

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

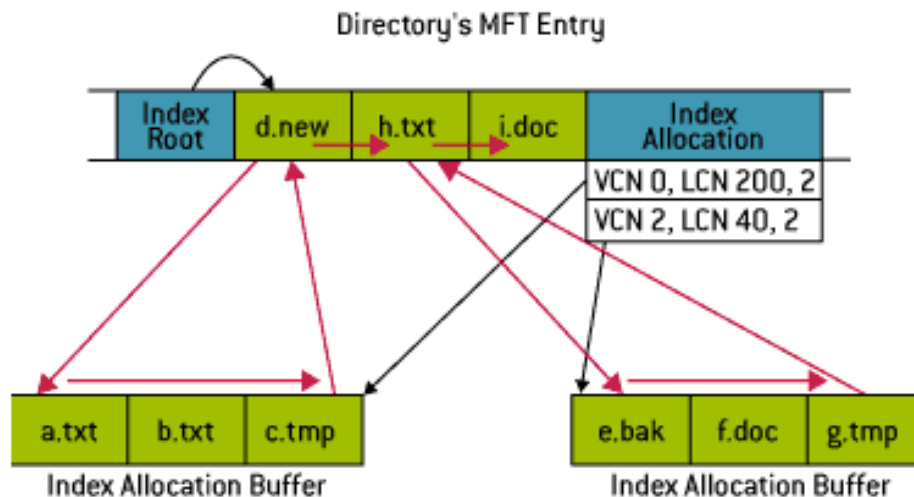
TFS, the native file-system format for Windows 2000, has continuously evolved since its release with Windows NT 3.1. Although NTFS's original features made it suitable as a high-end file-system format, the extensive and significant enhancements that Microsoft added for Win2K address enterprise-level requirements that Microsoft identified as more organizations adopted NT. One feature, consolidated security information, improves the efficiency of NTFS in everyday operation; others, such as quota management, must be leveraged by applications or enabled and managed by an administrator.

In this two-part series, I take you inside the NTFS features introduced by NTFS 5.0 (NTFS5), the version of NTFS included with Win2K. I don't describe administrative interfaces to the features or list programming APIs that let you access them; instead, I discuss how NTFS implements the features behaviorally and in its on-disk format. The features I cover in this column include consolidated security information, reparse points (including mount points and junctions), and quota management. In Part 2, I will conclude with a look at sparse-file support, the change journal, link tracking, and encryption.

Before you proceed, you should have a firm understanding of basic NTFS on-disk organization, including Master File Table (MFT) entries, attribute types, and resident and nonresident attributes. If you're not familiar with these concepts, you might want to read "Inside NTFS," January 1998, as a primer. For more background information, see "Related Articles in Previous Issues," page 46. The sidebar "Exploring NTFS On-disk Structures," page 46, describes several tools for viewing and sources of information about NTFS internal data structures such as those I describe in this column.

General Indexing

Several new NTFS5 features rely on a fundamental NTFS feature called attribute indexing. Attribute indexing consists of sorting entries of a particular attribute type, using an efficient storage mechanism for fast lookups. Pre-Win2K versions of NTFS support attribute indexing for only \$I30, the index attribute that stores directory entries. The attribute indexing process sorts directory entries by name and stores the entries as a B+ tree (a form of a binary tree that stores multiple items at each node in the tree). Figure 1, page 47, illustrates the MFT entry of a directory that contains nine entries stored in three nodes, each with three entries. The index root attribute contains the root of the B+ tree. Because the nine entries don't all fit in the directory's MFT entry, NTFS must store some of the entries elsewhere. Consequently, NTFS allocates two index allocation buffers to store two of the entries. (Index root and index allocation buffers typically can store entries for more than three files, depending on the length of the filenames.) As I explain in "Inside NTFS," an MFT entry is 1KB in size and index allocation buffers are 4KB in size.



Inside Win2K NTFS

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

The red arrows in the figure emphasize the way NTFS stores entries that are alphabetically sequential. If you ran a program that opened the file e.bak in the directory that the figure illustrates, NTFS would read the index root attribute, which contains entries for d.new, h.txt, and i.doc, and compare the string e.bak with the name in the first entry, d.new. NTFS would conclude that e.bak is alphabetically greater than d.new and would therefore proceed to the next entry, h.txt. After performing the same comparison, NTFS would find that e.bak is alphabetically less than h.txt. NTFS would then look in h.txt's directory entry for the virtual cluster number (VCN) of the index buffer that contains directory entries alphabetically less than h.txt (but greater than d.new). VCNs represent a cluster's order within a file or directory. NTFS uses mapping information to translate a VCN to a Logical Cluster Number (LCN), which is the number of a cluster relative to the start of a volume. If the directory entry for h.txt didn't store a VCN for an index buffer, NTFS would immediately know that the h.txt directory doesn't contain e.bak and would indicate that the lookup failed.

After obtaining the VCN of the starting cluster of the index buffer that NTFS will examine next, NTFS reads the index allocation buffer and scans it for a match. In Figure 1, the index buffer's first entry is the one NTFS is searching for, so NTFS reads the number of e.bak's MFT entry from e.bak's directory entry. Directory entries also store other information, such as the file's timestamps (e.g., created, last modified), size, and attributes. Although NTFS also stores this information in the file's MFT entry, duplicating the information in a directory entry saves NTFS the trouble of reading the file's MFT entry to obtain the information when listing directories and doing simple file queries.

Directory entries are sorted alphabetically, which explains why NTFS files are always printed alphabetically in directory listings. In contrast, the FAT file system doesn't sort directories, so FAT listings aren't sorted. Further, because NTFS stores entries as a B+ tree, lookups for particular files in a large directory are very efficient—typically, NTFS needs to scan only a fraction of a directory. This approach contrasts with FAT's linear lookups, which require FAT to potentially examine every entry in a directory while searching for a specific name.

Whereas pre-Win2K NTFS implements indexing only for filenames, NTFS5 implements general indexing, which lets NTFS5 store arbitrary data in indexes and sort the data entries by something other than a name. NTFS5 uses general indexing to manage security descriptors, quota information, reparse points, and file object identifiers—features that I explain in this series.

Consolidated Security

NTFS has always supported security, which lets an administrator specify which users can and can't access individual files and directories. In pre-Win2K NTFS, every file and directory stores its security descriptor in its own security attribute. In most cases, administrators apply the same security settings to an entire directory tree, which results in duplication of security descriptors across all the files and subdirectories to which the settings apply. This duplication can intensively utilize disk space in multiuser environments, such as Win2K Server Terminal Services and NT Server 4.0, Terminal Server Edition (WTS), in which security descriptors might contain entries for multiple accounts. NTFS5 optimizes disk utilization for security descriptors by using a central metadata file named \$Secure to store only one instance of each security descriptor on a volume.

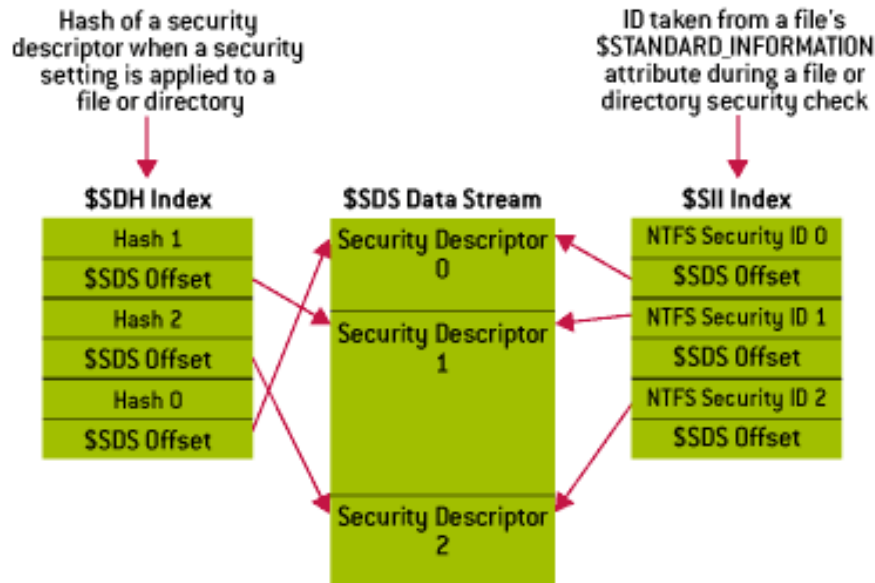
The \$Secure file contains two index attributes—\$SDH and \$SII—and a data-stream attribute named \$SDS, as Figure 2 shows. NTFS5 assigns every unique security descriptor on a volume an internal NTFS security ID (not to be confused with a SID, which uniquely identifies computers and user accounts) and hashes the security descriptor according to a simple hash algorithm. A hash is a potentially nonunique shorthand representation of a descriptor. Entries in the \$SDH index map the security descriptor hashes to the security descriptor's storage location within the \$SDS data attribute,

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

and the \$SII index entries map NTFS5 security IDs to the security descriptor's location in the \$SDS data attribute.



When you apply a security descriptor to a file or directory, NTFS obtains a hash of the descriptor and looks through the \$SDH index for a match. NTFS sorts the \$SDH index entries according to the hash of their corresponding security descriptor and stores the entries in a B+ tree. If NTFS finds a match for the descriptor in the \$SDH index, NTFS locates the offset of the entry's security descriptor from the entry's offset value and reads the security descriptor from the \$SDS attribute. If the hashes match but the security descriptors don't, NTFS looks for another matching entry in the \$SDH index. When NTFS finds a precise match, the file or directory to which you're applying the security descriptor can reference the existing security descriptor in the \$SDS attribute. NTFS makes the reference by reading the NTFS security identifier from the \$SDH entry and storing it in the file or directory's \$STANDARD_INFORMATION attribute. The NTFS \$STANDARD_INFORMATION attribute, which all files and directories have, stores basic information about a file, including its attributes and timestamp information. Win2K expands this attribute to accommodate additional data, such as the file's security identifier.

If NTFS doesn't find in the \$SDH index an entry that has a security descriptor that matches the descriptor you're applying, then the descriptor you're applying is unique to the volume and NTFS assigns the descriptor a new internal security ID. NTFS internal security IDs are 32-bit values, whereas SIDs are typically several times larger, so representing SIDs with NTFS security IDs saves space in the \$STANDARD_INFORMATION attribute. NTFS then adds the security descriptor to the \$SDS attribute, which is sorted in a B+ tree by NTFS security ID, and adds to the \$SDH and \$SII indexes entries that reference the descriptor's offset in the \$SDS data.

When an application attempts to open a file or directory, NTFS uses the \$SII index to look up the file or directory's security descriptor. NTFS reads the file or directory's internal security ID from the MFT entry's \$STANDARD_INFORMATION attribute, then uses the \$Secure file's \$SII index to locate the ID's entry in the SDS attribute. The offset into the \$SDS attribute lets NTFS read the security descriptor and complete the security check. NTFS5 doesn't delete entries in the \$Secure file, even if no file or directory on a volume references the entry. Not deleting these entries doesn't significantly decrease disk space because most volumes, even those used for long periods, have relatively few unique security descriptors.

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

NTFS5's use of general indexing lets files and directories that have the same security settings efficiently share security descriptors. The \$SII index lets NTFS quickly look up a security descriptor in the \$Secure file while performing security checks, and the \$SDH index lets NTFS quickly determine whether a security descriptor being applied to a file or directory is already stored in the \$Secure file and can be shared.

Reparse Points

Reparse points let an application associate a block of application data with a file or directory and let the Object Manager reparse, or reexecute a name lookup, when an application encounters a reparse point. (For information about the Object Manager's role in the OS's architecture, see "Inside NT's Object Manager," October 1997.) In addition to storing the reparse data, the reparse point stores a reparse code that identifies the reparse point as belonging to a particular application. Although not useful by themselves, reparse points let Win2K or third-party developers build functionality. Win2K provides several types of reparse-point functionality, including mount points, NTFS junctions, and Hierarchical Storage Management (HSM). I discuss the way each of these functionalities works, then delve into the implementation of reparse points.

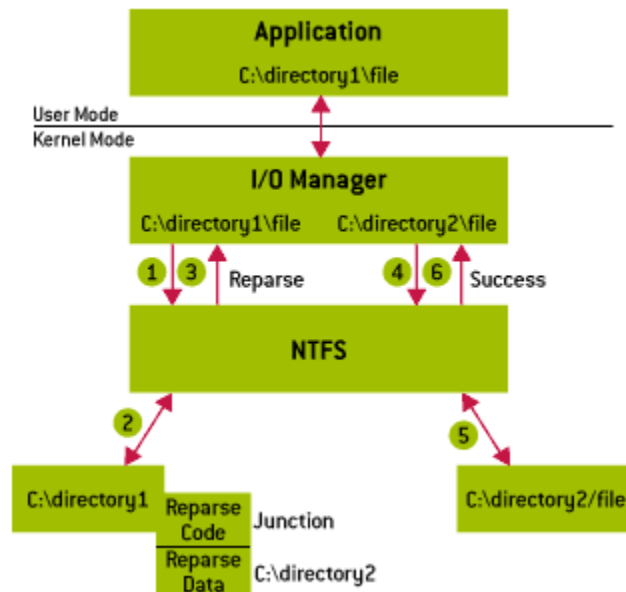
To be accessible, all NT volumes must have a drive letter. NTFS5 mount points let you connect a volume at a mount point directory on a parent NTFS5 volume without assigning a drive letter to the child volume. This capability lets you consolidate multiple volumes under one drive letter. For example, if you mount a volume that contains the directory \articles to a mount point named C:\documents, you can use the path C:\articles\documents to access the \documents directory's files. A mount point is a reparse point whose data consists of a volume's internal name. This internal name has the form \??\Volume{XX-XX-XX-XX}, where the Xs are the numbers that make up the globally unique ID (GUID) that Win2K assigned to the volume.

When you open the file C:\articles\documents\column.doc, NTFS encounters the mount point associated with the \documents directory. NTFS reads the mount point's reparse data (the volume name) and returns a reparse status to the Object Manager. The I/O manager intercepts the reparse status, examines the reparse data, and determines that NTFS encountered a mount point. The I/O manager modifies the name being looked up and directs the Object Manager (the kernel component that guides name lookups) to reissue the lookup using the modified path \??\Volume{GUID}\documents\column.doc. The reissued lookup causes name parsing for \documents\column.doc to continue on the mounted volume. To create and list mount points, you can use the Microsoft Management Console (MMC) Disk Management snap-in or the Mountvol command-line tool that comes with Win2K.

NTFS junctions are similar to mount points, and the I/O manager and Object Manager implement junctions as they implement mount points. However, junctions reference directories rather than volumes. Junctions are the Win2K equivalent of UNIX symbol links (although unlike UNIX symbolic links, junctions can't be applied to files). If you create the junction C:\articles\documents that references D:\documents, you can access files stored in D:\documents by using the path C:\articles\documents. The junction's reparse point stores the redirected path information, and as for mount-point traversal, the I/O manager modifies the name and reissues the name lookup when NTFS encounters a junction. Figure 3 illustrates how junctions work. When the application opens C:\directory1\file, NTFS encounters a reparse point on C:\directory1 that points at C:\directory2. The I/O manager changes the name to C:\directory2\file, and the application ultimately opens C:\directory2\file.

Inside Win2K NTFS

Mark Russinovich
(Reprinted from WindowsItPro Magazine)



Win2K doesn't include tools for creating junctions, and Microsoft doesn't officially support such tools because some applications might not function properly when they use paths that contain junctions. However, you can use either the Linkd tool from the Microsoft Windows 2000 Resource Kit or the free Junction tool (<http://www.sysinternals.com/misc.htm>) to create and list junctions.

Not all reparse points rely on path reparing functionality. The HSM system uses HSM reparse points to transparently migrate infrequently accessed files to offline storage. When HSM moves a file offline, the HSM system deletes the file's contents and creates a reparse point in the file's place. The reparse point data contains information the HSM system uses to locate the file's data on archival media. When an application later accesses an offline HSM file, the HSM driver RsFilter.sys (Remote Storage Filter) intercepts the reparse code that NTFS returns to the Object Manager. The driver deletes the reparse point, fetches the file data from archival storage, then reissues the original request. This time, NTFS accesses the file as it would any other, and the application doesn't realize that data shuffling occurred. When a file or directory has a reparse point associated with it, NTFS creates an attribute named `$Reparse` for the reparse point. This attribute stores the reparse code and data. So that NTFS can easily locate all reparse points on a volume, a metadata file named `\$Extend\$Reparse` stores entries that connect the reparse point file and directory MFT entry numbers to their associated reparse point codes. NTFS sorts the entries by MFT entry number in the `$R` index.

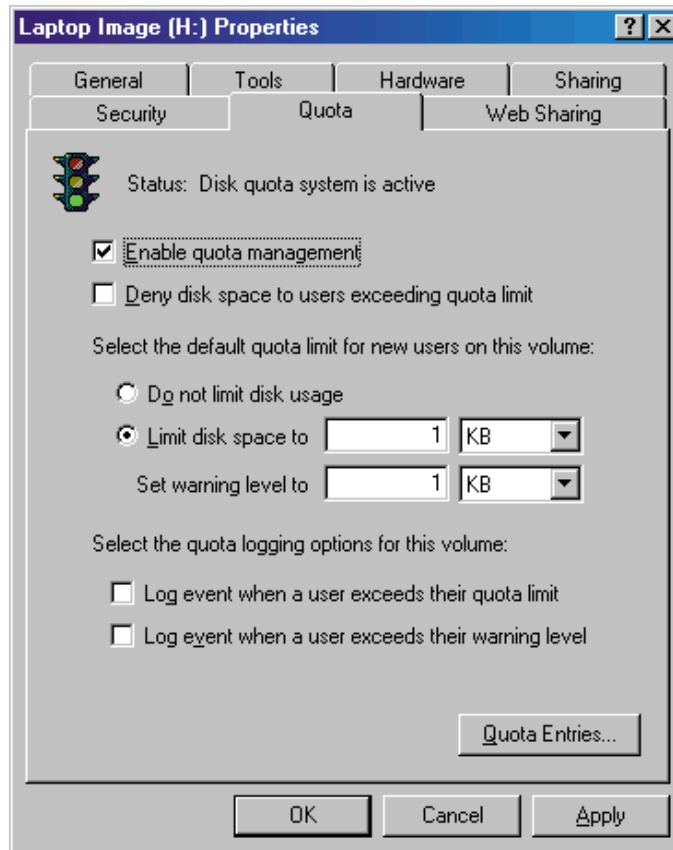
Quota Tracking

One of the most popular roles that Win2K Server and NT Server systems play is that of a file server. A deficiency of NT is that systems administrators have no built-in tools to monitor or control the amount of server disk space that individual users consume. Several third-party quota-management products are available to fill this gap in NT's functionality. However, Win2K provides basic quota management that in many cases alleviates the need for third-party tools. Win2K quota management is based on NTFS quota support (FAT volumes don't support quota management) and is a per-volume, per-user model. In other words, an administrator can use an interface like the one that Figure 4 shows to specify warning and limit thresholds on the amount of disk space a user can consume on an NTFS volume.

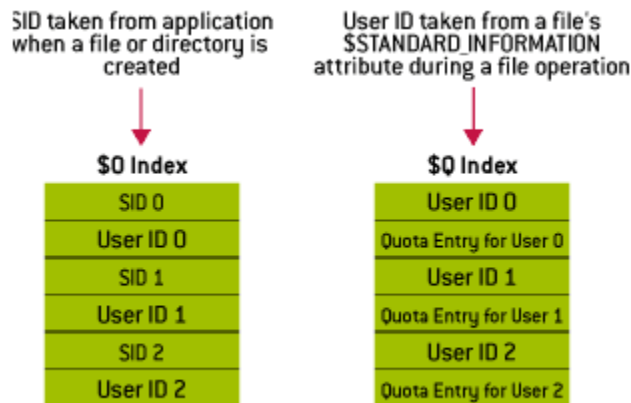
Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



NTFS stores quota information in the `\$Extend\$Quota` metadata file, which consists of the indexes `$O` and `$Q`. Figure 5 shows the organization of these indexes. Just as NTFS assigns each security descriptor a unique internal security ID, NTFS assigns each user a unique user ID. When an administrator defines quota information for a user, NTFS allocates a user ID that corresponds to the user's SID. In the `$O` index, NTFS creates an entry that maps a SID to a user ID and sorts the index by user ID; in the `$Q` index, NTFS creates a quota control entry. A quota control entry contains the value of the user's quota limits, as well as the amount of disk space the user consumes on the volume.



When an application creates a file or directory, NTFS obtains the application user's SID and looks up the associated user ID in the `$O` index. NTFS records the user ID in the new file or directory's `$STANDARD_INFORMATION` attribute, which counts all disk space allocated to the file or directory

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

against that user's quota. Then, NTFS looks up the quota entry in the \$Q index and determines whether the new allocation causes the user to exceed his or her warning or limit threshold. When a new allocation causes the user to exceed a threshold, NTFS takes appropriate steps, such as logging an event to the System event log or not letting the user create the file or directory. As a file or directory changes size, NTFS updates the quota control entry associated with the user ID stored in the \$STANDARD_INFORMATION attribute. NTFS uses general indexing to efficiently correlate user IDs with account SIDs, and, given a user ID, to efficiently look up a user's quota control information.

Yet More Features

This discussion of quota tracking brings us to the end of this issue's column. In my next column, I conclude this series about new NTFS features by describing the implementation of distributed link tracking support, sparse-file support, volume change tracking, and encryption.

In "Inside Win2K NTFS, Part 1," November 2000, I began this two-part series by describing general indexing, consolidated security, reparse points, and quota management, all of which are features new to NTFS 5.0 (NTFS5), the Windows 2000 version of NTFS. In this issue, I look at how NTFS5 implements Distributed Link Tracking (DLT), sparse-file support, volume change tracking, and encryption. I conclude with a look at alternate data streams, an NTFS feature rarely used before NTFS5.

Distributed Link Tracking

Windows Explorer supports a type of symbolic link called a shell link or shell shortcut. Shell links, which often appear on the Windows desktop and in the Start menu, let you easily access programs and files without having to navigate to their original location. Another type of Windows link is an OLE link. OLE links are links in one application's files that store data belonging to another application. For example, if you embed a Microsoft Excel spreadsheet in a Microsoft Word document, an OLE link in the Word document refers to the Excel document.

If you've ever moved a link source (the executable program that a shell or OLE link refers to) in a pre-Win2K version of Windows and then clicked the link that referenced the source, you've witnessed the heuristic-based search that Windows Explorer performs in its attempt to find the source's new location. If the search is unsuccessful, Windows Explorer gives up and requires you to tell it where you moved the source. In Win2K, DLT automatically updates shell links to point at moved link sources. The only requirement is that the link source's original and final locations are both on NTFS5 volumes and in the same domain.

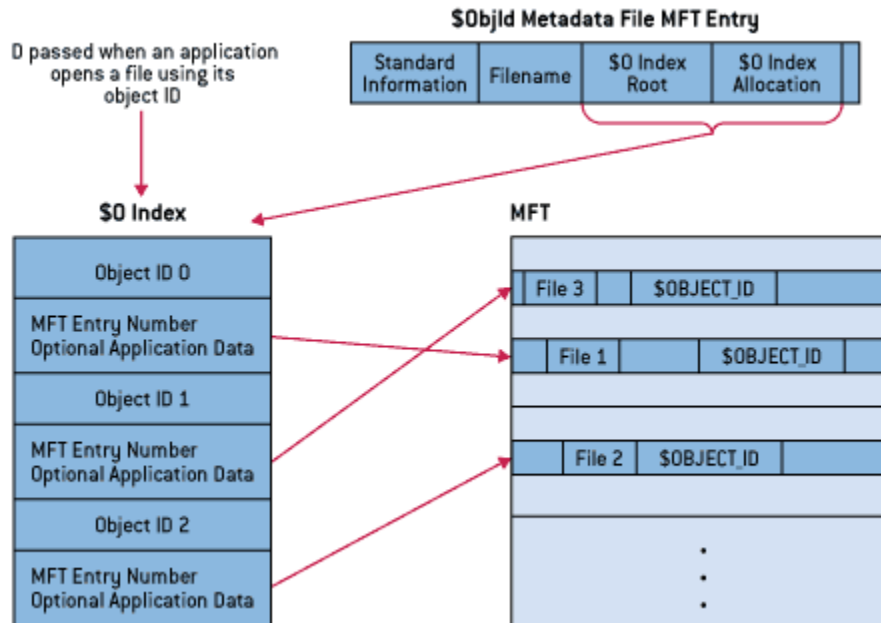
NTFS5 is required for automatic link updating because NTFS5 lets an application assign files and directories a 16-byte (128-bit) object ID. Not coincidentally, 16 bytes is the length of a Windows globally unique ID (GUID). Windows uses GUIDs as a general-purpose identification mechanism because a GUID's length and the algorithm that Windows uses to generate a GUID virtually guarantee that every GUID is statistically unique. The DLT service, which Win2K implements as a Win32 service, assigns a GUID to every link source that resides on an NTFS volume and directs NTFS to record the GUID as the source's object ID. When you click on a shell link or open a document with an embedded OLE link and Windows Explorer fails to find the link source at the path the link specifies, Windows Explorer queries the DLT service for the source's new location. DLT uses the object ID that Windows Explorer supplied to attempt to open the link source on each volume in the domain, starting with local volumes. When DLT finds the file (or directory), DLT asks NTFS for the name of the file and returns the result to Windows Explorer, which updates the link and follows it to the tracked location. Thus, even if the source moves to a volume on another computer in the domain, the link continues to work.

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

When the DLT service (or another application) assigns an object ID to a file or directory, NTFS creates an attribute of type \$OBJECT_ID in the file's Master File Table (MFT) entry, as Figure 1, page 46, shows. (For information about the MFT and its entries and attributes, see "Inside NTFS," January 1998.) This 16-byte attribute stores the GUID. At the same time, NTFS adds an entry to the \Extend\$ObjId metadata file that maps the GUID to the file's MFT entry. Win2K stores the metadata file's entries in an index named \$O, which NTFS sorts by object ID in a B+ tree. When DLT finds a moved link and attempts to use the link's object ID to open it, NTFS can look up the link's MFT entry number in the \$ObjId file's \$O index. After DLT uses the file's MFT entry number to open the file, DLT asks NTFS for the file's name, updates the link, and opens the link source. The \$ObjId file's use is an example of general indexing, a new NTFS feature I describe in "Inside Win2K NTFS, Part 1."



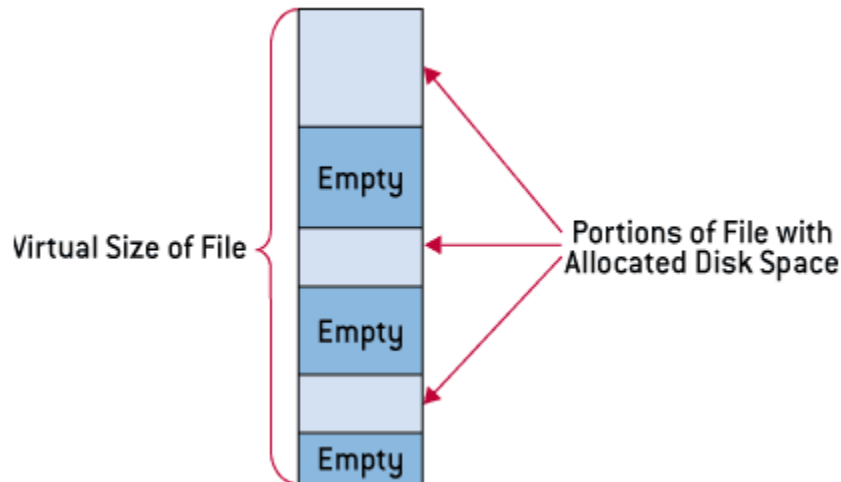
Sparse Files

Many applications consist of a server that logs data to a file and a client that reads the data that the server writes. This architecture usually requires the use of a technique called circular logging, in which the log file is a fixed size and the server rolls over, or returns to the beginning of the file, after reaching the file's end. A rollover can lead to overwriting data before the client has a chance to read it. Other applications, such as databases, allocate extremely large files, of which valid data fills only small portions, resulting in disk space that is allocated but unused. NTFS5 includes a feature called sparse-file support that Win2K applies to both these situations to minimize unnecessary disk utilization and avoid the rollover problems inherent in circular logging.

Sparse-file support lets an application designate unused portions of a file as empty, freeing the disk space allocated to the empty regions. In a logging application, the server appends log information to the file without needing to roll over, and the client marks file areas empty as it reads them. A log file might therefore appear to be very large but use only a small amount of disk space. When you view a file's properties, Windows Explorer shows both sizes and calls the amount of space the file takes on disk the Disk size. If an application reads from part of a sparse file that is designated as empty, the application receives zero-filled data. In the case of a database, the database application marks the unused parts of the database as empty, releasing the disk space. Figure 2 illustrates a sparse file.

Inside Win2K NTFS

Mark Russinovich
(Reprinted from WindowsItPro Magazine)



The NTFS implementation of compressed files has always had a type of sparse-file support. NTFS's compression algorithm compresses files in blocks of 16 virtual clusters, typically allocating fewer than 16 logical clusters on disk to store that data. A virtual cluster is a cluster within a file, and a logical cluster is a cluster on a volume. For an uncompressed file, virtual clusters correspond to logical clusters one-to-one, but in a compressed file, multiple virtual clusters might map to fewer logical clusters. When an application reads from a compressed file, NTFS decompresses the logical clusters that make up the 16-cluster block that the application is reading, recreating the 16 uncompressed virtual clusters in memory. If a 16-cluster block is filled with zeros, NTFS optimizes disk utilization by not allocating any logical clusters for the block. When an application reads from such a region, which is called a sparse region, NTFS returns zero-filled data.

NTFS5's sparse-file support relies on the same driver subroutines that NTFS uses for sparse regions of compressed files. However, two differences exist between sparse files and compressed files. One difference is that the nonempty portions of a sparse file aren't compressed. The second difference is that an application can indicate to NTFS that a region of a sparse file has become empty, freeing the logical clusters that were previously allocated to it. To make a sparse region of a compressed file, an application must write zero-filled data to the file.

You might think that users could use sparse files to exceed their disk quota by allocating large empty files and then filling them over time. However, NTFS counts a sparse file's virtual size, not the on-disk size, against a user's quota limit.

Volume Change Tracking

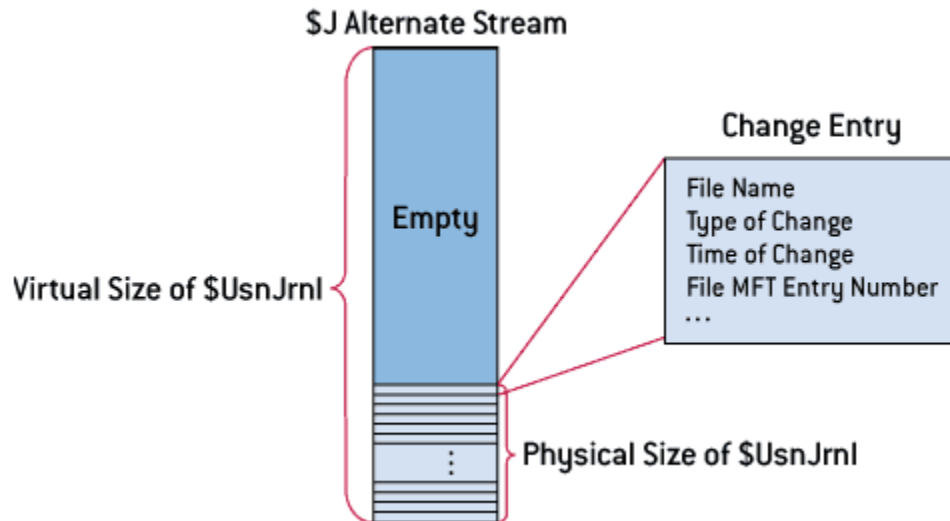
Before Win2K, an application that needed to monitor a volume for changes had limited options. The Win32 API exports functions that notify an application when the contents of a directory or directory tree change, but the application must either scan directories to determine the nature of changes as they occur or take the risk that the list of changes that Windows returns will overflow the application's buffer. Scanning directories after every change causes an unacceptable performance hit, yet many applications, such as incremental backup solutions, can't tolerate missed changes. NTFS5's new volume change tracking facility lets applications easily monitor changes to a volume and provides an effective alternative to earlier approaches for file- replication, incremental-backup, and virus-scanning applications.

Interestingly, volume change tracking relies on sparse-file support. NTFS disables volume change tracking by default, so applications that need to use it must enable it. When an application turns on

Inside Win2K NTFS

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

volume change tracking, NTFS creates a change journal, which is a sparse file named `\$Extend\$UsnJrnl`. Figure 3 depicts the change journal. Every change to a volume, such as creating, resizing, or deleting a file, logs an entry to the `$J` alternate data stream (I describe alternate data streams in more detail later) in the `$UsnJrnl` file. A change entry records the name of the file that changed, the type and time of the change, and several other pieces of useful information. Applications can request that NTFS notify them when it adds new entries to the journal and can read entries as the system logs them. The Win2K File Replication Service (FRS), which Win2K uses to replicate Dfs shares and to propagate group policies, is one of the change journal's clients.



NTFS monitors the on-disk space that the change journal consumes and limits this space to the amount the application specifies when it enables change tracking. When a journal exceeds the specified size, NTFS marks the change journal's oldest valid entries as empty. For even a moderately sized change journal, only an extremely high volume of file activity would cause an application to fall so far behind that it couldn't read entries before NTFS deletes them.

Encryption

Support for Encrypting File System (EFS) is an important part of NTFS5. Although I cover that support extensively in the series "Inside Encrypting File System" (June and July 1999), I provide a summary here. (For background information about EFS, see "Related Articles in Previous Issues." For an administrator's view of EFS management, see Mark Minasi, Inside Out, "Decrypting EFS," page 139.)

EFS isn't built in to NTFS but is an add-on driver (`\winnt\system32\driversefs.sys`). EFS provides transparent file-based encryption so that users can protect sensitive data that might fall into unauthorized hands. Although NTFS security prevents unauthorized access to a file while a Win2K system is online, a malicious user could bypass this security by booting a computer to another installation or by using NTFSDOS (available from [www.sysinternals.com/ntfs30 .htm](http://www.sysinternals.com/ntfs30.htm)) from a DOS boot floppy disk.

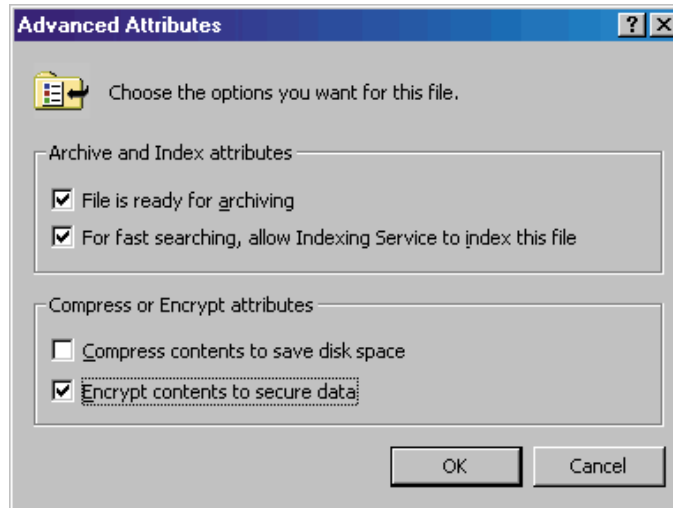
When someone uses the Advanced Attributes panel of a file's properties dialog (which Figure 4 shows) to initially encrypt a file, the EFS service, which runs in the Local Security Authority Subsystem (LSASS—`\winnt\system32\lsass .exe`), uses Crypto API services to assign the user a private/public key pair that EFS can use for file encryption. EFS randomly generates a file encryption key (FEK) for an encrypted file and uses the FEK and the Data Encryption Standard X (DESX) encryption algorithm to encrypt the file's data. Because DESX is a symmetric algorithm, decryption

Inside Win2K NTFS

Mark Russinovich

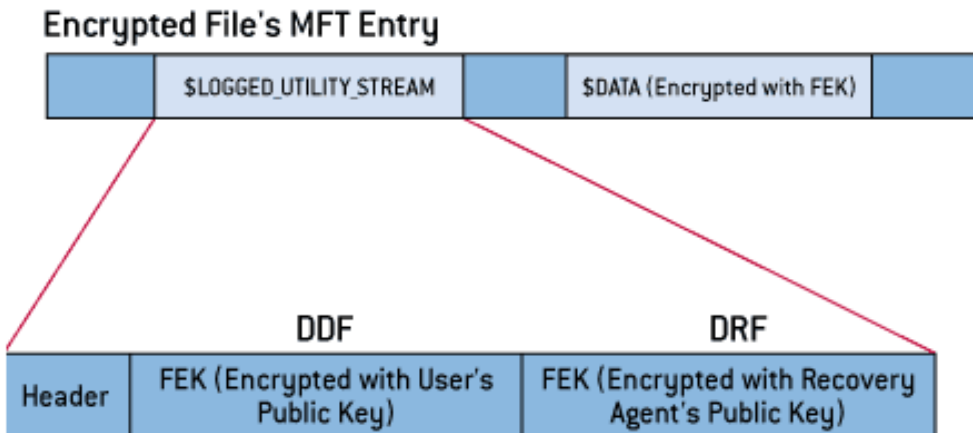
(Reprinted from WindowsItPro Magazine)

also uses the FEK, so EFS must protect the FEK from unauthorized access. To protect the FEK, EFS uses the RSA asymmetric encryption algorithm to encrypt the FEK with the user's EFS public key and stores the encrypted FEK with the file. When a user opens and reads an encrypted file, EFS uses the user's EFS private key to decrypt the FEK, then uses the decrypted FEK to decrypt the file data.



Because EFS uses the user's password to encrypt the user's EFS private key, an intruder would have to obtain the user's password to compromise encrypted file data. Further, although the initial release of Win2K stores encrypted EFS private keys on disk within a user's profile directory, subsequent releases will let users store their keys on smart cards.

EFS stores encrypted FEKs in a file's \$LOGGED_UTILITY_STREAM attribute (which is new to NTFS5), as Figure 5, page 50, shows. The stored data consists of a Data Decryption Field (DDF) and a Data Recovery Field (DRF). The DDF contains a copy of the FEK encrypted with the user's EFS public key, and the DRF contains a copy of the FEK encrypted with the system's Recovery Agent EFS public key. On a standalone system, the local administrator is the default system Recovery Agent; on a domain-based system, the default Recovery Agent is the domain administrator. Because EFS includes in every file an FEK encrypted with a Recovery Agent's EFS public key, an authorized user can decrypt the file and recover its contents if the user is unable to do so.



When the system boots, NTFS reads the Registry value:

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

```
HKEY_LOCAL_MACHINE\ SYSTEM\CurrentControlSet\Control\FileSystem\NtfsEncryptionService
```

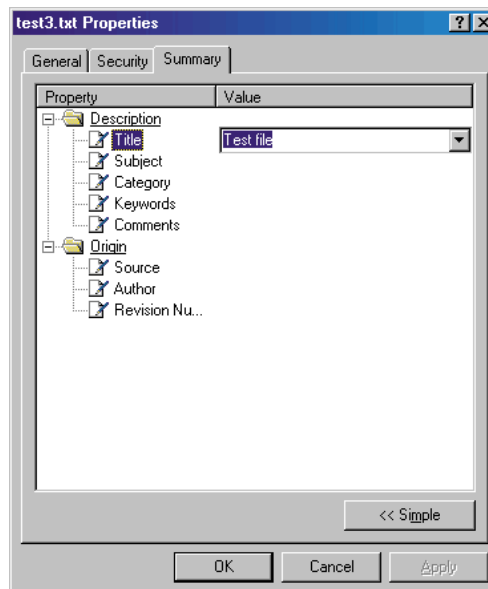
to obtain the EFS driver's name, then starts the driver. The EFS driver registers callbacks (i.e., routines that NTFS directly invokes) so that NTFS can hand off encrypted-file-related operations to the EFS driver. Internally, NTFS handles encrypted files much as it handles compressed files. NTFS decompresses a compressed file's data into the Win2K file-system cache and recompresses the data when it's written to the on-disk file. Similarly, NTFS decrypts an encrypted file's data into the cache and re-encrypts the data when writing it to disk.

Alternate Data Streams

An alternate data stream is a way to embed files within other files. Technically, every NTFS file contains an embedded file that has no name. This file, which is called the default data stream or unnamed data stream, is the file you see when you use Notepad to open a file and is the file that applications see when they use a file's standard name to open the file. Applications also can add embedded files, which are called alternate data streams, and give them different names. Applications use the Win32 API syntax File:Alternate Stream (in which AlternateStream is the name of an alternate stream) to access data within alternate streams.

Microsoft originally included alternate-stream functionality in NTFS to support Apple Macintosh (Mac) file system resource forks. Many Mac files use the Mac's Hierarchical File System (HFS) to store icon and other information in an alternate data stream. Because Windows NT Server comes with Services for Macintosh (SFM—a service that lets NT share files with Mac clients), NT must support alternate streams so that Mac clients can store files with resource forks on NT servers without losing the resource fork information. This relatively obscure role of alternate data streams has meant that few NT users have used alternate data streams. In Win2K, alternate data streams have a more prominent role.

The Summary tab, which Figure 6 shows, that the Win2K version of Windows Explorer displays when you edit the properties of an NTFS file lets you associate with a file arbitrary textual information such as a title, keywords, and revision number. Windows Explorer stores that information as an alternate data stream named ?SummaryInformation (the question mark represents an unprintable character).



Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Because alternate data streams are the exception, most applications and console commands aren't aware of alternate streams. For example, Windows Explorer and the Dir command show the size of only the file's unnamed data stream.

You can use the Echo and More commands, which are aware of alternate streams, to experiment with streams. First, create a file with an alternate stream:

```
echo hello > file.txt:alternatestream
```

Get a directory listing of the file to verify that Windows reports the file's size as zero, then use the More command to display the data in the alternate stream:

```
more < file.txt:alternatestream
```

(You can't view Windows Explorer summary information streams in this manner because the name of the summary information stream begins with a non-ASCII character.)

If you're curious about whether your files and directories contain alternate data streams, you might want to use the free Streams utility from www.sysinternals.com/misc.htm. This utility takes a file or directory name as a command-line parameter and reports the names of all alternate data streams that the file or directory contains. Streams accepts the wildcard character *, and you can use the /s switch to cause Streams to examine subdirectories, so you can easily see all the files and directories that contain alternate streams on a volume or in a directory.

NTFS Metadata Files

With the addition of new features, NTFS has acquired additional metadata files to store data related to those features. As I've discussed the new NTFS5 features, I've described the metadata file that each feature uses. Table 1 summarizes all the NTFS5 metadata files.

TABLE 1: NTFS5 Metadata Files	
Metadata File	Purpose
\	Root directory
\$AttrDef	Attribute definition list
\$BadClus	Bad cluster file
\$Bitmap	Volume cluster-allocation map
\$Boot	Boot sector
\$Extend	Optional metadata file directory
\$Extend\ObjId	Object ID file
\$Extend\Quota	Quota file
\$Extend\Reparse	Reparse point file
\$Extend\UsnJrnl	Change journal
\$LogFile	Volume Transaction log
\$MFT	MFT

Inside Win2K NTFS

Mark Russinovich
(Reprinted from WindowsItPro Magazine)

\$MFTMirr	Mirror of part of MFT
\$Secure	Security settings
\$UpCase	Uppercase mapping file
\$Volume	Volume name, volume dirty flag

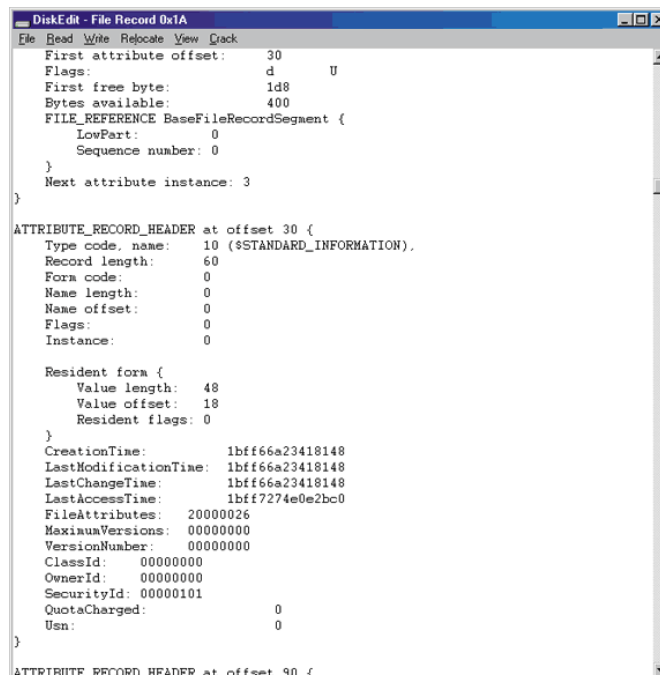
In "Inside NTFS," January 1998, I present a utility named NtfsInfo that shows you the size of the various NTFS metadata files. NtfsInfo relies on an undocumented feature of NTFS that lets applications explicitly specify the name of a metadata file to obtain a directory listing of the file. However, NTFS5 doesn't allow access to metadata files, so NtfsInfo doesn't work on Win2K. Instead, you can use NFI, the NTFS File Sector Information Utility that I introduced in "Inside Win2K NTFS, Part 1," to list information about all the files on a volume, including metadata files.

More to Come

NTFS5's powerful enhancements, including support for large files, encryption, and change tracking, will help extend Win2K's reach into the enterprise. However, you can be sure we haven't seen the end of NTFS's evolution.

Exploring NTFS On-disk Structures

Few publicly available tools let you explore the internals of the NTFS on-disk structure. One of those few is DiskEdit, an internal NTFS testing tool that Microsoft inadvertently shipped on the Windows NT 4.0 Service Pack 4 (SP4) CD-ROM and the most powerful NTFS viewer I've seen. DiskEdit, which Figure A shows, provides a detailed look at the structures that make up files and directories, translates file and directory paths to Master File Table (MFT) entry numbers, and lets you look at attribute data. No documentation accompanies DiskEdit, but I provide a tutorial for it in a back issue of my free Sysinternals Newsletter (<http://www.sysinternals.com/newsletter.htm>). To use DiskEdit on Windows 2000, you must copy the ifsutil.dll, ulib.dll, untfds.dll, and ufat.dll files from an NT 4.0 system's \winnt\system32 directory to the directory from which you want to run DiskEdit.



```
File Record 0x1A
File Edit Write Relocate View Crack
First attribute offset: 30
Flags: d U
First free byte: 1d8
Bytes available: 400
FILE_REFERENCE BaseFileRecordSegment {
  LowPart: 0
  Sequence number: 0
}
Next attribute instance: 3
}

ATTRIBUTE_RECORD_HEADER at offset 30 {
  Type code, name: 10 ($STANDARD_INFORMATION),
  Record length: 60
  Form code: 0
  Name length: 0
  Name offset: 0
  Flags: 0
  Instance: 0

  Resident form {
    Value length: 48
    Value offset: 18
    Resident flags: 0
  }
}
CreationTime: 1bff66a23418148
LastModificationTime: 1bff66a23418148
LastChangeTime: 1bff66a23418148
LastAccessTime: 1bff7274e0e2bc0
FileAttributes: 20000026
MaximumVersions: 00000000
VersionNumber: 00000000
ClassId: 00000000
OwnerId: 00000000
SecurityId: 00000101
QuotaCharged: 0
Usn: 0
}
ATTRIBUTE_RECORD_HEADER at offset 90 {
```

Inside Win2K NTFS

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

If you're interested in NTFS data structure information but can't get your hands on an SP4 CD-ROM, you can use NFI, the NTFS File Sector Information Utility, which Microsoft includes with its OEM Support Tools package at <http://support.microsoft.com/support/kb/articles/q253/0/66.asp>. (In addition to NFI, the OEM Support Tools include debugging utilities and kernel debugger extensions to help support personnel and developers analyze crash dumps.)

NFI accepts several command-line forms that let you dump information about a particular file or directory or about all the files on a volume, obtain the file or directory in which a particular logical sector on a drive resides, or obtain the file or directory in which a particular physical disk sector resides. For example, the command

```
nfi C: 123
```

reports the name of the file on volume C: that contains the volume's 123rd sector. To examine NTFS data structures, you can use the same command but omit the sector number to tell NFI to dump detailed information about attributes in all the disk's files. You can specify the name of the metadata files I describe in the column to view evidence of the index attributes they contain. For example, executing the command

```
nfi c:\$extend\$quota
```

on a volume that has quota management enabled results in output that shows that the \$Quota file contains indexes named \$O and \$Q:

File 24

```
\$Extend\$Quota
$STANDARD_INFORMATION (resident)
$FILE_NAME (resident)
$INDEX_ROOT $O (resident)
$INDEX_ROOT $Q (resident)
```

For another source of NTFS on-disk structure information, see Gary Nebbett, *Windows NT/2000 Native API Reference* (New Riders Publishing, 2000). An appendix lists the definitions for many NTFS on-disk data structures (some of which don't reflect changes in Win2K) and contains the source code for a Win32 program that interprets the data structures to bypass the NTFS file-system driver and dump a volume's contents.