NTFS Documentation

Richard Russon
Yuval Fledel

NTFS Documentation

by Richard Russon and Yuval Fledel

Abstract

This is technical documentation, created to help the programmer.

It was originally written to complement the Linux NTFS driver [http://linux-ntfs.sourceforge.net/].

The latest version is available online at: http://linux-ntfs.sourceforge.net/ntfs/index.html and can be downloaded from: http://sourceforge.net/project/showfiles.php?group_id=13956

We're confident that the information is correct. We think we know where there are gaps in our knowledge. We may be wrong. Beware.

For simple answers to common questions, try reading the NTFS FAQ [http://linux-ntfs.sourceforge.net/info/ntfs.html].

Table of Contents

List of Tables

Chapter 1. Prologue

1. NTFS Documentation Preface

This is version 0.5 of the NTFS Documentation and is available as part of the Linux-NTFS Project [http://linux-ntfs.sourceforge.net/]

This is technical documentation, created to help the programmer.

It was originally written to complement the Linux NTFS driver [http://linux-ntfs.sourceforge.net/].

The latest version is available online at: http://linux-ntfs.sourceforge.net/ntfs/index.html and can be downloaded from: http://sourceforge.net/project/showfiles.php?group_id=13956

For simple answers to common questions, try reading the NTFS FAQ [http://linux-ntfs.sourceforge.net/info/ntfs.html].

2. About the NTFS Documentation

2.1. Overview

NTFS is the filesystem of Windows NT, 2000 and XP. It supports almost all POSIX features, all HFS features, and all HPFS features.

- It can deal with large capacity (up to 2^{46} GB) storage units.
- It has built-in data compression.
- It uses log file for transactions.
- Byte order: everything is little-endian on-disk.

2.2. Documentation Layout

- Chapter 1 - Prologue: is information describing the documentation.
- Chapter 2 - Files: is a list of the Metadata files.
- Chapter 3 - Attributes: is a list of Metadata attributes.
- Chapter 4 - Concepts: is a list of objects that are neither file, nor attribute.
- Chapter 5 - Epilogue: is some more information about the documentation.
- Appendix I - License: is the license under which the documentation is distributed.
- The Glossary: is a what's what of technical terminology

2.3. Accuracy

Microsoft hasn't released any documentation for NTFS. These documents have been pieced together partly by carefully reading all the SDKs and Windows help but mostly by reverse-engineering the filesystem.

We're confident that the information is correct. We think we know where there are gaps in our knowledge. We may be wrong. Beware.

2.4. Contact Points

You can post questions to an open forum on SourceForge [http://sourceforge.net/] at: http://sourceforge.net/forum/forum.php?forum_id=44084

If you'd like to get more involved in the Linux project, then you can join one of the mailing lists (both low volume).

- A general list for NTFS: http://tiger.informatik.hu-berlin.de/cgi-bin/mailman/listinfo/linux-ntfs
- A bit more technical one: http://lists.sourceforge.net/lists/listinfo/linux-ntfs-dev

Alternatively, if you have any questions, suggestions or corrections, please email me.

Richard Russon

2.5. License

2.6. Thanks

Many thanks to the following for their help preparing these documents.

- Albert Cahalan
- Alex Ionescu
- Anton Altaparmakov
- Bram Moolenaar (for vim)
- Damon Casale

- David Dillard
- Domagoj Pensa
- Helen Custer
- Martin von Löwis
- Olof Wolgast
- Rani Assaf
- Régis Duchesne
- Richard Russon
- Yuval Fledel

3. Tables Legend

3.1. Overview

The tables in this documentation aren't completely consistant. Below is a key to the tables showing how various fields are represented.

3.2. Footnotes

Any table fields that have footnote marks, e.g. (a), (e), will have a fuller description immediately below the table.

3.3. Size Fields

In NTFS not all fields are of a fixed size. Some depend on the value of another field, some depend on the contents of the field.

All the numbers in size fields are in decimal format. e.g. 12 (twelve), 42 (forty-two).

Table 1.1. Size fields table legend

Key	Name	Description
12	Fixed	This field is twelve bytes long. Its size is constant.
P8	Padding	P8 means pad the field to an 8 byte boundary. The size of this field could be 0 - 7 bytes. P4 means 4 byte alignment, etc (a)
V	Variable	The length of this field depends on its contents. An example is a SID. To know its length, you must decode the structure.
S	X-Ref	A cross-reference shows that the size is defined elsewhere in the table. The size can be represented by any letter, except P or V.

(a) Any padding of a fixed size will be displayed as a fixed size.

3.4. Indexes

Where a table represents an index, the key and data will be shown as below:

Table 1.2. An example for an index table

Offset	**Size**	**Description**	
0x00	2	Offset to data	
0x02	2	Size of data	
0x04	4	Key	SID
0x08	4	Data	Owner Id
0x0C	4	Data	Hash

3.5. Operating System

Note that the fields are not all used in exactly the same way. NT indicates old fields whereas 2K and XP indicate new fields.

Table 1.3. NTFS volume versions for each OS

OS	**NTFS**	**Description**
blank	all	Used by all versions of Windows
NT	1.2	Only used in Windows NT
2K	3.0	Windows 2000 and later
XP	3.1	New to Windows XP

```
repeating groups?
link padding8, padding and other table features to help/tables
consistant use of padding/alignment fields
```

4. Volume Layout

4.1. Overview

A freshly formatted NTFS volume will look like:

Table 1.4. Layout of a freshly formatted NTFS volume

B O O	M F T	Free Space	More Meta data	Free Space

T				

4.2. Notes

4.2.1. Other information

Everything is a file in NTFS. The index to these files is the Master File Table (MFT). The MFT lists the Boot Sector file ($Boot), located at the beginning of the disk. $Boot also lists where to find the MFT. The MFT also lists itself.

Located in the centre of the disk, we find some more Metadata files. The interesting ones are: $MFTMirr and $LogFile. The MFT Mirror is an exact copy of the first 4 records of the MFT. If the MFT is damaged, then the volume could be recovered by finding the mirror. The LogFile is journal of all the events waiting to be written to disk. If the machine crashes, then the LogFile is used to return the disk to a sensible state.

Hidden at the end of the volume, is a copy of the boot sector (cluster 0). The only Metadata file that makes reference to it is $Bitmap, and that only says that the cluster is in use.

4.2.2. MFT Zone

To prevent the MFT becoming fragmented, Windows maintains a buffer around it. No new files will be created in this buffer region until the other disk space is used up. The buffer size is configurable and can be 12.5%, 25%, 37.5% or 50% of the disk. Each time the rest of the disk becomes full, the buffer size is halved.

```
MFT Zone Reservation IS NOT STORED ON DISK
MFT Zone (reserved space for MFT)
  1 = 12.5%
  2 = 25.0%
  3 = 37.5%
  4 = 50.0%
  Where is this stored on disk?
  volume?  mft?  boot?
  This is the 'system files' space at
  the beginning of the disk.
  NtfsMftZoneReservation

link in to mft and bitmap
```

- cluster size 512 bytes, 1k, 2k, 4k, 8k, 16k, 32k, 64k
- very flexible, all the system files can be relocated, except $Boot
- supports streams named data streams
- attributes for a file can span several MFT records not necessarily contiguous or in order
- everything is an attribute, including the data
- filenames stored in Unicode
- journalling file system

- compression
- security
- hard links
- encryption
- LCNs vs VCNs

Chapter 2. NTFS Attributes

1. Overview

Each MFT FILE Record is built up from Attributes.

The list of possible Attributes is defined in $AttrDef.

Table 2.1. Standard NTFS Attributes

Type	OS	Name
0x10		$STANDARD_INFORMATION
0x20		$ATTRIBUTE_LIST
0x30		$FILE_NAME
0x40	NT	$VOLUME_VERSION
0x40	2K	$OBJECT_ID
0x50		$SECURITY_DESCRIPTOR
0x60		$VOLUME_NAME
0x70		$VOLUME_INFORMATION
0x80		$DATA
0x90		$INDEX_ROOT
0xA0		$INDEX_ALLOCATION
0xB0		$BITMAP
0xC0	NT	$SYMBOLIC_LINK
0xC0	2K	$REPARSE_POINT
0xD0		$EA_INFORMATION
0xE0		$EA
0xF0	NT	$PROPERTY_SET
0x100	2K	$LOGGED_UTILITY_STREAM

1.1. Notes

1.1.1. Other Information

$PROPERTY_SET, $SYMBOLIC_LINK and $VOLUME_VERSION existed in NTFS v1.2, but weren't used. They no longer exist in NTFS v3.0 (that used by Win2K).

Each MFT record has a Standard Header, followed by a list of attributes (in order of ascending Attribute Id) and an end marker. The end marker is just four bytes: 0xFFFFFFFF.

2. Attribute - $STANDARD_INFORMATION (0x10)

2.1. Overview

In old version of NTFS this Attribute contained little more than the DOS File Permissions and the file times.

Windows 2000 introduced four new fields which are used to reference Quota, Security, File Size and Logging information.

As defined in $AttrDef, this attribute has a minimum size of 48 bytes and a maximum of 72 bytes.

2.2. Layout of the Attribute (Resident)

Table 2.2. Layout of the $STANDARD_INFORMATION (0x10) attribute

Offset	Size	OS	Description
~	~		Standard Attribute Header
0x00	8		C Time - File Creation
0x08	8		A Time - File Altered
0x10	8		M Time - MFT Changed
0x18	8		R Time - File Read
0x20	4		DOS File Permissions
0x24	4		Maximum Number of Versions
0x28	4		Version Number
0x2C	4		Class Id
0x30	4	2K	Owner Id
0x34	4	2K	Security Id
0x38	8	2K	Quota Charged
0x40	8	2K	Update Sequence Number (USN)

2.2.1. File Permissions

Also called attributes in DOS terminology.

Table 2.3. File Permissions

Flag	Description
0x0001	Read-Only
0x0002	Hidden
0x0004	System
0x0020	Archive
0x0040	Device
0x0080	Normal
0x0100	Temporary
0x0200	Sparse File
0x0400	Reparse Point

Flag	Description
0x0800	Compressed
0x1000	Offline
0x2000	Not Content Indexed
0x4000	Encrypted

- Maximum Number of Versions

  Maximum allowed versions for file. Zero means that version numbering is disabled.

- Version Number

  This file's version (if any). Will be zero if Maximum Number of Versions is zero.

- Class Id

  Class Id from bidirectional Class Id index.

- Owner Id

  Owner Id of the user owning the file. This Id is a key in the $O and $Q Indexes of the file $Quota. If zero, then quotas are disabled.

- Security Id

  This should not be confused with a Security Identifier. The Security Id is a key in the $SII Index and $SDS Data Stream in the file $Secure.

- Quota Charged

  The number of bytes this file user from the user's quota. This should be the total data size of all streams. If zero, then quotas are disabled.

- Update Sequence Number (USN)

  Last Update Sequence Number of the file. This is a direct index into the file $UsnJrnl. If zero, the USN Journal is disabled.

2.3. Notes

2.3.1. Other Information

If a NTFS volume is upgraded from v1.2 to v3.0, then this attribute won't be upgraded (lengthened) until it is accesssed.

2.3.2. Questions

Are the Version fields and the Class Id ever used?

3. Attribute - $ATTRIBUTE_LIST (0x20)

3.1. Overview

When there are lots of attributes and space in the MFT record is short, all those attributes that can be made non-resident are moved out of the MFT. If there is still not enough room, then an $ATTRIBUTE_LIST attribute is needed. The remaining attributes are placed in a new MFT record and the $ATTRIBUTE_LIST describes where to find them. It is very unusual to see this attribute.

3.2. Layout of the Attribute

After the standard header, this attribute contains a list of variable length records, describing the type and location (in the MFT) of all the other attributes belonging to this file. Each record is aligned on an 8-byte boundary.

The list is sorted by:

1. Attribute type
2. Attribute name (if present)
3. Sequence number

N.B. It does not list itself.

Table 2.4. Layout of the $ATTRIBUTE_LIST (0x20) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00	4	Type
0x04	2	Record length
0x06	1	Name length (N)
0x07	1	Offset to Name (a)
0x08	8	Starting VCN (b)
0x10	8	Base File Reference of the attribute
0x18	2	Attribute Id (c)
0x1A	2N	Name in Unicode (if N >0)

(a) If the name doesn't exist, does this point at the attribute or zero?

(b) Starting VCN, or zero if the attribute is resident

(c) Each attribute has a unique identifier

```
(a) it always points to where the name would be (0x1A)
0x04 record allocation (8 byte alignment)
(c) always seems to be zero, check
(c) no it's only shown the first time for a given attribute type
not sure about sorting by sequence number.  VCN definitely
```

3.3. Notes

3.3.1. $AttrDef

It can be either resident or non-resident. This attribute has a no minimum or maximum size.

3.3.2. Other Information

The offset at 0x07 is just one byte long, unusual for an attribute.

If this attribute is non-resident, then the data runs must fit into one MFT record.

The $ATTRIBUTE_LIST may be needed if the file:

- has a large number of hard links (lots of file name attributes present).
- becomes very fragmented, so the data runs overflow the MFT record.
- has a complex security descriptor (not applicable to NTFS v3.0+
- has many named streams, e.g. data streams.

3.3.3. To Do

```
8 VCN lowest_vcn;
Lowest virtual cluster number of this portion of the attribute value. This is
is non-zero for the case where one attribute does not fit into one mft record
several mft records are allocated to hold this attribute. In the latter case,
record holds one extent of the attribute and there is one attribute list entry
extent. NOTE: This is DEFINITELY a signed value! The windows driver uses cmp,
by jg when comparing this, thus it treats it as signed.

24 __u16 instance;
If lowest_vcn = 0, the instance of the attribute being referenced; otherwise 0

The attribute list is used in case where a file need extension FILE records in
MFT to be fully described, in order to find any file attribute of this file.
This file attribute may be non-resident because its stream is likely to grow.

The extents of one non-resident attribute (if present) immediately follow
after the initial extent. They are ordered by lowest_vcn and have their instan
```

4. Attribute - $FILE_NAME (0x30)

4.1. Overview

This Attribute stores the name of the file attribute anl is always resident.

As defined in $AttrDef, this attribute has a minimum size of 68 bytes and a maximum of 578 bytes. This equates to a maximum filename length of 255 Unicode characters.

4.2. Layout of the Attribute (Resident)

Table 2.5. Layout of the $FILE_NAME (0x30) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00	8	File reference to the parent directory.
0x08	8	C Time - File Creation
0x10	8	A Time - File Altered
0x18	8	M Time - MFT Changed
0x20	8	R Time - File Read
0x28	8	Allocated size of the file
0x30	8	Real size of the file
0x38	4	Flags, e.g. Directory, compressed, hidden
0x3c	4	Used by EAs and Reparse
0x40	1	Filename length in characters (L)
0x41	1	Filename namespace 0x42 2L File name in Unicode (not null terminated)

4.2.1. File Reference

This is a File Reference to the base record of the parent directory.

4.2.2. File Size

The allocated size of a file is the amount of disk space the file is taking up. It will be a multiple of the cluster size. The real size of the file is the size of the unnamed data attribute. This is the number that will appear in a directory listing.

N.B. The Real Size is only present if the Starting VCN is zero. See the Standard Attribute Header for more information.

4.2.3. Flags

Table 2.6. File Flags

Flag	Description
0x0001	Read-Only
0x0002	Hidden
0x0004	System
0x0020	Archive
0x0040	Device
0x0080	Normal
0x0100	Temporary
0x0200	Sparse File
0x0400	Reparse Point
0x0800	Compressed
0x1000	Offline
0x2000	Not Content Indexed
0x4000	Encrypted

Flag	Description
0x10000000	Directory (copy from corresponding bit in MFT record)
0x20000000	Index View (copy from corresponding bit in MFT record)

4.3. Notes

4.3.1. Other Information

NTFS implements POSIX-style Hard Links by creating a file with several Filename Attributes. Each Filename Attribute has its own details and parent. When a Hard Linked file is deleted, its filename is removed from the MFT Record. When the last link is removed, then the file is really deleted.

N.B. All fields, except the parent directory, are only updated when the filename is changed. Until then, they just become out of date. $STANDARD_INFORMATION Attribute, however, will always be kept up-to-date.

N.B. If the file has EAs (Extended Attributes), then the EA Field will contain the size of buffer needed.

N.B. If the file is a Reparse Point, then the Reparse Field will give its type.

5. Attribute - $OBJECT_ID (0x40)

5.1. Overview

The Object Id was introduced in Windows 2000. Every MFT Record is assigned a unique GUID. Additionally, a record may have a Birth Volume Id, a Birth Object Id and a Domain Id, all of which are GUIDs.

As defined in $AttrDef, this attribute has a no minimum size but a maximum of 256 bytes.

5.2. Layout of the Attribute

Table 2.7. Layout of the $OBJECT_ID (0x40) attribute

Offset	Size	Name	Description
~	~	Standard Attribute Header	
0x00	16	GUID Object Id	Unique Id assigned to file
0x10	16	GUID Birth Volume Id	Volume where file was created
0x20	16	GUID Birth Object Id	Original Object Id of file
0x30	16	GUID Domain Id	Domain in which object was created

5.2.1. Birth Volume Id

Birth Volume Id is the Object Id of the Volume on which the Object Id was allocated. It never changes.

5.2.2. Birth Object Id

Birth Object Id is the first Object Id that was ever assigned to this MFT Record. I.e. If the Object Id is changed for some reason, this field will reflect the original value of the Object Id.

5.2.3. Domain Id

Domain Id is currently unused but it is intended to be used in a network environment where the local machine is part of a Windows 2000 Domain. This may be used in a Windows 2000 Advanced Server managed domain.

5.3. Notes

5.3.1. Other Information

This Attribute may be just 16 bytes long (the size of one GUID).

Even if the Birth Volume, Birth Object and Domain Ids are not used, they may be present, but one or more may be zero.

Need examples where all the fields are used.

6. Attribute - $SECURITY_DESCRIPTOR (0x50)

6.1. Overview

```
Standard Attribute Header?
```

The security descriptor can be summarised as:

- A header (may be flags), followed by one or two ACLs and two SIDs.
- The first ACL contains auditing information and may be absent.
- The second ACL contains permissions (who can do what).
- Each ACL contains one or many ACEs.
- Each ACE contains a SID.
- The last two SIDs show the owner of the object (User and Group)

Table 2.8. Layout of the $SECURITY_DESCRIPTOR (0x50) attribute

<table>
<tr><th colspan="3">Component</th><th>Description</th></tr>
<tr><td colspan="3">Header</td><td>Offsets to various structures</td></tr>
<tr><td>Audit ACL</td><td>ACE</td><td>SID</td><td>ACEs for the Audit ACL</td></tr>
<tr><td rowspan="3">Permissions ACL</td><td>ACE</td><td>SID</td><td rowspan="3">ACEs for the Permissions ACL</td></tr>
<tr><td>ACE</td><td>SID</td></tr>
<tr><td>ACE</td><td>SID</td></tr>
<tr><td colspan="3">SID (User)</td><td rowspan="2">The owner of this object</td></tr>
<tr><td colspan="3">SID (Group)</td></tr>
</table>

The security descriptor is necessary to prevent unauthorised access to files. It stores information about:

- The owner of the file
- Permissions the owner has granted to other users
- What actions should be logged (auditing)

6.2. Layout of the Attribute

6.2.1. Notes

6.2.1.1. Size

As defined in $AttrDef, this attribute has a no minimum or maximum size.

6.3. Layout of the stream

6.3.1. Questions

- How are the ACEs of directories inherited?
- How can we fit the ACEs into a normal looking Unix file system?
- How can we tie the file permissions into PAM or SMB?
- Can we use NT authentication, somehow?

6.3.2. To Do

- Decide which Standard, and Specific, Rights relate to which filesystem activities, e.g. FILE_APPEND_DATA will allow a user to extend a file, but not create one.
- Experiment to see if the zeros we see are padding and that the flag-like fields are flags.
- Experiment with the Generic Read / Write / Execute / All flags.

6.3.3. Header

Table 2.9. Layout of the $SECURITY_DESCRIPTOR (0x50) attribute header

Offset	Size	Description
0x00	1	Revision (a)
0x01	1	Padding
0x02	2	Control Flags (b)
0x04	4	Offset to User SID
0x08	4	Offset to Group SID

Offset	Size	Description
0x0C	4	Offset to SACL
0x10	4	Offset to DACL

(a) 0x1 for now

(b) Usually 0x4 (DACL Present), or 0x14 (DACL Present + SACL Present). See Flags below.

(c) This refers to the Auditing ACL

(d) This refers to the Permissions ACL

6.4. ACL

Table 2.10. Layout of an ACL

Offset	Size	Description
0x00	1	ACL Revision
0x01	1	Padding (0x00)
0x02	2	ACL size
0x04	2	ACE count
0x06	2	Padding (0x0000)

The Access Control List (ACL) contains one or many ACEs.

The ACL revision is currently 0x02, on my machine.

The Win32 APIs suggest that 0x01 and 0x06 contain padding 0x00's for alignment purposes.

6.5. ACE

Table 2.11. Layout of an ACE

Offset	Size	Description
0x00	1	Type
0x01	1	Flags
0x02	2	Size
0x04	4	Access mask
0x08	V	SID

6.5.1. Types

The currently implemented (in NT) Types are:

Table 2.12. ACE types

Value	Description
0x00	Access Allowed
0x01	Access Denied
0x02	System Audit

6.5.2. Flags

Flags is a bit field. The possible values of Flags depend on the value of Type. When applied to a directory, Access Allowed or Access Denied can have flags of

Table 2.13. ACE flags

Value	Description
0x01	Object inherits ACE
0x02	Container inherits ACE
0x04	Don't propagate 'Inherit ACE'
0x08	Inherit only ACE

If the Type is System Audit, then the flags can be

Table 2.14. ACE audit flags

Value	Description
0x40	Audit on Success
0x80	Audit on Failure

6.5.3. Access Mask / Access Rights

The Access Mask / Rights is a bit field enumerating all the (dis)allowed actions.

Table 2.15. ACE access mask

Bit(Range)	Meaning	Description / Examples
0 - 15	Object Specific Access Rights	Read data, Execute, Append data
16 - 22	Standard Access Rights	Delete, Write ACL, Write Owner
23	Can access security ACL	
24 - 27	Reserved	
28	Generic ALL (Read, Write, Execute)	Everything below
29	Generic Execute	All things necessary to execute a program
30	Generic Write	All things necessary to write to a file
31	Generic Read	All things necessary to read a file

6.6. SID (Security Identifier)

A typical SID looks like: S-1-5-21-646518322-1873620750-619646970-1110

It's composed of 'S-p-q-r-s-t-u-v'

Table 2.16. SID contents

S	Security
p	Revision number (currently 1)
q	NT Authority. This number is divided into 6 bytes (48 bit big-endian number).
r-v	NT Sub-authorities (there can be many of these)

On disk the SID is stored as follows:

in dec: S-1-5-21-646518322-1873620750-619646970-1110

in hex: S-1-5-15-26891632-6fad2f0e-24ef0ffa-456 (5 Sub-authorities)

Table 2.17. SID example

0x00	01	05	00	00	00	00	00	05
0x08	15	00	00	00	32	16	89	26
0x10	0e	2f	ad	6f	fa	0f	ef	24
0x18	56	04	00	00				

NB This is a variable length structure. The could have been more, or fewer, sub-authorities making the structure larger, or smaller.

6.6.1. Security Descriptor Control Flags

Table 2.18. Security Descriptor Control Flags

Flag	Description
0x0001	Owner Defaulted
0x0002	Group Defaulted
0x0004	DACL Present
0x0008	DACL Defaulted
0x0010	SACL Present
0x0020	SACL Defaulted
0x0100	DACL Auto Inherit Req
0x0200	SACL Auto Inherit Req
0x0400	DACL Auto Inherited
0x0800	SACL Auto Inherited
0x1000	DACL Protected
0x2000	SACL Protected

Flag	Description
0x4000	RM Control Valid
0x8000	Self Relative

6.6.1.1. OWNER DEFAULTED

This boolean flag, when set, indicates that the SID pointed to by the Owner field was provided by a defaulting mechanism rather than explicitly provided by the original provider of the security descriptor. This may affect the treatment of the SID with respect to inheritence of an owner.

6.6.1.2. GROUP DEFAULTED

This boolean flag, when set, indicates that the SID in the Group field was provided by a defaulting mechanism rather than explicitly provided by the original provider of the security descriptor. This may affect the treatment of the SID with respect to inheritence of a primary group.

6.6.1.3. DACL PRESENT

This boolean flag, when set, indicates that the security descriptor contains a discretionary ACL. If this flag is set and the Dacl field of the SECURITY DESCRIPTOR is null, then a null ACL is explicitly being specified.

6.6.1.4. DACL DEFAULTED

This boolean flag, when set, indicates that the ACL pointed to by the Dacl field was provided by a defaulting mechanism rather than explicitly provided by the original provider of the security descriptor. This may affect the treatment of the ACL with respect to inheritence of an ACL. This flag is ignored if the DaclPresent flag is not set.

6.6.1.5. SACL PRESENT

This boolean flag, when set, indicates that the security descriptor contains a system ACL pointed to by the Sacl field. If this flag is set and the Sacl field of the SECURITY DESCRIPTOR is null, then an empty (but present) ACL is being specified.

6.6.1.6. SACL DEFAULTED

This boolean flag, when set, indicates that the ACL pointed to by the Sacl field was provided by a defaulting mechanism rather than explicitly provided by the original provider of the security descriptor. This may affect the treatment of the ACL with respect to inheritence of an ACL. This flag is ignored if the SaclPresent flag is not set.

6.6.1.7. SELF RELATIVE

This boolean flag, when set, indicates that the security descriptor is in self-relative form. In this form, all fields of the security descriptor are contiguous in memory and all pointer fields are expressed as offsets from the beginning of the security descriptor.

```
The SID structure is a variable-length structure used to uniquely identify
users or groups. SID stands for security identifier.

The standard textual representation of the SID is of the form:
    S-R-I-S-S...
Where:
   - The first "S" is the literal character 'S' identifying the following
    digits as a SID.
```

```
   - R is the revision level of the SID expressed as a sequence of digits
  either in decimal or hexadecimal (if the later, prefixed by "0x").
   - I is the 48-bit identifier_authority, expressed as digits as R above.
   - S... is one or more sub_authority values, expressed as digits as above.

Example SID; the domain-relative SID of the local Administrators group on
Windows NT/2k:
    S-1-5-32-544

This translates to a SID with:
    revision = 1,
    sub_authority_count = 2,
    identifier_authority = {0,0,0,0,0,5},   SECURITY_NT_AUTHORITY
    sub_authority[0] = 32,                  SECURITY_BUILTIN_DOMAIN_RID
    sub_authority[1] = 544                  DOMAIN_ALIAS_RID_ADMINS

ACE Types
ACCESS_MIN_MS_ACE_TYPE           = 0
ACCESS_ALLOWED_ACE_TYPE          = 0
ACCESS_DENIED_ACE_TYPE           = 1
SYSTEM_AUDIT_ACE_TYPE            = 2
SYSTEM_ALARM_ACE_TYPE            = 3 Not implemented as of Win2k.
ACCESS_MAX_MS_V2_ACE_TYPE        = 3

ACCESS_ALLOWED_COMPOUND_ACE_TYPE = 4
ACCESS_MAX_MS_V3_ACE_TYPE        = 4

The following are Win2k only.
ACCESS_MIN_MS_OBJECT_ACE_TYPE    = 5
ACCESS_ALLOWED_OBJECT_ACE_TYPE   = 5
ACCESS_DENIED_OBJECT_ACE_TYPE    = 6
SYSTEM_AUDIT_OBJECT_ACE_TYPE     = 7
SYSTEM_ALARM_OBJECT_ACE_TYPE     = 8
ACCESS_MAX_MS_OBJECT_ACE_TYPE    = 8

ACCESS_MAX_MS_V4_ACE_TYPE        = 8

This one is for WinNT&2k.
ACCESS_MAX_MS_ACE_TYPE           = 8

The ACE flags (8-bit) for audit and inheritance

SUCCESSFUL_ACCESS_ACE_FLAG is only used with system audit and alarm ACE
types to indicate that a message is generated (in Windows!) for successful
accesses.

FAILED_ACCESS_ACE_FLAG is only used with system audit and alarm ACE types
to indicate that a message is generated (in Windows!) for failed accesses.

The inheritance flags.
OBJECT_INHERIT_ACE             = 0x01
CONTAINER_INHERIT_ACE          = 0x02
NO_PROPAGATE_INHERIT_ACE       = 0x04
INHERIT_ONLY_ACE               = 0x08
INHERITED_ACE                  = 0x10  Win2k only
VALID_INHERIT_FLAGS            = 0x1f

The audit flags.
SUCCESSFUL_ACCESS_ACE_FLAG     = 0x40
FAILED_ACCESS_ACE_FLAG         = 0x80

The access mask defines the access rights.

The standard rights.
```

```
DELETE                     = 0x00010000
READ_CONTROL               = 0x00020000
WRITE_DAC                  = 0x00040000
WRITE_OWNER                = 0x00080000
SYNCHRONIZE                = 0x00100000

STANDARD_RIGHTS_REQUIRED = 0x000f0000

STANDARD_RIGHTS_READ     = 0x00020000
STANDARD_RIGHTS_WRITE    = 0x00020000
STANDARD_RIGHTS_EXECUTE  = 0x00020000

STANDARD_RIGHTS_ALL      = 0x001f0000

The access system ACL and maximum allowed access types.
ACCESS_SYSTEM_SECURITY   = 0x01000000
MAXIMUM_ALLOWED          = 0x02000000

The generic rights.
GENERIC_ALL              = 0x10000000
GENERIC_EXECUTE          = 0x20000000
GENERIC_WRITE            = 0x40000000
GENERIC_READ             = 0x80000000

The object ACE flags (32-bit).
ACE_OBJECT_TYPE_PRESENT            = 1
ACE_INHERITED_OBJECT_TYPE_PRESENT  = 2

ACL_CONSTANTS
Current revision.
ACL_REVISION         = 2
ACL_REVISION_DS      = 4

History of revisions.
ACL_REVISION1        = 1
MIN_ACL_REVISION     = 2
ACL_REVISION2        = 2
ACL_REVISION3        = 3
ACL_REVISION4        = 4
MAX_ACL_REVISION     = 4
```

Absolute security descriptor. Does not contain the owner and group SIDs, nor the sacl and dacl ACLs inside the security descriptor. Instead, it contains pointers to these structures in memory. Obviously, absolute security descriptors are only useful for in memory representations of security descriptors. On disk, a self-relative security descriptor is used.

Attribute: Security descriptor (0x50). A standard self-relative security descriptor.

NOTE: Always resident.
NOTE: Not used in NTFS 3.0+, as security descriptors are stored centrally in FILE_$Secure and the correct descriptor is found using the security_id from the standard information attribute.

On NTFS 3.0+, all security descriptors are stored in FILE_$Secure. Only one referenced instance of each unique security descriptor is stored.

FILE_$Secure contains no unnamed data attribute, i.e. it has zero length. It does, however, contain two indexes ($SDH and $SII) as well as a named data stream ($SDS).

Every unique security descriptor is assigned a unique security identifier (security_id, not to be confused with a SID). The security_id is unique for

```
the NTFS volume and is used as an index into the $SII index, which maps
security_ids to the security descriptor's storage location within the $SDS
data attribute. The $SII index is sorted by ascending security_id.

A simple hash is computed from each security descriptor. This hash is used
as an index into the $SDH index, which maps security descriptor hashes to
the security descriptor's storage location within the $SDS data attribute.
The $SDH index is sorted by security descriptor hash and is stored in a B+
tree. When searching $SDH (with the intent of determining whether or not a
new security descriptor is already present in the $SDS data stream), if a
matching hash is found, but the security descriptors do not match, the
search in the $SDH index is continued, searching for a next matching hash.

When a precise match is found, the security_id coresponding to the security
descriptor in the $SDS attribute is read from the found $SDH index entry and
is stored in the $STANDARD_INFORMATION attribute of the file/directory to
which the security descriptor is being applied. The $STANDARD_INFORMATION
attribute is present in all base mft records (i.e. in all files and
directories).

If a match is not found, the security descriptor is assigned a new unique
security_id and is added to the $SDS data attribute. Then, entries
referencing the this security descriptor in the $SDS data attribute are
added to the $SDH and $SII indexes.

Note: Entries are never deleted from FILE_$Secure, even if nothing
references an entry any more.

The $SDS data stream contains the security descriptors, aligned on 16-byte
boundaries, sorted by security_id in a B+ tree. Security descriptors cannot
cross 256kib boundaries (this restriction is imposed by the Windows cache
manager). Each security descriptor is contained in a SDS_ENTRY structure.
Also, each security descriptor is stored twice in the $SDS stream with a
fixed offset of 0x40000 bytes (256kib, the Windows cache manager's max size)
between them; i.e. if a SDS_ENTRY specifies an offset of 0x51d0, then the
the first copy of the security descriptor will be at offset 0x51d0 in the
$SDS data stream and the second copy will be at offset 0x451d0.

$SII index. The collation type is COLLATION_NTOFS_ULONG.
$SDH index. The collation rule is COLLATION_NTOFS_SECURITY_HASH.
```

7. Attribute - $VOLUME_NAME (0x60)

7.1. Overview

This attribute simply contains the name of the volume.

As defined in $AttrDef, this attribute has a minimum size of 2 bytes and a maximum of 256 bytes. This equates to a maximum volume name length of 127 Unicode characters.

7.2. Layout of the Attribute

Table 2.19. Layout of the $VOLUME_NAME (0x60) attribute

Offset	Size	Description
~	~	Standard Attribute Header

Offset	Size	Description
0x00		Unicode name

7.3. Notes

The Volume Name is not terminated with a Unicode NULL. Its name's length is the size of the attribute as stored in the header.

A Volume's Serial Number is stored in $Boot.

8. Attribute - $VOLUME_INFORMATION (0x70)

8.1. Overview

Indicates the version and the state of the volume.

As defined in $AttrDef, this attribute has a minimum and a maximum size of 12 bytes.

8.2. Layout of the Attribute

Table 2.20. Layout of the $VOLUME_INFORMATION (0x70) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00	8	Always zero?
0x08	1	Major version number
0x09	1	Minor version number
0x0A	2	Flags
0x0C	4	Always zero?

8.2.1. Flags

Table 2.21. Volume Flags

Value	Description
0x0001	Dirty
0x0002	Resize LogFile
0x0004	Upgrade on Mount
0x0008	Mounted on NT4
0x0010	Delete USN underway
0x0020	Repair Object Ids
0x8000	Modified by chkdsk

8.3. Notes

8.3.1. Dirty Flag

When the Dirty Flag is set, Windows NT must perform the chkdsk/F command on the volume when it next boots.

8.3.2. Version numbers

Table 2.22. Volume Version Numbers

Operating System	Version
Windows NT	1.2
Windows 2000	3.0
Windows XP	3.1

8.3.3. Other Information

A Volume's Serial Number is stored in $Boot.

9. Attribute - $DATA (0x80)

9.1. Overview

This Attribute contains the file's data. A file's size is the size of its unnamed Data Stream.

As defined in $AttrDef, this attribute has a no minimum or maximum size.

9.2. Layout of the Attribute

Table 2.23. Layout of the $DATA (0x80) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00		Any data

9.3. Notes

9.3.1. Common Data Stream Used By Windows

- [Unnamed]
- {4c8cc155-6c1e-11d1-8e41-00c04fb9386d}
- ^EDocumentSummaryInformation

- ^ESebiesnrMkudrfcoIaamtykdDa
- ^ESummaryInformation
- $MountMgrDatabase
- $Bad
- $SDS
- $J
- $Max

9.3.2. Other Information

Usually, a directory has no Data Attribute, and the Data Attribute of a file has no name.

```
must have (at least empty) unnamed data attr
```

NTFS has an advantage: as you can have several data attributes for a file, you can easily implement HFS whose files are made of two parts (also called forks in the HFS terminology): a resource part and a data part. For the data part, you use default unnamed data attribute, and for the resource part, you use a data attribute named e.g. 'resource'.

10. Attribute - $INDEX_ROOT (0x90)

10.1. Overview

This is the root node of the B+ tree that implements an index (e.g. a directory). This file attribute is always resident.

```
Always resident.
```

10.2. Layout of the Attribute

```
link up below
```

$INDEX_ROOT

- Standard Attribute Header
- Index Root
- Index Header
- Index Entry

- Index Entry
- ...

10.2.1. Index Root

Table 2.24. Layout of the $INDEX_ROOT (0x90) attribute: an Index Root

Offset	Size	Description
~	~	Standard Attribute Header
0x00	4	Attribute Type
0x04	4	Collation Rule
0x08	4	Size of Index Allocation Entry (bytes)
0x0C	1	Clusters per Index Record
0x0D	3	Padding (Align to 8 bytes)

10.2.2. Index Header

Table 2.25. Layout of the $INDEX_ROOT (0x90) attribute: an Index Header

Offset	Size	Description
0x00	4	Offset to first Index Entry
0x04	4	Total size of the Index Entries
0x08	4	Allocated size of the Index Entries
0x0C	1	Flags
0x0D	3	Padding (align to 8 bytes)

10.2.3. Flags

Table 2.26. Index flags

Flag	Description
0x00	Small Index (fits in Index Root)
0x01	Large index (Index Allocation needed)

```
silly to have a flag of 0x00, remove
```

The large index flag indicates whether the file attributes index allocation and bitmap are present (when the index is small enough to be stored completely in the root node, these two file attributes are missing).

10.3. Notes

10.3.1. Size

As defined in $AttrDef, this attribute has a no minimum or maximum size.

10.3.2. Sequence of index entries

This is a sequence of index entries that has a variable length. The sequence is terminated with a special index entry whose last entry flag is set.

```
This is the header for indexes, describing the INDEX_ENTRY records, which
follow the INDEX_HEADER. Together the index header and the index entries
make up a complete index.

This is followed by a sequence of index entries (INDEX_ENTRY structures)
as described by the index header.

When a directory is small enough to fit inside the index root then this
is the only attribute describing the directory. When the directory is too
large to fit in the index root, on the other hand, two aditional attributes
are present: an index allocation attribute, containing sub-nodes of the B+
directory tree (see below), and a bitmap attribute, describing which virtual
cluster numbers (vcns) in the index allocation attribute are in use by an
index block.

NOTE: The root directory (FILE_$root) contains an entry for itself.

struct {
        ATTR_TYPES type;
        Type of the indexed attribute. Is $FILENAME for directories, zero
        for view indexes. No other values allowed.
        COLLATION_RULES collation_rule;        Collation rule used to sort the
        index entries. If type is $FILENAME, this must be COLLATION_FILENAME.

        __u32 bytes_per_index_block;
        Byte size of each index block (in the index allocation attribute).

        __u8 clusters_per_index_block;
        Cluster size of each index block (in the index allocation attribute),
        an index block is >= than a cluster, otherwise this will be the log of
        the size (like how the encoding of the mft record size and the index
        record size found in the boot sector work). Has to be a power of 2.
}  INDEX_ROOT;
```

10.4. List of Common Indexes

Table 2.27. Common Indexes

Name	Index Of	Used By
$I30	Filenames	Directories
$SDH	Security Descriptors	$Secure
$SII	Security Ids	$Secure
$O	Object Ids	$ObjId
$O	Owner Ids	$Quota
$Q	Quotas	$Quota

Name	Index Of	Used By
$R	Reparse Points	$Reparse

```
which elements are shared between indexes?
not relevant for index root
```

11. Attribute - $INDEX_ALLOCATION (0xA0)

11.1. Overview

This is the basic component of an index (e.g. a directory). This is the storage location for all sub-nodes of the B+ tree that implements an index (e.g. a directory). This file attribute is always non-resident.

As defined in $AttrDef, this attribute has a no minimum or maximum size.

```
this attribute is never resident - would use index root instead
```

11.2. Layout of the Attribute

It is simply a sequence of all index buffers that belong to the index.

Table 2.28. Layout of the $INDEX_ALLOCATION (0xA0) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00	...	Data runs

11.2.1. Index Entry

```
split into two tables, at least
```

Table 2.29. Layout of a data entry in the $INDEX_ALLOCATION (0xA0) attribute

Offset	Size	Description
~	~	Standard Attribute Header
The next field is only valid when the last entry flag is not set		
0x00	8	File reference
0x08	2	L = Length of the index entry
0x0A	2	M = Length of the stream

Offset	Size	Description
0x0C	1	Flags
The next field is only present when the last entry flag is not set		
0x10	M	Stream
The next field is only present when the sub-node flag is set		
L - 8	8	VCN of the sub-node in the index allocation attribute

11.2.2. Flags

Table 2.30. Data entry flags

Flag	Description
0x01	Index entry points to a sub-node
0x02	Last index entry in the node

The last entry flag is used to indicate the end of a sequence of index entries. Although it does not represent a valid file, it can point to a sub-node.

11.3. Notes

11.3.1. Length of the stream

A copy of the field at offset 10 in the header part of the resident file attribute indexed by the index entry. But why the hell haven't these 2 fields the same size?

11.3.2. Stream

A copy of the stream of the resident file attribute indexed by the index entry (e.g. for a directory, the file name attribute).

```
Always non-resident (doesn't make sense to be resident anyway!).

This is an array of index blocks. Each index block starts with an
INDEX_BLOCK structure containing an index header, followed by a sequence of
index entries (INDEX_ENTRY structures), as described by the INDEX_HEADER.

When creating the index block, we place the update sequence array at this
offset, i.e. before we start with the index entries. This also makes sense,
otherwise we could run into problems with the update sequence array
containing in itself the last two bytes of a sector which would mean that
multi sector transfer protection wouldn't work. As you can't protect data
by overwriting it since you then can't get it back...
When reading use the data from the ntfs record header.
```

12. Attribute - $BITMAP (0xB0)

12.1. Overview

This file attribute is a sequence of bits, each of which represents the status of an entity.

As defined in $AttrDef, this attribute has a no minimum or maximum size.

12.2. Layout of the Attribute

This attribute is currently used in two places: indexes (e.g. directories), $MFT. N.B. The index entries and the FILE records also have flags in them to show if they are in use or not.

In an index, the bit field shows which index entries are in use. Each bit represents one VCN of the index allocation.

In the $MFT, the bit field shows which FILE records are in use.

Table 2.31. Layout of the $BITMAP (0xB0) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00		Bit field

13. Attribute - $REPARSE_POINT (0xC0)

13.1. Overview

As defined in $AttrDef, this attribute has a no minimum size but a maximum of 16384 bytes.

13.2. Layout of the Attribute (Microsoft Reparse Point)

Table 2.32. Layout of the $REPARSE_POINT (0xC0) attribute (Microsoft Reparse Point)

Offset	Size	Description
~	~	Standard Attribute Header
0x00	4	Reparse Type (and Flags)
0x04	2	Reparse Data Length
0x06	2	Padding (align to 8 bytes)
0x08	V	Reparse Data (a)

13.3. Layout of the Attribute (Third-Party Reparse Point)

Table 2.33. Layout of the $REPARSE_POINT (0xC0) attribute (Third-Party Reparse Point)

Offset	Size	Description
~	~	Standard Attribute Header

Offset	Size	Description
0x00	4	Reparse Type (and Flags)
0x04	2	Reparse Data Length
0x06	2	Padding (align to 8 bytes)
0x08	16	Reparse GUID
0x18	V	Reparse Data (a)

```
(a) The structure of the Reparse Data depends on the Reparse Type. There are
    three defined Reparse Data (SymLinks, VolLinks and RSS) + the Generic Reparse.
```

13.3.1. Symbolic Link Reparse Data

Table 2.34. Symbolic Link Reparse Data

Offset	Size	Description
0x00	2	Substitute Name Offset
0x02	2	Substitute Name Length
0x04	2	Print Name Offset
0x08	2	Print Name Length
0x10	V	Path Buffer

13.3.2. Volume Link Reparse Data

Table 2.35. Volume Link Reparse Data

Offset	Size	Description
0x00	2	Substitute Name Offset
0x02	2	Substitute Name Length
0x04	2	Print Name Offset
0x08	2	Print Name Length
0x10	V	Path Buffer

13.3.3. Reparse Tag Flags

```
These are just the predefined reparse flags
```

Table 2.36. Reparse Tag Flags

Flag	Description
0x20000000	Is alias
0x40000000	Is high latency
0x80000000	Is Microsoft

Flag	Description
0x68000005	NSS
0x68000006	NSS recover
0x68000007	SIS
0x68000008	DFS
0x88000003	Mount point
0xA8000004	HSM
0xE8000000	Symbolic link

13.4. Notes

13.4.1. Other Information

```
The reparse point tag defines the type of the reparse point. It also
includes several flags, which further describe the reparse point.

The reparse point tag is an unsigned 32-bit value divided in three parts:

1. The least significant 16 bits (i.e. bits 0 to 15) specifiy the type of
   the reparse point.
2. The 13 bits after this (i.e. bits 16 to 28) are reserved for future use.
3. The most significant three bits are flags describing the reparse point.
   They are defined as follows:
     bit 29: Name surrogate bit. If set, the filename is an alias for
             another object in the system.
     bit 30: High-latecny bit. If set, accessing the first byte of data will
             be slow. (E.g. the data is stored on a tape drive.)
     bit 31: Microsoft bit. If set, the tag is owned by Microsoft. User
             defined tags have to use zero here.

The system file FILE_$Extend/$Reparse contains an index named $R listing
all reparse points on the volume. The index entry keys are as defined
below. Note, that there is no index data associated with the index entries.

The index entries are sorted by the index key file_id. The collation rule is
COLLATION_NTOFS_ULONGS. FIXME: Verify whether the reparse_tag is not the
primary key / is not a key at all. (AIA)
```

14. Attribute - $EA_INFORMATION (0xD0)

14.1. Overview

Used to implement under NTFS the HPFS extended attributes used by the information subsystem of OS/2 and OS/2 clients of Windows NT servers. This file attribute may be non-resident because its stream is likely to grow.

As defined in $AttrDef, this attribute has a minimum and a maximum size of 8 bytes.

14.2. Layout of the Attribute

Table 2.37. Layout of the $EA_INFORMATION (0xD0) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00	2	Size of the packed Extended Attributes
0x02	2	Number of Extended Attributes which have NEED_EA set
0x04	4	Size of the unpacked Extended Attributes

15. Attribute - $EA (0xE0)

15.1. Overview

Used to implement the HPFS extended attribute under NTFS. This file attribute may be non-resident because its stream is likely to grow.

As defined in $AttrDef, this attribute has a no minimum size but a maximum of 65536 bytes.

15.2. Layout of the Attribute

The Extended Attribute is a collection of name, value pairs.

Table 2.38. Layout of the $EA (0xE0) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00	4	Offset to next Extended Attribute
0x04	1	Flags
0x05	1	Name Length (N)
0x06	2	Value Length (V)
0x08	N	Name
N+0x08	V	Value

Conversely, the *Offset to next EA* is the size of this EA.

15.2.1. Flags

Table 2.39. EA flags

Value	Description
0x80	Need EA

15.3. Notes

15.3.1. Other Information

What is the role and the layout of the stream of this file attribute? It could be valuable to have a look at HPFS documentation.

15.3.2. Questions

Is it true that the EA name is not unicode?

16. Attribute - $LOGGED_UTILITY_STREAM (0x100)

16.1. Overview

As defined in $AttrDef, this attribute has a no minimum size but a maximum of 65536 bytes.

16.2. Layout of the Attribute

```
As an attribute it's no different to a named data attribute
Contents depend on the name of the $DATA stream
```

Table 2.40. Layout of the $LOGGED_UTILITY_STREAM (0x100) attribute

Offset	Size	Description
~	~	Standard Attribute Header
0x00		Any data

16.3. Notes

16.3.1. Other Information

Information needed

```
Operations on this attribute are logged to the journal ($LogFile) like
normal metadata changes.

Used by the Encrypting File System (EFS). All encrypted files have this
attribute with the name $EFS.

Can be anything the creator chooses.
EFS uses it as follows:
FIXME: Type this info, verifying it along the way. (AIA)
```

Chapter 3. NTFS Files

1. Overview

Everything on an NTFS volume is a file. There are two categories: Metadata and Normal. The Metadata files contain information about the volume and the Normal files contain your data.

1.1. Layout of the Volume

Below is a table of files found on a Win2K volume (Key).

Table 3.1. Layout of files on the Volume

Inode	Filename	OS	Description
0	$MFT		Master File Table - An index of every file
1	$MFTMirr		A backup copy of the first 4 records of the MFT
2	$LogFile		Transactional logging file
3	$Volume		Serial number, creation time, dirty flag
4	$AttrDef		Attribute definitions
5	. (dot)		Root directory of the disk
6	$Bitmap		Contains volume's cluster map (in-use vs. free)
7	$Boot		Boot record of the volume
8	$BadClus		Lists bad clusters on the volume
9	$Quota	NT	Quota information
9	$Secure	2K	Security descriptors used by the volume
10	$UpCase		Table of uppercase characters used for collating
11	$Extend	2K	A directory: $ObjId, $Quota, $Reparse, $UsnJrnl
12-15	<Unused>		Marked as in use but empty
16-23	<Unused>		Marked as unused
Any	$ObjId	2K	Unique Ids given to every file
Any	$Quota	2K	Quota information
Any	$Reparse	2K	Reparse point information
Any	$UsnJrnl	2K	Journalling of Encryption
>24	A_File		An ordinary file
>24	A_Dir		An ordinary directory
...	...		...

1.2. Notes

1.2.1. Unused Inodes

On a freshly formatted volume, inodes 0x0B to 0x0F are marked as in use, but empty. Inodes 0x10 to 0x17 are marked as free and not used. This doesn't change until the volume is under a lot of stress.

When the $MFT becomes very fragmented it won't fit into one FILE Record and an extension record is needed. If a new record was simply allocated at the end of the $MFT then we encounter a problem. The $DATA Attribute describing the location of the new record is in the new record.

The new records are therefore allocated from inode 0x0F, onwards. The $MFT is always a minimum of 16 FILE Records long, therefore always exists. After inodes 0x0F to 0x17 are used up, higher, unreserved, inodes are used.

This effect may not be limited to the $MFT, but more evidence is needed.

1.2.2. Other Information

For some reason $ObjId, $Quota, $Reparse and $UsnJrnl don't have inode numbers below 24, like the rest of the Metadata files.

Also, the sequence number for each of the system files is always equal to their mft record number and it is never modified.

2. NTFS Files: $MFT (0)

2.1. Overview

In NTFS, everything on disk is a file. Even the metadata is stored as a set of files.

The Master File Table (MFT) is an index of every file on the volume. For each file, the MFT keeps a set of records called attributes and each attribute stores a different type of information.

2.2. $MFT Attributes

Table 3.2. $MFT Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$MFT
0x80	$DATA	[Unnamed]
0xB0	$BITMAP	[Unnamed]

2.3. Layout of the File

2.3.1. Unnamed Data Stream

The description of each file is packed into FILE records.

If one record is not large enough (this is unusual), then an $ATTRIBUTE_LIST attribute is needed.

The first 24 FILE records are reserved for the system files. For a full list see the Files page.

Table 3.3. Sample records from the beginning of $MFT

Inode	Filename	Description
0	$MFT	Master File Table - An index of every file
1	$MFTMirr	A backup copy of the first 4 records of the MFT
2	$LogFile	Transactional logging file
3	$Volume	Serial number, creation time, dirty flag
...	...	...

2.4. Notes

2.4.1. MFT Zone

To prevent the MFT becoming fragmented, Windows maintains a buffer around it. No new files will be created in this buffer region until the other disk space is used up. The buffer size is configurable and can be 12.5%, 25%, 37.5% or 50% of the disk. Each time the rest of the disk becomes full, the buffer size is halved.

2.4.2. Other Information

The MFT is self-referencing.

The MFT has some space reserved for future expansion. MFT records 12 - 15 are marked as in use, but are empty. MFT records 16 - 23 are marked as not in use, however they are never used.

Under Windows, the MFT cannot shrink whilst the system is running.

3. NTFS Files: $MFTMirr (1)

3.1. Overview

This is a system file that duplicates at least the first four FILE records of the MFT for recovery purposes.

If the cluster size of the volume is less than or equal to four times the mft record size, i.e. usually the cluster size is less than or equal to 4096 bytes, then the first four MFT records are stored in the $MFTMirr.

If the cluster size is greater than four times the mft record size, then the size of $MFTMirr is one cluster and as many MFT records are stored in it as fit inside a cluster.

For example given an MFT record size of 1024 bytes and a cluster size of 8192 bytes the $MFTMirr would be 8192 bytes long and it would contain the first eight FILE records of the MFT.

3.2. $MFTMirr Attributes

Table 3.4. $MFTMirr Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$MFTMirr
0x80	$DATA	[Unnamed]

3.3. Layout of the File

3.3.1. Unnamed Data Stream

A copy of at least the first four FILE records of the $MFT.

Table 3.5. Layout of $MFTMirr

Inode	Filename	Description
0	$MFT	Master File Table - An index of every file
1	$MFTMirr	A backup copy of the first 4 records of the MFT
2	$Logfile	Transactional logging file
3	$Volume	Serial number, creation time, dirty flag
4	...	If present, further FILE records from the MFT (see $MFT)

4. NTFS Files: $LogFile (2)

4.1. Overview

4.2. $LogFile Attributes

Table 3.6. $LogFile Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$LogFile
0x80	$DATA	[Unnamed]

4.3. Layout of the File

4.3.1. Unnamed Data Stream

Little is known about the LogFile's structure.

4.4. Notes

4.4.1. Other Information

The logging area consists of a sequence of 4KB log records.

Each logrecord is structured as follows:

```
offset(length)   contents
0(4)             Magic number 'RCRD'
1E(12)           Fixup
```

The logrecord supposedly contains a sequence of variable sized records. The structuring of those is not clear. File 2 is $LogFile, which contains transaction records to guarantee data integrity in case of a system failure. As pp. 37 describe, it consists of 2 copies of the restart area, and the 'infinite' logging area.

When you want to write a file on a storage unit, you have to update the file itself plus some tables of the filesystem (say as an example the date of the file). At this point, you need a transaction made of 2 operations (update the file itself, update the date of the file).

If the transaction is realized, you are sure that your file is written on the storage unit, and that the filesystem has been left in a defined state.

If the transaction is not realized (in case of e.g. power failure, or system failure in general), the filesystem is in an undefined state. The only way for you to put it back in a defined (and sane) state (this operation is called a roll-back) is to log in a special file, the log file, which operations of the transaction have been successfully completed.

At the first access to the disk after a system failure, the system read the log file and rolls back all the operations to the beginning of the last transaction.

- When the system writes to the log file, the operation must be atomic and immediate.
- You can put back your volume in sane state in a short time which is not related to the size of your disk but only to the complexity of the transaction that failed. Note: This operation is not performed by the Windows NT chkdsk utility, but by the system: this normal and reliable operation is a feature of NTFS.
- If your hardware is reliable, you are sure that you always have access to all the files of your volume, because it is consistent. But you can't restore eventual data losses.

Log file organization:

Two restart areas present in the first two pages (restart pages). When the volume is unmounted they should be identical.

These are followed by log records organized in pages headed by a record header going up to log file size.

Not all pages contain log records when a volume is first formatted, but as the volume ages, all records will be used.

When the log file fills up, the records at the beginning are purged (by modifying the oldest_lsn to a higher value presumably) and writing begins at the beginning of the file. Effectively, the log file is viewed as a circular entity.

Log file restart page header (begins the restart area):

```
struct {
  NTFS_RECORD;              The magic is "RSTR".
  __u64 chkdsk_lsn;         The check disk log file sequence
                            number for this restart page.
                            Only used when the magic is changed
                            to "CHKD". = 0
  __u32 system_page_size;   Byte size of system pages, has to be
                                      >= 512 and a power of 2. Use this
                            to calculate the required size of the
```

```
                             usa and add this to the
                             ntfs.usa_offset value. Then verify
                             that the result is less than the
                             value of the restart_offset. = 0x1000
    __u32 log_page_size;     Byte size of log file records,
                             has to be >= 512 and a power of 2.
                             = 0x1000
    __u16 restart_offset;    Byte offset from the start of the
                             record to the restart record.
                             Value has to be aligned to 8-byte
                             boundary. = 0x30
    __s16 minor_ver;         Log file minor version. Only check if
                             major version is 1. (=1 but >=1 is
                             treated the same and <=0 is also
                             ok)
    __u16 major_ver;         Log file major version (=1 but =0 is
                             ok)
  } RESTART_PAGE_HEADER;
```

Log file restart area record:

The offset of this record is found by adding the offset of the RESTART_PAGE_HEADER to the restart_offset value found in it.

```
  struct {
    __u64 current_lsn;       Log file record. = 0x700000, 0x700808
    __u16 log_clients;       Number of log client records
                             following the restart_area. = 1
    __u16 client_free_list;  How many clients are free(?). If !=
                             0xffff, check that log_clients >
                             client_free_list. = 0xffff
    __u16 client_in_use_list; How many clients are in use(?).
                             If != 0xffff check that log_clients
                                       > client_in_use_list. = 0
    __u16 flags;             ??? = 0
    __u32 seq_number_bits;   ??? = 0x2c or 0x2d
    __u16 restart_area_length;Length of the restart area.
                             Following checks required if version
                             matches. Otherwise, skip them.
                             restart_offset + restart_area_length
                             has to be <lt;= system_page_size.
                             Also, restart_area_length has to be
                                       >= client_array_offset +
                             (log_clients * 0xa0). = 0xd0
    __u16 client_array_offset;Offset from the start of this record
                             to the first client record if versions
                             are matched. The offset is otherwise
                             assumed to be (sizeof(RESTART_AREA) +
                             7) & ~7, i.e. rounded up to first
                             8-byte boundary. Either way, the
                             offset to the client array has to be
                             aligned to an 8-byte boundary. Also,
                             restart_offset + offset to the client
                             array have to be <lt;= 510. Also,
                             the offset to the client array +
                             (log_clients * 0xa0) have to be
                                       <lt;= SystemPageSize. = 0x30
    __u64 file_size;         Byte size of the log file. If the
                             restart_offset + the offset of the
                             file_size are > 510 then corruption
                             has occured. This is the very first
                             check when starting with the
```

```
                                restart_area as if it fails it means
                                that some of the above values will be
                                corrupted by the multi sector transfer
                                protection! If the structure is
                                deprotected then these checks are
                                futile of course.
                                Calculate the file_size bits and check
                                that seq_number_bits == 0x43 -
                                file_size bits. = 0x400000
    __u32 last_lsn_data_length;??? = 0, 0x40
    __u16 record_length;       Byte size of this record. If the
                                version matches then check that the
                                value of record_length is a multiple
                                of 8, i.e. (record_length + 7) &
                                ~7 == record_length. = 0x30
    __u16 log_page_data_offset;??? = 0x40
} RESTART_AREA;
```

Log file client record:

Starts at 0x58 even though AFAIU the above it should start at 0x60. Something fishy is going on. /-:

```
struct {
    __u64 oldest_lsn;          Oldest log file sequence number for
                                this client record. = 0xbd16951d
    __u64 client_restart_lsn;  ??? = 0x700000, 0x700827, 0x700d07
    __u16 prev_client;         ??? = 0x808, 0xd07, 0xd5d
    __u16 next_client;         ??? = 0x70
    __u16 seq_number;          ??? = 0, 4 size uncertain, Regis
                                calls this "volume clear flag" and
                                gives a size of one byte.
    __u16 client_name;         ??? = empty string??? size uncertain
} RESTART_CLIENT;
```

NOTE: Above client record is followed by 0xffffffff probably to indicate the end of the restart area. Then there are 8 bytes = 0, then one __u32 = 8, followed by the Unicode string "NTFS" and then zeroes till the end of the page. Is this important at all?

Log page record page header:

Each log page begins with this header and is followed by several LOG_RECORD structures.

```
struct {
    NTFS_RECORD;                          The magic is "RCRD".
    union {
        __u64 last_lsn;
        __u32 file_offset;
    } copy;
    __u32 flags;
    __u16 page_count;
    __u16 page_position;
    union {
        struct {
            __u64 next_record_offset;
            __u64 last_end_lsn;
        } packed;
    } header;
} RECORD_PAGE_HEADER;
```

Possible flags for log records:

```
enum {
   LOG_RECORD_MULTI_PAGE = 1,        ???
   LOG_RECORD_SIZE_PLACE_HOLDER = 0xffff,
         This has nothing to do with the log record.
         It is only so gcc knows to make the flags 16-bit.
}  LOG_RECORD_FLAGS;
```

Log record header:

```
struct {
   __u64 this_lsn;
   __u64 client_previous_lsn;
   __u64 client_undo_next_lsn;
   __u32 client_data_length;
   struct {
      __u16 seq_number;
      __u16 client_index;
   }  client_id;
   __u32 record_type;
   __u32 transaction_id;
   LOG_RECORD_FLAGS flags;
   __u16 reserved_or_alignment[3];
*** Now are at ofs 0x30 into struct. ***
   __u16 redo_operation;
   __u16 undo_operation;
   __u16 redo_offset;
   __u16 redo_length;
   __u16 undo_offset;
   __u16 undo_length;
   __u16 target_attribute;
   __u16 lcns_to_follow;             Number of lcn_list entries
                                     following this entry.
   __u16 record_offset;
   __u16 attribute_offset;
   __u32 alignment_or_reserved;
   __u32 target_vcn;
   __u32 alignment_or_reserved1;
   struct {              Only present if lcns_to_follow is not 0.
      __u32 lcn;
      __u32 alignment_or_reserved;
   }  lcn_list[0];
}  LOG_RECORD;
```

The restart area (supposedly) has a pointer into the log area, such as the first and last log records written and the last checkpoint record written. If the restart area is screwed, recovery will be very hard - therefore you have two copies of the restart areas.

Individual log records are identified by logical sequence numbers (LSNs). The log area wraps around, but the LSNs don't (at least not anytime soon), so they are used for identifying log records instead of the offset in the log file.

Any modification of meta data (such as updating the time stamp that the file system was opened) will result in log file actions, which in turn result in restart area changes. It might well be that the dirty bit is implicit rather than explicit: The file system is clean if the last log record says that there are no pending transactions.

5. NTFS Files: $Volume (3)

5.1. Overview

This is a system file containing information about the volume.

5.2. $Volume Attributes

Table 3.7. $Volume Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Volume
0x50	$SECURITY_DESCRIPTOR	
0x60	$VOLUME_NAME	
0x70	$VOLUME_INFORMATION	
0x80	$DATA	[Unnamed]

5.3. Layout of the File

5.3.1. Unnamed Data Stream

The $DATA attribute has zero length.

5.4. Notes

5.4.1. Other Information

The Serial Number of a volume is stored in $Boot.

This is the only Metadata File that uses the $VOLUME_NAME and $VOLUME_INFORMATION file attributes.

6. NTFS Files: $AttrDef (4)

6.1. Overview

This is a system file containing information about all the file attributes usable in a volume.

```
Attribute end marker 0xFFFFFFFF
```

6.2. $AttrDef Attributes

Table 3.8. $AttrDef Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$AttrDef
0x50	$SECURITY_DESCRIPTOR	
0x80	$DATA	[Unnamed]

6.3. Layout of the File

6.3.1. Unnamed Data Stream

Its layout is a sequence of records. Each record defines one file attribute, and its layout is:

Table 3.9. Layout of $AttrDef

Offset	Size	Description
0x00	128	Label in Unicode
0x80	4	Type
0x84	4	Display rule
0x88	4	Collation rule
0x8C	4	Flags
0x90	8	Minimum size
0x98	8	Maximum size

6.3.2. Display Rule

At the moment this is always zero

6.3.3. Collation Rule

At the moment this is always zero, but the possible values are:

Table 3.10. $AttrDef Collation Rules

Flag	Description
0x00	Binary
0x01	Filename
0x02	Unicode String
0x10	Unsigned Long
0x11	SID
0x12	Security Hash
0x13	Multiple Unsigned Longs

6.3.4. Flags

We've only witnessed three flags: 0x02, 0x40 and 0x80. It seems that 0x40 and 0x80 are never seen together. Therefore, the guess is that:

Table 3.11. $AttrDef Flags

Flag	Description
0x02	Indexed
0x40	Resident (always)
0x80	Non-Resident (allowed to be)

6.4. Notes

- $PROPERTY_SET existed, briefly, in NTFS v3.0. It was intended to support Native Structure Storage (NSS).
- obsolete: $VOLUME_VERSION and $SYMBOLIC_LINK appeared in WinNT but weren't used. They don't appear in either Win2K or WinXP.

6.4.1. Other Information

It should be possible to add user-defined attributes to this file.

$AttrDef has big WAS it? 36K?

yep in nt4 = 36K mostly blank

now 2560 = 15attrs + 1 blank (2.5K)

6.5. Examples

6.5.1. Windows NT Example

Table 3.12. $AttrDef example from Windows NT

Type	Name	Flags	IRN	Min Size	Max Size
0x10	$STANDARD_INFORMATION	0x40	R	0x30	0x30
0x20	$ATTRIBUTE_LIST	0x80	N	-	-
0x30	$FILE_NAME	0x42	IR	0x44	0x242
0x40	$VOLUME_VERSION	0x40	R	0x8	0x8
0x50	$SECURITY_DESCRIPTOR	0x80	N	-	-
0x60	$VOLUME_NAME	0x40	R	0x2	0x100
0x70	$VOLUME_INFORMATION	0x40	R	0xC	0xC
0x80	$DATA	0x00		-	-
0x90	$INDEX_ROOT	0x40	R	-	-
0xA0	$INDEX_ALLOCATION	0x80	N	-	-
0xB0	$BITMAP	0x80	N	-	-

Type	Name	Flags	IRN	Min Size	Max Size
0xC0	$SYMBOLIC_LINK	0x80	N	-	-
0xD0	$EA_INFORMATION	0x40	R	0x8	0x8
0xE0	$EA	0x00		-	0x10000

6.5.2. Windows 2000 and Windows XP Example

Table 3.13. $AttrDef example from Windows 2000/XP

Type	Name	Flags	IRN	Min Size	Max Size
0x10	$STANDARD_INFORMATION	0x40	R	0x30	0x48
0x20	$ATTRIBUTE_LIST	0x80	N	-	-
0x30	$FILE_NAME	0x42	IR	0x44	0x242
0x40	$OBJECT_ID	0x40	R	-	0x100
0x50	$SECURITY_DESCRIPTOR	0x80	N	-	-
0x60	$VOLUME_NAME	0x40	R	0x2	0x100
0x70	$VOLUME_INFORMATION	0x40	R	0xC	0xC
0x80	$DATA	0x00		-	-
0x90	$INDEX_ROOT	0x40	R	-	-
0xA0	$INDEX_ALLOCATION	0x80	N	-	-
0xB0	$BITMAP	0x80	N	-	-
0xC0	$REPARSE_POINT	0x80	N	-	0x4000
0xD0	$EA_INFORMATION	0x40	R	0x8	0x8
0xE0	$EA	0x00		-	0x10000
0xF0	$PROPERTY_SET	?	?	?	?
0x100	$LOGGED_UTILITY_STREAM	0x80	N	-	0x10000

7. NTFS Files: . (Root Directory) (5)

7.1. Overview

The Root Directory of an NTFS, called . (dot) is an ordinary directory. If the volume has Reparse Points then the directory will have a Named Data Stream called *$MountMgrDatabase*. See the Directory Page for more information.

7.2. Dot (.) Attributes

Table 3.14. Dot (.) Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	.
0x50	$SECURITY_DESCRIPTOR	

Type	Description	Name
0x80	$DATA	$MountMgrDatabase
0x90	$INDEX_ROOT	$I30
0xA0	$INDEX_ALLOCATION	$I30
0xB0	$BITMAP	$I30

7.3. Layout of the File

7.3.1. $MountMgrDatabase Data Stream

This Data Stream only exists when there are Reparse Points on the Volume. It consists of repeating groups of:

Table 3.15. Layout of Dot (.)

Offset	Size	Description
0x00	4	Size of entry
0x04	4	Flags? (bitfield?)
0x08	2	Offset to UNC Path
0x0A	2	Size of UNC Path
0x0C	2	Offset to data
0x0E	2	Size of data

7.4. Notes

7.4.1. Other Information

See the Directory Page for more information.

8. NTFS Files: $Bitmap (6)

8.1. Overview

This file lists which clusters are in use. Each bit in this file represents one LCN.

8.2. $Bitmap Attributes

Table 3.16. $Bitmap Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Bitmap
0x80	$DATA	[Unnamed]

8.3. Layout of the File

8.3.1. Unnamed Data Stream

The lowest bit represents the lowest numbered LCN. Thus:

Table 3.17. Layout of $Bitmap

Bit	**LCN**
00000001_2	0
00000010_2	1
00000100_2	2
...	etc

8.4. Notes

8.4.1. MFT Zone

To prevent the MFT becoming fragmented, Windows maintains a buffer around it. No new files will be created in this buffer region until the other disk space is used up.

The buffer size is configurable and can be 12.5%, 25%, 37.5% or 50% of the disk. Each time the rest of the disk becomes full, the buffer size is halved.

8.4.2. Other Information

The size of this file is always a multiple of 8 bytes (64 clusters). Because of this rounding-up, the $Bitmap will represent slightly more clusters than the disk has. These bit are always set to 1.

The backup copy of the boot sector lies in this no-mans-land the cluster is hence marked as in use.

In theory, on very small volume, this attribute could be resident. In practice, Windows crashes.

9. NTFS Files: $Boot (7)

9.1. Overview

This is the system file that allows the system to boot.

This metadata file points at the boot sector of the volume.

It contains information about the size of the volume, clusters and the MFT.

It is the only file that cannot be relocated.

9.2. $Boot Attributes

Table 3.18. $Boot Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Boot
0x50	$SECURITY_DESCRIPTOR	
0x80	$DATA	[Unnamed]

9.3. Layout of the File

9.3.1. Unnamed Data Stream

Table 3.19. Layout of $Boot

Offset	Size	Description
0x0000	3	Jump to the boot loader routine
0x0003	8	System Id: "NTFS "
0x000B	2	Bytes per sector
0x000D	1	Sectors per cluster
0x000E	7	Unused
0x0015	1	Media descriptor (a)
0x0016	2	Unused
0x0018	2	Sectors per track
0x001A	2	Number of heads
0x001C	8	Unused
0x0024	4	Usually 80 00 80 00 (b)
0x0028	8	Number of sectors in the volume
0x0030	8	LCN of VCN 0 of the $MFT
0x0038	8	LCN of VCN 0 of the $MFTMirr
0x0040	4	Clusters per MFT Record (c)
0x0044	4	Clusters per Index Record (c)
0x0048	8	Volume serial number
~	~	~
0x0200		Windows NT Loader

9.4.

(a) A media descriptor of 0xF8 means a hard disk.

(b) A value of 80 00 00 00 has been seen on a USB thumb drive which is formatted with NTFS under Windows XP. Note this is removable media and is not partitioned, the drive as a whole is NTFS formatted.

(c) This can be negative, which means that the size of the MFT/Index record is smaller than a cluster. In this case the size of the MFT/Index record in bytes is equal to 2^(-1 * Clusters per MFT/Index record). So for example if Clusters per MFT Record is 0xF6 (-10 in decimal), the MFT record size is 2^(-1 * -10)

= 2^10 = 1024 bytes.

9.5. Notes

9.5.1. Other Information

The first 40 bytes are the same as for FAT boot sectors, except that unused fields are zeroed.

Because this file begins with a boot sector, it must start at physical cluster 0 (this is the only cluster that NTFS can not move). This forces the data attribute of this file to be non-resident. Consequently, the copy of the boot sector (critical data) can be located anywhere on the volume.

For crash recovery purposes Windows NT 3.51 saves a copy of the boot sector and puts it in the logical middle of the volume. Windows NT and later put it at the end of the volume.

10. NTFS Files: $BadClus (8)

10.1. Overview

This Metadata file contains a list of all the bad clusters on the volume.

The file is sparse, with the only data runs pointing at bad clusters.

Naturally the file cannot be read.

10.2. $BadClus Attributes

Table 3.20. $BadClus Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$BadClus
0x80	$DATA	[Unnamed]
0x80	$DATA	$Bad

10.3. Layout

10.3.1. Unnamed Data Stream

This is always zero length.

10.3.2. $Bad Data Stream

It is a file the size of the volume. Any cluster that is OK, is represented by a sparse (zero) cluster. Any bad cluster points to that cluster on disk.

10.4. Notes

10.4.1. Other Information

A cluster is bad if it contains at least one bad sector.

Because this system file works as any other file, all the bad clusters are marked as used in the $Bitmap system file, so they can never ever be used by any other file.

NTFS support hot-fixing: no more FAT's "Abort, Retry, Fail?". If a new bad cluster is found while the system is running, it is silently added to this file. If the cluster was on a fault tolerant volume, ftdisk (the fault tolerant volume driver) reconstitutes the data and NTFS stores them in another free cluster.

- Has an empty unnamed data stream.
- Has a named ($Bad) data stream, the size of the volume (sparse)
- entire volume of clusters (VCN)
- allocated size = volume size (bytes)
- attribute size = volume size (bytes)
- initialised size = 0 (or is one of above, redundant)
- runs imply sparse file size of volume
- initialised = 0 implies completely sparse

This file deals with Clusters not sectors. The Cluster is the smallest unit of disk space that NTFS will use.

11. NTFS Files: $Secure (9)

11.1. Overview

In NTFS v1.2, every file had a $SECURITY_DESCRIPTOR Attribute. It was inefficient to read and check these for every file access and most of them were the same.

NTFS v3.0 introduced a new Metadata File $Secure.

A new field in $STANDARD_INFORMATION, the Security Id, is a index into $Secure.

There is a Data Stream, $SDS, and two indexes $SII and $SDH.

The Data Stream has a copy of every $SECURITY_DESCRIPTOR Attribute on the volume, and the indexes cross-reference everything.

11.2. $Secure Attributes

Table 3.21. $Secure Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Secure

Type	Description	Name
0x80	$DATA	$SDS
0x90	$INDEX_ROOT	$SDH
0x90	$INDEX_ROOT	$SII
0xA0	$INDEX_ALLOCATION	$SDH
0xA0	$INDEX_ALLOCATION	$SII
0xB0	$BITMAP	$SDH
0xB0	$BITMAP	$SII

11.3. Layout of the File

11.3.1. $SDS Data Stream

The Security Descriptor Stream ($SDS) contains a list of all the Security Descriptors on the volume.

Each entry is padded to a 16 byte boundary and has a hash for indexing purposes.

Table 3.22. Layout of $Secure:$SDS

Offset	Size	Description
0x00	4	Hash of Security Descriptor
0x04	4	Security Id
0x08	8	Offset of this entry in this file
0x10	4	Size of this entry
0x04	V	Self-relative Security Descriptor
V+0x04	P16	Padding

```
sorted by security id
Self-relative? == has 2 * SID
generally a large file, not all used
there may be missing entries - test
large block of ids at start, then junk, then another block at 256KB
```

11.3.2. $SDH Index

The Security Descriptor Hash ($SDH) Index

Table 3.23. Layout of $Secure:$SDH

Offset	Size	Value	Description
~	~	~	Standard Index Header
0x00	2	0x18	Offset to data
0x02	2	0x14	Size of data
0x04	4	0x00	Padding
0x08	2	0x30	Size of Index Entry

Offset	Size	Value	Description	
0x0A	2	0x08	Size of Index Key	
0x0C	2		Flags	
0x0E	2	0x00	Padding	
0x10	4		Key	Hash of Security Descriptor
0x14	4		Key	Security Id
0x18	4		Data	Hash of Security Descriptor
0x1C	4		Data	Security Id
0x20	8		Data	Offset to Security Descriptor (in $SDS)
0x28	4		Data	Size of Security Descriptor (in $SDS)
0x2C	P8		Data	Padding

```
Last padding is always 4 bytes and always appears to be the Unicode string "II".
```

11.3.3. $SII Index

```
The Security Id Index ($SII)
```

Table 3.24. Layout of $Secure:$SII

Offset	Size	Value	Description	
~	~	~	Standard Index Header	
0x00	2	0x14	Offset to data	
0x02	2	0x14	Size of data	
0x04	4	0x00	Padding	
0x08	2	0x28	Size of Index Entry	
0x0A	2	0x04	Size of Index Key	
0x0C	2		Flags	
0x0E	2	0x00	Padding	
0x10	4		Key	Security Id
0x14	4		Data	Hash of Security Descriptor
0x18	4		Data	Security Id
0x1C	8		Data	Offset to Security Descriptor (in $SDS)
0x24	4		Data	Size of Security Descriptor (in $SDS)

```
This file is sorted by the hash.
The security descriptors are stored in the $SDS data stream.
surprisingly the offset (64 bit isn't 8 byte aligned)
```

11.4. Notes

11.4.1. Questions

- Why do some files still have a $SECURITY_DESCRIPTOR Attribute?
- How is the Security Hash generated?

12. NTFS Files: $UpCase (10)

12.1. Overview

This is a 128KB file full of capital letters.

For each character in the Unicode alphabet, there is an entry in this file.

It is used to compare and sort filenames.

12.2. $UpCase Attributes

Table 3.25. $UpCase Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$UpCase
0x80	$DATA	[Unnamed]

12.3. Layout of the File

12.3.1. Unnamed Data Stream

Table 3.26. Layout of $UpCase

Offset	Character
~	~
0x82	A
0x84	B
0x86	C
~	~

12.4. Notes

12.4.1. Other Information

This file allows the system to compare filenames independently of their code page.

13. NTFS Files: $Extend (11)

13.1. Overview

Windows 2K has introduced a new directory for metadata files.

This is a directory containing the Metadata files: $ObjId, $Quota, $Reparse and $UsnJrnl.

13.2. $Extend Attributes

Table 3.27. $Extend Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Extend
0x90	$INDEX_ROOT	$I30

13.3. Layout of the File

13.3.1. $I30 Index

This is an ordinary directory. There is no data stream for this file.

13.4. Notes

13.4.1. Other Information

Because there are only up to four files in this directory, there's never any need for an $INDEX_ALLOCATION and a $BITMAP.

14. NTFS Files: $ObjId (Any)

14.1. Overview

This system file is an index of all the $OBJECT_ID Attributes in use on the volume. See the $OBJECT_ID page for more details.

14.2. $ObjId Attributes

Table 3.28. $ObjId Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$ObjId
0x90	$INDEX_ROOT	$O
0xA0	$INDEX_ALLOCATION	$O

Type	Description	Name
0xB0	$BITMAP	$O

14.3. Layout of the File

14.3.1. $O Index

Table 3.29. Layout of $ObjId:$O

Offset	Size	Value	Description	
~	~	~	Standard Index Header	
0x00	2	0x20	Offset to data	
0x02	2	0x38	Size of data	
0x04	4	0x00	Padding	
0x08	2	0x58	Size of Index Entry	
0x0A	2	0x10	Size of Index Key	
0x0C	2		Flags	
0x0E	2	0x00	Padding	
0x10	16		Key	GUID Object Id
0x20	8		Data	MFT Reference
0x28	16		Data	GUID Birth Volume Id
0x38	16		Data	GUID Birth Object Id
0x48	16		Data	GUID Domain Id

14.3.2. Flags

Table 3.30. $ObjId flags

Flag	Description
0x01	Entry has subnodes
0x02	Last Entry

14.4. Notes

14.4.1. Other Information

The index is called *$O*. This is an index of Object Ids. It should not be confused with the index of the same name, used by the Metadata File $Quota.

The index, $O, is sorted by GUID (0x13). This Collation Rule is specified in the Index Root.

A file's $OBJECT_ID Attribute has a GUID that can be found in this Index. The Index's data provides an MFT reference back to the file.

15. NTFS Files: $Quota (NT:9, 2K:Any)

15.1. Overview

This file first appeared in Window NT, but wan't used. In Windows 2000, and later, it keeps track of file quotas.

Quotas are kept per person and per volume.

15.2. $Quota Attributes

Table 3.31. $Quota Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Quota
0x90	$INDEX_ROOT	$O
0x90	$INDEX_ROOT	$Q
0xA0	$INDEX_ALLOCATION	$O
0xA0	$INDEX_ALLOCATION	$Q
0xB0	$BITMAP	$O
0xB0	$BITMAP	$Q

15.3. Layout of the File

15.3.1. $O Index

Table 3.32. Layout of $Quota:$O

Offset	Size	Value	Description	
~	~	~	Standard Index Header	
0x00	2	0x1C	Offset to data	
0x02	2	0x04	Size of data	
0x04	4	0x00	Padding	
0x08	2	0x20	Size of Index Entry	
0x0A	2	0x0C	Size of Index Key (K)	
0x0C	2		Flags	
0x0E	2	0x00	Padding	
0x10	K		Key	SID
K+0x10	4		Data	Owner Id
K+0x14	P		Data	Padding8

```
Flags?
```

15.3.2. $Q Index

```
header & repeating group
```

Table 3.33. Layout of $Quota:$Q

Offset	Size	Value	Description	
~	~	~	Standard Index Header	
0x00	2	0x14	Offset to data	
0x02	2		Size of data	
0x04	4	0x00	Padding	
0x08	2		Size of Index Entry	
0x0A	2	0x04	Size of Index Key	
0x0C	4	0x00	Padding	
0x10	4		Key	Owner Id
0x14	4	0x02	Data	Version
0x18	4		Data	Flags
0x1C	8		Data	Bytes Used
0x24	8		Data	Change Time
0x2C	8		Data	Warning Limit
0x34	8		Data	Hard Limit
0x3C	8		Data	Exceeded Time
0x44	V		Data	SID
V+0x44	P	0x00	Data	Padding8

```
sid may be missing (quota flags = default limit => no SID, just padding)
padding may not be necessary
index key - xref to which index?
change time - date/time
exceeded time - 10/4/01 (not +5 days)
in the last (null) entry, the padding at 0x0C = 0x02
```

15.3.3. Flags

Table 3.34. $Quota flags

Flag	Description
0x0001	Default Limits
0x0002	Limit Reached
0x0004	Id Deleted
0x0010	Tracking Enabled
0x0020	Enforcement Enabled
0x0040	Tracking Requested

Flag	Description
0x0080	Log Threshold
0x0100	Log Limit
0x0200	Out Of Date
0x0400	Corrupt
0x0800	Pending Deletes

15.4. Notes

15.4.1. Other Information

The index is called *$O*. This is an index of Owner Ids. It should not be confused with the index of the same name, used by the Metadata File $ObjId.

A file's Owner Id is stored in the $STANDARD_INFORMATION Attribute. The Owner Id can be looked up in $O, to give a Security Id (SID) or looked up in $Q to provide quota information.

```
The $Q index contains one entry for each existing user_id on the
volume. The index key is the user_id of the user/group owning this
quota control entry, i.e. the key is the owner_id. The user_id of
the owner of a file, i.e. the owner_id, is found in the standard
information attribute. The collation rule for $Q is
COLLATION_NTOFS_ULONG.

The $O index contains one entry for each user/group who has been
assigned a quota on that volume. The index key holds the SID of
the user_id the entry belongs to, i.e. the owner_id. The collation
rule for $O is COLLATION_NTOFS_SID.

The $O index entry data is the user_id of the user corresponding
to the SID.
This user_id is used as an index into $Q to find the quota control
entry associated with the SID.
```

16. NTFS Files: $Reparse (Any)

16.1. Overview

Win2K can mount volumes and shares on top of existing directories. This is managed part in software and part by the volume itself.

16.2. $Reparse Attributes

Table 3.35. $Reparse Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	$Reparse
0x90	$INDEX_ROOT	$R

Type	Description	Name
0xA0	$INDEX_ALLOCATION	$R
0xB0	$BITMAP	$R

16.3. Layout of the File

16.3.1. $R Index

Table 3.36. Layout of $Reparse:$R

Offset	Size	Value	Description
~	~	~	Standard Index Header
0x00	2	0x1C	Offset to data
0x02	2	0x00	Size of data
0x04	4	0x00	Padding
0x08	2	0x20	Size of Index Entry
0x0A	2	0x0C	Size of Index Key
0x0C	2		Flags
0x0E	2	0x00	Padding
0x10	4		Key Reparse Tag (and Flags)
0x14	8		Key MFT Reference of Reparse Point
0x1C	4	0x00	Key Padding (align to 8 bytes)

```
0xA000003 flags - see $REPARSE_POINT
No data!
```

16.4. Notes

16.4.1. Other Information

More information needed.

17. NTFS Files: $UsnJrnl (Any)

17.1. Overview

A user-readable equivalent of the LogFile.

17.2. $UsnJrnl Attributes

Table 3.37. $UsnJrnl Attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	

Type	Description	Name
0x30	$FILE_NAME	$UsnJrnl
0x80	$DATA	$J
0x80	$DATA	$Max

17.3. Layout of the File

17.3.1. $J Data Stream

```
repeating group
```

Table 3.38. Layout of $UsnJrnl:$J

Offset	Size	Description
0x00	4	Size of entry
0x04	2	Major Version
0x06	2	Minor Version
0x08	8	MFT Reference
0x10	8	Parent MFT Reference
0x18	8	Offset of this entry in $J
0x20	8	Timestamp
0x28	4	Reason
0x2B	4	SourceInfo
0x30	4	SecurityID
0x34	4	FileAttributes
0x38	2	Size of filename (in bytes)
0x3A	2	Offset to filename
0x3C	V	Filename
V+0x3C	P	Padding (align to 8 bytes)

17.4.

17.4.1. $Max Data Stream

Table 3.39. Layout of $UsnJrnl:$Max

Offset	Size	Description
0x00	8	Maximum Size
0x08	8	Allocation Delta
0x10	8	USN ID (a)
0x18	8	Lowest Valid USN

(a) In version 2.0 of the USN Journal, Microsoft uses a FILETIME 64-bit value to randomize the USN ID. However, future versions might use another way to generate the ID, so it is not safe to assume this to be the time of the journals creation.

17.5. Notes

Version Number

The current version number is 2.0 (Major = 2, Minor = 0).

Reason Flags

Table 3.40. $UsnJrnl reason flags

Flag	Description
0x01	Data in one or more named data streams for the file was overwritten.
0x02	The file or directory was added to. 0x04 The file or directory was truncated.
0x10	Data in one or more named data streams for the file was overwritten.
0x20	One or more named data streams for the file were added to.
0x40	One or more named data streams for the file was truncated.
0x100	The file or directory was created for the first time.
0x200	The file or directory was deleted.
0x400	The user made a change to the file's or directory's extended attributes. These NTFS attributes are not accessible to Windows-based applications.
0x800	A change was made in the access rights to the file or directory.
0x1000	The file or directory was renamed, and the file name in this structure is the previous name.
0x2000	The file or directory was renamed, and the file name in this structure is the new name.
0x4000	A user changed the FILE_ATTRIBUTE_NOT_CONTENT_INDEXED attribute. That is, the user changed the file or directory from one that can be content indexed to one that cannot, or vice versa.
0x8000	A user has either changed one or more file or directory attributes or one or more time stamps.
0x10000	An NTFS hard link was added to or removed from the file or directory.
0x20000	The compression state of the file or directory was changed from or to compressed.
0x40000	The file or directory was encrypted or decrypted.
0x80000	The object identifier of the file or directory was changed.
0x100000	The reparse point contained in the file or directory was changed, or a reparse point was added to or deleted from the file or directory.
0x200000	A named stream has been added to or removed from the file, or a named stream has been renamed.
0x80000000	The file or directory was closed.

Source Info Flags

Table 3.41. $UsnJrnl source info flags

Flag	Description
0x01	The operation provides information about a change to the file or directory made by the operating system. A typical use is when the Remote Storage system moves data from external to local storage. Remote Storage is the hierarchical storage management software. Such a move usually at a minimum adds the USN_REASON_DATA_OVERWRITE (0x01) flag to a USN record.
0x02	The operation adds a private data stream to a file or directory. An example might be a virus detector adding checksum information. As the virus detector modifies the item, the system generates USN records. USN_SOURCE_AUXILIARY_DATA (0x02) indicates that the modifications did not change the application data.
0x04	The operation creates or updates the contents of a replicated file. For example, the file replication service sets this flag when it creates or updates a file in a replicated directory.

Chapter 4. NTFS Concepts

1. Overview

When a few lines in the glossary aren't enough.

1.1. Index

Table 4.1. NTFS Concepts

Concept	Description
Attribute Header	Standard Attribute Header
Attribute Id	Attribute Ids used in the MFT FILE Record
B*Tree	Balanced tree data structure, holds the NTFS directory tree
Clusters	LCNs, VCNs, sizes
Collation	Sorting and searching
Compression	File and directory level compression
Data Runs	Data runs
Directory	A typical directory on NTFS
File	A typical file on NTFS
FILE Record	An MFT File Record
Filename Namespace	Allowable filenames
FileRef	File References
Fixup	Sector fixups
Index Header	Standard Index Header
INDX Record	A directory index
Links	Hard and Symbolic links
Restart	LogFile Restart Area
SID	Built-in Security Identifiers
Sparse	Sparse files

2. Concept - Attribute Header

2.1. Overview

Every attribute in every MFT record has a standard header. The header stores information about the attribute's type, size, name (optional) and whether it is resident, or not.

The size of the attribute depends on two things. Does it have a name? Is it resident? To simplify the tables, all four possibilities will be shown in full (with some values already filled in).

2.2. Standard Attribute Header

2.2.1. Resident, No Name

Table 4.2. Layout of a resident unnamed attribute header

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x10, 0x60)
0x04	4		Length (including this header)
0x08	1	0x00	Non-resident flag
0x09	1	0x00	Name length
0x0A	2	0x00	Offset to the Name
0x0C	2	0x00	Flags
0x0E	2		Attribute Id (a)
0x10	4	L	Length of the Attribute
0x14	2	0x18	Offset to the Attribute
0x16	1		Indexed flag
0x17	1	0x00	Padding
0x18	L		The Attribute

(a) Each attribute has a unique identifier

2.2.2. Resident, Named

Table 4.3. Layout of a resident named attribute header

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x90, 0xB0)
0x04	4		Length (including this header)
0x08	1	0x00	Non-resident flag
0x09	1	N	Name length
0x0A	2	0x18	Offset to the Name
0x0C	2	0x00	Flags
0x0E	2		Attribute Id (a)
0x10	4	L	Length of the Attribute
0x14	2	2N+0x18	Offset to the Attribute (b)
0x16	1		Indexed flag
0x17	1	0x00	Padding
0x18	2N	Unicode	The Attribute's Name
2N+0x18	L		The Attribute (c)

(a) Resident attributes cannot be compressed.

(b) Each attribute has a unique identifier.

(c) Rounded up to a multiple of 4 bytes.

2.2.3. Non-Resident, No Name

Table 4.4. Layout of a non-resident unnamed attribute header

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x20, 0x80)
0x04	4		Length (including this header)
0x08	1	0x01	Non-resident flag
0x09	1	0x00	Name length
0x0A	2	0x00	Offset to the Name
0x0C	2		Flags
0x0E	2		Attribute Id (a)
0x10	8		Starting VCN
0x18	8		Last VCN
0x20	2	0x40	Offset to the Data Runs
0x22	2		Compression Unit Size (b)
0x24	4	0x00	Padding
0x28	8		Allocated size of the attribute (c)
0x30	8		Real size of the attribute
0x38	8		Initialized data size of the stream (d)
0x40	...		Data Runs

(a) Each attribute has a unique identifier

(b) Compression unit size = 2^{x} clusters. 0 implies uncompressed

(c) This is the attribute size rounded up to the cluster size

(d) When is this not equal to the allocated size?

2.2.4. Non-Resident, Named

Table 4.5. Layout of a non-resident named attribute header

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x80, 0xA0)
0x04	4		Length (including this header)
0x08	1	0x01	Non-resident flag
0x09	1	N	Name length
0x0A	2	0x40	Offset to the Name
0x0C	2		Flags
0x0E	2		Attribute Id (a)
0x10	8		Starting VCN

Offset	Size	Value	Description
0x18	8		Last VCN
0x20	2	2N+0x40	Offset to the Data Runs (b)
0x22	2		Compression Unit Size (c)
0x24	4	0x00	Padding
0x28	8		Allocated size of the attribute (d)
0x30	8		Real size of the attribute
0x38	8		Initialized data size of the stream (e)
0x40	2N	Unicode	The Attribute's Name
2N+0x40	...		Data Runs (b)

(a) Each attribute has a unique identifier

(b) Rounded up to a multiple of 4 bytes

(c) Compression unit size = 2^{x} clusters. 0 implies uncompressed

(d) This is the attribute size rounded up to the cluster size

(e) When is this not equal to the allocated size?

2.2.5. Flags

Table 4.6. Attribute flags

Flag	Description
0x0001	Compressed
0x4000	Encrypted
0x8000	Sparse

2.3. Notes

2.3.1. Other Information

Only the data attribute can be compressed, or sparse, and only when it is non-resident.

Although the compression flag is stored in the header, it does not affect the size of the header.

```
name isn't null terminated
```

```
FIXME
0x40 __s64 compressed_size;
Byte size of the attribute value after compression.
Only present when compressed. Always is a multiple of the cluster
size. Represents the actual amount of disk space being used on the disk.
```

FIXME: The indexed flag only appears in the resident attributes. Does this mean you can only index resident attributes?

3. Concept - Attribute Id

3.1. Overview

Every Attribute in every FILE Record has an Attribute Id. This Id is unique within the FILE Record and is used to maintain data integrity.

link to file record a field of the FILE Record each attribute has an id reused when zero skipped

Next Attribute Id

The Attribute Id that will be assigned to the next Attribute added to this MFT Record.

N.B. Incremented each time it is used.

N.B. Every time the MFT Record is reused this Id is set to zero.

N.B. The first instance number is always 0.

4. Concept - B*Trees

4.1. Overview

B+Trees

```
fixed order
height balanced
during add/remove of keys
minimal disturbance
pointers downwards only
```

4.2. Basic Terminology

- Key

  An object bearing data

- Leaf

  A key with no children

- Node

  A collection of keys

- Order

  A node of order n, has a maximum of n-1 keys

- Tree

An ordered data structure

- Root Node

  A node with no parent

- Median

  The ceil((n-1)/2)th key in a node

- Siblings

  Two keys in the same node, or two nodes with the same parent

- Depth

  The number of layers in the tree. Grandparent, parents, children = 3

- b-tree

  A balanced tree

- b+tree

  A balanced tree whose nodes are at least 1/2 full

- b*tree

  A balanced tree whose nodes are at least 2/3 full

4.3. NTFS Trees

```
    index root
    index allocation
    dummy keys
    data in non-leaf keys
    on-disk pointer only point down

What we have so far

    ...

Overview

    ...

Add Rules

    Find the first key that is larger than the new key
    (this will be a necessarily be a leaf)
    Insert the new key before this key (in the same node)
    While the node is full
        Split the current node in two
        Promote the median key to the parent
        Now consider the parent
    End

Delete Rules
```

```
Delete the key
If the key had children
    Find the successor and move it to this node
    Now consider the successor's old node
End
While the node isn't full enough
    If a sibling has enough keys
      steal one
    Else
      Combine with one of the sibling
    End
End
```

4.4. Discussion

A discussion log from #ntfs on IRC.

```
flatcap : hi _Oracle_
_Oracle_: hi there
flatcap : anything I can do for you?
_Oracle_: I was wondering about the B+ trees of ntfs
_Oracle_: they seem to be a bit awkward, or at least - not what I expected :)
flatcap : they _do_ seem strange, but they are perfect for filesystems
_Oracle_: no, i meant their on-disk representation
_Oracle_: they have a dummy node of sorts?
flatcap : the trees in ntfs aren't proper b+trees
flatcap : a dummy key
_Oracle_: that's exactly what I was hoping to hear!
flatcap : (thinking is still a bit hard this morning, bear with me :-)
_Oracle_: no problem ;-)
flatcap : the trees consist of a node, which contains keys
flatcap : the keys in a real (ideal world) b+tree are just separators, and the d
_Oracle_: right
_Oracle_: btw - how big is a node under ntfs? i mean, how many keys fit in there
flatcap : the INDX record is 4k, an you can get 10's of filenames in it
flatcap : but..., that depends on the lengths of the filenames
_Oracle_: i thought the number of keys in a node was a fixed property of a b+ tr
flatcap : hehe, usually, yes
flatcap : the keys of ntfs actually contain data and also a pointer to their chi
_Oracle_: so i noticed
AntonA  : one should add that INDX records of 2k size have also been seen in the
_Oracle_: really?
_Oracle_: what OS?
AntonA  : NT4
flatcap : because there's one more child than key, there has to be a dummy key (
_Oracle_: interesting...
AntonA  : some of my directories (e.g. c:\winnt and c:\program files) have 2k IN
_Oracle_: so the dummy key is always the "largest"?
flatcap : yes
_Oracle_: i see...
_Oracle_: so if the non-leaf nodes have data of themselves, wouldn't that make t
flatcap : I've just written a test program to help me understand the trees, whic
_Oracle_: I'd love to see that
flatcap : I read a lots of webpages and I think that the nearest term is a b*tre
_Oracle_: and how is it different from a b-tree?
flatcap : a b-tree maintains a minimum of 1/2 full nodes (except for the root no
flatcap : a b*tree changes the rules slightly and maintains 2/3 full
_Oracle_: so it just changes the rules of combining two nodes to one and such?
flatcap : exactly
```

```
_Oracle_: hmmm...
_Oracle_: let me think about that for a moment :)
flatcap : in a true b+tree, the data keys (leaves) should also have pointers to
flatcap : I'm going to write up everything I know about ntfs trees soon
_Oracle_: let me see if i got that...
_Oracle_: the index root points to the root INDX record
flatcap : you can see my test prog at:  http://linux-ntfs.bkbits.net:8080/tng-su
_Oracle_: each INDX record contains keys that have pointers to the files themsel
flatcap : yes
_Oracle_: I see
flatcap : the index root lives in the MFT record
_Oracle_: Yeah, this I managed to discover :)
flatcap : all the rest (index allocations) are non-res
_Oracle_: and the number of keys in a single INDX record is completely flexible?
AntonA  : yes
flatcap : yes, but there's a minimum
AntonA  : a minimum?
flatcap : yes, that's part of the tree algorithm
AntonA  : surely the minimum is a non-data containing terminator entry?
_Oracle_: what's the minimum?
flatcap : the minimum for a b+tree is 1/2 full, b* 2/3 full
flatcap : only the root node may contain fewer
_Oracle_: oh.
_Oracle_: yeah
AntonA  : and the last node...
flatcap : the keys are moved about to keep this true
flatcap : even the last node will have the "right number" in it
AntonA  : that would mean that in a really large directory deleting one file cou
flatcap : no, you might think that, but the balancing doesn't affect many other
flatcap : if the tree is 4 deep (NTFS equiv say 10^5 files), you'd only be alter
flatcap : I'll draw lots of pictures when I have a moment (probably tomorrow)
_Oracle_: that should be interesting to read!
flatcap : are you on our dev mailing list, _Oracle_
_Oracle_: What mailing list? (er... no.)
AntonA  : the major question that springs to my mind is what would windows ntfs
flatcap : hehe, I hate to think :-)
_Oracle_: I wouldn't want to be there, that's for sure
flatcap : chkdsk would probably try and rebalance it and you might find that ntf
_Oracle_: how do i join the list?
flatcap : http://lists.sourceforge.net/lists/listinfo/linux-ntfs-dev
AntonA  : um, it would be a lot easier to get directory operations working while
flatcap : I'll mail the list and answer questions there
AntonA  : if windows is able to pickup the pieces without complaint / failure, i
flatcap : yes possibly, but I think I know enough now to build something close e
flatcap : (I just wanted a big project where I could start without tripping over
AntonA  : cool
_Oracle_: I've got a few more questions if you have the time
AntonA  : As I said before. I am not going anywhere near directories. (-:
flatcap : sure
_Oracle_: Smaller ones, though
```

4.5. References

Here are some sites that I found helpful whilst writing the B-Tree code.

http://tide.it.bond.edu.au/inft320/003/lectures/physical.htm
http://cis.stvincent.edu/carlsond/swdesign/btree/btree.html
http://www.fit.qut.edu.au/~maire/baobab/baobab.html
http://www.fit.qut.edu.au/~maire/baobab/lecture/index.htm

5. Concept - Clusters

5.1. Overview

In NTFS, the Cluster is the fundamental unit of disk usage. The number of sectors that make up a cluster is always a power of 2, and the number is fixed when the volume is formatted. This number is called the Cluster Factor and is usually quoted in bytes, e.g. 8KB, 2KB. NTFS addresses everything by its Logical Cluster Number.

5.1.1. Logical Cluster Number (LCN)

Each cluster in a volume is given a sequential number. This is its Logical Cluster Number. LCN 0 (zero) refers to the first cluster in the volume (the boot sector).

To convert from an LCN to a physical offset in the volume, multiply the LCN by the Cluster Size.

5.1.2. Virtual Cluster Number (VCN)

Each cluster of a non-resident stream is given a sequential number. This is its Virtual Cluster Number. VCN 0 (zero) refers to the first cluster of the stream.

To locate the stream on disk, it's necessary to convert from a VCN to an LCN. This is done with the help of data runs.

5.1.3. Data Runs

Each contiguous block of LCNs is given a Data Run, which contains a VCN, an LCN and a length. When NTFS needs to to find an object on disk, it looks up the VCN in the Data Runs to get the LCN.

5.2. Notes

5.2.1. Other information

The Cluster Size can be chosen when the volume is formatted.

The Cluster Size for a volume is stored in $Boot. Also defined there is the size, in clusters, of an MFT File Record and an Index Record.

By using Cluster Numbers, NTFS can address larger disks than if sectors numbers were used.

A list of allowed and default cluster sizes is shown below.

Windows NT

512bytes, 1KB, 2KB or 4KB

Windows 2000, Windows XP

512bytes, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB or 64KB

Table 4.7. Default cluster size

Volume Size	Default Cluster Size
<512MB	Sector size
<1GB	1KB
<2GB	2KB
>2GB	4KB

5.2.2. Questions

Why does NTFS use Virtual Cluster Numbers?

6. Concept - Collation

6.1. Overview

To be able to search and sort objects under NTFS

Table 4.8. Collation types

Value	Name	Compare the Values as:
0x00	Binary	Binary, where the first byte is most significant
0x01	Filename	Unicode strings
0x02	Unicode	Unicode strings, except that upper case letters should come first
0x10	ULONG	An unsigned long (32 bits, little-endian)
0x11	SID	A security identifier
0x12	Security Hash	First compare by the Security Hash, then by Security Identifier
0x13	ULONGS	A set of unsiged longs (32 bits, little-endian)

6.2. Usage

Here are some examples of where various collation rules are used.

Table 4.9. Default collations types for standard indexes

Name	Used By
ULONG	$SII in file $Secure
SID	$O in file $Extend/$Quota
Security Hash	$SDH in file $Secure
ULONGS	$O in file $Extend/$ObjId

6.3. Notes

6.3.1. Questions

When comparing by ULONGS, where is the maximum length specified? Or, can two objects never have identical ULONGS?

```
0x13 ULONGS refers to GUIDs TEST
```

7. Concept - Compression

7.1. Overview

here's a short summary of the mechanism: data. These are compressed using a modified LZ77 algorithm. The basic idea is that substrings of the block which have been seen before are compressed by referencing the string rather than mentioning it again. For example, Consider the Plain text

```
#include <ntfs.h>\n
#include <stdio.h>\n
```

This is compressed to #include <ntfs.h>\n (-18,10)stdio(-17,4)

So the algorithm recognizes that -18 bytes from the current position, it has already seen the text '#include <'. Then, stdio is new, but '.h>\n' has been seen before.

The interesting details are in the question? How to encode the pair (-18,10), and how to mix this with plain-text strings. The first thing to understand is that such a pair is recorded in two bytes. Because a back-reference takes two bytes, there is no point in back-referencing one- or two-byte substrings. This means the shortest possible substring is 3. This means that length values of 0, 1, and 2 are not possible. So you can subtract 3 of the length before encoding it. Also, the references are always backward, and never 0. So you can store them as positive numbers, and subtract one. The first back-reference is stored as (17,7), and the second one as (16,1).

Given that a block is 4096 in size, you might need 12 bits to encode the back reference. This means that you have only for bits left to encode the length, allowing for a maximum length of 19. This is not desirable as it limits to compression ratio to 1:19. OTOH, if the current offset is, say, 123, a back reference of -512 is not possible. Some clever MS engineer decided to dynamically allocate more bits for the back-reference and less for the length. The exact split can be written as a table, or as

```
for(i=clear_pos-1,lmask=0xFFF,dshift=12;i>=0x10;i>>=1){
        lmask >>= 1; /* bit mask for length */
        dshift--;    /* shift width for delta */
}
```

Now that we can encode a (offset,length) pair as two bytes, we still have to know whether a token is a back-reference, or plain-text. This is one bit per token. Eight tokens are grouped together and preceded with the tags byte. So the group

```
>\n(18,10)stdio
```

would be encoded as

```
00000100 > \n 0A 90 s t d i o
```

(the 1 bit indicates the back reference). As an extreme case, a block of all space (' ') is compressed as

```
00000010 ' ' FC 0F
```

or ' ' (-1,4095). This works because you always read data you just stored. As a compression unit consists of 16 clusters, it usually contains more than one of these blocks. If you want to access the second block, it would be a waste of time to decompress the first one. Instead, each block is preceded by a 2-byte length. The lower twelve bits are the length, the higher 4 bits are of unknown purpose.

```
FIXME: Compression unit's size 2^4 in attribute header.
The compression method is based on independently compressing blocks of X
clusters, where X is determined from the compression_unit value found in the
non-resident attribute record header (more precisely: X = 2^compression_unit
clusters). On Windows NT/2k, X always is 16 clusters (compression_unit = 4).

  1) The data in the block is all zero (a sparse block):
    This is stored as a sparse block in the run list, i.e. the run list
    entry has length = X and lcn = -1. The mapping pairs array actually
    uses a delta_lcn value length of 0, i.e. delta_lcn is not present at
    all, which is then interpreted by the driver as lcn = -1.
    NOTE: Even uncompressed files can be sparse on NTFS 3.0 volumes, then
    the same principles apply as above, except that the length is not
    restricted to being any particular value.

  2) The data in the block is not compressed:
    This happens when compression doesn't reduce the size of the block
    in clusters. I.e. if compression has a small effect so that the
    compressed data still occupies X clusters, then the uncompressed data
    is stored in the block.
    This case is recognised by the fact that the run list entry has
    length = X and lcn >= 0. The mapping pairs array stores this as
    normal with a run length of X and some specific delta_lcn, i.e.
    delta_lcn has to be present.

  3) The data in the block is compressed:
    The common case. This case is recognised by the fact that the run
    list entry has length L < X and lcn >= 0. The mapping pairs array
    stores this as normal with a run length of X and some specific
    delta_lcn, i.e. delta_lcn has to be present. This run list entry is
    immediately followed by a sparse entry with length = X - L and
    lcn = -1. The latter entry is to make up the vcn counting to the
    full compression block size X.

In fact, life is more complicated because adjacent entries of the same type
can be coalesced. This means that one has to keep track of the number of
clusters handled and work on a basis of X clusters at a time being one
block. An example: if length L > X this means that this particular run list
entry contains a block of length X and part of one or more blocks of length
```

```
L - X. Another example: if length L < X, this does not necessarily mean that
the block is compressed as it might be that the lcn changes inside the block
and hence the following run list entry describes the continuation of the
potentially compressed block. The block would be compressed if the
following run list entry describes at least X - L sparse clusters, thus
making up the compression block length as described in point 3 above. (Of
course, there can be several run list entries with small lengths so that the
sparse entry does not follow the first data containing entry with
length < X.)

NOTE: At the end of the compressed attribute value, there most likely is not
just the right amount of data to make up a compression block, thus this data
is not even attempted to be compressed. It is just stored as is.
```

If you look at the algorithm, you will notice that it will not always reduce the data size. If there are no back references, each byte plain-text will remain as-is. However, every 8 bytes, a tag bit is inserted, which then will be zero. So, in the worst case, a block might grow to 4610 bytes (counting the length of the block). If the block grows in size, it will be stored uncompressed. A length of exactly 4095 is used to indicate this case. It might be still possible that the following block will compress well, reducing the total size of the chunk. If it doesn't, the entire chunk is stored uncompressed, which is indicated in the run list.

> each block is preceded by a 2-byte length. The lower twelve bits are the >length, the higher 4 bits are of unknown purpose.#

Bit 0x8000 is the flag specifying that the block is compressed. The compression code OR's in the value 0xB000 (if its compressed), but the decompression code only looks at bit 0x8000.

Also, the length is actually stored as (n-3) in the low 12 bits. Actually, (n-1) if you don't count the two bytes used to store the length itself. So for an uncompressed block the length is stored as 0xFFF, meaning the length is 4096 + 2 more bytes holding the length itself.

A 0x1000 length block compressed to length 0x500 would require 0x502 bytes, and have an advertised length of 0x4FF.

What I don't know is whether a 16 cluster file that doesn't compress at all requires 17 clusters to store, in order to accommodate the extra 2 bytes per block.

I believe it will take only 16 clusters. The fact that it is not compressed will be expressed in the run list. For example, the compressed file will look like

```
(1000 A) (0 6)       //(rel.VCN length)
```

whereas the uncompressable file will look like

```
(1000 10)
```

or

```
(1000 A) (1040 6)
```

IOW, if you don't have any runs with VCN==0 in the 16 clusters, the chunk is entirely uncompressed and plain. Given the compression algorithm, it is fairly easy to create such a file:

```
s=""
for i in range(0,16):   #adjust to clusters >512 if necessary

s=s+chr(i)+chr(j)
open("uncompressable","w").write(s)
```

8. Concept - Data Runs

8.1. Overview

Non-resident attributes are stored in intervals of clusters called runs. Each run is represented by its starting cluster and its length. The starting cluster of a run is coded as an offset to the starting cluster of the previous run.

Normal, compressed and sparse files are all defined by runs.

The examples start simple, then quickly get complicated.

This is a table written in the content part of a non-resident file attribute, which allows to have access to its stream.

NB Assume a 1KB cluster size, throughout. And little endian disk storage.

8.2. Layout

The runlist is a sequence of elements: each element stores an offset to the starting LCN of the previous element and the length in clusters of a run.

To save space, Offset and Length are variable size fields (probably up to 8 bytes), and an element is written in this crunched format:

Table 4.10. Layout of a data run

Offset in nibble to the beginning of the element	Size	Description
0	1	F=Size of the Offset field
1	1	L=Size of the Length field
2	2*L	Length of the run
2+2*L	2*F	Offset to the starting LCN of the previous element

Offset to the starting LCN of the previous element

This is a signed value. For the first element, consider the offset as relative to the LCN 0, the beginning of the volume.

The layout of the runlist must take account of the data compression: the set of VCNs containing the stream of a compressed file attribute is divided in compression units (also called chunks) of 16 clusters: VCNs 0 to 15 constitutes the 1st compression unit, VCNs 16 to 31 the 2nd one, and so on... For each compression unit,

- The alpha stage of compression is very simple and is independent of the compression engine used to compress the file attribute: if all the 16 clusters of a compression unit are full of zeroes, this compression unit is called a sparse unit and is not physically stored. Instead, an element with no Offset field (F=0, the Offset is assumed to be 0 too) and a Length of 16 clusters is put in the runlist.
- Else, the beta stage of compression is done by the compression engine used to compress the file attribute: if the compression of the unit is possible, N (< 16) clusters are physically stored, and an element with a Length of N is put in the runlist, followed by another element with no Offset field (F=0, the Offset is assumed to be 0 too) and a Length of 16 - N.
- Else, the unit is not compressed, 16 clusters are physically stored, and an element with a Length of 16 is put in the runlist.

In practice, this is a bit more complicated because some of the element can be gathered. But let's take an ...

8.2.1. ...Example

We have to decode the following runlist:

```
Runlist:
    21 14 00 01 11 10 18 11 05 15 01 27 11 20 05

Decode
    0x14   at   0x100   21 0x100, 0x14
    0x10   at  + 0x18   11  0x18, 0x10
    0x05   at  + 0x15   11  0x15, 0x05
    0x27   at  + none   01  0x27, none
    0x20   at  + 0x05   11  0x05, 0x20

Absolute LCNs
    0x14   at   0x100
    0x10   at   0x118
    0x05   at   0x12D
    0x27   at   none
    0x20   at   0x132

Regroup
    0x10   at   0x100

    0x04   at   0x110
    0x0C   at   0x118

    0x04   at   0x118
    0x05   at   0x12D
    0x07   at   none

    0x10   at   none

    0x10   at   none

    0x10   at   0x132

    0x10   at   0x142
```

```
Compression unit beginning at VCN 0x0
 0x10 clusters at LCN 0x100
 Unit not compressed

Compression unit beginning at VCN 0x10
 0x4 clusters at LCN 0x110
 0xC clusters at LCN 0x118
 Unit not compressed

Compression unit beginning at VCN 0x20
 0x4 clusters at LCN 0x124
 0x5 clusters at LCN 0x12D
 0x7 unused clusters: compressed unit

Compression unit beginning at VCN 0x30
 0x10 zeroed clusters: sparse unit

Compression unit beginning at VCN 0x40
 0x10 zeroed clusters: sparse unit

Compression unit beginning at VCN 0x50
 0x10 clusters at LCN 0x132
 Unit not compressed

Compression unit beginning at VCN 0x60
 0x10 clusters at LCN 0x142
 Unit not compressed

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

file.txt 31KB bytes (disk has a 1KB cluster size)

it's stored at clusters 10-26, 45-49, 100-108

17 clusters at LCN 10
5  clusters at LCN 45
9  clusters at LCN 100

next make the offsets relative

17 clusters at LCN 10
5  clusters at LCN 45
9  clusters at LCN 100

is encoded as

11

working in unit of 16 clusters
relative offsets (including -ve)
compressed sparse
variable length structures
stored as:
save space implies wherever MFT places
data it's best not to spread it too far.

-ve implies an offset of +129 would have to use two bytes
therefore -10 = 0xF6
0x80 = -128
0XFF7F = -129

21 14 00 01 11 10 18 11 05 15 01 27 11 20 05
```

8.3. data runs

Length and starting cluster are variable size fields. The first byte of a run indicates the size of both. The size of the offset is stored in the high nibble, and the size of the length in the low nibble.

For compressed or sparse runs, the offset is 0, and the size of the offset is also 0. Each compression unit starts at a multiple of 16 clusters. If compression is possible, at the VCN of a unit will be one or more data runs followed by an empty run. If there are data runs for more than 16 clusters, the unit was not compressible. If there is no data run at all (only a large empty run), the unit Consists of All zeroes.

```
Example: 21 20 ED 05 22 48 07 48 22 21 28 C8 DB
First run: 20 clusters starting from 5ED (5ED to 60D)
2nd run: 748 clusters starting from 5ED+2248 (2835 to 2F7D)
3rd run: 28 clusters starting from 2835+DBC8 (3FD to 425)
```

Note that the offset is interpreted as signed value.

Take a file of size 0x80 clusters (anywhere on disk). This is represented by VCN (virtual cluster numbers) 0x00 to 0x7F. These VCNs are mapper to LCN (logical cluster numbers) in runs (or extents), eg 21 80 30 60 00.

These runs are variable length, terminated with a zero. The low nibble of the first byte determines the length of the next number (1 byte) namely 80. The high nibble determines the length of the following number (2 bytes) namely 6030.

Outcome: 80 clusters, starting at cluster 6030.

The "sizes" are stored in one byte. The length is unsigned. The offset is signed and relative to the previous offset.

11 30 60 - 21 10 00 01 - 11 20 E0 - 00

```
Run 1 length 30 offset 60 (first run relative to 0)
Run 2 length 10 offset 100 + 60
Run 3 length 20 offset 160 - 20 (EO == -20)
             ==
             80
```

Files are represented by a set of VCNs. Sparse files, simply, have VCNs missing, eg

```
21 09 F5 47  9 clusters from 47F5
01 07        7 clusters from nowhere (0)
11 07 09     7 clusters from 47F5 + 9
  ====
  0x17

123456789ABCDEFG1234... VCN
RRRRRRRRRZZZZZZZRRRR... Real/Zero
```

Compresses files are first broken into blocks of 16 (0x10) clusters. Imagine:

```
VCN0123...
```

```
XXXXXXXXXXOOOOO  X=DATA O=SPACE
```

The data is compressed, here, into just ten clusters (If we can't save 1 cluster in 16, we don't bother) The above is coded as:

```
21 0A 10 F6   10 clusters of compressed data at F610
01 06         6 clusters of nothing to round up this block to 16
```

The 6 extra clusters aren't actually taking up any disk space. The VCNs are bunched into 16s. {{ If a block cannot be compressed, it would be represented by:

```
21 10 10 F6   16 clusters of compressed data at F610
```

```
FIXME:
In fact, life is more complicated because adjacent entries of the same type
can be coalesced. This means that one has to keep track of the number of
clusters handled and work on a basis of X clusters at a time being one
block. An example: if length L > X this means that this particular run list
entry contains a block of length X and part of one or more blocks of length
L - X. Another example: if length L > X, this does not necessarily mean that
the block is compressed as it might be that the lcn changes inside the block
and hence the following run list entry describes the continuation of the
potentially compressed block. The block would be compressed if the
following run list entry describes at least X - L sparse clusters, thus
making up the compression block length as described in point 3 above. (Of
course, there can be several run list entries with small lengths so that the
sparse entry does not follow the first data containing entry with
length < X.)

NOTE: At the end of the compressed attribute value, there most likely is not
just the right amount of data to make up a compression block, thus this data
is not even attempted to be compressed. It is just stored as is.
```

Compressed and sparse runs can be intermixed. All this to save space.

8.4. Examples

8.4.1. Example 1 - Normal, Unfragmented File

Data runs: 21 18 34 56 00

Regrouped: 21 18 34 56 - 00

Table 4.11. Parsed data runs: Example 1 - Normal, Unfragmented File

Num	Group	Header		Data	
		Length size	Offset size	Length	Offset
1	21 18 34 56	1 byte	2 bytes	0x18 (1 byte)	0x5634 (2 bytes)

Num	Group	Header		Data	
		Length size	Offset size	Length	Offset
2	00	End			

Summary:

- 0x18 Clusters @ LCN 0x5634

Therefore, Data1 is a unfragmented file, of size 0x18 clusters, starting at LCN 0x5634.

8.4.2. Example 2 - Normal, Fragmented File

Data runs: 31 38 73 25 34 32 14 01 E5 11 02 31 42 AA 00 03 00

Regrouped: 31 38 73 25 34 - 32 14 01 E5 11 02 - 31 42 AA 00 03 - 00

Table 4.12. Parsed data runs: Example 2 - Normal, Fragmented File

Num	Group	Header		Data	
		Length size	Offset size	Length	Offset
1	31 38 73 25 34	1 byte	3 bytes	0x38	0x342573 (3 bytes)
2	32 14 01 E5 11 02	2 bytes	3 bytes	0x114	0x363758 (0x211E5 relative to 0x342573)
3	31 42 AA 00 03	1 byte	3 bytes	0x42	0x393802 (0x300AA relative to 0x363758)
4	00	End			

Summary:

- 0x38 Clusters @ LCN 0x342573
- 0x114 Clusters @ LCN 0x363758
- 0x42 Clusters @ LCN 0x393802

Therefore, Data2 is a fragmented file, of size 0x18E clusters, with data blocks at LCNs 0x342573, 0x363758 and 0x393802.

8.4.3. Example 3 - Normal, Scrambled File

Data runs: 11 30 60 21 10 00 01 11 20 E0 00

Regrouped: 11 30 60 - 21 10 00 01 - 11 20 E0 - 00

Table 4.13. Parsed data runs: Example 3 - Normal, Scrambled File

Num	Group	Header		Data	
		Length size	Offset size	Length	Offset
1	11 30 60	1 byte	1 byte	0x30 (1 byte)	0x60 (1 byte)
2	21 10 00 01	1 byte	2 bytes	0x10	0x160 (0x100 relative to 0x60)
3	11 20 E0	1 byte	1 byte	0x20	0x140 (-0x20 relative to 0x160)
4	00	End			

Summary:

- 0x30 Clusters @ LCN 0x60
- 0x10 Clusters @ LCN 0x160
- 0x20 Clusters @ LCN 0x140

Therefore, Data3 is a badly fragmented file of size 0x60 clusters, with data blocks at LCNs 0x60, 0x160 and 0x140. Furthermore, the third block of data is physically located between the first and second blocks. (The third run has a negative offset, placing it before the previous run).

8.4.4. Example 4 - Sparse, Unfragmented File

Data runs: 11 30 20 01 60 11 10 30 00

Regrouped: 11 30 20 - 01 60 - 11 10 30 - 00

Table 4.14. Parsed data runs: Example 4 - Sparse, Unfragmented File

Num	Group	Header		Data	
		Length size	Offset size	Length	Offset
1	11 30 20	1 byte	1 byte	0x30 (1 byte)	0x20 (1 byte)
2	01 60	1 byte	0 bytes	0x60	N/A
3	11 10 30	1 byte	1 byte	0x10	0x50 (0x30 relative to 0x20)
4	00	End			

Summary:

- 0x30 Clusters @ LCN 0x20
- 0x60 Clusters (sparse)
- 0x10 Clusters @ LCN 0x50

Therefore, Data4 is a sparse, unfragmented file, of size 0xA0 clusters, with data blocks at LCNs 0x20 and 0x50.

This file has a sparse part in the middle of size 0x60 clusters. It takes up no space on disk, but it it represented by 0x60 VCNs.

8.4.5. Example 5 - Compressed, Unfragmented File

Data runs: 11 08 40 01 08 11 10 08 11 0C 10 01 04 00

Regrouped: 11 08 40 - 01 08 - 11 10 08 - 11 0C 10 - 01 04 - 00

Table 4.15. Parsed data runs: Example 5 - Compressed, Unfragmented File

Num	Group	Header		Data	
		Length size	Offset size	Length	Offset
1	11 08 40	1 byte	1 byte	0x08 (1 byte)	0x40 (1 byte)
2	01 08	1 byte	0 bytes	0x08	N/A
3	11 10 08	1 byte	1 byte	0x10	0x48 (0x08 relative to 0x40)
4	11 0C 10	1 byte	1 byte	0x0C	0x58 (0x10 relative to 0x48)
5	01 04	1 byte	0 bytes	0x04	N/A
6	00	End			

Summary:

- 0x08 Clusters @ LCN 0x40
- 0x08 Clusters (sparse)
- 0x10 Clusters @ LCN 0x48
- 0x0C Clusters @ LCN 0x58
- 0x04 Clusters (sparse)

Therefore, Data5 is a compressed, unfragmented, file of length 0x30, with data blocks at LCNs 0x40, 0x48 and 0x58.

The data, as stored on disk, is contiguous. The sparse runs pad out the compression units to blocks of 16 clusters (0x10).

8.4.6. Example 6 - Compressed, Sparse, Fragmented File

brain damaged file

9. Concept - Directory

9.1. Overview

Under NTFS every object on the volume is a file, even directories. A directory is an index of filenames.

9.2. Attributes

Table 4.16. A directory record attributes

Type	Description	Name
Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	dirname
0x50	$SECURITY_DESCRIPTOR	
0x90	$INDEX_ROOT	$I30
0xA0	$INDEX_ALLOCATION	$I30
0xB0	$BITMAP	$I30

9.2.1.

9.2.1.1. Index Entry

An index is a list of index entries. Each entry contains the name of the file, the standard information and a pointer to the security information. The correct starting place is the Index Entry.

9.2.1.2. Index Root

This attribute, which is always resident, holds several index entries. It forms the root of the index tree.

9.2.1.3. Index Allocation

A set of runs telling the system where the other indexes are. (preposition!)

9.2.1.4. Index Bitmap

Which clusters (indexes) are in use.

```
A directory can even have a named data stream
```

9.3. Definition

From an human's point of view, a directory is a particular kind of file that can contain other files. It is a file folder, used in a nested way to create a logical file hierarchy on a volume.

9.4. Properties

From NTFS' point of view, a directory is an index of file names, or more accurately a sequence of index entries containing a filename attribute. An index entry is created for each file name attribute of each file contained in the folder. This kind of index entries can be compared together using the alphabetical order on their upper-cased (thanks to $UpCase) file name attribute.

A directory has no data attribute. But, as an index, it has instead three other file attributes: index root, index allocation, and bitmap. The index is stored in the nodes of a B+ tree in the following manner:

- Each node of the tree contains one or several index entries. Within a node, index entries are sorted in increasing order

- Each index entry may point to another (sub-)node containing only lower index entries
- The root node is in the stream of the index root attribute, the other (sub-)nodes are index buffers.

9.5. Interest

When an application reads a directory, NTFS returns a list of file names which is already sorted.

The B+ tree structure (which is used in HPFS too), when built in a balanced way, is far more efficient than a linear structure to perform a file name lookup in a folder containing a large number of files.

Although the duplication of the stream of the indexed attribute in an index entry can cost some time, it is worthy because you can browse an index without actually opening all the indexed files (FAT and HPFS do that, too).

In a directory, the three file attributes: index root, index allocation, and bitmap are named "$I30", and a directory is just an Index of file attributes whose type is 30. But NTFS has been thought as a database filesystem, and it can actually create indexes based on any file attribute that is always resident. E.g., you could create a new file attribute labeled "author name", and sort your files according to that criteria.

10. Concept - File

10.1. Overview

It is composed of attributes including its name and its data.

10.2. Attributes

Table 4.17. A file record attributes

Type	Description	Name
0x10	$STANDARD_INFORMATION	
0x30	$FILE_NAME	filename
0x50	$SECURITY_DESCRIPTOR	
0x80	$DATA	[Unnamed]

10.2.1. Standard Information

This contains the DOS-style file permission, such as read-only and archive. It also contains four different types of modification time.

- File creation time
- Last modification time
- Last modification time for FILE record
- Last access time

10.2.2. File Name

The file's name is stored as an attribute, too. A file can have several filenames. This is Windows' equivalent to hard linking files together.

10.2.3. Security Descriptor

This stores all of Windows' permissions. ACLs, ACEs, auditing.

```
May not exist on Win2K (std info, $secure)
```

10.2.4. Data

This, finally, is the actual data of the file. It, too, is stored in an attribute

```
unnamed data stream compulsory (chkdsk will put it back if missing)
named data streams optional (any limit to the number?)
```

10.3. Named Data Streams

```
access with "jim.txt:stream"
```

Table 4.18. Fictional named data streams

Type	Description	Name
0x80	$DATA	icon
0x80	$DATA	author

10.4. Summary Information

Windows 2000 introduced the idea of summary information on files. This information is stored as a set of four named data streams.

Description

- Title
- Subject
- Category
- Keywords (multi-line)

- Comments (multi-line)

Origin

- Source
- Author
- Revision Number

Table 4.19. Summary Information named data streams

Type	Description	Name
0x80	$DATA	{4c8cc155-6c1e-11d1-8e41-00c04fb9386d}
0x80	$DATA	^EDocumentSummaryInformation
0x80	$DATA	^ESebiesnrMkudrfcoIaamtykdDa
0x80	$DATA	^ESummaryInformation

N.B. Three of the names begin with CTRL-E (0x05). This is probably to discourage people from reading the streams directly.

The first stream *{4c..* is always empty. This is probably just a marker to

Table 4.20. contents of Summary Information named data streams

Data Stream	Summary Field	Data Type	Code
^EDocumentSummaryInformation	Unknown1	Numeric?	0x00
	Unknown2	Numeric	0x01
	Category	ASCII	0x02
^ESebiesnrMkudrfcoIaamtykdDa	Unknown3	Numeric?	0x00
	Unknown4	Numeric?	0x01
	Source	Unicode	0x04
^ESummaryInformation	Unknown5	Numeric?	0x00
	Unknown6	Numeric?	0x01
	Title	ASCII	0x02
	Subject	ASCII	0x03
	Author	ASCII	0x04
	Keywords	ASCII	0x05
	Comments	ASCII	0x06
	Revision Number	ASCII	0x09

11. Concept - File Record

11.1. Overview

The MFT is a set of FILE records. Each file of the volume is completely described by one or more of these FILE Records. File Records are equivalent to inodes in Unix terminology. The first FILE Record that describes a given file is called the Base FILE record and the others are called Extension FILE Records.

A FILE Record is built up from a header, several variable length attributes and an end marker (simply 0xFFFFFFFF).

`link table to notes`

See also: Attributes, Standard Attribute Header, $MFT, $Boot, File, Fixup, Attribute Id, Directory,

11.2. Layout

FILE Record

Header

Attribute

Attribute

...

End Marker (0xFFFFFFFF)

Table 4.21. Layout of a file record

Offset	Size	OS	Description
0x00	4		Magic number 'FILE'
0x04	2		Offset to the update sequence
0x06	2		Size in words of Update Sequence Number & Array (S)
0x08	8		$LogFile Sequence Number (LSN)
0x10	2		Sequence number
0x12	2		Hard link count
0x14	2		Offset to the first Attribute
0x16	2		Flags
0x18	4		Real size of the FILE record
0x1C	4		Allocated size of the FILE record
0x20	8		File reference to the base FILE record
0x28	2		Next Attribute Id
0x2A	2	XP	Align to 4 byte boundary
0x2C	4	XP	Number of this MFT Record
	2		Update Sequence Number (a)
	2S-2		Update Sequence Array (a)

(a) The offset to these two fields depends on your operating system.

$LogFile Sequence Number (LSN)

This is changed every time the record is modified.

Sequence Number

Number of times this mft record has been reused.

N.B. The increment (skipping zero) is done when the file is deleted.

N.B. If this is set to zero it is left as zero.

Hard Link Count

Number of hard links, i.e. the number of directory entries referencing this record.

N.B. Only used in mft base records.

Flags

Table 4.22. File record flags

Flag	Description
0x01	Record is in use
0x02	Record is a directory
0x04	Don't know
0x08	Don't know

Real / Allocated Size

The Allocated Size is how much space the Record takes up on disk. This should be a multiple of the cluster size and should probably be equal to the size of an MFT File Record. The Real Size is a count of how many bytes of the Record are actually used.

N.B. The Real Size will be padded to an 8 byte boundary.

Base MFT Record

This is zero for Base MFT Records. When it is not zero it is a MFT Reference pointing to the Base MFT Record to which this Record belongs. The Base Record contains the information about the Extension Record. This information is stored in an ATTRIBUTE_LIST attribute.

Next Attribute Id

The Attribute Id that will be assigned to the next Attribute added to this MFT Record.

N.B. Incremented each time it is used.

N.B. Every time the MFT Record is reused this Id is set to zero.

N.B. The first instance number is always 0.

The master file table record consists of a header and the attribute list. It has a size of 400 (=1K), or the cluster size (whichever is larger). The header has the following fields:

11.3. Notes

The attribute list is of variable length and terminated with FFFFFFFF. For 1K MFT records, the attribute list starts at offset 0x30.

```
The sequence number is a circular counter (skipping 0) describing how many
times the referenced mft record has been (re)used. This has to match the
sequence number of the mft record being referenced, otherwise the reference
is considered stale and removed (FIXME: only ntfsck or the driver itself?).

If the sequence number is zero it is assumed that no sequence number
consistency checking should be performed.

FIXME: The mft zone is defined as the first 12% of the volume. This space is
reserved so that the mft can grow contiguously and hence doesn't become
fragmented. Volume free space includes the empty part of the mft zone and
when the volume's free 88% are used up, the mft zone is shrunk by a factor
of 2, thus making more space available for more files/data. This process is
repeated everytime there is no more free space except for the mft zone until
there really is no more free space.

The mft record header present at the beginning of every record in the mft.
This is followed by a sequence of variable length attribute records which
is terminated by an attribute of type $END which is a truncated attribute
in that it only consists of the attribute type code $END and none of the
other members of the attribute structure are present.

When (re)using the mft record, we place the update sequence array at this
offset, i.e. before we start with the attributes. This also makes sense,
otherwise we could run into problems with the update sequence array
containing in itself the last two bytes of a sector which would mean that
multi sector transfer protection wouldn't work. As you can't protect data
by overwriting it since you then can't get it back...
When reading we obviously use the data from the ntfs record header.
```

The sequence of attributes part

This is a sequence of file attributes that has a variable length. In each FILE record, the sequence is ordered by increasing order of the attribute type. The sequence is terminated with FF FF FF FF.

```
Size defined in $Boot.
A FILE record is 1 KB large or the cluster size if larger (as far as Helen is
concerned, its maximum size is 4 KB, but Windows NT 4 limit is 64 KB). It fall
2 parts:
```

Extension FILE records are used when all information about a file doesn't fit into the base FILE record (e.g. if the sequence of file attributes grows because the file has a lot of file attributes or because the data attribute of the file has a long runlist because its stream is very fragmented). Only the base FILE record is used for referencing the file it describes. Since the type of the Attribute List file attribute is small enough, we are sure that this file attribute will be in the base FILE record. And this file attribute provides the references to all the extension FILE records describing the file.

When a file is deleted, NTFS can't simply remove the associated FILE records from the MFT, otherwise FILE record numbers wouldn't be constant over time, and all file references would have to be updated! Instead, the in-use flag of a FILE record indicates when it is no longer in use. When a file is created, an unused FILE record can be re-used for it, but its sequence number is incremented by one. This mechanism allow NTFS to check that file references don't point to deleted files.

```
seq num = inode for 0x00 < i < 0x10 (inode 0 (MFT) has seq num of 1)

see also attribute id page and file reference page

flags 1 in use, 2 dir, 4 ???, 8??? (4+8 ARE used)
```

12. Concept - File Reference

12.1. Overview

A unique identifier for a FILE record in the MFT.

12.2. Layout

Table 4.23. Layout of a file reference

Offset	Size	Description
0x00	6	FILE record number
0x06	2	Sequence number

12.3. Notes

12.3.1. Sequence number

If the filesystem is consistent, this number must match the sequence number of the FILE record referenced by the FILE record number.

```
mft references (aka file references or file record segment references) are
used whenever a structure needs to refer to a record in the mft.

A reference consists of a 48-bit index into the mft and a 16-bit sequence
number used to detect stale references.

when is the seq num incremented
```

13. Concept - Filename Namespace

13.1. Overview

Old versions of the FAT filesystem had strict limits on filenames. Many characters were forbidden, and the length was restricted to 11 characters (a small namespace). Newer versions of FAT allowed more characters and longer filenames. NTFS has almost no restrictions.

Filenames are given a flag to show which namespace the name belongs to. In order to support old applications, NTFS allocates a short DOS-friendly name to any file with an DOS-incompatible name.

13.2. Possible Namespaces

0: POSIX

This is the largest namespace. It is case sensitive and allows all Unicode characters except for NULL (0) and Forward Slash '/'. The maximum name length is 255 characters. N.B. There are some characters, e.g. Colon ':', which are valid in NTFS, but Windows will not allow you to use.

1: Win32

Win32 is a subset of the POSIX namespace and is case insensitive. It uses all the Unicode characters, except: '"' '*' '/' ':' '<' '>' '?' '\' '|' N.B. Names cannot end with Dot '.', or Space ' '.

2: DOS

DOS is a subset of the Win32 namespace, allowing only 8 bit upper case characters, greater than Space ' ', and excluding: '"' '*' '+' ',' '/' ':' ';' '<' '=' '>' '?' '\'. N.B. Names must match the following pattern: 1 to 8 characters, then '.', then 1 to 3 characters.

3: Win32 &DOS

This namespace means that both the Win32 and the DOS filenames are identical and hence have been saved in this single filename record.

To convert a POSIX or Win32 filename to a DOS-friendly filename, follow these steps:

1. Remove all Unicode characters
2. Remove all '.' but the last one if *it is not the first character*
3. Uppercase all letters
4. Remove forbidden characters
5. Truncate everything before the potential '.' to 6 characters, and add the string "~1"
6. Truncate everything after the potential '.' to 3 characters
7. While the name already exists, increment the string "~1"
8. N.B. Step 7 means that although the generated DOS name is unique, it is impossible to deduce it from the Win32 name only.

14. Concept - Fixup

14.1. Overview

The smallest unit of disk space that NTFS uses is a Cluster. This can vary from one sector to 128 sectors, the usual number is 8 (4KB). Naturally this is dependent on the sector and Cluster. sizes declared in $Boot.

Because a single sector could fail, it's important for NTFS to be able to detect errors in a cluster. For this purpose the sectors have *Fixups*, which are kept in an *Update Sequence Array*.

Many important Metadata Records use fixups to protect data integrity

- FILE Records in the $MFT
- INDX Records in directories and other indexes
- RCRD Records in the $LogFile
- RSTR Records in the $LogFile

14.2. What Does It Do?

The header of each of these records contains a Update Sequence Number and a buffer. The last two bytes of each sector of the record are copied into the buffer and the Update Sequence Number is written in their place.

When the record is read, the Update Sequence Number is read from the header and compared against the last two bytes of each sector. If it succeeds, then it copies the bytes in the buffer back to their original places.

14.3. Example

Here's an example before the fixup is applied, with a cluster size of 2KB (4 Sectors).

Table 4.24. Fixup example: before

Offset	Data								Description
0x0000	...								Header
0x0028	CD	AB							Update Sequence Number
0x002A	00	00	00	00	00	00	00	00	Update Sequence Array
...	...								
0x01F8	11	12	13	14	15	16	17	18	End of Sector 1
...	...								
0x03F8	21	22	23	24	25	26	27	28	End of Sector 2
...	...								
0x05F8	31	32	33	34	35	36	37	38	End of Sector 3
...	...								

0x07F8	41	42	43	44	45	46	47	48	End of Sector 4

Here the Update Sequence Number is 0xABCD and the Update Sequence Array is still empty.

Table 4.25. Fixup example: after

Offset	Data								Description
0x0000	...								Header
0x0028	CD	AB							Update Sequence Number
0x002A	17	18	27	28	37	38	47	48	Update Sequence Array
...	...								
0x01F8	11	12	13	14	15	16	CD	AB	End of Sector 1
...	...								
0x03F8	21	22	23	24	25	26	CD	AB	End of Sector 2
...	...								
0x05F8	31	32	33	34	35	36	CD	AB	End of Sector 3
...	...								
0x07F8	41	42	43	44	45	46	CD	AB	End of Sector 4

The last two bytes of each sector have been copied into the Update Sequence Array, and the Update Sequence Number has been written over the last two bytes of each sector.

14.4. The Details

14.4.1. Writing

Before writing a fixup-protected record:

1. Add one to the Update Sequence Number (0x0000 must be skipped)
2. For each sector, copy the last two bytes into the Update Sequence Array
3. Write the new Update Sequence Number to the end of each sector
4. Write the record to disk

14.4.2. Reading

When reading a fixup-protected record:

1. Read the record from disk
2. Check the magic number is correct
3. Read the Update Sequence Number
4. Compare it against the last two bytes of every sector
5. Copy the contents of the Update Sequence Array to the correct places
6. If any of the checks fail when reading, it could mean there is: a bad sector, disk corruption or a fault in the driver.

15. Concept - Index Header

15.1. Overview

Every Index Record has a standard header and a set of blocks containing an Index Key and Index Data.

The size of an Index Record is defined in $Boot and always seems to be 4KB.

15.2. Layout

15.2.1. Standard Index Header

Table 4.26. Layout of a Standard Index Header

Offset	Size	Description
0x00	4	Magic number 'INDX'
0x04	2	Offset to the Update Sequence
0x06	2	Size in words of the Update Sequence Number &Array (S)
0x08	8	$LogFile sequence number
0x10	8	VCN of this INDX buffer in the Index Allocation
0x18	4	Offset to the Index Entries (a)
0x1C	4	Size of Index Entries (a)
0x20	4	Allocated size of the Index Entries (a)
0x24	1	1 if not leaf node (b)
0x25	3	Padding (always zero)
0x28	2	Update sequence
0x2A	2S-2	Update sequence array

```
(a) These values are relative to 0x18
(b) Has children
```

15.3. Notes

15.3.1. List of Common Indexes

Table 4.27. List of Common Indexes

Name	Index Of	Description
$I30	Filenames	Used by Directories
$SDH	Security Descriptors	$Secure
$SII	Security Ids	$Secure
$O	Object Ids	$ObjId
$O	Owner Ids	$Quota
$Q	Quotas	$Quota
$R	Reparse Points	$Reparse

15.3.2. Other Information

There is no information contained in the Index Record describing what the index is storing (this is kept in the Index Root).

16. Concept - Index Record

16.1. Overview

```
This is only applicable to a file index ($I30)

    indx help describe as "index = key + data"

    given an INDX record, it's difficult to work out what's
    being indexed (that info is in the index root)
```

16.2. Definition

This is a sub-node of the B+ tree that implements an index (e.g. a directory). It is stored in the stream of the index allocation attribute associated to the index it belongs to.

16.3. Layout

An INDX buffer is at least 2 KB large or the cluster size if larger (this seems to be a per-index parameter). It falls into 2 parts:

16.3.1. The header part

this ISN'T just the header...

Table 4.28. Layout of an Index record header

Offset	Size	Description
~	~	Standard Index Header
0x00	8	MFT Reference of the file
0x08	2	Size of this index entry
0x0A	2	Offset to the filename
0x0C	2	Index Flags
0x0E	2	Padding (align to 8 bytes)
0x10	8	MFT File Reference of the parent
0x18	8	File creation time
0x20	8	Last modification time
0x28	8	Last modification time for FILE record
0x30	8	Last access time
0x38	8	Allocated size of file
0x40	8	Real size of file
0x48	8	File Flags
0x50	1	Length of filename (F)
0x51	1	Filename namespace
0x52	2F	Filename
2F+0x52	P	Padding (align to 8 bytes)
P+2F+0x52	8	VCN of index buffer with sub-nodes

```
N.B. the filename is not null terminated
surely the flags can't be 8 bytes long
table for the flags
VCN of ib only exists when flags&1
last entry has a size of 0x10 (just large enough
for the flags (and a mft ref of zero))
```

16.3.2. The sequence of index entries part

This is a sequence of index entries similar to the one found in the index root attribute.

The index entry has the following structure:

```
Index entry flags (16-bit).

INDEX_ENTRY_NODE = cpu_to_le16(1), This entry contains a sub-node,
                  i.e. a reference to an index
                  block in form of a virtual
                  cluster number (see below).
INDEX_ENTRY_END  = cpu_to_le16(2), This signifies the last entry in
                  an index block. The index entry
                  does not represent a file but it
                  can point to a sub-node.

This is an index entry. A sequence of such entries follows each INDEX_HEADER
structure. Together they make up a complete index. The index follows either
an index root attribute or an index allocation attribute.
```

```
NOTE: Before NTFS 3.0 only filename attributes were indexed.
```

Most entries are not valid (and present) if the entry is the last one. This entry does not represent a file and is used only for subnodes. The pointer to the subnode buffer is only present if the entry has subnodes.

17. Concept - Links

17.1. Overview

17.2. Interest

NTFS doesn't manage POSIX symbolic links. Nevertheless, this file attribute let us think that NTFS will manage symbolic links (or Reparse point, in Microsoft terminology) in Windows NT 5.0, like all modern Unix filesystems (e.g. Ext2, the Linux filesystem) do.

17.3. Questions

What is the role and the layout of the stream of this file attribute?

```
NTFS represents POSIX-style hard links as files with multiple filename
NTFS represents hard links with multiple filenames.
This is different to one file with names in different namespaces.
Delete a name from a hard linked file and only the name will be removed.
```

18. Concept - Restart

18.1. Overview

Each copy of the restart area is 4KB in size, and has the following structure:

```
Offset(length)         Description
0(4)                   Magic number 'RSTR'
1E(12)                 Fixup
30(4)                  LSNa
58(4)                  LSNb
60(4)                  LSNc (==LSNa?)
6C(1)                  Volume clear flag
78(8)                  Unicode string 'NTFS'
```

The purpose of the various LSNs is unclear. It appears that the data around offset 3C deal with the clear/ dirty state of the volume, too.

19. Concept - SID

19.1. Overview

There are several SIDs reserved for NT.

```
link back to sec page
```

sec

```
S-1-5-21-646518322-1873620750-619646970-1110
S for security id
1 Revision level
5 Identifier Authority (48 bit) 5 = logon id
21 Sub-authority (21 = nt non unique)
646518322        SA
1873620750        SA domain id
619646970        SA
1110        user id
```

Table 4.29. Common well known SIDs

SID	Description
S-1-5-32-544	Local admin.
S-1-1-0	World (everybody)
S-1-5-21	NT non-unique ids

Identifier Authorities

Table 4.30. Identifier Authorities

Identifier Authority	Abbr.
Null SID	S-1-0
World SID	S-1-1
Local SID	S-1-2
Creator SID	S-1-3
Non-unique	S-1-4
NT SID	S-1-5

Relative Identifiers (RIDs)

```
These relative identifiers (RIDs) are used with the above identifier
authorities to make up universal well-known SIDs.

Note: The relative identifier (RID) refers to the portion of a SID, which
identifies a user or group in relation to the authority that issued the SID.
For example, the universal well-known SID Creator Owner ID (S-1-3-0) is
made up of the identifier authority SECURITY_CREATOR_SID_AUTHORITY (3) and
```

```
the relative identifier SECURITY_CREATOR_OWNER_RID (0).
```

Relative Identifiers

Table 4.31. Relative Identifiers

Relative Identifier	**Code**	**SID**
Null	0	S-1-0-0
World	0	S-1-1-0
Local	0	S-1-2-0
Creator Owner	0	S-1-3-0
Creator Group	1	S-1-3-1
Creator Owner Server	2	S-1-3-2
Creator Group Server	3	S-1-3-3
Dialup	1	S-1-5-1
Network	2	S-1-5-2
Batch	3	S-1-5-3
Interactive	4	S-1-5-4
Logon Ids	5	S-1-5-5-X-Y
Service	6	S-1-5-6
Anonymous Logon	7	S-1-5-7
Proxy	8	S-1-5-8
Enterprise Controllers	9	S-1-5-9
Server Logon	9	S-1-5-9
Principal Self	10	S-1-5-10
Authenticated User	11	S-1-5-11
Restricted Code	12	S-1-5-12
Terminal Server	13	S-1-5-13
Local System	18	S-1-5-18
NT Non-unique	21	S-1-5-21
Builtin Domain	32	S-1-5-32

Well-known domain relative sub-authority values (RIDs).

Domain Users

Table 4.32. Domain Users

Domain User	**Code**
Admin	500
Guest	501
Kerberos Target	502

Domain Groups

Table 4.33. Domain Groups

Domain Group	Code
Admins	512
Users	513
Guests	514
Computers	515
Controllers	516
Cert Admins	517
Schema Admins	518
Enterprise Admins	519
Policy Admins	520

Domain Aliases

Table 4.34. Domain Aliases

Domain Alias	Code
Admins	544
Users	545
Guests	546
Power Users	547
Account Ops	548
System Ops	549
Print Ops	550
Backup Ops	551
Replicator	552
RAS Servers	553
Pre W2K Comp Access	554

Universal well-known SIDs

Table 4.35. Universal well-known SIDs

SID	Abbr.
Null	S-1-0-0
World	S-1-1-0
Local	S-1-2-0
Creator Owner	S-1-3-0
Creator Group	S-1-3-1

SID	Abbr.
Creator Owner Server	S-1-3-2
Creator Group Server	S-1-3-3
Non-unique IDs	S-1-4

NT well-known SIDs

Table 4.36. NT well-known SIDs

SID	Abbr.
NT Authority	S-1-5
Dialup	S-1-5-1
Network	S-1-5-2
Batch	S-1-5-3
Interactive	S-1-5-4
Service	S-1-5-6
Anonymous Logon (Null Logon)	S-1-5-7
Proxy	S-1-5-8
Server Logon (Domain Controller)	S-1-5-9
Self	S-1-5-10
Authenticated User	S-1-5-11
Restricted Code	S-1-5-12
Terminal Server	S-1-5-13
Logon IDs	S-1-5-5-X-Y
NT Non-unique IDs	S-1-5-21-...
Built-in Domain	S-1-5-32

20. Concept - Sparse

20.1. Overview

Sparse files

```
fix the data runs page for NT4 (old style)
13 b8 ae 04 ff 00     old
03 b8 ae 04 00        new
bad clus on NT4 sparse data runs use -1!
```

Chapter 5. Epilogue

1. ToDo

Unless otherwise specified, each item is a rewrite / overhaul.

Urgent

- Security
- Log
- Index Root
- Attribute Id
- FILE Record

Medium

- Cross-ref $Secure
- Cross-ref $Quota
- Attribute List
- Logged Utility Stream
- Compression
- Data Runs
- Directory
- File
- Index Header
- Index Record
- Sparse files

Low

- Res/Non-res in Overview
- Table (P8) sizes
- Data
- Reparse Point
- File Reference

- USN confusion
- Remove Links?
- Restart
- SID
- Glossary

2. Unanswered Questions

This, final, section of the documentation is the place for all the unanswered questions. Some relate to Windows' use of NTFS and some are very technical.

Your help is needed to fill in the blanks.

- Why do some Metadata files on NTFS 3.0+ still have Security Descriptors?

  On NTFS 3.0+, $Volume, $AttrDef, dot and $Boot have Security Descriptors. Is this to save time at boot up? Perhaps to reduce the number of files it has to parse? Or is this the same as the previous question?

- $STANDARD_INFORMATION: Max Versions, Version Number and Class Id?

  Are any of the three fields used?

- Is $UsnJrnl's $J Data Stream a fixed size?

  Is it a fixed size? Does it wrap around like $LogFile?

- What does $UsnJrnl's $Max Data Stream do?

  There's a time stamp, two fields that might be flags and a field that might be a length.

- Attribute Header

  When is "Initialised" not the same size as "Real"?

- $MountMgrDatabase

  What is the format of this stream?

- MFT (FILE) Records

  Will we only see MFT Extension records with inodes <23? Is the sequence number always equal to the inode number for the Metadata?

- MFT Mirr

  How large is this if the cluster size is greater than 4kB?

- Index Records

  Are they always 4kB?

- Collation

Is a collation type ULONGS equivalent to GUID?

- Security Descriptors

  How are ACEs inherited?

ToDo: copy questions to relevant pages and x-link.

3. History

Version 0.6

- Conversion to DocBook:
  - Reordering files as chapters and sections.
  - Removed/Reauthored paragraphs related specifically to the html format.
  - Titled sections and tables.
  - Line breaks became new paragraphs or removed.
- Presentation changes:
  - Moving "Notes" and "Other information" to the bottom of each section.
  - Removed illustrations from the Tree concept (will be returned in the future).
  - Removed empty sections.
  - Data Runs examples are described now in tables.

Version 0.5

- New:
  - Added a link to the NTFS FAQ.
  - Added the tree concept.
- Tidied, Fixed or Rewritten:
  - Fixed the 3rd data run example.
  - Add directory & Index View flags to the FileName attribute.
  - More info about Reparse Points.
  - More info about $usnjrnl.
  - Updated $boot.
  - Updated $mftmirr.

  - Updated Security Descriptor attribute.
  - Fixed a minor error in the attribute header concept.
  - Fixed a minor error in the file record concept.
  - Fixed a type in the clusters concept.
  - Updated the thanks page.
- HTML Improvements:
  - Added an icon to the html meta
  - Moved the help menu to the front page
  - Added the SF logo and a copyright to the footer
  - Change the contact email to a picture.
  - Removed a link from the glossary to the obsolete property_set page.
  - Removed the contact info from the footer
  - Fixed a link to sourceforge (removed the www. prefix).
  - CSS updates.
  - Whitespace cleanup

Version 0.4

- New:
  - List of all Data Streams and Indexes
  - Pages: About, Collation and SID
  - (Some) info about XP
  - Info about $Q, $O and $R
  - Info about the MFT Zone
  - More info about Indexes
  - Load of new Glossay entries
- Tidied, Fixed or Rewritten:
  - Standard Information, Filename, Fixup
  - Standardise naming of the four time fields
  - Standardise naming of the three file size fields
  - Minor improvements to Bitmap and Quota

- HTML Improvements:
  - Standardised tables
  - Footnote links on every page: Validate HTML, CSS and Online
  - Next / Prev links cycle through the index
  - Better CSS compliance
  - Added keywords to aid search engines
  - Tweaked fonts

Version 0.3

- Worked in Anton's header files
- New page for Collation
- New page for Index Header
- New page for $UsnJrnl
- Reworked Index Record page
- New info for $ObjId
- New info for $Quota
- New info for $Secure
- New info for $Reparse
- $MountMgrDatabase added to dot
- Reworked $MFT page
- Lots of tidying up

Version 0.2

- Put everything under CVS control on SourceForge
- Added $Id CVS tag to the end of every file
- Added full path to the beginning of every file
- Fixed up CSS so old version of Netscape should look OK
- Updated $AttrDef
- Updated $EA
- Updated $EA_INFORMATION

- Updated $FILE_NAME
- Updated $STANDARD_INFORMATION
- Updated $VOLUME_INFORMATION
- Wrote entries for all the glossary items
- Access keys for Previous and Next , and .
- Fixed lots of typos

Version 0.1

- First public release, based on the very old "original docs"

Appendix Appendix I. License

1. GNU Free Documentation License

1.1. Version 1.1, March 2000

1.2. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of *copyleft*, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1.3. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The *Document*, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as *you*.

A *Modified Version* of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A *Secondary Section* is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The *Invariant Sections* are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The *Cover Texts* are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A *Transparent* copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not *Transparent* is called *Opaque*.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The *Title Page* means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, *Title Page* means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

1.4. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

1.5. 3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

1.6. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled *History*, and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled *History* in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the *History* section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled *Acknowledgements* or *Dedications*, preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled *Endorsements*. Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as *Endorsements* or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified

Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled *Endorsements*, provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

1.7. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled *History* in the various original documents, forming one section entitled *History*; likewise combine any sections entitled *Acknowledgements*, and any sections entitled *Dedications*. You must delete all sections entitled *Endorsements.*

1.8. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

1.9. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an *aggregate*, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

1.10. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

1.11. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

1.12. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License *or any later version* applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Glossary

This is a glossary of all terms.

Some entries refer to other entries, e.g. *See also*.

Some entries have an entire page of their own, e.g. *More*...

Glossary

. (See Dot, Root Directory)
: See Dot, Root Directory.

Access Control Entry (ACE)
: An Access Control Entry is the smallest unit of security. It contains a SID (either a user or a group) and permissions information.

  The permission will be one of *Access Allowed*, *Access Denied* or *System Audit*. This object has flags to determine how the permissions should be inherited.
  See Also Security Identifier (SID), Access Control List (ACL), Audit, Auditing.

Access Control List (ACL)
: This security structure contains a list of ACEs.
  See Also $SECURITY_DESCRIPTOR, Security Identifier (SID), Access Control List (ACL), Audit, Auditing.

ACE (See Access Control Entry)
: See Access Control Entry (ACE).

ACL (See Access Control List)
: See Access Control List (ACL).

$AttrDef
: This metadata file contains the definitions of all the attributes that are allowed on an NTFS volume.

  (More...)

Attribute
: on disk a file is stored as a set of attributes resident / non res

$ATTRIBUTE_LIST
: This attribute is used when a file's attributes won't fit in a single MFT File Record. It has a list of all the attributes and where they can be found.

  The $ATTRIBUTE_LIST is always stored in the Base FILE Record.

  (More...)
  See Also FILE Record, $MFT, Base FILE Record.

Audit, Auditing
: As part of the security permissions of a file, any actions performed on the file can be recorded.

  For example a file could be required to log all the people who tried to read it, but didn't have the permissions to do so.

B+ Tree
: A B+ tree is a variant of the binary tree.

  Instead of one data element per node, there are many.

  In NTFS the actual number depends on the lengths of the names and the cluster size).

The B+ tree retains the efficiency of a binary tree and also performs well with large numbers of data elements (because the tree tends to grow wide rather than deep).
See Also Binary Tree, Balanced Tree.

BAAD

During chkdsk, if NTFS finds a multi-sector item (MFT, INDEX BLOCK, etc) where the multi-sector header doesn't match the values at the end of the sector, it marks the item with the magic number 'BAAD', and fill it with zeroes (except for a short at the end of each sector...)

```
FIXME
"BAAD" == corrupt record
"CHKD" == chkdsk ???
"FILE" == mft entry
"HOLE" == ??? (NTFS 3.0+?)
"INDX" == index buffer
RSTR & ???
```

See Also chkdsk, fsck.

$Bad

This is the named Data Stream representing bad clusters on a volume.
See Also $BadClus.

$BadClus

This metadata file lists all the unreadable clusters on the volume.

(More...)

Balanced Tree

Often binary trees can become very uneven. By reorganising the data, the tree can be balanced such that no a node has similar numbers of children to it's left and right.
See Also B+ Tree, Binary Tree.

Base FILE Record

If the attributes don't fit into a single MFT record then the Base FILE Record holds enough information to locate the other records.
See Also $ATTRIBUTE_LIST, FILE Record, $MFT.

Binary

Maths carried out in base two. In this documentation, certain flags fields are represented in binary, for the sake of clarity.

e.g. 00001000_2, 010000000_2.
See Also Decimal, Hex, Hexadecimal, Units.

Binary Tree

This is an efficient way of storing sorted data in order.

Each node in the tree represents a data element.

The left child node is a collection of all the elements that come before it.

The right child node is a collection of all the elements that come after it.
See Also B+ Tree, Balanced Tree.

Bit	One binary digit, one or zero. See Also Units.
$Bitmap	This metadata file keeps track of which clusters are in use on the volume. (More...)
$BITMAP	This attribute keeps track of which records are in use in an index. (More...)
Block	In Linux terminology, this is a cluster. Block device In Linux terminology, this is a storage unit. Cluster is the minimum allocation unit. Clusters are a fixed power of 2 of the sector size (called the cluster factor), and their size can be between 512 bytes and 4 KB (Sometimes 64 KB, but 4 KB is the largest cluster size that the current NTFS compression engine can operate with. That limit may be related to the 4 KB page size used on the Intel i386 CPU). This size can be set with the Windows NT format utility, whose default is: Volume size Cluster size 1 to 512 MB Sector size 512 MB to 1 GB 1 KB 1 GB to 2 GB 2 KB more than 2 GB 4 KB
$Boot	This metadata file points at the boot sector of the volume. It contains information about the size of the volume, clusters and the MFT. (More...)
Byte (See Units)	See Units.
chkdsk	This is a DOS and Windows utility to check and repair filesystems. Its name is an abbreviation of check disk. See Also fsck.
Cluster	This is the smallest unit of disk that NTFS uses and it is a multiple of the sector size. It is determined when the volume is formatted and cannot be altered afterwards. See Also Sector, $Boot, Volume.
Compression	NTFS supports file- and directory-level compression. The compression is performed transparently when the file is read or written. Any new files in a compressed directory will automatically be compressed. See Also Compression Unit.

Compression Unit	Each file marked to be compressed is divided into sixteen cluster blocks, known as compression units. If one of these blocks cannot be compressed into fifteen clusters or less it is left uncompressed. This division also helps accessing a file randomly, ie it isn't necessary to decompress the whole file. See Also Cluster, Compression.
$DATA	This attribute contains the actual data for a file. This stream may also have a name. (More...)
Data Runs	Non-resident attributes are stored in intervals of clusters called runs. Each run is represented by its starting cluster and its length. The runs map the VCNs of a file to the LCNs of a volume. (More...) See Also Attribute, Cluster, Logical Cluster Number (LCN), Virtual Cluster Number (VCN), Volume.
Decimal	Maths carried out in base ten. In this documentation, numbers that are neither in hex, nor binary, are in decimal, e.g. 16 (sixteen), 23 (twenty-three). See Also Binary, Hex, Hexadecimal, Units.
Directory	An NTFS directory is an index attribute. NTFS uses index attributes to collate file names. A directory entry contains the name of the file and a copy of the file's standard information attribute (time stamp information). This approach provides a performance boost for directory browsing because NTFS does not need to read the files' MFT records to print directory information. (More...)
DOS File Permissions (see File Permissions)	See File Permissions.
Dot, Root Directory	Root directory of the disk (More...)
Drive (See Volume)	See Volume.
Dynamic Disk	`Dynamic disk SDS, win2k`
$EA	This attribute is used to implement the HPFS extended attribute under NTFS.

It is only used for OS/2 compatibity.

(More...)

$EA_INFORMATION — This attribute is used to implement the HPFS extended attribute under NTFS.

It is only used for OS/2 compatibity.

(More...)

$EFS — $EFS is the named $LOGGED_UTILITY_STREAM of any encrypted file.
See Also $LOGGED_UTILITY_STREAM.

$Extend — This metadata directory contains the metadata files:

- $ObjId
- $Quota
- $Reparse

(More...)

File — In the NTFS terminology, a file can be a normal file, directory (like in Linux) or a system file.

(More...)

$FILE_NAME — This attribute represents the file's name.

A file can have one or more names, which can be in any directory.

This is the NTFS equivalent to Unix's hard links.

(More...)

Filename Namespace — Not all characters are valid in DOS filenames.

For compatibity NTFS stores which namespace the name belongs to.

(More...)

File Permissions — NTFS supports the standard set of DOS file permissions, namely *Archive*, *System*, *Hidden* and *Read Only*.

In addition, NTFS supports *Compressed* and *Encrypted*.
See Also $SECURITY_DESCRIPTOR, Compression.

FILE Record — The $MFT is made up of FILE records, so named because of a magic number of *FILE*.

Each record has a standard header and a list of attributes.

If the attributes don't fit into a single record, then more records will be used and a $ATTRIBUTE_LIST attribute will be needed.

See Also Attribute, $ATTRIBUTE_LIST, Magic Number, $MFT.

File Record Segment (FRS)

```
FRS = MFT File Record
```

File Reference — Each file record has a unique number identifying it.

The first 48 bits are a sequentially allocated number which is the offset in the $MFT.

The last 16 bits are a sequence number.

Every time the record is altered this number is incremented.

The sequence number can help detect errors on the volume.
See Also FILE Record, $MFT, Volume.

File Runs (See Data Runs) — See Data Runs.

File Size — There are three file sizes that NTFS records.

Each of them stores the number of bytes.

- R) Real. The number of bytes of data.
- A) Allocated. The size taken up on disk.
- I) Initialised. Size of compressed file.

If the file is compressed, the Initialised Size may be smaller than the Real Size.

Filesystem — The physical structure an operating system uses to store and organize files on a storage unit.

A commonly used filesystem is FAT (used by DOS).

Fixup (See Update Sequence) — See Update Sequence.

Fork (See Resource Fork) — See Resource Fork.

Fragmented — (un)f file

FRS (See File Record Segment) — See File Record Segment (FRS).

fsck — This is a utility to check and repair filesystems.

Its name is an abbreviation of filesystem check.

GB (See Units) — See Units.

GUID (See Units)

```
The valid format for a GUID is {XXXXXXXX-XXXX-XXXX-X
Globally Unique Identifier (GUID)
```

```
GUID structures store globally unique identifiers (G
128-bit value consisting of one group of eight hexad
by three groups of four hexadecimal digits each, fol
twelve hexadecimal digits. GUIDs are Microsoft's imp
distributed computing environment (DCE) universally
Example of a GUID:
        1F010768-5A73-BC91-0010A52216A7

order stored on disk?

01020304-0506-0708-090A0B0C0D0E0F010

0x00  04030201
0x04  0605
0x06  0807
0x08  090A0B0C0D0E0F010
```

Hex, Hexadecimal

Maths carried out in base sixteen.

In this documentation, many numbers represented in hex, e.g. 0x02E0, 0xF100.
See Also Binary, Decimal, Units.

HFS (See Hierarchical File System)

See Hierarchical File System (HFS).

Hierarchical File System (HFS)

The MacOS filesystem.

High Performance File System (HPFS)

The OS/2 filesystem.

Remember: once upon a time, OS/2 had to be the operating system developed by both IBM and Microsoft.

There was a break between the 2 giants. IBM continued to develop OS/2 (it became OS/2 Warp), and that explains why OS/2 knows how to execute Windows applications. Microsoft decided to make its own operating system: Windows NT.

HPFS design influenced NTFS design, so the 2 filesystems share many features.

HPFS (See High Performance File System)

See High Performance File System (HPFS).

$I30

This is the named index used by directories.

The name refers to attribute 0x30 ($FILE_NAME).
See Also Attribute, Directory, $FILE_NAME, Index.

Index

just the whole index idea)

$INDEX_ALLOCATION

This attribute contains the location of the entries that make up an index.

(More...)

$INDEX_ROOT

This attribute is the root of an index.

The index is stored as a balanced binary tree.

The only attribute which is indexed is $FILE_NAME and the index is called $I30.

(More...)

INDX Record
: Index records are used by directories, $Quota, $Reparse and $Secure.

The contents depend on the type of index being kept.

Directories store $FILE_NAME attributes.

(More...)
See Also Directory, $I30, $Quota, $Reparse, $Secure.

Infinite Logging Area
: Something contained in $LogFile. It consists of a sequence of 4KB log records.
See Also $LogFile.

Inode
: An inode is the filesystems representation of a file, directory, device, etc.

In NTFS every inode it represented by an MFT FILE record.
See Also Directory, File, FILE Record, Filesystem.

$J
: $J is a named data stream of the Metadata File $UsnJrnl.
See Also $UsnJrnl.

Junction Point
: Microsoft term for a mount point, available in NT 5.0.

KB (See Units)
: See Units.

LCN (See Logical Cluster Number)
: See Logical Cluster Number (LCN).

Log Record
: One 4KB chunk of the infinite logging area. It starts with the magic number 'RCRD' and a fixup, then has undocumented variable length data. [The log record might be further subdivided - I cannot imagine they waste 4KB if they only have to log a few bytes. Custer mentions high level and low level 'records'. High level are: - allocate inode n, - make a directory entry foo in directory m low level are: - modify inode n with the new contents of <1KB>]

$LogFile
: This metadata file is used to guarantee data integrity in case of a system failure.

It has two copies of the restart area and the infinite logging area.

The log file is near the centre of the volume, just after the second cluster of the boot file. [Better say 'run' than cluster. The boot file usually extends over several clusters at the beginning of the disk, and then has a single run of just one cluster (the copy of the boot sector). Also, isn't it 'infinite'?]

Transactional logging file

(More...)

AM	This attribute is used by encrypted files. (More...)
Logical Cluster Number (LCN)	A volume is divided into clusters. They are numbered sequentially, starting at zero. See Also Cluster, Volume.
Logical Sequence Number (LSN)	A serial number used to identify an NTFS log record.
LSN (See Logical Sequence Number)	See Logical Sequence Number (LSN).
Magic Number	Most of the on-disk structures in NTFS have a unique constant identifying them. This number is usually located at the beginning of the structure and can be used as a sanity check.
Master File Table	See Master File Table.
$Max	$Max is a named Data Stream of $UsnJrnl. See Also $UsnJrnl.
MB (See Units)	See Units.
Metadata	Data on the storage unit used by the filesystem only, as a frame to access user data. Metadata constitutes the structure of the filesystem). Metadata examples from various filesystems include FATs, inode tables, free block lists, free block bitmaps, logging areas, and the superblock.

```
meta-data

Data about data. In data processing, meta-data is
that provides information about or documentation o
within an application or environment.

For example, meta data would document data about d
attributes, (name, size, data type, etc) and data
data structures (length, fields, columns, etc) and
(where it is located, how it is associated, owners
data may include descriptive information about the
and condition, or characteristics of the data.
```

$MFT	This metadata file, the Master File Table, is an index of all the files on the volume. It contains the attributes of each file and the root of any indexes. (More...)
$MFTMirr	This metadata file stores a copy of the first four records of $MFT. It is a safety measure which probably only gets used when chkdsk is run.

	(More...)
$MountMgrDatabase	$MountMgrDatabase is a named Data Stream of dot (the root directory). It contains a list of mounted volumes. See Also Dot, Root Directory.
MST (See Multi-Sector Transfer)	See Multi-Sector Transfer.
Multi-Sector Transfer	`multiple sectors, fixup, safety checks`
Nibble	Half of a byte (4 bits).
NT Authority	The NT Authority defines the scope of the security identifier. Numbers 0 - 4 represent internal identifiers, e.g. World, Local. 5 represents the NT Authority. See Also NT Sub Authority, Security Identifier (SID), $SECURITY_DESCRIPTOR.
NTFS	NTFS is the file system of Windows NT, Windows 2000 and Windows XP. (More...) See Also Filesystem.
NT Sub Authority	The Sub Authority can contain any number of fields (five is usual). Sub Authorities beginning with 21 (0x15) denote a NT Domain identifier. See Also NT Authority, Security Identifier (SID), $SECURITY_DESCRIPTOR.
$O	This is one of the named indexes belonging to $Quota and $ObjId. See Also Index, $Q, $ObjId, $Quota.
$OBJECT_ID	This attribute stores a mapping between a SID and a Security Hash. (More...)
$ObjId	This attribute record's the unique identifiers given to files and directorys when using Distributed Link Tracking. (More...)
PAM	Pluggable Authentication Modules (PAM) are a set of libraries for validating security on Linux.
Partition (See Volume)	See Volume.
Partition Table	`partition table...`

SFS Win2K dynamic disk

Permissions	There are two mechanisms for storing permissions in NTFS. One is a superset of DOS File Permissions, which includes *Read Only* and *Hidden*. The other is based on ACEs and allows granting specific permissions to specific users. See Also Access Control Entry (ACE), File Permissions, $SECURITY_DESCRIPTOR.
POSIX	An acronym (pronounced like positive) for Portable Operating System Interface, suggested by Richard M. Stallman. It is a set of international standards (ISO/IEC 9945-1:1996(E), ANSI/IEEE Std 1003.1 1996 Edition) to interface with Unix-like exploitation systems, e.g. Linux. NTFS does not support Unix-like device files.
$PROPERTY_SET	An obsolete attribute (0xF0) from NT4
$Q	This is one of the named indexes belonging to $Quota. See Also Index, $O, $Quota.
$Quota	This metadata file stores information about file quotas. (More...)
$R	This is the named index belonging to $Reparse. See Also Index, $Reparse.
RCRD Record	This record is used in the $LogFile. Each represents an atomic transaction that is to be performed. See Also $LogFile, Transaction.
Record	There are several record types in NTFS. FILE Record are used in the $MFT, INDX Records in indexes, RCRD and RSTR Records in the $LogFile. See Also FILE Record, INDX Record, RCRD Record, RSTR Record.
Recursion	See Recursion.
Reference	file (are there any others?)
$Reparse	This metadata file stores information about reparse points. (More...)
$REPARSE_POINT	This attribute stores information about reparse points. (More...)

Resource Fork — In MacOS's filesystem, HFS, files are allowed to have multiple data streams.

These are called resource forks.
See Also Hierarchical File System (HFS), Stream.

Roll-back — When an NTFS volume is mounted, it is checked to see if it is in a consistant state.

If it isn't then the $LogFile is consulted and transactions are undone until the disk returns to a consistant state.

This does not guarantee data integrity, only disk integrity.
See Also $LogFile, Transaction, Volume.

Root Directory (See Dot, Root Directory) — See Dot, Root Directory.

RSTR Record — Two copies of this are in $LogFile.

A restart area has the magic number 'RSTR' followed by a fixup and some other data, including three LSNs.

A restart area has a pointer into the log area, such as the first and last log records written and the last checkpoint record written. (that is three - now which is which?)

Runs (See Data Runs) — See Data Runs.

$SDH — This is one of the named indexes belonging to $Secure.
See Also Index, $SII, $Secure.

$SDS — This is the named data stream belonging to $Secure.
See Also $Secure, Stream.

Sector — Unit of data on the physical storage unit.

The storage controller can only access data in multiples of this unit.

A sector is usually 512 bytes, but can be 1 KB on certain Asian hard disks.

$Secure — This metadata file stores a table of security descriptors used by the volume.

(More...)

Security — There are two levels of security in NTFS.

There are the DOS File Permissions, such as *Read Only* and *Hidden* and an ACL model which grants specific permissions to specific users.
See Also Access Control Entry (ACE), Access Control List (ACL), Permissions, $SECURITY_DESCRIPTOR, Security Identifier (SID).

$SECURITY_DESCRIPTOR — This attribute stores all the security information about a file or directory.

It contains an ACL for auditing, an ACL for permissions and a SID

	to show the user and group of the owner. (More...) See Also Attribute, Access Control List (ACL), Access Control Entry (ACE), Security Identifier (SID).
Security Identifier (SID)	This variable-length identifier uniquely identifies a user or a group on an NT domain. It is used in the security permissions. See Also Access Control Entry (ACE), Access Control List (ACL), $SECURITY_DESCRIPTOR.
Sequence Array (See Update Sequence)	See Update Sequence.
SID (See Security Identifier)	See Update Sequence.
$SII	This is one of the named indexes belonging to $Secure. See Also Index, $SDH, $Secure.
Sparse File	NTFS supports sparse files. If a file contains large, contiguous, blocks of zeros, then NTFS can choose to not waste any space storing these portions on disk. They are represented as data runs containing nothing. When read from disk, NTFS simply substitutes zeros. See Also Data Runs.
$STANDARD_INFORMATION	This attribute contains information about a file, such as its file permissions and when it was created. (More...)
Stream	All data on NTFS is stored in streams, which can have names. A file can have more than one data streams, but exactly one must have no name. The *size* of a file is the size of its unnamed data attribute.
$SYMBOLIC_LINK	This attribute, like $VOLUME_VERSION existed in NTFS v1.2, but wasn't used. It does not longer exist in NTFS v3.0+.
TB (See Units)	See Units.
Time Stamp	NTFS stores four significant times referring to files and directories. They are: File creation time; Last modification time; Last modification of the MFT record; Last access time. NTFS stores dates as the number of 100ns units since Jan 1 st 1601. Unix, stores dates as the number of seconds since Jan 1 st 1970.

```
standardise 4 time fields name & description conce
refer to 4 times as:
  C creation
```

```
  A alter (modification)
  M mft (mft changed)
  R read (last access)

FIXME:
NOTE: There is conflicting information about the m
fields but the meaning as defined below has been v
correct by practical experimentation on Windows NT
assumed to be the one and only correct interpretat

creation_time
Time file was created. Updated when a filename is

last_data_change_time
Time the data attribute was last modified.

last_mft_change_time
Time this mft record was last modified.

last_access_time
Approximate time when the file was last accessed (
updated on read-only volumes). In Windows this is
accessed if some time delta has passed since the l



N.B. There is conflicting information about the me
fields but the meaning as defined below has been v
correct by practical experimentation on Windows NT
assumed to be the one and only correct interpretat
```

See Also FILE Record.

Transaction

A transaction on a system is a set of operations (on that system) that constitutes a unit. This unit can't be divided.

Before the transaction, the state of the system is well defined. During the transaction, it is undefined. After the transaction, it is well defined again.

A transaction can't be half-realized: if no operation fails, the transaction is realized. If on the contrary an error occurs in one or more of the operations, the transaction is not realized.

A set of (even atomic) operations is not atomic by definition. A transaction is a model that provides a kind of atomicity to this set of operations.

Unfragmented (see Fragmented)

See Fragmented.

Unicode

International character set coded on 16 bits (ASCII is coded on 7 bits and Latin-1 coded on 8 bits). Unicode can represent every symbol of almost every language in the world.

Units

Every size in this document is measured in bytes (unless clearly marked).

The abbreviations for sizes are:

Table 122. Measurement Units

Abbr.	Name	Exactly	Approx.
KB	Kilobyte	2^{10}	10^{3}
MB	Megabyte	2^{20}	10^{6}
GB	Gigabyte	2^{30}	10^{9}
TB	Terabyte	2^{40}	10^{12}

see also Binary, Decimal, Hexadecimal

N.B. Technically, the correct abbreviation for 1024 bytes is KiB, which stands for kilobinary bytes.

$UpCase

This metadata file contains 128KB of capital letters.

For each character in the Unicode alphabet, there is an entry in this file.

It is used to compare and sort filenames.

(More...)

Update Sequence

Several structures in NTFS have sequence numbers in them to check for consistancy errors.

They are FILE, INDX, RCRD and RSTR records.

Before the record is written to disk, the last two bytes of each sector are copied to an array in the header.

The update sequence number is then incremented and written to the end of each sector.

If any disk corruption occurs, this technique could detect it.

```
The Update Sequence Array (usa) is an array of the
to the end of each sector protected by the update
this array is contained. Note that the first entry
Number (usn), a cyclic counter of how many times t
been written to disk. The values 0 and -1 (ie. 0xf
last __u16's of each sector have to be equal to th
are set to it (during writing). If they are not, a
transfer has occured when the data was written.
The maximum size for the update sequence array is
        maximum size = usa_ofs + (usa_count * 2) =
The 510 bytes comes from the fact that the last __
(obviously) finish before the last __u16 of the fi
This formula can be used as a consistency check in
(usa_count * 2) has to be less than or equal to 51
```

See Also FILE Record, INDX Record, RCRD Record, RSTR Record.

$UsnJrnl

`used for logging`

(More...)

VCN (See Virtual Cluster Number)
: See Virtual Cluster Number (VCN).

Virtual Cluster Number (VCN)
: When representing the data runs of a file, the clusters are given virtual cluster numbers.

  Cluster zero refers to the first cluster of the file.

  The data runs map the VCNs to LCNs so that the file can be located on the volume.
  See Also Cluster, Logical Cluster Number (LCN), Volume.

Volume
: (=drive=partition) (extended, striped, mirrored (not supported))

  A logical NTFS partition. It is a group of physical partitions (see the fdisk utility, you can set up mirroring and stripping) that act as one (somewhat like the Linux md block devices).

$Volume
: This metadata file contains information such as the name, serial number and whether the volume needs checking for errors.

  (More...)

$VOLUME_INFORMATION
: This attribute contains information such as the serial number, creation time and whether the volume needs checking for errors.

  (More...)

$VOLUME_NAME
: This attribute stores the name of the volume in Unicode.

  (More...)

$VOLUME_VERSION
: This attribute, like $SYMBOLIC_LINK existed in NTFS v1.2, but wasn't used.

  It does not longer exist in NTFS v3.0+.