Australian Journal of Basic and Applied Sciences, 6(7): 325-338, 2012
ISSN 1991-8178

# Review NTFS Basics

Behzad Mahjour Shafiei, Farshid Iranmanesh, Fariborz Iranmanesh

Bardsir Branch, Islamic Azad University, Bardsir, Iran

**Abstract:** The Windows NT file system (NTFS) provides a combination of performance, reliability, and compatibility not found in the FAT file system. It is designed to quickly perform standard file operations such as read, write, and search - and even advanced operations such as file-system recovery - on very large hard disks.

**Key words:** Format, NTFS, Volume, Fat, Partition

## INTRODUCTION

Formatting a volume with the NTFS file system results in the creation of several system files and the Master File Table (MFT), which contains information about all the files and folders on the NTFS volume.

The first information on an NTFS volume is the Partition Boot Sector, which starts at sector 0 and can be up to 16 sectors long. The first file on an NTFS volume is the Master File Table (MFT).

The following figure illustrates the layout of an NTFS volume when formatting has finished.


**Fig. 5-1:** Formatted NTFS Volume.

This chapter covers information about NTFS. Topics covered are listed below:

- NTFS Partition Boot Sector
- NTFS Master File Table (MFT)
- NTFS File Types
- NTFS File Attributes
- NTFS System Files
- NTFS Multiple Data Streams
- NTFS Compressed Files
- NTFS & EFS Encrypted Files
- Using EFS
- EFS Internals
- $EFS Attribute
- Issues with EFS
- NTFS Sparse Files
- NTFS Data Integrity and Recoverability

The NTFS file system includes security features required for file servers and high-end personal computers in a corporate environment. The NTFS file system also supports data access control and ownership privileges that are important for the integrity of critical data. While folders shared on a Windows NT computer are assigned particular permissions, NTFS files and folders can have permissions assigned whether they are shared or not. NTFS is the only file system on Windows NT that allows you to assign permissions to individual files.

The NTFS file system has a simple, yet very powerful design. Basically, everything on the volume is a file and everything in a file is an attribute, from the data attribute, to the security attribute, to the file name attribute. Every sector on an NTFS volume that is allocated belongs to some file. Even the file system metadata (information that describes the file system itself) is part of a file.

***What's New in NTFS5 (Windows 2000):***

***Encryption:***

The Encrypting File System (EFS) provides the core file encryption technology used to store encrypted files on NTFS volumes. EFS keeps files safe from intruders who might gain unauthorized physical access to sensitive, stored data (for example, by stealing a portable computer or external disk drive).

**Corresponding Author:** Behzad Mahjour Shafiei, Computer software engineering student, Islamic Azad University, Kerman Branch, Kerman, Iran
E-mail: B_m_Shafiei@hotmail.com

***Disk Quotas:***

Windows 2000 supports disk quotas for NTFS volumes. You can use disk quotas to monitor and limit disk-space use.

***Reparse Points:***

Reparse points are new file system objects in NTFS that can be applied to NTFS files or folders. A file or folder that contains a reparse point acquires additional behavior not present in the underlying file system. Reparse points are used by many of the new storage features in Windows 2000, including volume mount points.

***Volume Mount Points:***

Volume mount points are new to NTFS. Based on reparse points, volume mount points allow administrators to graft access to the root of one local volume onto the folder structure of another local volume.

***Sparse Files:***

Sparse files allow programs to create very large files but consume disk space only as needed.

***Distributed Link Tracking:***

NTFS provides a link-tracking service that maintains the integrity of shortcuts to files as well as OLE links within compound documents.

***Partition Boot Sector:***

Table 5-1 describes the boot sector of a volume formatted with NTFS. When you format an NTFS volume, the format program allocates the first 16 sectors for the boot sector and the bootstrap code.

**Table 5-1:** NTFS Boot Sector.

| Byte Offset | Field Length | Field Name |
|---|---|---|
| 0x00 | 3 bytes | Jump Instruction |
| 0x03 | LONGLONG | OEM ID |
| 0x0B | 25 bytes | BPB |
| 0x24 | 48 bytes | Extended BPB |
| 0x54 | 426 bytes | Bootstrap Code |
| 0x01FE | WORD | End of Sector Marker |

On NTFS volumes, the data fields that follow the BPB form an extended BPB. The data in these fields enables Ntldr (NT loader program) to find the master file table (MFT) during startup. On NTFS volumes, the MFT is not located in a predefined sector, as on FAT16 and FAT32 volumes. For this reason, the MFT can be moved if there is a bad sector in its normal location. However, if the data is corrupted, the MFT cannot be located, and Windows NT/2000 assumes that the volume has not been formatted.

The following example illustrates the boot sector of an NTFS volume formatted while running Windows 2000. The printout is formatted in three sections:

Bytes 0x00- 0x0A are the jump instruction and the OEM ID (shown in bold print).

Bytes 0x0B-0x53 are the BPB and the extended BPB.

The remaining code is the bootstrap code and the end of sector marker (shown in bold print).

```
Physical Sector:Cyl 0, Side 1, Sector 1
00000000:EB 52 90 4E 54 46 53 20 -20 20 20 00 02 08 00 00 .R.NTFS ........
00000010:00 00 00 00 00 F8 00 00 -3F 00 FF 00 3F 00 00 00 ........?...?...
00000020:00 00 00 00 80 00 80 00 -4A F5 7F 00 00 00 00 00 ........J.......
00000030:04 00 00 00 00 00 00 00 -54 FF 07 00 00 00 00 00 ........T.......
00000040:F6 00 00 00 01 00 00 00 -14 A5 1B 74 C9 1B 74 1C ...........t..t.
00000050:00 00 00 00 FA 33 C0 8E -D0 BC 00 7C FB B8 C0 07 .....3.....|....
00000060:8E D8 E8 16 00 B8 00 0D -8E C0 33 DB C6 06 0E 00 ..........3.....
00000070:10 E8 53 00 68 00 0D 68 -6A 02 CB 8A 16 24 00 B4 ..S.h..hj....$..
00000080:08 CD 13 73 05 B9 FF FF -8A F1 66 0F B6 C6 40 66 ...s......f...@f
00000090:0F B6 D1 80 E2 3F F7 E2 -86 CD C0 ED 06 41 66 0F .....?.......Af.
000000A0:B7 C9 66 F7 E1 66 A3 20 -00 C3 B4 41 BB AA 55 8A ..f..f....A..U.
000000B0:16 24 00 CD 13 72 0F 81 -FB 55 AA 75 09 F6 C1 01 .$...r...U.u....
000000C0:74 04 FE 06 14 00 C3 66 -60 1E 06 66 A1 10 00 66 t......f`..f...f
000000D0:03 06 1C 00 66 3B 06 20 -00 0F 82 3A 00 1E 66 6A ....f;....:..fj
000000E0:00 66 50 06 53 66 68 10 -00 01 00 80 3E 14 00 00 .fP.Sfh.....>...
000000F0:0F 85 0C 00 E8 B3 FF 80 -3E 14 00 00 0F 84 61 00 ........>.....a.
00000100:B4 42 8A 16 24 00 16 1F -8B F4 CD 13 66 58 5B 07 .B..$.......fX [..
```

```
00000110:66 58 66 58 1F EB 2D 66 -33 D2 66 0F B7 0E 18 00 fXfX.-f3.f......
00000120:66 F7 F1 FE C2 8A CA 66 -8B D0 66 C1 EA 10 F7 36 f......f..f....6
00000130:1A 00 86 D6 8A 16 24 00 -8A E8 C0 E4 06 0A CC B8 ......$.........
00000140:01 02 CD 13 0F 82 19 00 -8C C0 05 20 00 8E C0 66 ..............f
00000150:FF 06 10 00 FF 0E 0E 00 -0F 85 6F FF 07 1F 66 61 ..........o...fa
00000160:C3 A0 F8 01 E8 09 00 A0 -FB 01 E8 03 00 FB EB FE ................
00000170:B4 01 8B F0 AC 3C 00 74 -09 B4 0E BB 07 00 CD 10 .....<.t........
00000180:EB F2 C3 0D 0A 41 20 64 -69 73 6B 20 72 65 61 64 .....A disk read
00000190:20 65 72 72 6F 72 20 6F -63 63 75 72 72 65 64 00 error occurred.
000001A0:0D 0A 4E 54 4C 44 52 20 -69 73 20 6D 69 73 73 69 ..NTLDR is missi
000001B0:6E 67 00 0D 0A 4E 54 4C -44 52 20 69 73 20 63 6F ng...NTLDR is co
000001C0:6D 70 72 65 73 73 65 64 -00 0D 0A 50 72 65 73 73 mpressed...Press
000001D0:20 43 74 72 6C 2B 41 6C -74 2B 44 65 6C 20 74 6F Ctrl+Alt+Del to
000001E0:20 72 65 73 74 61 72 74 -0D 0A 00 00 00 00 00 00 restart........
000001F0:00 00 00 00 00 00 00 00 -83 A0 B3 C9 00 00 55 AA ..............U.
```

The following table describes the fields in the BPB and the extended BPB on NTFS volumes. The fields starting at 0x0B, 0x0D, 0x15, 0x18, 0x1A, and 0x1C match those on FAT16 and FAT32 volumes. The sample values correspond to the data in this example.

| Byte Offset | Field Length | Sample Value | Field Name |
|---|---|---|---|
| 0x0B | WORD | 0x0002 | Bytes Per Sector |
| 0x0D | BYTE | 0x08 | Sectors Per Cluster |
| 0x0E | WORD | 0x0000 | Reserved Sectors |
| 0x10 | 3 BYTES | 0x000000 | *always 0* |
| 0x13 | WORD | 0x0000 | *not used by NTFS* |
| 0x15 | BYTE | 0xF8 | Media Descriptor |
| 0x16 | WORD | 0x0000 | *always 0* |
| 0x18 | WORD | 0x3F00 | Sectors Per Track |
| 0x1A | WORD | 0xFF00 | Number Of Heads |
| 0x1C | DWORD | 0x3F000000 | Hidden Sectors |
| 0x20 | DWORD | 0x00000000 | *not used by NTFS* |
| 0x24 | DWORD | 0x80008000 | *not used by NTFS* |
| 0x28 | LONGLONG | 0x4AF57F0000000000 | Total Sectors |
| 0x30 | LONGLONG | 0x0400000000000000 | Logical Cluster Number for the file $MFT |
| 0x38 | LONGLONG | 0x54FF070000000000 | Logical Cluster Number for the file $MFTMirr |
| 0x40 | DWORD | 0xF6000000 | Clusters Per File Record Segment |
| 0x44 | DWORD | 0x01000000 | Clusters Per Index Block |
| 0x48 | LONGLONG | 0x14A51B74C91B741C | Volume Serial Number |
| 0x50 | DWORD | 0x00000000 | Checksum |

***Protecting the Boot Sector:***

Because a normally functioning system relies on the boot sector to access a volume, it is highly recommended that you run disk scanning tools such as Chkdsk regularly, as well as back up all of your data files to protect against data loss if you lose access to a volume.

***NTFS Master File Table (MFT):***

Each file on an NTFS volume is represented by a record in a special file called the master file table (MFT). NTFS reserves the first 16 records of the table for special information. The first record of this table describes the master file table itself, followed by a MFT *mirror record*. If the first MFT record is corrupted, NTFS reads the second record to find the MFT mirror file, whose first record is identical to the first record of the MFT. The locations of the data segments for both the MFT and MFT mirror file are recorded in the boot sector. A duplicate of the boot sector is located at the logical center of the disk.

The third record of the MFT is the log file, used for file recovery. The seventeenth and following records of the master file table are for each file and directory (also viewed as a file by NTFS) on the volume.

Figure provides a simplified illustration of the MFT structure:

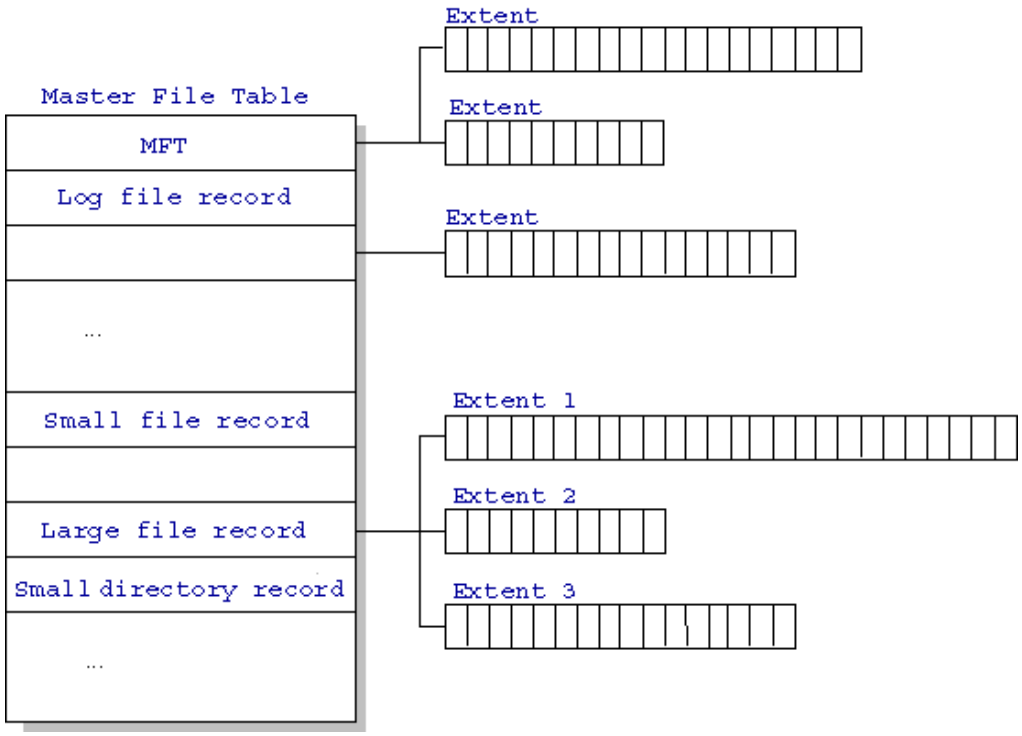



**Fig. 5-2:** MFT Structure.

The master file table allocates a certain amount of space for each file record. The attributes of a file are written to the allocated space in the MFT. Small files and directories (typically 1500 bytes or smaller), such as the file illustrated in next figure, can entirely be contained within the master file table record.

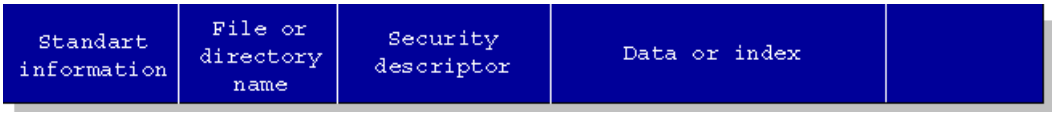



**Fig. 5-2:** MFT Record for a Small File or Directory:

This design makes file access very fast. Consider, for example, the FAT file system, which uses a file allocation table to list the names and addresses of each file. FAT directory entries contain an index into the file allocation table. When you want to view a file, FAT first reads the file allocation table and assures that it exists. Then FAT retrieves the file by searching the chain of allocation units assigned to the file. With NTFS, as soon as you look up the file, it's there for you to use.

Directory records are housed within the master file table just like file records. Instead of data, directories contain index information. Small directory records reside entirely within the MFT structure. Large directories are organized into B-trees, having records with pointers to external clusters containing directory entries that could not be contained within the MFT structure.

***NTFS File Types:***

NTFS File Attributes
NTFS System Files
NTFS Multiple Data Streams
NTFS Compressed Files
NTFS Encrypted Files
Using EFS
EFS Internals
$EFS Attribute
Issues with EFS
NTFS Sparse Files

***NTFS File Attributes:***

The NTFS file system views each file (or folder) as a set of file attributes. Elements such as the file's name, its security information, and even its data, are all file attributes. Each attribute is identified by an attribute type code and, optionally, an attribute name.

When a file's attributes can fit within the MFT file record, they are called resident attributes. For example, information such as filename and time stamp are always included in the MFT file record. When all of the information for a file is too large to fit in the MFT file record, some of its attributes are nonresident. The nonresident attributes are allocated one or more clusters of disk space elsewhere in the volume. NTFS creates the Attribute List attribute to describe the location of all of the attribute records.

Table 5-3 lists all of the file attributes currently defined by the NTFS file system. This list is extensible, meaning that other file attributes can be defined in the future.

**Table 5-3:** File Attributes Defined by NTFS.

| Attribute Type | Description |
|---|---|
| Standard Information | Includes information such as timestamp and link count. |
| Attribute List | Lists the location of all attribute records that do not fit in the MFT record. |
| File Name | A repeatable attribute for both long and short file names. The long name of the file can be up to 255 Unicode characters. The short name is the 8.3, case-insensitive name for the file. Additional names, or hard links, required by POSIX can be included as additional file name attributes. |
| Security Descriptor | Describes who owns the file and who can access it. |
| Data | Contains file data. NTFS allows multiple data attributes per file. Each file typically has one unnamed data attribute. A file can also have one or more named data attributes, each using a particular syntax. |
| Object ID | A volume-unique file identifier. Used by the distributed link tracking service. Not all files have object identifiers. |
| Logged Tool Stream | Similar to a data stream, but operations are logged to the NTFS log file just like NTFS metadata changes. This is used by EFS. |
| Reparse Point | Used for volume mount points. They are also used by Installable File System (IFS) filter drivers to mark certain files as special to that driver. |
| Index Root | Used to implement folders and other indexes. |
| Index Allocation | Used to implement folders and other indexes. |
| Bitmap | Used to implement folders and other indexes. |
| Volume Information | Used only in the $Volume system file. Contains the volume version. |
| Volume Name | Used only in the $Volume system file. Contains the volume label. |

***NTFS System Files:***

NTFS includes several system files, all of which are hidden from view on the NTFS volume. A *system file* is one used by the file system to store its metadata and to implement the file system. System files are placed on the volume by the Format utility.

**Table 5-4:** Metadata Stored in the Master File Table.

| System File | File Name | MFT Record | Purpose of the File |
|---|---|---|---|
| Master file table | $Mft | 0 | Contains one base file record for each file and folder on an NTFS volume. If the allocation information for a file or folder is too large to fit within a single record, other file records are allocated as well. |
| Master file table 2 | $MftMirr | 1 | A duplicate image of the first four records of the MFT. This file guarantees access to the MFT in case of a single-sector failure. |
| Log file | $LogFile | 2 | Contains a list of transaction steps used for NTFS recoverability. Log file size depends on the volume size and can be as large as 4 MB. It is used by Windows NT/2000 to restore consistency to NTFS after a system failure. |
| Volume | $Volume | 3 | Contains information about the volume, such as the volume label and the volume version. |
| Attribute definitions | $AttrDef | 4 | A table of attribute names, numbers, and descriptions. |
| Root file name index | $ | 5 | The root folder. |
| Cluster bitmap | $Bitmap | 6 | A representation of the volume showing which clusters are in use. |
| Boot sector | $Boot | 7 | Includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable. |
| Bad cluster file | $BadClus | 8 | Contains bad clusters for the volume. |
| Security file | $Secure | 9 | Contains unique security descriptors for all files within a volume. |
| Upcase table | $Upcase | 10 | Converts lowercase characters to matching Unicode uppercase characters. |
| NTFS extension file | $Extend | 11 | Used for various optional extensions such as quotas, reparse point data, and object identifiers. |

| | | | |
|---|---|---|---|
| | | 12–15 | Reserved for future use. |

***NTFS Multiple Data Streams:***

NTFS supports multiple data streams, where the stream name identifies a new data attribute on the file. A handle can be opened to each data stream. A data stream, then, is a unique set of file attributes. Streams have separate opportunistic locks, file locks, and sizes, but common permissions.

This feature enables you to manage data as a single unit. The following is an example of an alternate stream:

myfile.dat:**stream2**

A library of files might exist where the files are defined as alternate streams, as in the following example:

library:file1
:file2
:file3

A file can be associated with more than one application at a time, such as Microsoft® Word and Microsoft® WordPad. For instance, a file structure like the following illustrates file association, but not multiple files:

program:source_file
:doc_file
:object_file
:executable_file

To create an alternate data stream, at the command prompt, you can type commands such as:

echo text>program:source_file
more<program:source_file

***Important:***

When you copy an NTFS file to a FAT volume, such as a floppy disk, data streams and other attributes not supported by FAT are lost.

***NTFS Compressed Files:***

Windows NT/2000 supports compression on individual files, folders, and entire NTFS volumes. Files compressed on an NTFS volume can be read and written by any Windows-based application without first being decompressed by another program.

Decompression occurs automatically when the file is read. The file is compressed again when it is closed or saved. Compressed files and folders have an attribute of **C** when viewed in Windows Explorer.

Only NTFS can read the compressed form of the data. When an application such as Microsoft® Word or an operating system command such as **copy** requests access to the file, the compression filter driver decompresses the file before making it available. For example, if you copy a compressed file from another Windows NT/2000–based computer to a compressed folder on your hard disk, the file is decompressed when read, copied, and then recompressed when saved.

This compression algorithm is similar to that used by the Windows 98 application DriveSpace 3, with one important difference — the limited functionality compresses the entire primary volume or logical volume. NTFS allows for the compression of an entire volume, of one or more folders within a volume, or even one or more files within a folder of an NTFS volume.

The compression algorithms in NTFS are designed to support cluster sizes of up to 4 KB. When the cluster size is greater than 4 KB on an NTFS volume, none of the NTFS compression functions are available.

Each NTFS data stream contains information that indicates whether any part of the stream is compressed. Individual compressed buffers are identified by "holes" following them in the information stored for that stream. If there is a hole, NTFS automatically decompresses the preceding buffer to fill the hole.

NTFS provides real-time access to a compressed file, decompressing the file when it is opened and compressing it when it is closed. When writing a compressed file, the system reserves disk space for the uncompressed size. The system gets back unused space as each individual compression buffer is compressed.

***EFS - Encrypting File System. Encrypted Files and Folders (NTFS5 only):***

The Encrypting File System (EFS) provides the core file encryption technology used to store encrypted files on NTFS volumes. EFS keeps files safe from intruders who might gain unauthorized physical access to sensitive, stored data (for example, by stealing a portable computer or external disk drive).

Users work with encrypted files and folders just as they do with any other files and folders. Encryption is transparent to the user who encrypted the file; the system automatically decrypts the file or folder when the user accesses. When the file is saved, encryption is reapplied. Users who are not authorized to access the encrypted files or folders transparently receive an “Access denied” message if they try to open, copy, move, or rename the

encrypted file or folder. The exact message text may vary depending on application which tries to access the file, because it is related not to user rights for file but to ability of EFS to decrypt file using user's private key.
EFS has the following benefits over 3rd party encrypting applications:

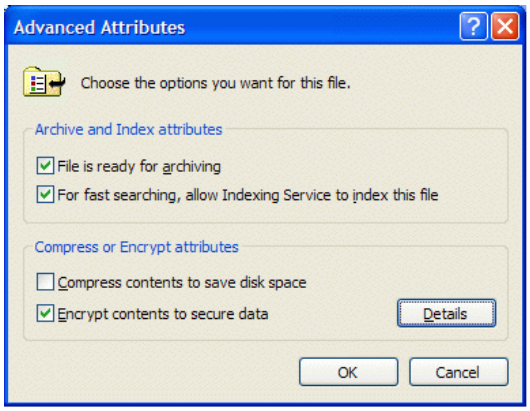
1. It is transparent for user and any applications. There's no risk for user to forget to encrypt file and leave data unprotected. Once file or folder is marked as encrypted, it will be encrypted in background without interaction with user. User does not need to remember password to decrypt files.
2. Strong key security. In contrast to other solutions when keys are based on user entered pass-phrase, EFS generates keys which are tolerant to dictionary based attacks.
3. All encrypting/decrypting processes are performed in kernel mode, excluding the risk of leaving key in paging file, from where it could be possibly extracted.
4. EFS provides data recovery mechanism which is valuable in business environment, giving an organization an opportunity to restore data even if the employee who encrypted it left the company.



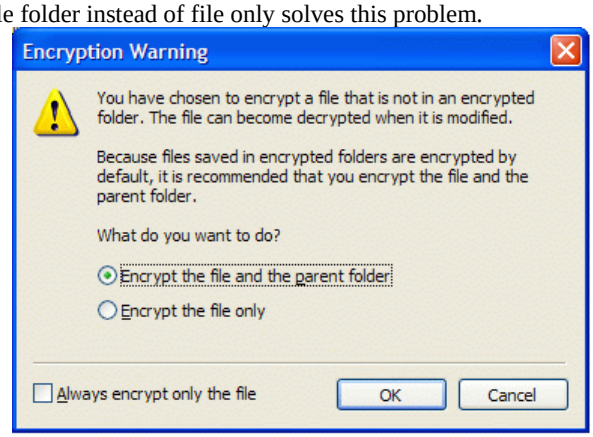
**EFS - Encrypting File System. Encrypted Files and Folders (NTFS5 only)**

**Using EFS**

User can invoke EFS features through Windows Explorer or by using a command-line utility called **cipher.exe**. To use Windows Explorer to encrypt file, open File property window by right clicking on file name. Click **Advanced...** button - Advanced Attributes dialog will be opened allowing you to mark file as encrypted.




Before saving new settings Windows will prompt user to encrypt file only or the whole folder. It address very important issue - while the file itself could be perfectly protected, the application which opens the file may create a temporary copies of the file while working with the document. The example is Microsoft Word. When user opens encrypted document, EFS decrypts it transparently for Word. Then during the work, Word creates temporary hidden file where it automatically saves the document in the process of editing and deletes it on the exit. This hidden file presents a real breach in security because it contains user data in plain (not encrypted) form. Encrypting the whole folder instead of file only solves this problem.




**EFS - Encrypting File System. Encrypted Files and Folders (NTFS5 only)**

**EFS Internals**

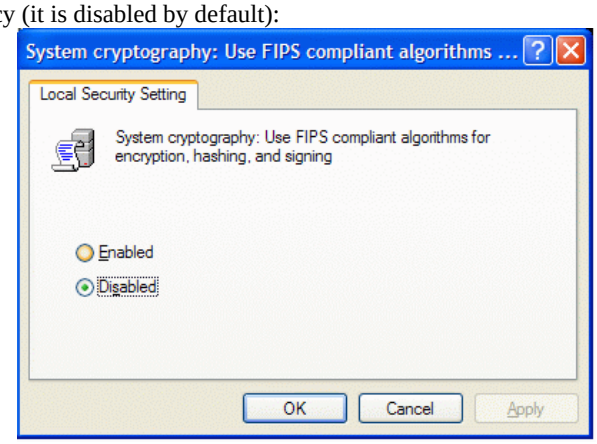
**EFS uses symmetric key encryption in combination with public key technology to protect files.** File data is being encrypted with symmetric algorithm (DESX). The key, used in symmetric encryption is called *File Encryption Key* (FEK). The FEK in its own turn is encrypted with a public/private key algorithm (RSA) and stored along with the file. The reason why two different algorithms are used is the speed of encryption. The performance burden of asymmetric algorithms is too much to use them for encrypting a large amount of data. Symmetric algorithms are about 1000 times faster making their suitable for encrypting of large amounts of data.

As a first setp to encrypt file, NTFS creates a log file called Efs0.log in System Volume Information folder on the same drive, as encrypted file. Then EFS aquires access CryptoAPI context. It uses Microsoft Base Cryptographic Provider 1.0 as cryptographic provider. Having the crypto context open, EFS generate File Encryption Key (FEK).

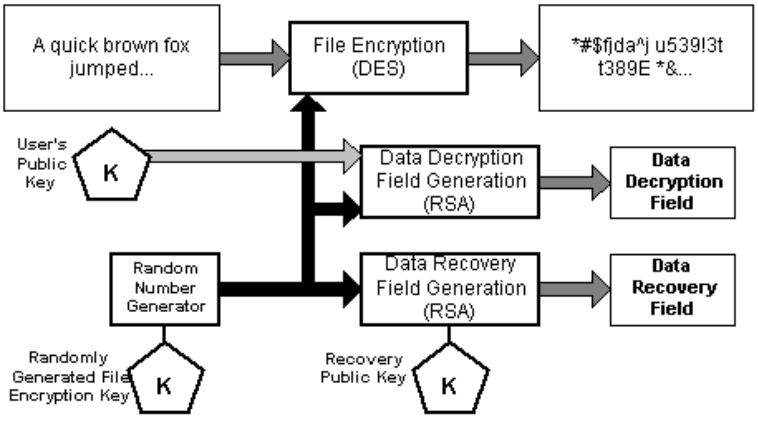
The next step is to get public/private key pair; if it does not exist at this stage (the case when EFS invoked first time), EFS generate a new pair. EFS uses 1024-bit RSA algorithm to encrypt FEK.

Then, EFS creates Data Decryption Field (DDF) for the current user, where it places FEK and encrypts it with public key. If recovery agent is defined by system policy, EFS creates also Data Recovery Field (DRF) and places there FEK encrypted with public key of recover agent. A separate DRA is created for every recovery agent defined. Please note, that on Windows XP not included into domain, there's no recovery agent is defined, so this step is omitted.

Now a temporary file Efs0.tmp is created in the same folder as the file being encrypted. The contents of original file (plain text) is copied into temporary file, after that the original is overwritten with encrypted data. By default, EFS uses DESX algorithm with 128-bit key to encrypt file data, but Windows could be also configured to use stronger 3DES algorithm with 168-bit key. In that case FIPS compliant algorithms usage must be turned on in LSA policy (it is disabled by default):

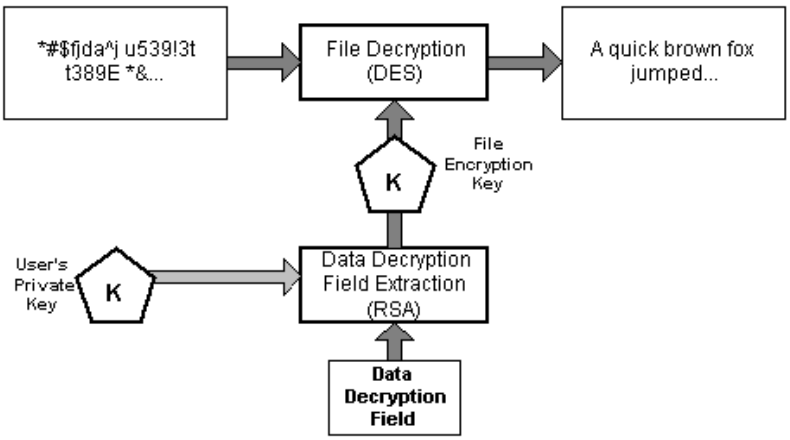



EFS uses the registry to determine if it will use DESX or 3DES. If HKLM\SYSTEM\CurrentControlSet\Control\LSA\FipsAlgorithmPolicy = 1, then 3DES will be used. If not, then EFS checks HKLM\Software\Microsoft\Windows NT\CurrentVersion\EFS\AlgorithmID (this value may not be present); if present, it will have ID CALG_3DES or CALG_DESX, otherwise, DESX should be used. After encryption is done, temporary and log files are deleted.



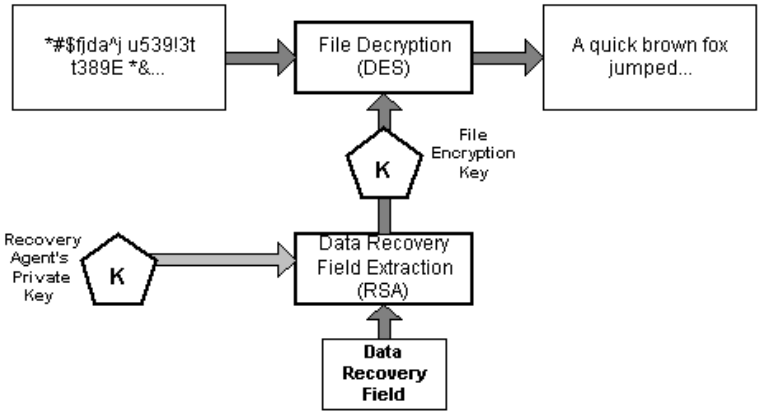
After file is encrypted, only users who has correspondent DDF or DRF can access the file. This mechanism is separate from common security meaning that beside rights to access file, the file must have its FEK encrypted with user's public key. Only user who can decrypt FEK with his own private key, can access the file. The consequence is, that user, who has access to the file, can encrypt it thus preventing the owner to access his own file. Initially only one DDF is created for user who encrypts the file, but later he can add extra users to key ring. In this case EFS simply decrypts FEK with private key of user who wants to give access to the file to another user, and encrypts FEK with public key of target user, thus creating a new DDF which is stored along with the first one.

The decryption process is opposite to encryption:




First, system checks if user has a private key used by EFS. If yes, it reads EFS attributes and walk through the DDF ring looking for DDF for current user. If DDF is found, user's private key is used to decrypt FEK extracted from DDF. Using decrypted FEK, EFS decrypts file data. It should be noticed that file never decrypted in whole but rather by sectors when upper level module requests particular sector.

Recovery process is similar to decryption, except that it uses the recovery agent's private key to decrypt the FEK in the DRF, not in DDF:



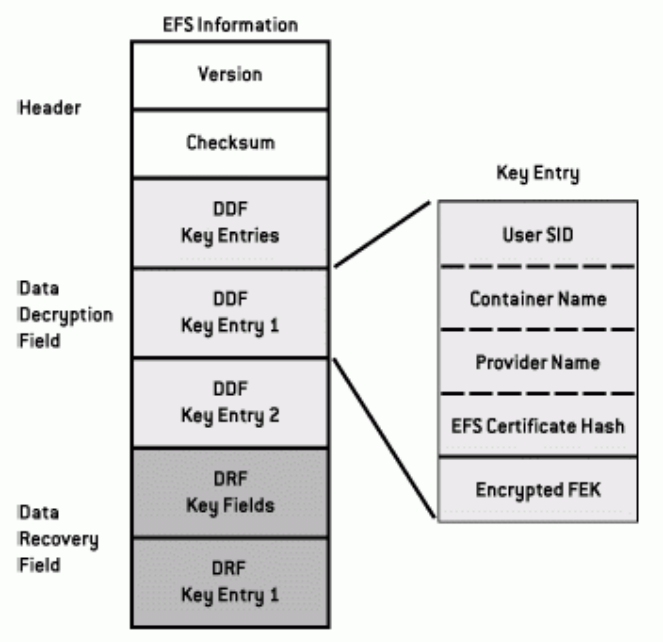

DRA policy is implemented differently for Windows 2000 and Windows XP. In Windows 2000 by default on computers, not included into domain, local Administrator is added to Public Key Policy as Encrypted Data Recovery Agent. So, when user encrypts file, both DDF and DRF fields are created. If the last DRA is deleted, the whole EFS functionality is turned off and it is not possible to encrypt file anymore.

In Windows XP the situation is different. Since majority of home users working standalone do not need anybody else to be able to decrypt file except themselves, there's no need in data recovery agents, so there's no DRA included into Public Key Policy and EFS works without DRA. In this case only DDF field is created for encrypted file.

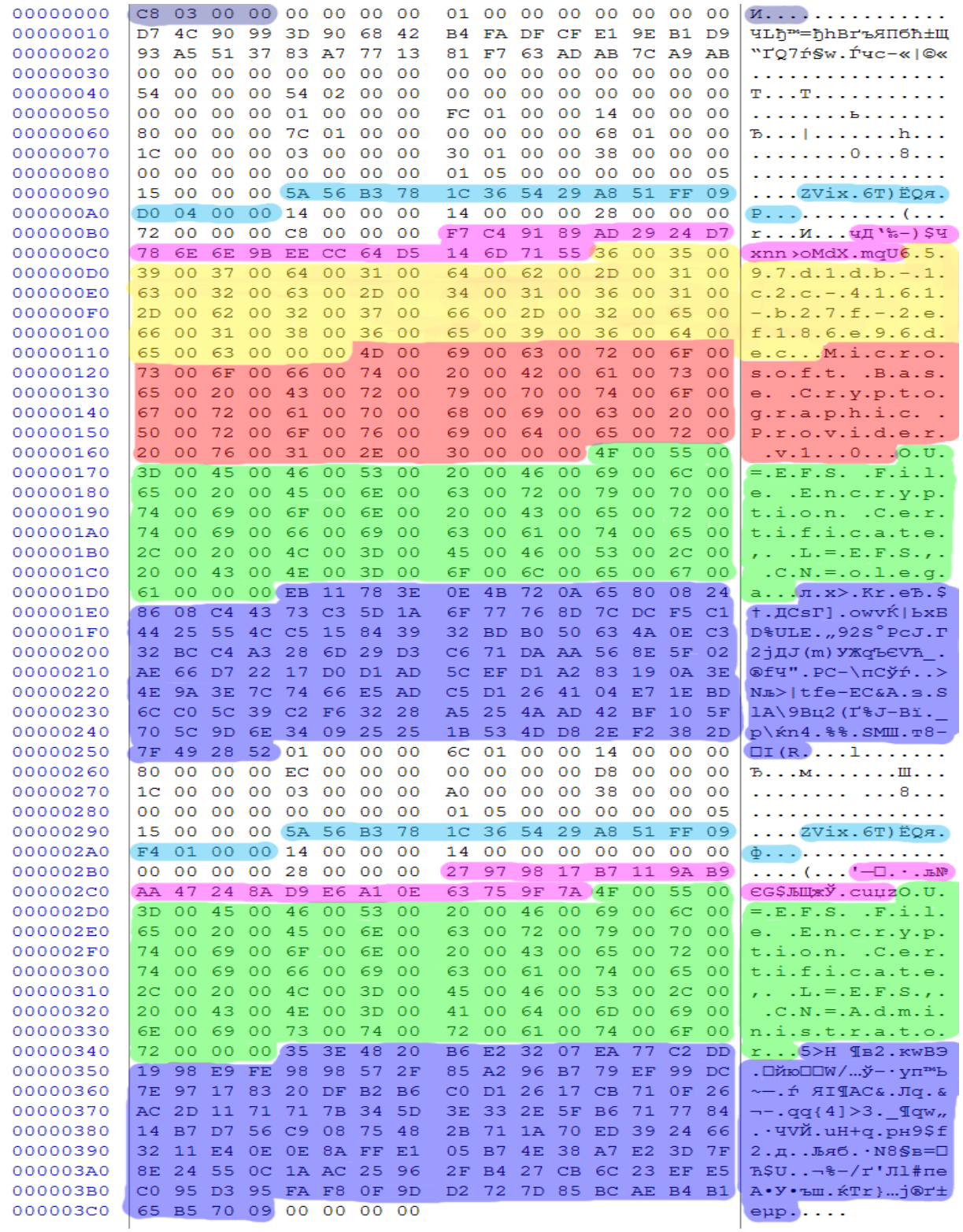
**EFS - Encrypting File System. Encrypted Files and Folders (NTFS5 only)**

**$EFS Attribute**

When NTFS encrypts file, it sets flag Encrypted (0x4000) for the file and creates $EFS attribute for the file where it stores DDFs and DDRs. This attribute has Attribute ID = 0x100 in NTFS and can be pretty lengthy, occupying from 0.5K to several kilobytes depending on number of DDFs and DRFs.




Here's an example of $EFS attribute with more details.



$EFS attribute size

Computer SID and user number. It specifies folder where EFS stores certificates. In order to get folder name EFS makes some manipulations:
5A56B378 1C365429 A851FF09 D040000 - data stored in $EFS,

78B3565A 2954361C 09FF15A8 000004D0 - reversed
2025018970-693384732-167712168-1232 - converte to decimal
S-1-5-21-2025018970-693384732-167712168-1232 - SID prefix added
So, the folder will be %User Profile%\Application Data\Microsoft\Crypto\RSA\S-1-5-21-2025018970-693384732-167712168-1232\

Public key thumbprint

Private key GUID (also used as container name). This name EFS uses when it aquires context from CryptoAPI provider. If there's only one DDFin $EFS attribute, container name can be figured out from $EFS (this field), but as more users added to the file (more DDFs or DRFs), PK GUID is not stored for all of them and must be retrieved from certificate storage based on public key thumbprint.

Cryptographic provider name = Microsoft Base Cryptographic Provider v.1.0

User name, to whom current DDF or DRF belongs

Encrypted FEK. Usually FEK is 128-bit long (in case of DESX) but since it's encrypted with 1024-bit RSA key, its encrypted length is 1024 bits.

**EFS - Encrypting File System. Encrypted Files and Folders (NTFS5 only)**

**Issues with EFS**

Temporary file is not erased. When EFS encrypts file, it copies its contents into temporary hidden file named Efs0.tmp in the same folder, as encrypting file. Then, it encrypts plain text by blocks and writes encrypted data into original file. After the process is done, temporary file is deleted. The problem is that EFS simply marks it as deleted without actually erasing its contents, which makes possible easy access to unprotected data by low-level data recovery software like Active@ Undelete. Solution - to wipe free disk space. Usually, even if plain text overwritten ones, small magnetic traces remain detectible, thus giving a chance to read erased data with proper equipment. To minimize this possibility, use commercially available software providing sophisticated data erasing algorithms like Active@ Eraser or ZDelete.NET.

File names in encrypted folder are not protected. Actually, encrypting folder contents means automatically applying encryption to all files in the folder, not encrypting directory data itself. Since the file name itself could contain sensitive information, it could be a breach in security. One of the solutions would be using encrypted *.zip* archives instead of folders, which are treated by Windows XP almost like folders. Thus, only one file is needed to be encrypted and archived data themselves are harder to crack.

EFS security relies on public/private key pair which is stored on local computer. Windows protects all private keys by encrypting them through ***Protected Storage*** **service**. Protected Storage encrypts all private keys with *Session Key*, derived from 512 bit *Master Key*, and stores them in %User Profile%\Application Data\Microsoft\Crypto\RSA\User SID. The Master Key is encrypted by *Master Key Encryption Key*, which is derived from user password by using a Password Based Key Derivation Function and stored in %User Profile%\Application Data\Microsoft\Protect\User SID. Despite the efforts Windows takes to protect keys, the fact, that all information is stored on local computer, gives an attacker, who's got an access to hard drive, a chance to figure out keys and use them to decrypt protected data. The overall security could be significantly enhanced by encrypting private keys with *System Key*. The **syskey.exe** utility can be used to store System Key on a floppy disk and remove it from computer. In this case user must insert a diskette with System Key when computer boots up. Nevertheless, this method should be taken with precautions since if key diskette is lost, there's no way to get access to computer.

Forgot Password? Windows XP/2003 password recovery ...

***NTFS Sparse Files (NTFS5 only):***

A sparse file has an attribute that causes the I/O subsystem to allocate only meaningful (nonzero) data. Nonzero data is allocated on disk, and non-meaningful data (large strings of data composed of zeros) is not. When a sparse file is read, allocated data is returned as it was stored; non-allocated data is returned, by default, as zeros.

NTFS deallocates sparse data streams and only maintains other data as allocated. When a program accesses a sparse file, the file system yields allocated data as actual data and deallocated data as zeros.

NTFS includes full sparse file support for both compressed and uncompressed files. NTFS handles read operations on sparse files by returning allocated data and sparse data. It is possible to read a sparse file as allocated data and a range of data without retrieving the entire data set, although NTFS returns the entire data set by default.

With the sparse file attribute set, the file system can deallocate data from anywhere in the file and, when an application calls, yield the zero data by range instead of storing and returning the actual data. File system application programming interfaces (APIs) allow for the file to be copied or backed as actual bits and sparse

stream ranges. The net result is efficient file system storage and access. Next figure shows how data is stored with and without the sparse file attribute set.

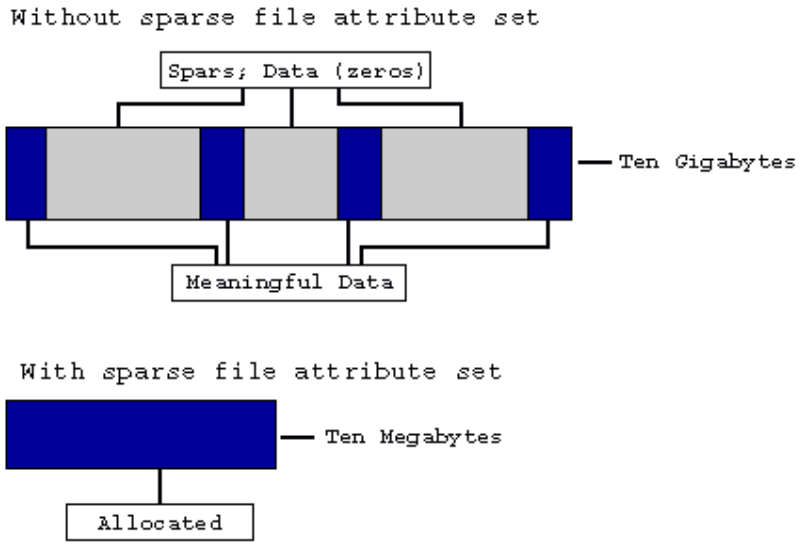



**Fig. 5-4:** Windows 2000 Data Storage

***Important:***

If you copy or move a sparse file to a FAT or a non-Windows 2000 NTFS volume, the file is built to its originally specified size. If the required space is not available, the operation does not complete.

***Data Integrity and Recoverability with NTFS:***

NTFS is a recoverable file system that guarantees the consistency of the volume by using standard transaction logging and recovery techniques. In the event of a disk failure, NTFS restores consistency by running a recovery procedure that accesses information stored in a log file. The NTFS recovery procedure is exact, guaranteeing that the volume is restored to a consistent state. Transaction logging requires a very small amount of overhead.

NTFS ensures the integrity of all NTFS volumes by automatically performing HDD recovery operations the first time a program accesses an NTFS volume after the computer is restarted following a failure.

NTFS also uses a technique called cluster remapping to minimize the effects of a bad sector on an NTFS volume.

***Important:***

If either the master boot record (MBR) or boot sector is corrupted, you might not be able to access data on the volume.

***Recovering Data with NTFS:***

NTFS views each I/O operation that modifies a system file on the NTFS volume as a transaction, and manages each one as an integral unit. Once started, the transaction is either completed or, in the event of a disk failure, rolled back (such as when the NTFS volume is returned to the state it was in before the transaction was initiated).

To ensure that a transaction can be completed or rolled back, NTFS records the suboperations of a transaction in a log file before they are written to the disk. When a complete transaction is recorded in the log file, NTFS performs the suboperations of the transaction on the volume cache. After NTFS updates the cache, it commits the transaction by recording in the log file that the entire transaction is complete.

Once a transaction is committed, NTFS ensures that the entire transaction appears on the volume, even if the disk fails. During recovery operations, NTFS redoes each committed transaction found in the log file. Then NTFS locates the transactions in the log file that were not committed at the time of the system failure and undoes each transaction suboperation recorded in the log file. Incomplete modifications to the volume are prohibited.

NTFS uses the Log File service to log all redo and undo information for a transaction. NTFS uses the redo information to repeat the transaction. The undo information enables NTFS to undo transactions that are not complete or that have an error.

***Important:***

NTFS uses transaction logging and recovery to guarantee that the volume structure is not corrupted. For this reason, all system files remain accessible after a system failure. However, user data can be lost because of a system failure or a bad sector.

***Conclusion:***

In the event of a bad-sector error, NTFS implements a recovery technique called cluster remapping. When Windows 2000 detects a bad-sector, NTFS dynamically remaps the cluster containing the bad sector and allocates a new cluster for the data. If the error occurred during a read, NTFS returns a read error to the calling program, and the data is lost. If the error occurs during a write, NTFS writes the data to the new cluster, and no data is lost.

NTFS puts the address of the cluster containing the bad sector in its bad cluster file so the bad sector is not reused.

Cluster remapping is not a backup alternative. Once errors are detected, the disk should be monitored closely and replaced if the defect list grows. This type of error is displayed in the Event Log.

**Resources**

Inside the Windows NT File System  Author: Helen custer 1994