01	00	C8	01	00	00	00	04	00	00
001ED020	00	00	00	00	00	00	00	00	06	00	00	00	B4	07	00	00
001ED030	29	15	01	02	47	11	00	00	10	00	00	00	60	00	00	00
001ED040	00	00	00	00	00	00	00	00	48	00	00	00	18	00	00	00
001ED050	F4	77	FA	53	29	95	CB	01	76	59	CB	00	4D	A9	CB	01
001ED060	4E	AB	02	F9	4D	A9	CB	01	78	23	D4	E4	4D	A9	CB	01
001ED070	20	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00
001ED080	00	00	00	00	1A	07	00	00	00	00	00	00	00	00	00	00
001ED090	D8	B5	1D	59	00	00	00	00	20	00	00	00	B8	00	00	00
001ED0A0	00	00	00	00	00	00	04	00	A0	00	00	00	18	00	00	00
001ED0B0	10	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
001ED0C0	B4	07	00	00	00	00	37	01	00	00	00	00	20	00	00	00
001ED0D0	30	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
001ED0E0	B4	07	00	00	00	00	37	01	05	00	00	00	00	00	00	00
001ED0F0	80	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
001ED100	BE	1C	00	00	00	00	4F	01	00	00	00	00	00	00	00	00
001ED110	80	00	00	00	20	00	00	1A	F1	01	00	00	00	00	00	00
001ED120	3E	0B	00	00	00	00	F2	03	00	00	00	00	00	00	00	00
001ED130	80	00	00	00	20	00	00	1A	69	02	00	00	00	00	00	00
001ED140	99	0A	00	00	00	00	C4	04	00	00	00	00	00	00	00	00
001ED150	30	00	00	00	70	00	00	00	00	00	00	00	00	00	05	00
001ED160	54	00	00	00	18	00	01	00	E7	25	00	00	00	00	01	00
001ED170	F4	77	FA	53	29	95	CB	01	76	59	CB	00	4D	A9	CB	01
001ED180	76	59	CB	00	4D	A9	CB	01	76	59	CB	00	4D	A9	CB	01
001ED190	00	E0	32	00	00	00	00	00	DA	D6	32	00	00	00	00	00
001ED1A0	20	20	00	00	00	00	00	00	09	03	75	00	73	00	6E	00
001ED1B0	74	00	72	00	2E	00	6C	00	6F	00	67	00	00	00	00	00
001ED1C0	FF	FF	FF	FF	82	79	47	11	50	FF	21	01	40	FF	41	01

The $Attribute_List attribute has the same header layout of any other attribute: its content is a datalist of one or more MFT extension records. All attributes in the record are present in the datalist, even though they may be held in the base record.

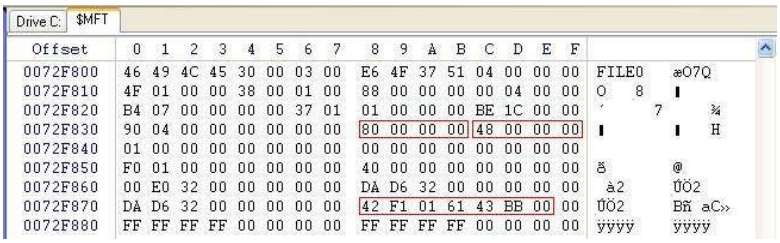
In the example above the $Attribute_List attribute is resident (the list is held entirely within the attribute) and the contents start at offset 0x18. Each entry in the datalist is 32 bytes long, and starts with the four-byte attribute ID followed by a two-byte entry length field. The attributes listed are 0x10, 0x30, and three 0x80. (The following 0x30 attribute is not part of the $Attribute_List attribute.)

At offset 0x10 in each entry is the MFT extension record ID, which is the record number concatenated with the sequence number. For attributes 0x10 and 0x30 the record number is 0x07B4, which when multiplied by 0x400 gives 0x1ED000. This is the

base record, the record in fact that we're examining, so these entries are self-referential.

The third attribute has an id of 0x80, $Data, and points to a different MFT record. This is the first MFT extension record for the file. Using the MFT record number of 0x1CBE we can find the extension record at 0x72F800, and confirm the sequence number of 0x014F. We can also see at offset 0x20 in the extension record the base record number 0x7B4 and sequence number 0x137.

MFT Extension record



Drive C: $MFT

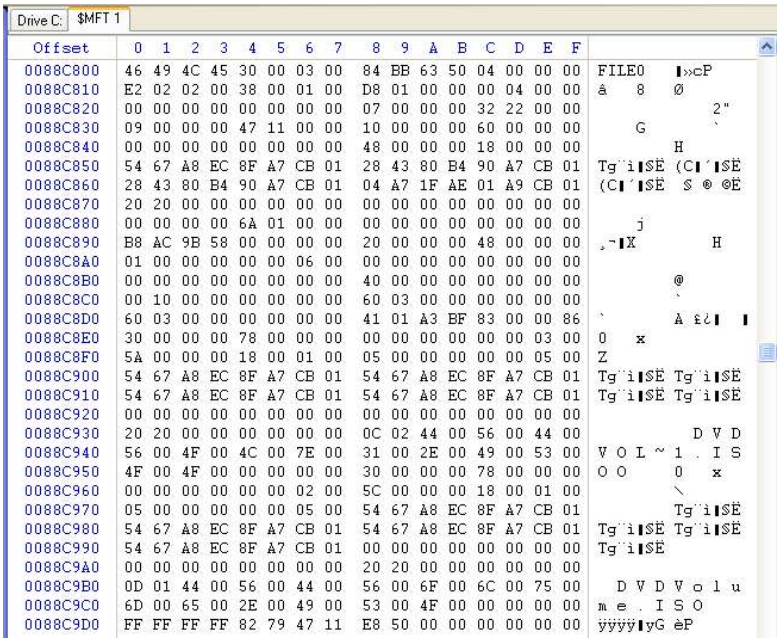
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0072F800	46	49	4C	45	30	00	03	00	E6	4F	37	51	04	00	00	00
0072F810	4F	01	00	00	38	00	01	00	88	00	00	00	00	04	00	00
0072F820	B4	07	00	00	00	00	37	01	01	00	00	00	BE	1C	00	00
0072F830	90	04	00	00	00	00	00	00	80	00	00	00	48	00	00	00
0072F840	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0072F850	F0	01	00	00	00	00	00	00	40	00	00	00	00	00	00	00
0072F860	00	E0	32	00	00	00	00	00	DA	D6	32	00	00	00	00	00
0072F870	DA	D6	32	00	00	00	00	00	42	F1	01	61	43	BB	00	00
0072F880	FF	FF	FF	FF	00	00	00	00	FF	FF	FF	FF	00	00	00	00

An MFT extension record obeys the same rules as any other record, it has the standard header but contains only one attribute, in this case and in most cases 0x80. The attribute is non-resident, and holds a datalist containing one datarun, 42 F1 01 61 43 BB. This identifies 0x1F1 clusters at 0xBB4381. Why does the extension record hold only one datrun when there are three extension records for the $Data attribute? I don't know, possibly because this is a log file and has frequent cluster allocations.

MFT base record with Non-resident $atttribute_List:

If a file has an extensive number of fragments then it may require many extension MFT records, so many that the $Attribute_List attribute itself becomes too large to be held in the base MFT record. In this case the $Attribute_List attribute is made non-resident, and its content, the list of extension records, held in an external cluster. Although this may seem to be rare, I loaded a 800 mb DVD ISO file for these tests and surprisingly, on a disk with over 90% free space, the file was created with almost 4000 fragments, and indeed the MFT base record posessed a non-resident $Attribute_List attribute.

Non-resident $Atttribute_List attribute



Drive C: $MFT 1

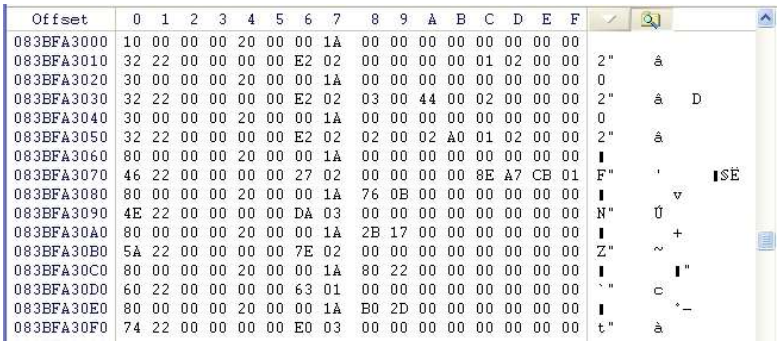
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0088C800	46	49	4C	45	30	00	03	00	84	BB	63	50	04	00	00	00
0088C810	E2	02	02	00	38	00	01	00	D8	01	00	00	00	04	00	00
0088C820	00	00	00	00	00	00	00	00	07	00	00	00	32	22	00	00
0088C830	09	00	00	00	47	11	00	00	10	00	00	00	60	00	00	00
0088C840	00	00	00	00	00	00	00	00	48	00	00	00	18	00	00	00
0088C850	54	67	A8	EC	8F	A7	CB	01	28	43	80	B4	90	A7	CB	01
0088C860	28	43	80	B4	90	A7	CB	01	04	A7	1F	AE	01	A9	CB	01
0088C870	20	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0088C880	00	00	00	00	6A	01	00	00	00	00	00	00	00	00	00	00
0088C890	B8	AC	9B	58	00	00	00	00	20	00	00	00	48	00	00	00
0088C8A0	01	00	00	00	00	00	06	00	00	00	00	00	00	00	00	00
0088C8B0	00	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
0088C8C0	00	10	00	00	00	00	00	00	60	03	00	00	00	00	00	00
0088C8D0	60	03	00	00	00	00	00	00	41	01	A3	BF	83	00	00	86
0088C8E0	30	00	00	00	78	00	00	00	00	00	00	00	00	00	03	00
0088C8F0	5A	00	00	00	18	00	01	00	05	00	00	00	00	00	05	00
0088C900	54	67	A8	EC	8F	A7	CB	01	54	67	A8	EC	8F	A7	CB	01
0088C910	54	67	A8	EC	8F	A7	CB	01	54	67	A8	EC	8F	A7	CB	01
0088C920	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0088C930	20	20	00	00	00	00	00	00	0C	02	44	00	56	00	44	00
0088C940	56	00	4F	00	4C	00	7E	00	31	00	2E	00	49	00	53	00
0088C950	4F	00	4F	00	00	00	00	00	30	00	00	00	78	00	00	00
0088C960	00	00	00	00	00	00	02	00	5C	00	00	00	18	00	01	00
0088C970	05	00	00	00	00	00	05	00	54	67	A8	EC	8F	A7	CB	01
0088C980	54	67	A8	EC	8F	A7	CB	01	54	67	A8	EC	8F	A7	CB	01
0088C990	54	67	A8	EC	8F	A7	CB	01	00	00	00	00	00	00	00	00
0088C9A0	00	00	00	00	00	00	00	00	20	20	00	00	00	00	00	00
0088C9B0	0D	01	44	00	56	00	44	00	56	00	6F	00	6C	00	75	00
0088C9C0	6D	00	65	00	2E	00	49	00	53	00	4F	00	00	00	00	00
0088C9D0	FF	FF	FF	FF	82	79	47	11	E8	50	00	00	00	00	00	00

A non-resident $Attribute_List attribute holds a single datarun in its datalistat at offset 0x40, one cluster in size and located in our example at cluster number 0x83BFA3. Multiplying the cluster number by the cluster sixe (0x1000) gives 0x83BFA3000, which is named in WinHex as Misc non-resident atttributes. This is not an MFT extension record but is the list of extension records.

In fact this cluster is the last cluster in the allocation for the file data. So it belongs to the MFT, is allocated to the file, but is not part of the file data.

Surprisingly the fields for file allocated space and actual size are set to zero in the $File_Name attribute, and the $Data attribute is not present in the base record. So it is not possible to extract the file size without retrieving the first MFT extension record, where the file size - 0x2F218000 - is held in the $Data attribute header. This has also been seen in a resident $Attribute_List attribute and in a base record with a $Data attribute (although this record had the correct file sizes held in the $Data attribute): the reason why the fields in the $File_Name attribute are sometimes left as zero is unknown.

Non-resident atttributes cluster

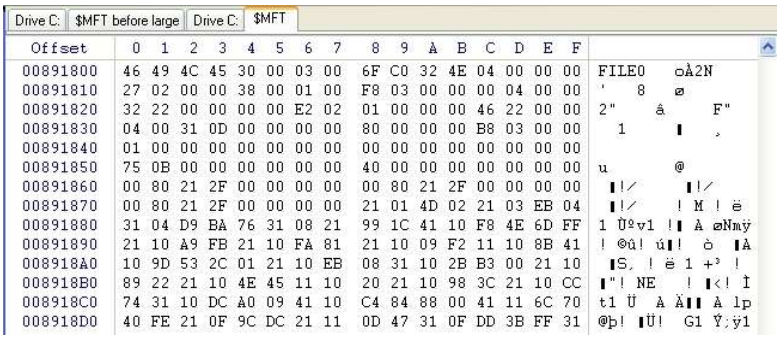


Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
083BFA3000	10	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
083BFA3010	32	22	00	00	00	00	E2	02	00	00	00	00	01	02	00	00
083BFA3020	30	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
083BFA3030	32	22	00	00	00	00	E2	02	03	00	44	00	02	00	00	00
083BFA3040	30	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
083BFA3050	32	22	00	00	00	00	E2	02	02	00	02	A0	01	02	00	00
083BFA3060	80	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
083BFA3070	46	22	00	00	00	00	27	02	00	00	00	00	8E	A7	CB	01
083BFA3080	80	00	00	00	20	00	00	1A	76	0B	00	00	00	00	00	00
083BFA3090	4E	22	00	00	00	00	DA	03	00	00	00	00	00	00	00	00
083BFA30A0	80	00	00	00	20	00	00	1A	2B	17	00	00	00	00	00	00
083BFA30B0	5A	22	00	00	00	00	7E	02	00	00	00	00	00	00	00	00
083BFA30C0	80	00	00	00	20	00	00	1A	80	22	00	00	00	00	00	00
083BFA30D0	60	22	00	00	00	00	63	01	00	00	00	00	00	00	00	00
083BFA30E0	80	00	00	00	20	00	00	1A	B0	2D	00	00	00	00	00	00
083BFA30F0	74	22	00	00	00	00	E0	03	00	00	00	00	00	00	00	00

This cluster holds extension record information in exactly the same way as it would be held if it were inside the $Attribute_List attribute itself. Again the entries for attributes 0x10 and 0x30 refer back to the MFT base record.

The fourth attribute section has an id of 0x80, $Data, and points to a different MFT record, the first of the MFT extension records for the file. Using the MFT record number we can find the extension record at 0x891800, and confirm the sequence number of 0x0227.

MFT extension record



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00891800	46	49	4C	45	30	00	03	00	6F	C0	32	4E	04	00	00	00
00891810	27	02	00	00	38	00	01	00	F8	03	00	00	00	04	00	00
00891820	32	22	00	00	00	00	E2	02	01	00	00	00	46	22	00	00
00891830	04	00	31	0D	00	00	00	00	80	00	00	00	B8	03	00	00
00891840	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00891850	75	0B	00	00	00	00	00	00	40	00	00	00	00	00	00	00
00891860	00	80	21	2F	00	00	00	00	00	80	21	2F	00	00	00	00
00891870	00	80	21	2F	00	00	00	00	21	01	4D	02	21	03	EB	04
00891880	31	04	D9	BA	76	31	08	21	99	1C	41	10	F8	4E	6D	FF
00891890	21	10	A9	FB	21	10	FA	81	21	10	09	F2	11	10	8B	41
008918A0	10	9D	53	2C	01	21	10	EB	08	31	10	2B	B3	00	21	10
008918B0	89	22	21	10	4E	45	11	10	20	21	10	98	3C	21	10	CC
008918C0	74	31	10	DC	A0	09	41	10	C4	84	88	00	41	11	6C	70
008918D0	40	FE	21	0F	9C	DC	21	11	0D	47	31	0F	DD	3B	FF	31

As before the base record reference number, at offset 0x20, refers back to the base record and sequence number. This extension record has one attribute, ID 0x80, $Data. At offset 0x40 in the attribute is the start of the datalist. The datalist in each MFT extension record holds approximately 160 dataruns. As there are so many extents in this file there are 24 extension records listed in the Misc non-resident atttributes cluster.

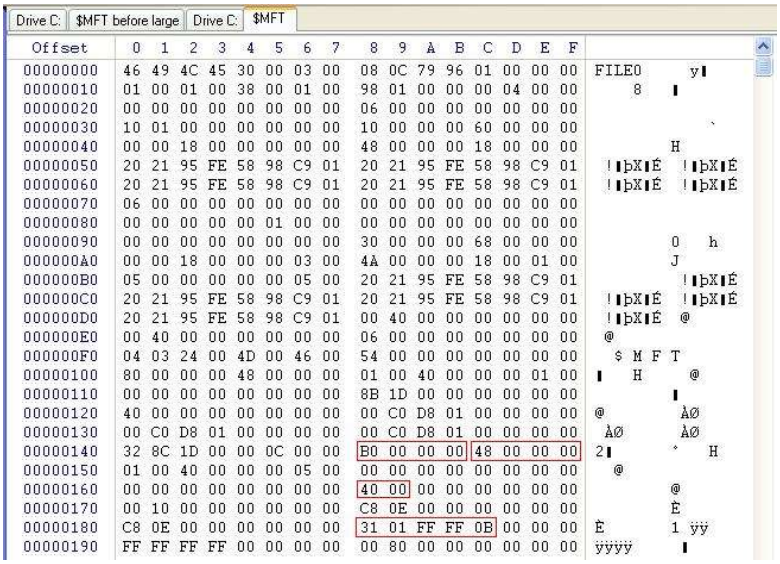
Although the non-resident atttributes cluster is part of the file allocation there is not a datarun for it in the $Data attribute in the non-resident atttributes cluster itself.

Bitmaps:

NTFS uses bitmaps for several purpose, so it is important to know which one, or type, is being discussed. The system file $Bitmap is the most obvious: this holds the map of logical clusters in use - and of course not in use - and is used for finding free space when a file is allocated. The less obvious bitmap is the MFT record attribute $Bitmap, with an attribute ID of 0xB0. This is used for mapping MFT records in - and out - of use. This bitmap attribute is also used in directory and INDX records in the MFT.

Any bit set to 1 indicates an in-use cluster or record. However the bits in each bitmap byte do not read sequentially but are read from the low-order bit first. For instance FF 13 translates as bit pattern 1111 1111 0001 0011, but represents 1111 1111 1100 1000 - clusters one to ten and thirteen in use, clusters eleven and twelve and fourteen onwards available for use.

Bitmap in MFT



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	46	49	4C	45	30	00	03	00	08	0C	79	96	01	00	00	00
00000010	01	00	01	00	38	00	01	00	98	01	00	00	00	04	00	00
00000020	00	00	00	00	00	00	00	00	06	00	00	00	00	00	00	00
00000030	10	01	00	00	00	00	00	00	10	00	00	00	60	00	00	00
00000040	00	00	18	00	00	00	00	00	48	00	00	00	18	00	00	00
00000050	20	21	95	FE	58	98	C9	01	20	21	95	FE	58	98	C9	01
00000060	20	21	95	FE	58	98	C9	01	20	21	95	FE	58	98	C9	01
00000070	06	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	01	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	30	00	00	00	68	00	00	00
000000A0	00	00	18	00	00	00	03	00	4A	00	00	00	18	00	01	00
000000B0	05	00	00	00	00	00	05	00	20	21	95	FE	58	98	C9	01
000000C0	20	21	95	FE	58	98	C9	01	20	21	95	FE	58	98	C9	01
000000D0	20	21	95	FE	58	98	C9	01	00	40	00	00	00	00	00	00
000000E0	00	40	00	00	00	00	00	00	06	00	00	00	00	00	00	00
000000F0	04	03	24	00	4D	00	46	00	54	00	00	00	00	00	00	00
00000100	80	00	00	00	48	00	00	00	01	00	40	00	00	00	01	00
00000110	00	00	00	00	00	00	00	00	8B	1D	00	00	00	00	00	00
00000120	40	00	00	00	00	00	00	00	00	C0	D8	01	00	00	00	00
00000130	00	C0	D8	01	00	00	00	00	00	C0	D8	01	00	00	00	00
00000140	32	8C	1D	00	00	0C	00	00	B0	00	00	00	48	00	00	00
00000150	01	00	40	00	00	00	05	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
00000170	00	10	00	00	00	00	00	00	C8	0E	00	00	00	00	00	00
00000180	C8	0E	00	00	00	00	00	00	31	01	FF	FF	0B	00	00	00
00000190	FF	FF	FF	FF	00	00	00	00	00	80	00	00	00	00	00	00

The MFT bitmap is one cluster at LCN 0xBFFFF. This is the cluster (in my case) physically prior to the start of the MFT records. It is actually part of the MFT allocation as a whole, with (again in my case) the MFT having two extents: one of over 7500 clusters for the MFT records, defined in the $Data attribute, and a single cluster for the bitmap, defined in the $Bitmap attribute. As with the non-resident atttributes cluster above, there is not a datarun for the MFT bitmap in the $Data attribute.

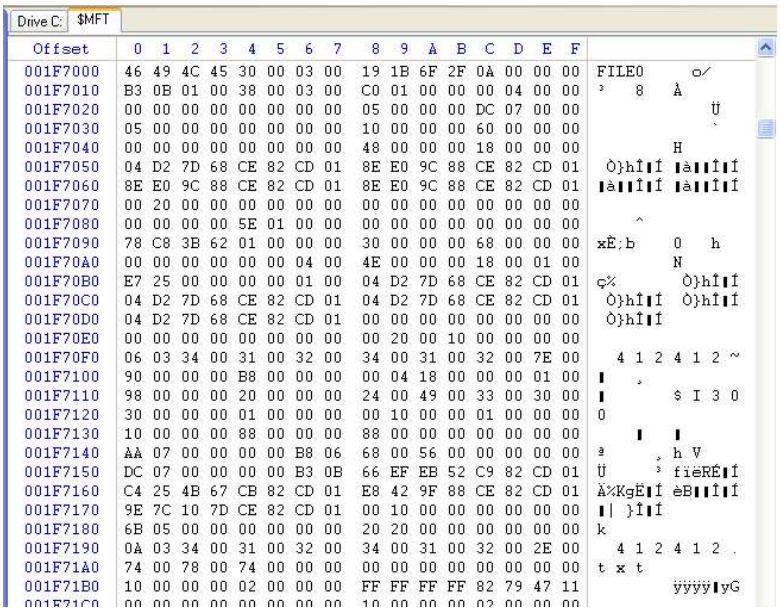
Indexes:

Index, or folder, records in the MFT are identified by 0x0003 in the flags field in the record header. They are made up of the same header and attributes as all other MFT records.

File names within a folder are held in ascending name order in the $Index_Root attribute (0x90). When a file is added to the folder it is inserted in the correct sequence in the $Index_Root attribute, the following file names shuffled down, and the attribute length increased. If a file is deleted the name is removed and the following names shuffled up, overwriting the data from the deleted file.

Basic file information is maintained along with the file name in the $Index_Root attribute, and is used to display folder views in Explorer: otherwise the records for every file in the folder would have to be read. Although this information is lost when a file is deleted, the folder structure can be constructed from the lowest level upwards by following the parent directory pointers in the deleted file's base record. Similarly file size and other information can be extracted from the file's record.

A simple MFT Index record



This is the index record for the folder 412412 which has one file within it, 412412.txt. The folder's name is in the $File_Name attribute, and the names of the file in the $Index_Root attribute. At offset 0x40 in the $Index_Root attribute is the record sequence number for the file of 0x07AA, at offset 0x078 the allocated size of 0x1000, and at offset 0x80 the actual data length 0f 0x056B.

Larger folders will have MFT extension records and very large folders will have non-resident attribute clusters.

File deletion:

What happens whan a file is deleted? This is of interest if you're trying to recover a deleted file, or trying to understand why you can't recover a particular deleted file.

The Recycler:

When files and folders are sent to the Recycler they remain as live files, taking up allocated space and being protected from being overwritten. The file data is untouched but the file and folder names are amended.

In Windows XP files sent to the Recycler are relocated to a protected system directory named \Recycler\SID where SID is the SID of the user that performed the deletion. The files are renamed to D followed by the drive letter followed by an sequential index number, followed by the original file extension, e.g. Dc145.doc. There is also a file named INFO2 which contains entries, identified by the index numbers, that describe the original file and folder sizes and path/names, etc of all the files in the recycler.

When a folder is sent to the Recycler the folder name is changed in the same way as a file (but without any extension). Files and folders within that folder retain their original names, but are not shown separately in the recycler. Files previously deleted from the deleted folder have their own D name in the Recycler and are not grouped with the files that were deleted as part of the subsequent folder deletion.

In Vista, Windows 7 and Windows 8 the way the files are named and indexed within the Recycler is different. The Recycler is in a protected system directory named \$Recycle.Bin\SID, where SID is the SID of the user that performed the deletion. When a file is sent to the Recycler it is renamed to $R followed by a set of random characters, followed by the original file extension, e.g. $Rhdxenv.doc. A matching file is created as $I followed by the same random characters and extension as the $R file. This file contains the original filename/path and file size, and the date and time that the file was moved to the Recycle Bin (there is no INFO2 file). The $I files are all 544 bytes long.

When a folder is sent to the Recycler the folder name is changed in the same way as a file. Files and folders within that folder retain their original names, but are not shown separately in the recycler. A $I file is created to match the folder's $R file. As with XP,

files previously deleted from the deleted folder have their own $R and $I files in the Recycler and are not grouped with the files that were deleted as part of the subsequent folder deletion.

Neither the INFO2 file nor the $I files are in readable text. $I files are structured as follows:

- 0x00 - $I File header – always set to 0x01 followed by seven bytes of 0x00.
- 0x08 - Original file size
- 0x16 - Deleted date and time stamp – represented in number of seconds since Midnight, January 1, 1601
- 0x24 - Original file/path name

When a file is deleted from the Recycler the same action is taken as with a shift/del, the Recycler is just an interim stage before the actual deletion. However the file was deleted, NTFS treats them all the same. Aardvark.txt has to go.

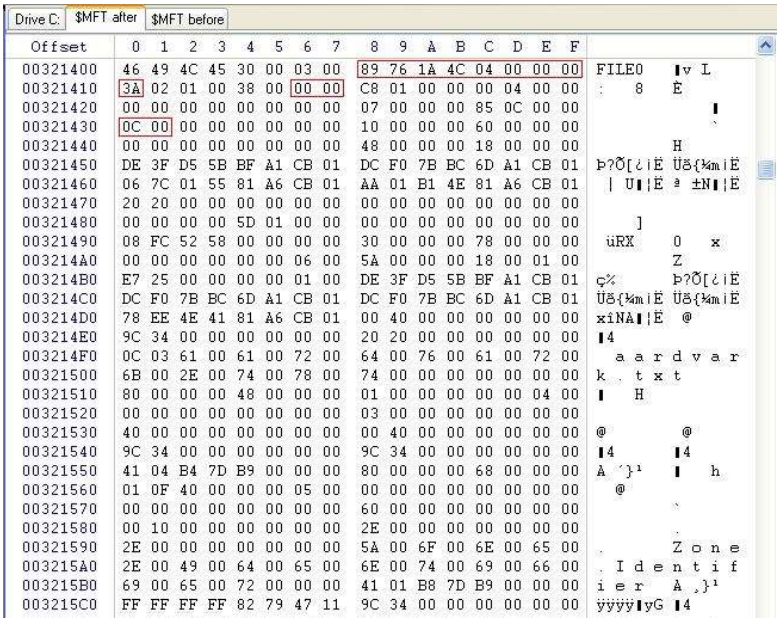
Deletion:

When a file is physically deleted, either by shift/del or emptying the Recycler, NTFS will modify its system files to reflect the deletion. The file's data, the clusters allocated to the file, are not modified or even accessed (and it is this that enables file recovery to be attempted). On file deletion NTFS will:

- Update all the file's MFT records to deleted state
- Remove the entry for the file from the owning folder's MFT record
- Update the MFT's bitmap to set the file's records as free for reuse
- Update the cluster bitmap ($Bitmap) to set the file's clusters as free for reuse

The pertinent MFT records aren't removed when a file is deleted, due to the use of absolute record numbers to relate records to indexes, and to themselves, etc.

Aardvark's MFT record after deletion


Drive C: $MFT after $MFT before

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00321400	46	49	4C	45	30	00	03	00	89	76	1A	4C	04	00	00	00
00321410	3A	02	01	00	38	00	00	00	C8	01	00	00	00	04	00	00
00321420	00	00	00	00	00	00	00	00	07	00	00	00	85	0C	00	00
00321430	0C	00	00	00	00	00	00	00	10	00	00	00	60	00	00	00
00321440	00	00	00	00	00	00	00	00	48	00	00	00	18	00	00	00
00321450	DE	3F	D5	5B	BF	A1	CB	01	DC	F0	7B	BC	6D	A1	CB	01
00321460	06	7C	01	55	81	A6	CB	01	AA	01	B1	4E	81	A6	CB	01
00321470	20	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00321480	00	00	00	00	5D	01	00	00	00	00	00	00	00	00	00	00
00321490	08	FC	52	58	00	00	00	00	30	00	00	00	78	00	00	00
003214A0	00	00	00	00	00	00	06	00	5A	00	00	00	18	00	01	00
003214B0	E7	25	00	00	00	00	01	00	DE	3F	D5	5B	BF	A1	CB	01
003214C0	DC	F0	7B	BC	6D	A1	CB	01	DC	F0	7B	BC	6D	A1	CB	01
003214D0	78	EE	4E	41	81	A6	CB	01	00	40	00	00	00	00	00	00
003214E0	9C	34	00	00	00	00	00	00	20	20	00	00	00	00	00	00
003214F0	0C	03	61	00	61	00	72	00	64	00	76	00	61	00	72	00
00321500	6B	00	2E	00	74	00	78	00	74	00	00	00	00	00	00	00
00321510	80	00	00	00	48	00	00	00	01	00	00	00	00	00	04	00
00321520	00	00	00	00	00	00	00	00	03	00	00	00	00	00	00	00
00321530	40	00	00	00	00	00	00	00	00	40	00	00	00	00	00	00
00321540	9C	34	00	00	00	00	00	00	9C	34	00	00	00	00	00	00
00321550	41	04	B4	7D	B9	00	00	00	80	00	00	00	68	00	00	00
00321560	01	0F	40	00	00	00	05	00	00	00	00	00	00	00	00	00
00321570	00	00	00	00	00	00	00	00	60	00	00	00	00	00	00	00
00321580	00	10	00	00	00	00	00	00	2E	00	00	00	00	00	00	00
00321590	2E	00	00	00	00	00	00	00	5A	00	6F	00	6E	00	65	00
003215A0	2E	00	49	00	64	00	65	00	6E	00	74	00	69	00	66	00
003215B0	69	00	65	00	72	00	00	00	41	01	B8	7D	B9	00	00	00
003215C0	FF	FF	FF	FF	82	79	47	11	9C	34	00	00	00	00	00	00

This is a rather uncomplicated file: after deletion there are relatively few changes to the record header, being:

- 0x08 - the logfile sequence number has changed
- 0x10 - the record usage count increases by one
- 0x16 - the flags field changes to 0x0000
- 0x30 - the fixup array value increases by one

The significant indication that this record refers to a deleted file and can be reused is the flags, being set to 0x0000 (or 0x0002 for a folder). The logfile sequence number reflects the entry in the logfile for this deletion (to ensure completion of the task or backout in case of error), and the increase in the record usage count is to ensure that this record no longer matches any record/seqno combination. The fixup array change is normal sector write validation.

There are of course other changes when a file is deleted. The bit in the MFT bitmap for this record will be set to zero, as will be the bits in the $Bitmap file that represent the file's data clusters. The MFT directory record for the file's parent folder will be updated with the file entry removed and the following entries moved up to take its place.

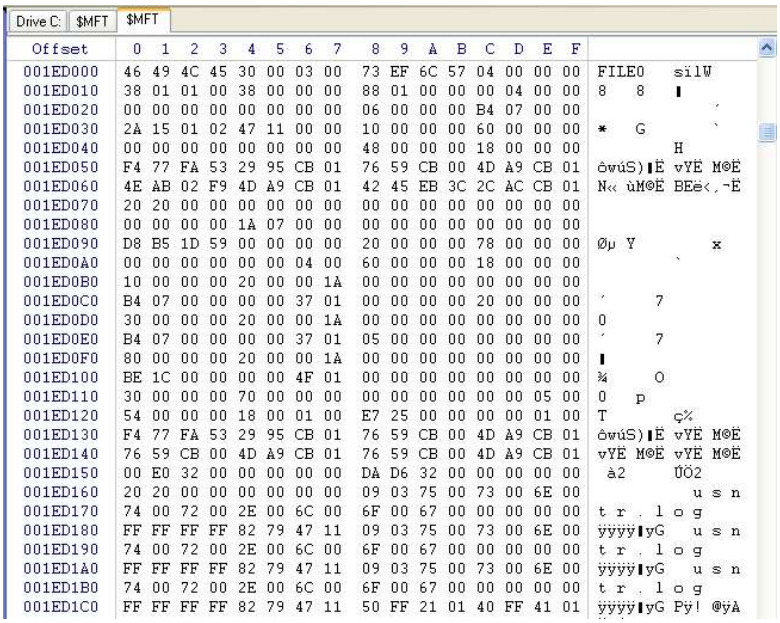
None of the file's MFT record's attributes have been altered. The record still contains the file name and data cluster location, enabling the record to be found and the data retrieved: at least until this MFT record is reused, or the data clusters are allocated in whole or part to another file.

With a larger file, with multiple dataruns, things are different.

Deleting a file with MFT extension records:

When a file which has an MFT record with extensions is deleted there are more extensive changes made to the record. Deleting such a record, again with shift/del, produces:

MFT record with extension after deletion



Drive C: | $MFT | $MFT

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
001ED000	46	49	4C	45	30	00	03	00	73	EF	6C	57	04	00	00	00
001ED010	38	01	01	00	38	00	00	00	88	01	00	00	00	04	00	00
001ED020	00	00	00	00	00	00	00	00	06	00	00	00	B4	07	00	00
001ED030	2A	15	01	02	47	11	00	00	10	00	00	00	60	00	00	00
001ED040	00	00	00	00	00	00	00	00	48	00	00	00	18	00	00	00
001ED050	F4	77	FA	53	29	95	CB	01	76	59	CB	00	4D	A9	CB	01
001ED060	4E	AB	02	F9	4D	A9	CB	01	42	45	EB	3C	2C	AC	CB	01
001ED070	20	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00
001ED080	00	00	00	00	1A	07	00	00	00	00	00	00	00	00	00	00
001ED090	D8	B5	1D	59	00	00	00	00	20	00	00	00	78	00	00	00
001ED0A0	00	00	00	00	00	00	04	00	60	00	00	00	18	00	00	00
001ED0B0	10	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
001ED0C0	B4	07	00	00	00	00	37	01	00	00	00	00	20	00	00	00
001ED0D0	30	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
001ED0E0	B4	07	00	00	00	00	37	01	05	00	00	00	00	00	00	00
001ED0F0	80	00	00	00	20	00	00	1A	00	00	00	00	00	00	00	00
001ED100	BE	1C	00	00	00	00	4F	01	00	00	00	00	00	00	00	00
001ED110	30	00	00	00	70	00	00	00	00	00	00	00	00	00	05	00
001ED120	54	00	00	00	18	00	01	00	E7	25	00	00	00	00	01	00
001ED130	F4	77	FA	53	29	95	CB	01	76	59	CB	00	4D	A9	CB	01
001ED140	76	59	CB	00	4D	A9	CB	01	76	59	CB	00	4D	A9	CB	01
001ED150	00	E0	32	00	00	00	00	00	DA	D6	32	00	00	00	00	00
001ED160	20	20	00	00	00	00	00	00	09	03	75	00	73	00	6E	00
001ED170	74	00	72	00	2E	00	6C	00	6F	00	67	00	00	00	00	00
001ED180	FF	FF	FF	FF	82	79	47	11	09	03	75	00	73	00	6E	00
001ED190	74	00	72	00	2E	00	6C	00	6F	00	67	00	00	00	00	00
001ED1A0	FF	FF	FF	FF	82	79	47	11	09	03	75	00	73	00	6E	00
001ED1B0	74	00	72	00	2E	00	6C	00	6F	00	67	00	00	00	00	00
001ED1C0	FF	FF	FF	FF	82	79	47	11	50	FF	21	01	40	FF	41	01

This time there are five changes in the MFT header, with the introduction of the reduction of the record size:

- 0x08 - the logfile sequence number has changed
- 0x10 - the record usage count increases by one
- 0x16 - the flags field changes to 0x0000
- 0x18 - the MFT record size is reduced from 0x1C8 to 0x188.
- 0x30 - the fixup array value increases by one

In the $Standard_Information attribute the MFT Record changed time has been updated.

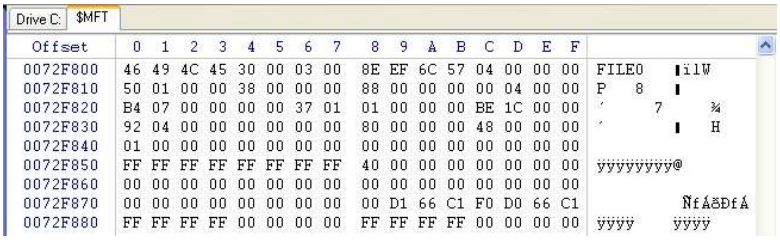
In the $Attribute_List attribute the length has been reduced so that it now holds only one datarun - the first - instead of three.

The $File_Name attribute has been moved to follow the shortened $Attribute_List but is otherwise unaltered.

Looking at the record it appears that the removal of two extension record addresses has been done in two steps: we can see the original 0xFFFFFFFF end attribute at 0x1C0, then another at 0x1A0 before the final resting place at 0x180, well and truly overwriting the other dataruns.

In the extension record there have been even more drastic chages.

MFT extension record after deletion



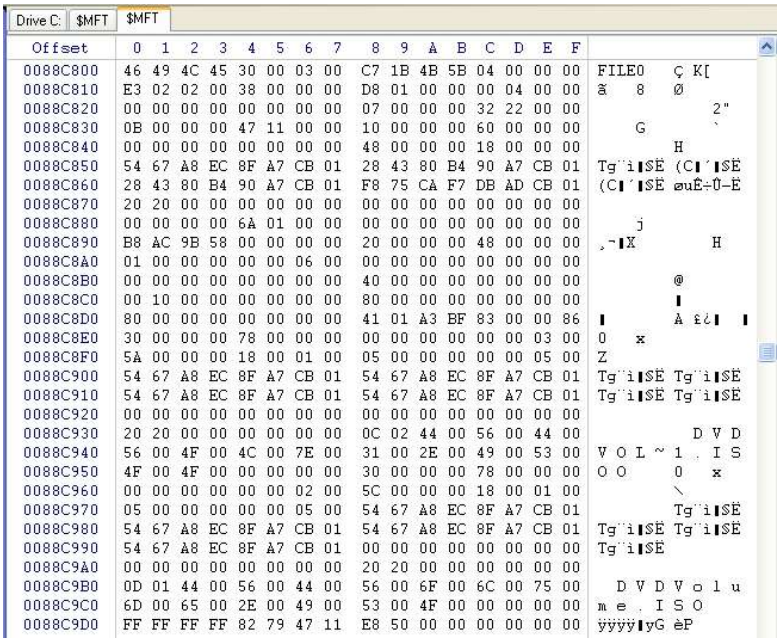
Once again there are the four standard changes to the MFT header, and in the $Data attribute

- 0x18 - the end virtual cluster number is set to 0xFF
- 0x28 - the allocated size of the data is set to 0x00
- 0x30 - the actual size of the data is set to zero
- 0x38 - the initialised size of the data is set to zero
- 0x40 - the start of the first datarun is set to zero, indicating a null run

From this we can see that there is no easy way to recover this deleted file. Although the files sizes are still available in the base MFT record's $File_Name attribute, the first extension record does not match the base record reference number, it has no valid datarun to identify the data, and even more disastrous the two other extension records are lost.

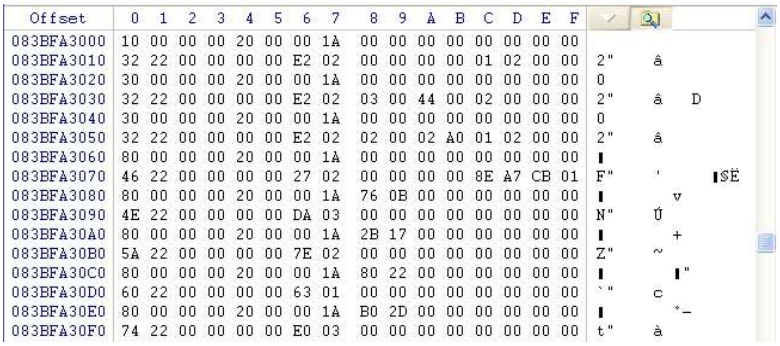
When the file with the non-resident $Attribute_list attribute is deleted the damage doesn't at first sight appear to be too bad:

MFT record with non-resident $Attribute_list after deletion



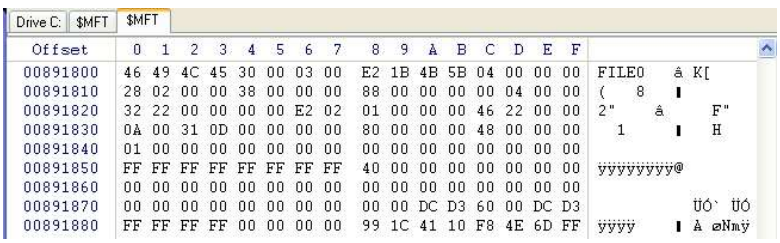
The standard four changes have been made, but the address of the Misc non-resident attributes cluster is unchanged.

Non-resident attribute cluster after deletion



And looking inside the Misc non-resident attributes cluster, this seems unchanged as well, although as part of the file's data allocation the cluster is now available for reuse.

MFT extension record after deletion



The extension record however has been wiped out as before, with the start of the datarun set to zero. Again there is no way to recover this file. The MFT base record for this file will not only show that it's been deleted, but the datruns have gone and - in this specific case at least - the file size in $File_Name is zero.

File recovery:

As long as the clusters holding the file data have not been re-allocated to another file then there's every chance that the file can be recovered, using one of the many applications available such as Piriform's Recuva. File recovery is greatly enhanced if the MFT record is still available. With the MFT record (and any extension records) intact the dataruns for all the file's extents can be followed. A recovery at a cluster level - i.e. reading each cluster and identifying the file signature - will usually only recover the first extent, as the file data does not commonly hold any file extent or even name information. Files with no signature, such as .txt or .bak files, won't be recognised during a cluster search, nor will files with unusual signatures: these files can't be easily recovered.

Undeleting a file:

File recovery is often incorrectly called undeletion. A file is recovered by copying its deleted data, and any metadata such as name etc., to a new location. For all practicable purposes a file can't be undeleted, it is not possible to 'switch' a file's MFT record back to live status.

It might appear that an intact simple MFT record for a deleted file - no extension records - could relatively easily be recovered by resetting the deletion fields (with the exception of the logfile sequence number, which would have to remain unchanged). However the MFT record for the owning folder would present difficulties as all evidence for the file would have been overwritten. Then the MFT and cluster bitmaps would have to be amended - correctly. Reconstruction would be an intensive and probably insuperable task.

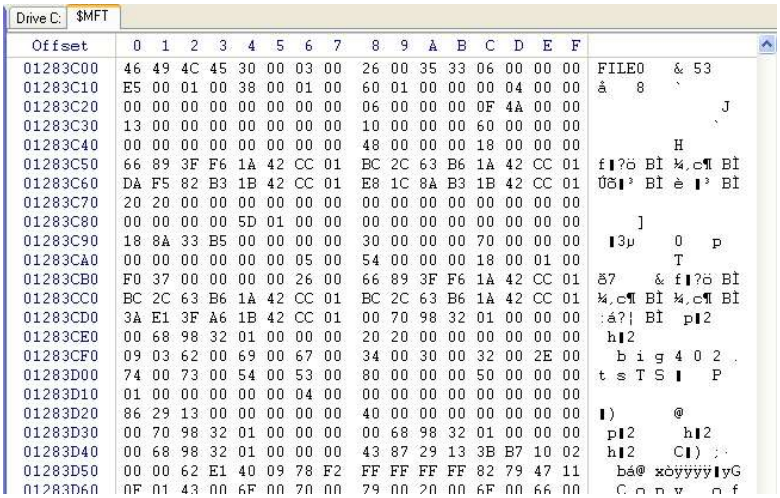
There is of course nothing to prevent editing the MFT with a hex editor. However NTFS is not so easily fooled, and any changes applied are simply backed out a few moments later, making the whole exercise, if carried out within Windows, pointless.

Why can't large files be recovered?

NTFS truncates MFT extension records and overwrites file size and location values. These files with many fragments, whatever their size, are unrecoverable after deletion (although a sector scan might still find the file clusters). However when a large file has only one fragment and one datarun, when it would be expected that the MFT record contents would still be accessible after deletion, it can also be unrecoverable.

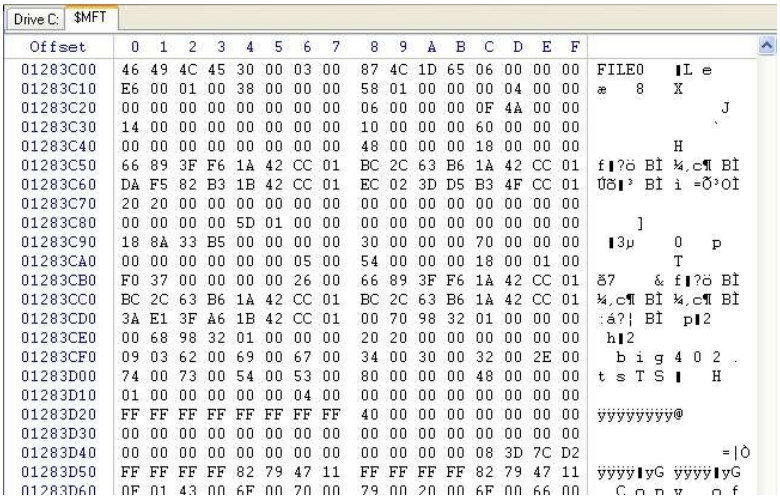
NTFS treats deletion of files over 4 gb in size in a different way from smaller files, with significant changes to data values.

Large file (4gb plus) before deletion



This is the MFT record for a file of 4.29 gb with one datarun - in one fragment. It is a straight-forward record, with the usual attributes and one datarun in the $Data attribute.

Large file (4gb plus) after deletion



The MFT record header has five changes, similar to the MFT record with extension records:

- 0x08 - the logfile sequence number has changed
- 0x10 - the record usage count increases by one
- 0x16 - the flags field changes to 0x0000
- 0x18 - the MFT record size is reduced from 0x160 to 0x158.
- 0x30 - the fixup array value increases by one

In the $Standard_Information attribute the MTF Record Changed time has updated.

In the $Data attribute

- 0x04 - the attribute size has been reduced from 0x50 to 0x48
- 0x18 - the end virtual cluster number is set to 0xFF
- 0x28 - the allocated size of the data is set to 0x00
- 0x30 - the actual size of the data is set to zero
- 0x38 - the initialised size of the data is set to zero
- 0x40 - the start of the first datarun is set to zero, indicating a null run

The MFT record for this file will show that it's been deleted and that the datrun has gone, but the file size is retrievable from the $File_Name attribute, although this is not necessarily up to date. Again there is no way to recover this file, apart from trying to read each data sector.

A similar file of 3.86 gb with a single datarun acted in the same way on deletion as a smaller file, with the four now familiar changes as described in previous examples. The $Data attribute remained unchanged, and the file size and cluster address could easily be obtained from the MFT record. It appears that the destruction of the $Data attribute occurs at or around 4 gb. Why does NTFS do more 'tidying up' on MFT records for large files? I really don't know.

Secure File Deletion:

Secure File Deletion is simply overwriting the MFT record data and the file cluster data with another byte string, usually zeroes but any byte pattern is equally effective. Disk manufacturers have spent countless millions of currency and hours to ensure that what was last written to a particular sector is what is returned to the user when read. There is no file system, and no software in the world, that will return what was previously written. The internet is full of stories about electron microscopes or

suchlike recovering overwritten data: no recovery company claims to be able to do this and no evidence is available showing that this has ever been done - there is overwhelming evidence that it can't be.

Multiple overwrites:

Despite the entirely misconcieved belief that muliple overwrites offer a 'more secure' method of deletion this is a complete fallacy. One overwrite of any data renders the data unrecoverable. The myth is based on a paper by Peter Gutmann published in 1996 that applied to what is now (and indeed was then) ancient Winchester disk technology. Even so he offered no evidence that it had been done in practice. Unfortunately cleanup software vendors still offer multiple overwrites, presumably for the gullible, misinformed, or paraniod.

Recovery after secure deletion:

It is not possible to recover data from a sector that has been overwritten. If any data is recovered, and quite often it can be, this is from edit copies, previous saves, defragging, autobackups, etc. In other words, copies from somewhere else.

Solid State Devices (SSD):

NTFS is NTFS, whatever the storage medium is. File deletion, and the chances of deleted file recovery, are entirely different when that storage medium is an SSD. All the above was written within the warm and familiar aegis of hard drives (HDDs). If you want to know about Solid State Devices (SSDs), then go to the Everything I Know About SSDs page *here*

You can return to my home page *here*

If you have any questions, comments or criticisms at all then I'd be pleased to hear them: please email me at kes at kcall dot co dot uk.