September 1999

MICROSOFT SYSTEMS JOURNAL

Keeping an Eye on Your NTFS Drives: the Windows 2000 Change Journal Explained

Jeffrey Cooperstein and Jeffrey Richter

The Windows 2000 Change Journal is a database that contains a list of every change made to the files or directories on an NTFS 5.0 volume. Each volume has its own Change Journal database that contains records reflecting the changes occurring to that volume's files and directories.

This article assumes you're familiar with Windows NT, Platform SDK

Code for this article: ChangeJournal.exe (4KB)

Jeffrey Cooperstein is an author, trainer, and Windows programming consultant. He can be reached at http://www.cooperstein.com. Jeffrey Richter wrote Advanced Windows, Third Edition *(Microsoft Press, 1997) and* Windows 95: A Developer's Guide *(M&T Books, 1995). Jeff is a consultant and teaches Win32 programming courses (http://www.solsem.com). He can be reached at http://www.JeffreyRichter.com.*

Windows 2000 is packed with new and exciting technologies, and the Change Journal is one of them. The Change Journal is going to open up a whole new world of features in future Windows-based applications, and it will provide the opportunity for dramatic performance improvements in many of today's applications. Everything from enterprise-class applications to your personal virus scanner will make use of the Change Journal.

We will explain the technology, its implementation, and introduce the API used to access the Change Journal. Our sample application will get you started with examining the features of the Change Journal. In a future article, we'll cover all the subtleties of programming the Change Journal and provide a full-fledged Change Journal sample that can be used as a template for your own application.

In simple terms, the Change Journal is a database that contains a list of every change made to the files or

contains a list of every change made to the files or directories on an NTFS 5.0 volume. When any file or directory is written to, NTFS guarantees that a record will be added to the Change Journal. Each volume has its own Change Journal database that contains records reflecting the changes occurring to that volume's files and directories. If you have more than one NTFS volume, each one will have its own journal. Of course, FAT volumes do not maintain a Change Journal.

The Change Journal is fairly easy to use. The Change Journal will be used most often by services, but there is nothing to prevent normal applications from reading it. It is accessed though documented functions, making it available to any application running on Windows 2000.

Any number of applications or services can have simultaneous access to this information. A backup service can read the journal to find out what files need to be backed up. At the same time, a security program might be watching to make sure no one tampers with the files in the system directory. On Windows NT 4.0, these tasks were accomplished with functions like FindFirstChangeNotification and ReadDirectoryChangesW. Anyone who has attempted to use these functions knows how limited they can be. The Change Journal provides a new level of detailed information for applications that need to monitor changes on an NTFS volume.

The Change Journal can also reduce the need for applications to walk the entire hard drive (full rescans). Many utilities rely on full rescans to occasionally gather up-to-date information. Now an application can do a full rescan just once and then rely on the Change Journal to tell it exactly what files or directories have changed and when.

Implementation Details

The Change Journal is actually a special file on an NTFS volume. The system hides this file so that you cannot view it using familiar tools like the Explorer and the CMD shell. Whenever the file system makes a change to a file or directory, it appends a record to the journal. The record identifies the file name, the time of the change, and what type of change occurred. The actual data that changed is not kept in the journal, so don't get your hopes up about being able to roll back changes—this keeps the size of the journal as small as possible.

The Change Journal is initially an empty file on the disk volume. As changes occur to the volume, records are appended to the end of this file. Each record is assigned a 64-bit identifier called an Update Sequence Number (USN). When Microsoft was first developing the Change Journal, it was internally called the USN Journal. That's why the structures and defines in the winioctl.h header file refer to the Change Journal as the USN Journal. When a record is added to the journal, it is assigned a USN. USNs are generated in increasing order, so that you can compare USNs to find out the order of events (lower USNs are older events). USNs are not contiguous, so it's possible that the first USN record might be 0 and the second USN record might be 128.

The Change Journal always writes new records to the end of the file, so the implementors chose to use the file offset of a record as its USN. This makes querying the journal fast since the system can simply seek the desired record using the USN. Since records include a file name they vary in length, so you'll notice varying distances between USNs of adjacent records. A typical record might be 100 bytes long. For performance, the system writes to the journal in 4KB blocks that contain groups of 30 or 40 records (as defined by USN_PAGE_SIZE in winioctl.h). The system will not allow a single record to span the boundaries of a page, so you'll sometimes see a gap in USNs where empty space was used to pad the end of a block.

On an NTFS volume, file and directory information is stored in the Master File Table (MFT). Each record in the MFT describes a file or directory's name, location, size, attributes, and more. With NTFS 5.0, each file's MFT entry records the Last USN generated for that file. This is also true for directories. As records are appended to the Change Journal, the file system updates the MFT's Last USN value for the changed file or directory. In our next article, we'll show how this information is useful with a technique that can quickly scan the MFT for all files that changed over a range of time.

If the journal file gets too big (as defined by the MaximumSize parameter), the system will purge the oldest records at the start of the file. Traditionally, truncating data at the start of a file requires lots of file I/O. The end of the file must be copied to a new location, which is a time-consuming task. Fortunately, NTFS 5.0 supports sparse files, a mechanism that allows unneeded portions of a file to be deleted while retaining the logical offsets of the remaining data. The Change Journal is a sparse file, allowing the purging of records without any performance penalty. In addition, remaining records can still be quickly located using the USN since they remain at the same logical offset. For more information on sparse files see the article "A File System for the 21st Century: Previewing the Windows NT 5.0 File System" by Jeffrey Richter and Luis Felipe Cabrera in the November 1998 issue of *MSJ*.

A Change Journal can be disabled on a given volume, preventing the system from logging file and directory changes. By default, an NTFS volume will have its Change Journal disabled. Some application must explicitly activate the journal. Also note that any application can activate or disable the volume's journal at any time. An application must be able to gracefully handle the situation when a journal is disabled while the first application is still using the journal. We'll describe how applications can handle this in a future article. When an application disables the Change Journal for a volume, the system will also purge any existing records to prevent recovery of the information. This prevents applications from inadvertently reading unreliable records. The journal will only contain records as long as the journal is continuously active.

In the current implementation of the Change Journal, the journal file on disk is actually deleted when the Change Journal is disabled. A new journal file is created the next time an application activates the Change

the next time an application activates the Change Journal. Although applications should not care about this implementation detail, it is why the terms "creating" and "deleting" the journal are used in the Platform SDK. We prefer to think of a Change Journal as being active or disabled since it describes the Change Journal as a service provided by the system. Terms such as "create" and "delete" are useful when trying to understand the implementation of the Change Journal as a file on disk. We've found that thinking about the Change Journal as active or disabled helps in understanding how it is used by applications.

Change Journals are assigned a unique 64-bit Journal ID (not to be confused with a USN number). The system will change a journal's ID when there is a chance that file or directory changes were not logged in the journal. For example, if a volume's journal is disabled, then activated, the Journal ID will be changed. As long as the Journal ID does not change, applications can be assured that the Change Journal has recorded every file and directory change. Even if the system is rebooted, the Journal ID will typically not need to change. In other words, if the Journal ID does not change after a reboot, the system has recorded all file and directory changes throughout the shutdown and boot sequence. Observant developers may discover that Journal IDs are actually standard 64-bit UTC time stamps generated from the system time. Applications should not derive any meaning from this (and remember, Microsoft may change how Journal IDs are generated before Windows 2000 ships).

Windows NT 4.0 Service Pack 4 provides limited access to NTFS 5.0 volumes. Unfortunately, the Change Journal cannot be accessed and changes to the volume will not be recorded. On dual boot systems, a boot to Windows NT 4.0 will cause all Journal IDs to be changed when Windows 2000 is restarted. Again, this allows applications running on Windows 2000 to know that they may have missed some file or directory changes.

Usage

All features of the Change Journal are accessed via the DeviceIoControl function.

```
BOOL DeviceIoControl(
  HANDLE hDevice,          // handle to device/file/
                           // directory
  DWORD dwIoControlCode,   // control code of operation
                           // to perform
  LPVOID lpInBuffer,       // pointer to buffer of
                           // input data
  DWORD nInBufferSize,     // size, in bytes, of input
                           // buffer
  LPVOID lpOutBuffer,      // pointer to buffer for
                           // output data
  DWORD nOutBufferSize,    // size, in bytes, of output
                           // buffer
  LPDWORD lpBytesReturned, // receives number of bytes
                           // written to lpOutBuffer
  LPOVERLAPPED lpOverlapped// for asynchronous
                           // operation
);
```

The first parameter is a handle to a file, directory, or device obtained by calling CreateFile. DeviceIoControl is

device obtained by calling CreateFile. DeviceIoControl is a common method used to pass device-specific requests to the driver managing hDevice. The parameter dwIoControlCode specifies what operation to perform and defines the structure of input/output buffers. If CreateFile is called with FILE_FLAG_OVERLAPPED, DeviceIoControl will operate asynchronously in the same way as ReadFile/WriteFile.

The NTFS driver manages the Change Journal. To communicate with a volume about its Change Journal, call DeviceIoControl with a handle to the volume. Call CreateFile as shown to get a volume's handle:

```
// Get a handle to access the Change Journal on the
// 'C' volume
HANDLE hcj = CreateFile("\\\\.\\C:", GENERIC_READ,
   FILE_SHARE_READ | FILE_SHARE_WRITE,
   NULL, OPEN_EXISTING, 0, NULL);
```

Access to this volume handle is restricted to the system and members of the Administrators group, so typical users will not be able to run Change Journal applications. This means that these applications will most likely be services or utilities run by administrators.

The control codes that are supported for Change Journals are documented in the Platform SDK. They can be located in the index, but are not listed directly in the documentation for DeviceIoControl. The best way to locate the documentation is to search for "Change Journal."

Change Journal Statistics

An application can query a volume for Change Journal statistics by calling DeviceIoControl and passing the FSCTL_QUERY_USN_JOURNAL code. If DeviceIoControl returns TRUE, the USN_JOURNAL_DATA structure shown in **Figure 1** is filled in. If DeviceIoControl returns FALSE, GetLastError may return one of the codes shown in **Figure 2**.

Change Journal Records

Let's take a closer look at the information stored in a single Change Journal record.

Applications deal with records using the USN_RECORD structure. This is not the on-disk structure of a record, but it contains all of the information that is available from a single record.

```
// Version 2.0 USN_RECORD structure
typedef struct {
    DWORD         RecordLength;
    WORD          MajorVersion;
    WORD          MinorVersion;
    DWORDLONG     FileReferenceNumber;
    DWORDLONG     ParentFileReferenceNumber;
    USN Usn;
    LARGE_INTEGER TimeStamp;
    DWORD         Reason;
    DWORD         SourceInfo;
    DWORD         SecurityId;
    DWORD         FileAttributes;
    WORD          FileNameLength;
    WORD          FileNameOffset;
    WCHAR         FileName[1];
} USN_RECORD, *PUSN_RECORD;
```

Applications will never have to fill in this structure. Instead, the system populates an output buffer with USN_ RECORDs when an application reads from the journal.

RecordLength is the total length of the record, in bytes, including the file name. Multiple records will be provided in an output buffer, so RecordLength should be used to calculate the location of the next record.

```
PUSN_RECORD pNext;
pNext = (PUSN_RECORD) (((PBYTE) pRecord) +
    pRecord->RecordLength);
```

Major Version and MinorVersion

It is easy to ignore the importance of version checking, but even easier to make a careless error that will infuriate users. Anyone who installed software on Windows NT 4.0 and received the message "Requires Windows NT 3.5 or greater" will testify to the disasters caused by the misuse of the GetVersion function! GetVersionEx was added to help clarify the versioning mess for developers, but even that was not enough. Windows 2000 has added VerifyVersionInfo to provide an even safer method for what should be a simple procedure.

For the sake of this article, we don't really care about what version of Windows is running, but the Change Journal has its own version control. There aren't any fancy functions to help you out, so it's all the more important that you take the time to understand this information. (We only mention VerifyVersionInfo as a public service announcement. If you want more information, see the current Platform SDK documentation.)

The initial release of Windows 2000 is expected to use version 2.0 Change Journal records (MajorVersion is 2, MinorVersion is 0). As we are writing this article, the Platform SDK contains only the version 2.0 definition of the USN_RECORD structure (defined in winioctl.h). Your application is responsible for knowing the version of the structure that was used at compile time. Winioctl.h does not currently provide any defined constants that have this information, so the best bet is to look in this header file for comments. For safety, it is a good idea to create your own compile-time constants and perform a runtime check to verify that newer structure definitions were not inadvertently included.

```
#include <winioctl.h>
#define CJ_MAJOR_VERSION_EXPECTED       2
#define CJ_MINOR_VERSION_EXPECTED       0
#define CJ_SIZEOF_USN_RECORD_EXPECTED  64
void RunTimeSanityCheck() {
   if (sizeof(USN_RECORD) !=
       CJ_SIZEOF_USN_RECORD_EXPECTED) {
      // YIKES! Someone probably updated winioctl.h
      // or changed the default structure packing.
      // Any code placed here will run if we are
      // compiling with a different size USN_RECORD
      // than when we wrote this module. We'd better
      // take a look at it!
   }
}
```

At runtime, applications examine the MajorVersion and MinorVersion of journal records to determine compatibility with the information. If a change in MajorVersion is detected, the USN_RECORD structure has changed dramatically and the only members you can still use are Record- Length, MajorVersion, and MinorVersion. Unfortunately, the system does not provide any ability to negotiate a compatible version at runtime. In other words, if the system fills an output buffer with records using a different MajorVersion than expected, the information cannot be used at all! Change Journal records with a MajorVersion of 1 existed on earlier betas of Windows 2000, but they are no longer supported.

If a change in the MinorVersion is detected, new members have been added after the penultimate member of the older structure. Applications can assume that USN_ RECORD structure members are valid up to the penultimate member of the older version. For example, consider the hypothetical version 2.3 USN_RECORD structure shown in **Figure 3**. If an application is compiled with today's version 2.0 USN_RECORD, it can still examine a memory buffer filled with the hypothetical version 2.3 USN_RECORD. It can reference all the members up to and including the FileNameOffset member. (We'll discuss the proper way to access FileName later.) On the other hand, imagine an application is compiled using version 2.3 USN_RECORD. If an output buffer has version 2.1 records, the version 2.3 USN_RECORD structure can still be used for members up to and including the ExtraInfo1 member (the penultimate member of version 2.1).

Even though the record version information is provided in every record, an application only has to check it once each time it is started. The version number will not vary between volumes on the same physical machine, and will only change during a system reboot after a service pack is installed with new Change Journal software.

Does this sound like a lot of work? Maybe, but consider the consequences if you incorrectly read a buffer provided by the system. Since most likely your software will be running as a service, an access violation will bring down the service! Fortunately, only version 2.0 structures currently exist.

FileNameLength, FileNameOffset, and FileName

A journal record describes a change to a particular file or directory on the volume. For convenience, "full path of a record" refers to the full path of the file or directory whose change is described by the record. The full path of a record is not stored in the record itself. To save space, the file or directory's name is stored without path information. Three members of USN_RECORD provide access to this name (see **Figure 4**).

Here's the proper method to copy the name from a USN_RECORD to another buffer so that you'll have a zero-terminated string to work with:

```
WCHAR szName[MAX_PATH];
CopyMemory(szName,
    ((PBYTE) pRecord) + pRecord->FileNameOffset,
     pRecord->FileNameLength);
// Let's zero-terminate it
szName[pRecord->FileNameLength/sizeof(WCHAR)] = 0;
```

FileReferenceNumber and ParentFileReferenceNumber

The file or directory name is pretty useless without knowing what directory it was found in. ParentFileReferenceNumber specifies this directory. A File Reference Number (FRN) is a 64-bit ID that uniquely identifies any file or directory on an NTFS volume. Here's what we want to do to find the full path of the record (assuming szName already contains the file or directory name of the record):

```
TCHAR szFullPath[MAX_PATH];
// Fill in the path of the parent directory
PathFromParentFRN(pRecord->ParentFileReferenceNumber,
                  szFullPath);

// Append name to path using the Win32 function PathAppend
PathAppend(szFullPath, szName);
```

Unfortunately, the function PathFromParentFRN does not exist. In fact, there is no currently exposed API that directly converts a FRN to a full path. A large portion of our next article will be devoted to doing just this.

You may now be wondering about the FileReferenceNumber member. If we could convert this FRN to a full path, it would be the full path of the record we are trying to find (and we would never need to discuss FileNameOffset, FileNameLength, or ParentFileReferenceNumber). It turns out that finding the full path from a directory FRN is much easier than finding the full path from a file FRN. The FileReferenceNumber may be either a file or directory FRN (depending on whether the record describes a change to a file or directory), but the ParentFileReferenceNumber will always be a directory FRN. Because of this, the easiest way to find the full path of a record is to examine the ParentFileReferenceNumber and append the name using the FileNameOffset and FileNameLength members.

Usn, TimeStamp, and Reason

As you might guess, the Usn member tells you the USN of the record. TimeStamp is a standard UTC time stamp of this record, in 64-bit format. The Reason member tells you what sorts of changes have occurred to the file or directory. **Figure 5** shows the types of changes (reason codes) that generate entries in the Change Journal. The Reason member may have one or more of the reason codes set. To interpret this member, let's go over how the system decides to write a record to the journal.

The system keeps track of a Reason variable for every open file. When the system first opens a file, it sets the Reason variable to zero. No record is added to the

journal when a file is opened, even if it is opened with write access. If a change actually does occur, the system checks whether the reason code is already marked in the Reason variable. If this is a new reason code, the code bit is set in the Reason variable and a record is added to the Change Journal (the Reason variable is copied to the Reason member of the record). It is possible for more than one application to be modifying a file or directory, and the Reason variable will accumulate the reason codes for all changes to the file. The Reason variable continues to accumulate the list of change reason bits until all handles to the file are closed. At that point, a final record is added to the Change Journal with the accumulated reason codes and the USN_REA-SON_CLOSE code. **Figure 6** illustrates this process.

It is possible to tell the system to clear the Reason variable of an open file using the FSCTL_WRITE_USN_CLOSE_RECORD code. The DeviceIoControl function is called using the handle of an open file (not the volume handle as with other journal functions), and a close record will be generated for that file immediately.

```
DWORD cb;
USN usn;
// Force a close record for
// the open file specified
// by 'hFile'
DeviceIoControl(hFile, FSCTL_WRITE_USN_CLOSE_RECORD,
NULL, 0, &usn, sizeof(usn), &cb, NULL);
```

There is no input buffer, and the output buffer will be filled with sizeof(USN) bytes of data representing the USN of the generated close record. When this is done, the system immediately writes a record to the journal with the accumulated reason codes and the USN_REASON_CLOSE code, but it does not actually close the file. The Reason variable is reset to zero, and it will start accumulating changes all over again. If the Reason variable is zero when FSCTL_WRITE_USN_CLOSE_RECORD is used, it will still generate a journal record; this means you will see a record with only the USN_REASON_CLOSE code.

The only reason code that does not follow the previous rules is the USN_REASON_RENAME_OLD_NAME code. When a file is renamed, two records are added to the journal. First, the USN_REASON_RENAME_OLD_NAME code is added to the Reason variable, and a record is created. The members FileNameOffset and FileNameLength will specify the original name, and ParentFileReferenceNumber will specify the original directory. (Moving a file or directory to another location on the same volume is considered a rename.)

Next, the USN_REASON_RENAME_OLD_NAME flag is removed from the Reason variable and replaced with USN_REASON_RENAME_NEW_NAME. A second record is generated with the new file name and new ParentFileReferenceNumber. Up through the next close record for the file or directory, the Reason member will continue to have the USN_REASON_RENAME_NEW_NAME code, but not the USN_REASON_RENAME_OLD_NAME code. The FileReferenceNumber of a file or directory will not

change if it is renamed or moved to another location on the same volume.

Suppose you rename and move the file D:\dir1\before.txt to D:\dir2\after.txt with the command:

```
move D:\dir1\before.txt D:\dir2\after.txt
```

You'll see the following three records in the journal:

FileNameOffset/ Length	Parent FRN points to	Reason
before.txt	D:\dir1	Rename Old Name
after.txt	D:\dir2	Rename New Name
after.txt	D:\dir2	Rename New Name | Close

What happens if you rename a directory that has hundreds of files and subdirectories? Say you rename D:\Program Files to D:\Pfiles. The system will only generate the following three records:

FileNameOffset/ Length	Parent FRN points to	Reason
Program Files	D:\	Rename Old Name
Pfiles	D:\	Rename New Name
Pfiles	D:\	Rename New Name | Close

There is no need to create records for all the child files or directories since this information can be inferred by following the ParentFileReferenceNumber. For just this reason, you'll find that maintaining a database of files and directories is easier if entries are stored as a name and parent ID. The main drawback occurs when you try to monitor a specific file; you need to monitor all of its parent directories up to the root directory on the volume or you might miss a move or rename.

When a directory is deleted, you do not have to worry about inferring what child files or directories are affected. The system will not allow a directory to be deleted if it has any children. If you delete a whole tree in Explorer, you'll see delete records for all the children before the delete record for any directory.

It is important to understand that the Change Journal does not provide a superset of the Change Notification functionality provided through functions like FindFirstChangeNotification or ReadDirectoryChangesW. The Change Journal is designed to report all explicit actions on files or directories. Not all side effects are reported in the journal. For example, if an application calls the SetFileTime function, the Change Journal will report a Basic Information Change. However, if an application writes to a file, the Change Journal reports only the Data Overwrite (the explicit action), but not the time stamp change (the side effect). In a similar scenario, when a file or directory is created, the change to the parent directory's time stamp is not reported in the Change Journal. The Change Notification APIs, on the other hand, are designed to report all changes that they are aware of, even if it is the side effect of some other action.

SourceInfo, SecurityId, and

FileAttributes

If the SourceInfo member is not zero, it will specify a reason that the file is changing (as opposed to the Reason member, which indicates the reason, or type of change, that caused a record to be generated). The distinction between Reason and SourceInfo is subtle. Consider the statement, "The virus checker removed a macro virus from your document." The virus checker probably opened the file and then overwrote the infected portion. This generates a record with the USN_REASON_DATA_OVERWRITE code. The record exists because of a Data Overwrite (Reason), but this was done to Remove a Virus (SourceInfo). An application can use this information to decide what to do about a file or directory change. If the virus program is trusted to leave the document contents intact, the change can probably be ignored.

This information does not come from the system. It is provided by the application that opened the file. (The discussion of FSCTL_MARK_HANDLE in our next article will explain how an application provides this information). Currently, there are only three flags supported (see **Figure 7**).

SecurityId is an identifier that the system uses to identify the security descriptor of a file. It is used along with the FSCTL_SECURITY_ID_CHECK device I/O control code.

FileAttributes is the same value that would be returned by calling GetFileAttributes for the file or directory. It is useful to have this information because you can easily determine if a USN_RECORD refers to a file or directory by looking for the FILE_ATTRIBUTE_DIRECTORY flag.

Reading from the Change Journal

We're finally ready to read records from the journal. First, we need two things: the volume handle and a valid USN_JOURNAL_DATA structure retrieved by using the FSCTL_QUERY_USN_JOURNAL code. Let's say these are in the following variables:

```
HANDLE hcj;
USN_JOURNAL_DATA ujd;
```

To read some records, we call DeviceIoControl with the FSCTL_READ_USN_JOURNAL code. The input buffer must point to the following structure:

```
typedef struct {
    USN StartUsn;
    DWORD ReasonMask;
    DWORD ReturnOnlyOnClose;
    DWORDLONG Timeout;
    DWORDLONG BytesToWaitFor;
    DWORDLONG UsnJournalID;
} READ_USN_JOURNAL_DATA, *PREAD_USN_JOURNAL_DATA;
```

Set StartUsn to the USN of the first record you want to read. It must be either zero, the USN of an existing record in the journal, or the USN of the next record

that will be written to the journal. If StartUsn is zero, the system will start reading from the first record available. If StartUsn is the USN of an existing record, the system will start reading at that location. If it's the USN of the next record that will be written (such as ujd.NextUsn), the system waits for more data to appear in the journal, as specified by the Timeout/BytesToWaitFor members we'll describe later.

Since there is no way to know if the record identified by StartUsn will match the filter criteria (see our discussion of ReasonMask/ReturnOnlyOnClose), the output buffer may not contain that specific record. Applications must examine the Usn member of returned USN_ RECORD structures to find out the USNs of the records actually returned.

Since the system writes to the journal in 4KB blocks (USN_PAGE_SIZE), all 4KB aligned values from ujd.FirstUsn to ujd.NextUsn are guaranteed to be the USN of a record in the journal. Therefore, these are valid values for StartUsn. Other than that, the only way to get a valid value for StartUsn is through Change Journal APIs that return USNs.

ReasonMask and ReturnOnlyOnClose

The system will only return journal records that have at least one of the reason codes specified by ReasonMask. In other words, you can filter the amount of information you need to process by specifying only the reason codes you care about. Records that do not contain the specified codes are not returned in the output buffer.

The system uses the following logic to determine whether to return a record:

```
// This function will return
// TRUE if it meets the filter
// criteria specified by the
// ReasonMask member of the
// READ_USN_JOURNAL_DATA structure
BOOL ReturnRecord(PREAD_USN_JOURNAL_DATA prujd,
   PUSN_RECORD precord) {

   if ((prujd->ReasonMask & precord->Reason) != 0)
      return TRUE; // The user wants this record
   return FALSE; // Skip it
}
```

ReturnOnlyOnClose is another member that allows you to filter which records will be put in the output buffer. If this value is nonzero, only records with the USN_REASON_ CLOSE code will be returned. This works in tandem with the ReasonMask member (both conditions must be satisfied). To retrieve just the close records, set ReasonMask to reason codes of interest, and ReturnOnlyOnClose to 1. The system will return just close records, and only close records with one or more of the reason codes specified by ReasonMask. The ReturnRecord function really looks like **Figure 8**.

Timeout and BytesToWaitFor

Timeout is a value for use with the BytesToWaitFor member. It does not guarantee that DeviceIoControl will return after the specified timeout, but rather it

return after the specified timeout, but rather it specifies how often the system should check whether requested data is available. This member is not like other conventional Win32¨ timeout parameters that use milliseconds. Instead, this member uses the same resolution as the Win32 FILETIME structure (100-nanosecond intervals—one second has ten million intervals). A value of zero specifies no timeout (or infinite). A fixed timeout is specified using negative values (even though this is an unsigned variable). For example, a timeout of 25 seconds can be expressed as (DWORDLONG)(-2500000000). The Timeout member is ignored if DeviceIoControl is called with an asynchronous request.

Don't confuse the BytesToWaitFor member with the output buffer size or the count of bytes returned by DeviceIoControl. If this member is set to zero, the function will return immediately, even if it found no matching records in the journal. If this member is nonzero, the system will not return until it has found at least one record to return. BytesToWaitFor specifies how often the system will recheck the journal to see whether any matching records have been created. For example, if you specify 16384, the system will only examine the journal for new records after a new 16KB block of raw data has been added. This prevents a process from using too many resources when many records are being added. If the Timeout and BytesToWaitFor members are both nonzero, the system also checks records if the timeout period expires before the journal has grown by the specified number of bytes.

If BytesToWaitFor is nonzero, but records are found that match the user's request, the DeviceIoControl function will return immediately; that is, the BytesToWaitFor and TimeOut members only have an effect when there are not any existing records that fulfill the ReasonMask/ReturnOnlyOnClose requirements.

UsnJournalID

The UsnJournalID should be set to ujd.UsnJournalID. If the Journal ID of the active journal has been changed by the system, the call to DeviceIoControl will fail. This protects applications from reading journal records if there is a chance that some data is missing.

The purpose of FSCTL_READ_USN_JOURNAL is to fill the output buffer with an array of zero or more records that match the criteria specified by ReasonMask and ReturnOnClose. There is no way to know how many matching records will be found, so the system will just fill your output buffer with as many as possible. The function's behavior depends on the number of records found, the size of the output buffer, and the BytesToWaitFor and Timeout members. The output buffer specified by lpOutBuffer and nOutBufferSize must be at least sizeof(USN) bytes long, and aligned on a 32-bit boundary; otherwise DeviceIoControl will fail. If DeviceIoControl succeeds, lpOutBuffer will be filled in with a USN in the first sizeof(USN) bytes, followed by an array of zero or more records. See **Figure 9** for the layout of the output buffer from the call

```
DeviceIOControl(hcj, FSCTL_READ_USN_JOURNAL, &InBuf,
```

```
sizeof(InBuf), pOut, cbOut, &cbReturned, NULL);
```

The USN returned at the start of the buffer is the USN of the next record following the last record returned. This is used to walk journal records without knowing exactly how much space is required. Use this USN as StartUsn on the next call to DeviceIoControl with the FSCTL_READ_USN_ JOURNAL code. **Figure 10** shows how to get all the data between two USNs, as well as how to walk the records in the output buffer.

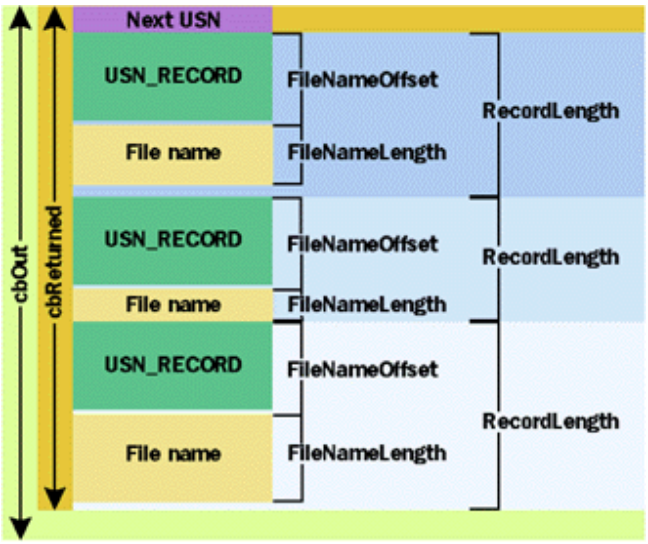

Figure 9 Output Buffer Data

The code in **Figure 10** should be used to read records that are known to exist in the journal. The usnStart and usnEnd parameters should be between or equal to the values StartUsn and NextUsn determined by FSCTL_QUERY_ USN_JOURNAL.

The Sample Application

The sample application CJDump uses everything we've discussed so far to dump the contents of the Change Journal. Since each volume may have its own Change Journal, CJDump just uses the current drive letter when picking a volume to examine. CJDump is a console application, and all the work is done in the main function.

The first thing that CJDump will do is print the information returned by using the FSCTL_QUERY_USN_JOURNAL code. CJDump will then read all available records from the Change Journal and print the USN, the reason code, and the file name of each record. CJDump can be easily modified to show other members of the USN_RECORD structure, or you can look at them in the debugger.

Since you'll want to look at a volume that has an active Change Journal, use a machine that has the Indexing service started. This service lets you perform full text searches across all the documents on your hard drives. It uses the Change Journal on NTFS volumes to monitor when documents are created, deleted, or moved. The Indexing service is available on both Windows 2000 Professional and Windows 2000 Server. For more information, see the Windows 2000 help files.

help files.

What's Coming Up

In our next article, we'll build a fully functional Change Journal application. We'll cover the correct programmatic ways to activate or disable the Change Journal, how to receive notification of journal changes, and how to use the information in journal records to maintain an accurate database of the files and directories on disk. In addition, we'll show how an application can persist information to disk when it is shut down and use the Change Journal to find out what's changed the next time it is launched. We'll also show how to convert file reference numbers to full paths by using the Change Journal itself to maintain a database of all the directories on a volume.

As you can see, the Change Journal provides a powerful new capability for applications to monitor changes to an NTFS volume without resorting to the costly process of full rescans. Applications such as virus checkers, search engines, and backup software can obviously benefit from this information. Perhaps the Change Journal will encourage the development of new classes of applications that we haven't even considered. We hope that this information is useful to you as you come up with your own killer app.

For related information see:
For related information see the *NTFS File System* page at http://msdn.microsoft.com/library/sdkdoc/winbase/fsys_538t.htm. Also check http://msdn.microsoft.com for daily updates on developer programs, resources and events.

From the September 1999 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.