E
NTFS On-Disk Structure

One of the interesting file system control operations defined in winioctl.h is `FSCTL_GET_NTFS_FILE_RECORD`, which retrieves a file record from the Master File Table (MFT) on an NTFS volume. When calling **`ZwFsControlFile`** (or the Win32 function `DeviceIoControl`) with this control code, the `InputBuffer` parameter points to a `NTFS_FILE_RECORD_INPUT_BUFFER` structure, and the `OutputBuffer` parameter points to a buffer large enough to hold a `NTFS_FILE_RECORD_OUTPUT_BUFFER` structure and a file record.

```
typedef struct {
    ULONGLONG FileReferenceNumber;
} NTFS_FILE_RECORD_INPUT_BUFFER, *PNTFS_FILE_RECORD_INPUT_BUFFER;

typedef struct {
    ULONGLONG FileReferenceNumber;
    ULONG FileRecordLength;
    UCHAR FileRecordBuffer[1];
} NTFS_FILE_RECORD_OUTPUT_BUFFER, *PNTFS_FILE_RECORD_OUTPUT_BUFFER;
```

Strictly speaking, a `FileReferenceNumber` consists of a 48-bit index into the Master File Table and a 16-bit sequence number that records how many times the entry in the table has been reused, but the sequence number is ignored when using `FSCTL_GET_NTFS_FILE_RECORD`. Therefore, to retrieve the file record at index 30, the value 30 should be assigned to `FileReferenceNumber`. If the table entry at index 30 is empty, `FSCTL_GET_NTFS_FILE_RECORD` retrieves a nearby entry that is not empty. To verify that the intended table entry has been retrieved, it is necessary to compare the low order 48 bits of `FileReferenceNumber` in the output buffer with that in the input buffer.

The remainder of this chapter describes the data structures that represent the on-disk structure of NTFS. It includes a sample utility that interprets the data structures to recover the data of a deleted file. The descriptions of the on-disk data structures also serve to explain the contents of the `FileRecordBuffer` returned by `FSCTL_GET_NTFS_FILE_RECORD`.

NTFS_RECORD_HEADER

```
typedef struct {
    ULONG Type;
    USHORT UsaOffset;
    USHORT UsaCount;
    USN Usn;
} NTFS_RECORD_HEADER, *PNTFS_RECORD_HEADER;
```

Members

Type
The type of NTFS record. When the value of `Type` is considered as a sequence of four one-byte characters, it normally spells an acronym for the type. Defined values include:

```
'FILE'
'INDX'
'BAAD'
'HOLE'
'CHKD'
```

UsaOffset
The offset, in bytes, from the start of the structure to the Update Sequence Array.

UsaCount
The number of values in the Update Sequence Array.

Usn
The Update Sequence Number of the NTFS record.

Remarks

None.

FILE_RECORD_HEADER

```
typedef struct {
    NTFS_RECORD_HEADER Ntfs;
    USHORT SequenceNumber;
    USHORT LinkCount;
    USHORT AttributesOffset;
    USHORT Flags;               // 0x0001 = InUse, 0x0002 = Directory
    ULONG BytesInUse;
    ULONG BytesAllocated;
    ULONGLONG BaseFileRecord;
    USHORT NextAttributeNumber;
} FILE_RECORD_HEADER, *PFILE_RECORD_HEADER;
```

Members

Ntfs
An `NTFS_RECORD_HEADER` structure with a Type of '`FILE`'.

SequenceNumber
The number of times that the MFT entry has been reused.

LinkCount
The number of directory links to the MFT entry.

AttributeOffset
The offset, in bytes, from the start of the structure to the first attribute of the MFT entry.

Flags
A bit array of flags specifying properties of the MFT entry. The values defined include:

```
InUse      0x0001  // The MFT entry is in use
Directory  0x0002  // The MFT entry represents a directory
```

BytesInUse
The number of bytes used by the MFT entry.

BytesAllocated
The number of bytes allocated for the MFT entry.

BaseFileRecord
If the MFT entry contains attributes that overflowed a base MFT entry, this member contains the file reference number of the base entry; otherwise, it contains zero.

NextAttributeNumber
The number that will be assigned to the next attribute added to the MFT entry.

Remarks

An entry in the MFT consists of a `FILE_RECORD_HEADER` followed by a sequence of attributes.

ATTRIBUTE

```
typedef struct {
    ATTRIBUTE_TYPE AttributeType;
    ULONG Length;
    BOOLEAN Nonresident;
    UCHAR NameLength;
    USHORT NameOffset;
    USHORT Flags;               // 0x0001 = Compressed
    USHORT AttributeNumber;
} ATTRIBUTE, *PATTRIBUTE;
```

Members

AttributeType

The type of the attribute. The following types are defined:

```
typedef enum {
    AttributeStandardInformation = 0x10,
    AttributeAttributeList = 0x20,
    AttributeFileName = 0x30,
    AttributeObjectId = 0x40,
    AttributeSecurityDescriptor = 0x50,
    AttributeVolumeName = 0x60,
    AttributeVolumeInformation = 0x70,
    AttributeData = 0x80,
    AttributeIndexRoot = 0x90,
    AttributeIndexAllocation = 0xA0,
    AttributeBitmap = 0xB0,
    AttributeReparsePoint = 0xC0,
    AttributeEAInformation = 0xD0,
    AttributeEA = 0xE0,
    AttributePropertySet = 0xF0,
    AttributeLoggedUtilityStream = 0x100
} ATTRIBUTE_TYPE, *PATTRIBUTE_TYPE;
```

Length

The size, in bytes, of the resident part of the attribute.

Nonresident

Specifies, when true, that the attribute value is nonresident.

NameLength

The size, in characters, of the name (if any) of the attribute.

NameOffset

The offset, in bytes, from the start of the structure to the attribute name. The attribute name is stored as a Unicode string.

Flags

A bit array of flags specifying properties of the attribute. The values defined include:

```
Compressed     0x0001  // The attribute is compressed
```

AttributeNumber

A numeric identifier for the instance of the attribute.

Remarks

None.

RESIDENT_ATTRIBUTE

```
typedef struct {
    ATTRIBUTE Attribute;
    ULONG ValueLength;
    USHORT ValueOffset;
    USHORT Flags;               // 0x0001 = Indexed
} RESIDENT_ATTRIBUTE, *PRESIDENT_ATTRIBUTE;
```

Members

Attribute
An `ATTRIBUTE` structure containing members common to resident and nonresident attributes.

ValueLength
The size, in bytes, of the attribute value.

ValueOffset
The offset, in bytes, from the start of the structure to the attribute value.

Flags
A bit array of flags specifying properties of the attribute. The values defined include:

```
Indexed        0x0001  // The attribute is indexed
```

Remarks

None.

NONRESIDENT_ATTRIBUTE

```
typedef struct {
    ATTRIBUTE Attribute;
    ULONGLONG LowVcn;
    ULONGLONG HighVcn;
    USHORT RunArrayOffset;
    UCHAR CompressionUnit;
    UCHAR AlignmentOrReserved[5];
    ULONGLONG AllocatedSize;
    ULONGLONG DataSize;
    ULONGLONG InitializedSize;
    ULONGLONG CompressedSize;    // Only when compressed
} NONRESIDENT_ATTRIBUTE, *PNONRESIDENT_ATTRIBUTE;
```

Members

Attribute
An `ATTRIBUTE` structure containing members common to resident and nonresident attributes.

LowVcn
The lowest valid Virtual Cluster Number (VCN) of this portion of the attribute value. Unless the attribute value is very fragmented (to the extent that an attribute list is needed to describe it), there is only one portion of the attribute value, and the value of `LowVcn` is zero.

HighVcn
The highest valid VCN of this portion of the attribute value.

RunArrayOffset
The offset, in bytes, from the start of the structure to the run array that contains the mappings between VCNs and Logical Cluster Numbers (LCNs).

CompressionUnit
The compression unit for the attribute expressed as the logarithm to the base two of the number of clusters in a compression unit. If `CompressionUnit` is zero, the attribute is not compressed.

AllocatedSize
The size, in bytes, of disk space allocated to hold the attribute value.

DataSize
The size, in bytes, of the attribute value. This may be larger than the `AllocatedSize` if the attribute value is compressed or sparse.

InitializedSize
The size, in bytes, of the initialized portion of the attribute value.

CompressedSize
The size, in bytes, of the attribute value after compression. This member is only present when the attribute is compressed.

Remarks

None.

AttributeStandardInformation

```
typedef struct {
    ULONGLONG CreationTime;
    ULONGLONG ChangeTime;
    ULONGLONG LastWriteTime;
    ULONGLONG LastAccessTime;
    ULONG FileAttributes;
    ULONG AlignmentOrReservedOrUnknown[3];
    ULONG QuotaId;                      // NTFS 3.0 only
    ULONG SecurityId;                   // NTFS 3.0 only
    ULONGLONG QuotaCharge;              // NTFS 3.0 only
    USN Usn;                            // NTFS 3.0 only
} STANDARD_INFORMATION, *PSTANDARD_INFORMATION;
```

Members

CreationTime
The time when the file was created in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

ChangeTime
The time when the file attributes were last changed in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

LastWriteTime
The time when the file was last written in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

LastAccessTime
The time when the file was last accessed in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601).

FileAttributes
The attributes of the file. Defined attributes include:

```
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_SPARSE_FILE
FILE_ATTRIBUTE_REPARSE_POINT
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
FILE_ATTRIBUTE_ENCRYPTED
```

AlignmentOrReservedOrUnknown
Normally contains zero. Interpretation unknown.

QuotaId
A numeric identifier of the disk quota that has been charged for the file (probably an index into the file "`\$Extend\$Quota`"). If quotas are disabled, the value of `QuotaId` is zero. This member is only present in NTFS 3.0. If a volume has been upgraded from an earlier version of NTFS to version 3.0, this member is only present if the file has been accessed since the upgrade.

SecurityId
A numeric identifier of the security descriptor that applies to the file (probably an index into the file "`\$Secure`"). This member is only present in NTFS 3.0. If a volume has been upgraded from an earlier version of NTFS to version 3.0, this member is only present if the file has been accessed since the upgrade.

QuotaCharge
The size, in bytes, of the charge to the quota for the file. If quotas are disabled, the value of `QuotaCharge` is zero. This member is only present in NTFS 3.0. If a volume has been upgraded from an earlier version of NTFS to version 3.0, this member is only present if the file has been accessed since the upgrade.

Usn
The Update Sequence Number of the file. If journaling is not enabled, the value of `Usn` is zero. This member is only present in NTFS 3.0. If a volume has been upgraded from an earlier version of NTFS to version 3.0, this member is only present if the file has been accessed since the upgrade.

Remarks

The standard information attribute is always resident.

AttributeAttributeList

```
typedef struct {
    ATTRIBUTE_TYPE AttributeType;
    USHORT Length;
    UCHAR NameLength;
    UCHAR NameOffset;
    ULONGLONG LowVcn;
    ULONGLONG FileReferenceNumber;
    USHORT AttributeNumber;
    USHORT AlignmentOrReserved[3];
} ATTRIBUTE_LIST, *PATTRIBUTE_LIST;
```

Members

AttributeType
The type of the attribute.

Length
The size, in bytes, of the attribute list entry.

NameLength
The size, in characters, of the name (if any) of the attribute.

NameOffset
The offset, in bytes, from the start of the `ATTRIBUTE_LIST` structure to the attribute name. The attribute name is stored as a Unicode string.

LowVcn
The lowest valid Virtual Cluster Number (VCN) of this portion of the attribute value.

FileReferenceNumber
The file reference number of the MFT entry containing the `NONRESIDENT_ATTRIBUTE` structure for this portion of the attribute value.

AttributeNumber
A numeric identifier for the instance of the attribute.

Remarks

The attribute list attribute is always nonresident and consists of an array of `ATTRIBUTE_LIST` structures.

An attribute list attribute is only needed when the attributes of a file do not fit in a single MFT record. Possible reasons for overflowing a single MFT entry include:

- The file has a large numbers of alternate names (hard links)
- The attribute value is large, and the volume is badly fragmented
- The file has a complex security descriptor (does not affect NTFS 3.0)
- The file has many streams

AttributeFileName

```
typedef struct {
    ULONGLONG DirectoryFileReferenceNumber;
    ULONGLONG CreationTime;   // Saved when filename last changed
    ULONGLONG ChangeTime;     // ditto
    ULONGLONG LastWriteTime;  // ditto
    ULONGLONG LastAccessTime; // ditto
    ULONGLONG AllocatedSize;  // ditto
    ULONGLONG DataSize;       // ditto
    ULONG FileAttributes;     // ditto
    ULONG AlignmentOrReserved;
    UCHAR NameLength;
    UCHAR NameType;           // 0x01 = Long, 0x02 = Short
    WCHAR Name[1];
} FILENAME_ATTRIBUTE, *PFILENAME_ATTRIBUTE;
```

Members

DirectoryFileReferenceNumber
The file reference number of the directory in which the filename is entered.

CreationTime
The time when the file was created in the standard time format (that is. the number of 100-nanosecond intervals since January 1, 1601). This member is only updated when the filename changes and may differ from the field of the same name in the `STANDARD_INFORMATION` structure.

ChangeTime
The time when the file attributes were last changed in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601). This member is only updated when the filename changes and may differ from the field of the same name in the `STANDARD_INFORMATION` structure.

LastWriteTime
The time when the file was last written in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601). This member is only updated when the filename changes and may differ from the field of the same name in the `STANDARD_INFORMATION` structure.

LastAccessTime
The time when the file was last accessed in the standard time format (that is, the number of 100-nanosecond intervals since January 1, 1601). This member is only updated when the filename changes and may differ from the field of the same name in the `STANDARD_INFORMATION` structure.

AllocatedSize
The size, in bytes, of disk space allocated to hold the attribute value. This member is only updated when the filename changes.

DataSize
The size, in bytes, of the attribute value. This member is only updated when the filename changes.

FileAttributes
The attributes of the file. This member is only updated when the filename changes and may differ from the field of the same name in the `STANDARD_INFORMATION` structure.

NameLength
The size, in characters, of the filename.

NameType
The type of the name. A type of zero indicates an ordinary name, a type of one indicates a long name corresponding to a short name, and a type of two indicates a short name corresponding to a long name.

Name
The name, in Unicode, of the file.

Remarks

The filename attribute is always resident.

AttributeObjectId

```
typedef struct {
    GUID ObjectId;
    union {
        struct {
            GUID BirthVolumeId;
            GUID BirthObjectId;
            GUID DomainId;
        } ;
        UCHAR ExtendedInfo[48];
    };
} OBJECTID_ATTRIBUTE, *POBJECTID_ATTRIBUTE;
```

Members

ObjectId
The unique identifier assigned to the file.

BirtVolumeId
The unique identifier of the volume on which the file was first created. Need not be present.

BirthObjectId
The unique identifier assigned to the file when it was first created. Need not be present.

DomainId
Reserved. Need not be present.

Remarks

The object identifier attribute is always resident.

AttributeSecurityDescriptor

The security descriptor attribute is stored on disk as a standard self-relative security descriptor. This attribute does not normally appear in MFT entries on NTFS 3.0 format volumes.

AttributeVolumeName

The volume name attribute just contains the volume label as a Unicode string.

AttributeVolumeInformation

```
typedef struct {
    ULONG Unknown[2];
    UCHAR MajorVersion;
    UCHAR MinorVersion;
    USHORT Flags;
} VOLUME_INFORMATION, *PVOLUME_INFORMATION;
```

Members

Unknown
Interpretation unknown.

MajorVersion
The major version number of the NTFS format.

MinorVersion
The minor version number of the NTFS format.

Flags
A bit array of flags specifying properties of the volume. The values defined include:

```
VolumeIsDirty    0x0001
```

Remarks

Windows 2000 formats new volumes as NTFS version 3.0. Windows NT 4.0 formats new volumes as NTFS version 2.1.

AttributeData

The data attribute contains whatever data the creator of the attribute chooses.

AttributeIndexRoot

```
typedef struct {
    ATTRIBUTE_TYPE Type;
    ULONG CollationRule;
    ULONG BytesPerIndexBlock;
    ULONG ClustersPerIndexBlock;
    DIRECTORY_INDEX DirectoryIndex;
} INDEX_ROOT, *PINDEX_ROOT;
```

Members

Type
The type of the attribute that is indexed.

CollationRule
A numeric identifier of the collation rule used to sort the index entries.

BytesPerIndexBlock
The number of bytes per index block.

ClustersPerIndexBlock
The number of clusters per index block.

DirectoryIndex
A `DIRECTORY_INDEX` structure.

Remarks

An `INDEX_ROOT` structure is followed by a sequence of `DIRECTORY_ENTRY` structures.

AttributeIndexAllocation

```
typedef struct {
    NTFS_RECORD_HEADER Ntfs;
    ULONGLONG IndexBlockVcn;
    DIRECTORY_INDEX DirectoryIndex;
} INDEX_BLOCK_HEADER, *PINDEX_BLOCK_HEADER;
```

Members

Ntfs
An `NTFS_RECORD_HEADER` structure with a Type of '`INDX`'.

IndexBlockVcn
The VCN of the index block.

DirectoryIndex
A `DIRECTORY_INDEX` structure.

Remarks

The index allocation attribute is an array of index blocks. Each index block starts with an `INDEX_BLOCK_HEADER` structure, which is followed by a sequence of `DIRECTORY_ENTRY` structures.

DIRECTORY_INDEX

```
typedef struct {
    ULONG EntriesOffset;
    ULONG IndexBlockLength;
    ULONG AllocatedSize;
    ULONG Flags;         // 0x00 = Small directory, 0x01 = Large directory
} DIRECTORY_INDEX, *PDIRECTORY_INDEX;
```

Members

EntriesOffset
The offset, in bytes, from the start of the structure to the first `DIRECTORY_ENTRY` structure.

IndexBlockLength
The size, in bytes, of the portion of the index block that is in use.

AllocatedSize
The size, in bytes, of disk space allocated for the index block.

Flags
A bit array of flags specifying properties of the index. The values defined include:

```
SmallDirectory   0x0000  // Directory fits in index root
LargeDirectory   0x0001  // Directory overflows index root
```

Remarks

None.

DIRECTORY_ENTRY

```
typedef struct {
    ULONGLONG FileReferenceNumber;
    USHORT Length;
    USHORT AttributeLength;
    ULONG Flags;           // 0x01 = Has trailing VCN, 0x02 = Last entry
    // FILENAME_ATTRIBUTE Name;
    // ULONGLONG Vcn;      // VCN in IndexAllocation of earlier entries
} DIRECTORY_ENTRY, *PDIRECTORY_ENTRY;
```

Members

FileReferenceNumber
The file reference number of the file described by the directory entry.

Length
The size, in bytes, of the directory entry.

AttributeLength
The size, in bytes, of the attribute that is indexed.

Flags
A bit array of flags specifying properties of the entry. The values defined include:

```
HasTrailingVcn   0x0001  // A VCN follows the indexed attribute
LastEntry        0x0002  // The last entry in an index block
```

Remarks

Until NTFS version 3.0, only filename attributes were indexed.

If the `HasTrailingVcn` flag of a `DIRECTORY_ENTRY` structure is set, the last eight bytes of the directory entry contain the VCN of the index block that holds the entries immediately preceding the current entry.

AttributeBitmap

The bitmap attribute contains an array of bits. The file "`\$Mft`" contains a bitmap attribute that records which MFT table entries are in use, and directories normally contain a bitmap attribute that records which index blocks contain valid entries.

AttributeReparsePoint

```
typedef struct {
    ULONG ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    UCHAR ReparseData[1];
} REPARSE_POINT, *PREPARSE_POINT;
```

Members

ReparseTag
The reparse tag identifies the type of reparse point. The high order three bits of the tag indicate whether the tag is owned by Microsoft, whether there is a high latency in accessing the file data, and whether the filename is an alias for another object.

ReparseDataLength
The size, in bytes, of the reparse data in the `ReparseData` member.

ReparseData
The reparse data. The interpretation of the data depends upon the type of the reparse point.

Remarks

None.

AttributeEAInformation

```
typedef struct {
    ULONG EaLength;
    ULONG EaQueryLength;
} EA_INFORMATION, *PEA_INFORMATION;
```

Members

EaLength
The size, in bytes, of the extended attribute information.

EaQueryLength
The size, in bytes, of the buffer needed to query the extended attributes when calling **`ZwQueryEaFile`**.

Remarks

None.

AttributeEA

```
typedef struct {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
    // UCHAR EaData[];
} EA_ATTRIBUTE, *PEA_ATTRIBUTE;
```

Members

NextEntryOffset
The number of bytes that must be skipped to get to the next entry.

Flags
A bit array of flags qualifying the extended attribute.

EaNameLength
The size, in bytes, of the extended attribute name.

EaValueLength
The size, in bytes, of the extended attribute value.

EaName
The extended attribute name.

EaData
The extended attribute data.

Remarks

None.

AttributePropertySet

Intended to support Native Structured Storage (NSS)—a feature that was removed from NTFS 3.0 during beta testing.

AttributeLoggedUtilityStream

A logged utility stream attribute contains whatever data the creator of the attribute chooses, but operations on the attribute are logged to the NTFS log file just like NTFS metadata changes. It is used by the Encrypting File System (EFS).

Special Files

The first sixteen entries in the Master File Table (MFT) are reserved for special files. NTFS 3.0 uses only the first twelve entries.

\$MFT (entry 0)

The Master File Table. The data attribute contains the MFT entries, and the bitmap attribute records which entries are in use.

\$MFTMirr (entry 1)

A mirror (backup copy) of the first four entries of the MFT.

\$LogFile (entry 2)

The volume log file that records changes to the volume structure.

\$Volume (entry 3)

The data attribute of `$Volume` represents the whole volume. Opening the Win32 pathname "`\\.\C:`" opens the volume file on drive C: (presuming that C: is an NTFS-formatted volume).

The `$Volume` file also has volume name, volume information, and object identifier attributes.

\$AttrDef (entry 4)

The data attribute of `$AttrDef` contains an array of attribute definitions.

```
typedef struct {
    WCHAR AttributeName[64];
    ULONG AttributeNumber;
    ULONG Unknown[2];
    ULONG Flags;
    ULONGLONG MinimumSize;
    ULONGLONG MaximumSize;
} ATTRIBUTE_DEFINITION, *PATTRIBUTE_DEFINITION;
```

\ (entry 5)

The root directory of the volume.

\$Bitmap (entry 6)

The data attribute of `$Bitmap` is a bitmap of the allocated clusters on the volume.

\$Boot (entry 7)

The first sector of `$Boot` is also the first sector of the volume. Because it is used early in the system boot process (if the volume is bootable), space is at a premium and the data stored in it is not aligned on natural boundaries. The format of the first sector can be represented by a `BOOT_BLOCK` structure.

```
#pragma pack(push, 1)

typedef struct {
    UCHAR Jump[3];
```

```
    UCHAR Format[8];
    USHORT BytesPerSector;
    UCHAR SectorsPerCluster;
    USHORT BootSectors;
    UCHAR Mbz1;
    USHORT Mbz2;
    USHORT Reserved1;
    UCHAR MediaType;
    USHORT Mbz3;
    USHORT SectorsPerTrack;
    USHORT NumberOfHeads;
    ULONG PartitionOffset;
    ULONG Reserved2[2];
    ULONGLONG TotalSectors;
    ULONGLONG MftStartLcn;
    ULONGLONG Mft2StartLcn;
    ULONG ClustersPerFileRecord;
    ULONG ClustersPerIndexBlock;
    ULONGLONG VolumeSerialNumber;
    UCHAR Code[0x1AE];
    USHORT BootSignature;
} BOOT_BLOCK, *PBOOT_BLOCK;

#pragma pack(pop)
```

\$BadClus (entry 8)

Bad clusters are appended to the data attribute of this file.

\$Secure (entry 9)

The data attribute of `$Secure` contains the shared security descriptors. `$Secure` also has two indexes.

\$UpCase (entry 10)

The data attribute of `$Upcase` contains the uppercase equivalent of all 65536 Unicode characters.

\$Extend (entry 11)

`$Extend` is a directory that holds the special files used by some of the extended functionality of NTFS 3.0. The (semi-) special files which are stored in the directory include "`$ObjId`," "`$Quota`," "`$Reparse`" and "`$UsnJrnl`."

Opening Special Files

Although the special files are indeed files, they cannot normally be opened by calling **`ZwOpenFile`** or **`ZwCreateFile`** because even though the ACL on the special files grants read access to Administrators, ntfs.sys (the NTFS file system driver) always returns `STATUS_ACCESS_DENIED`. There are two variables in ntfs.sys that affect this behavior: `NtfsProtectSystemFiles` and `NtfsProtectSystemAttributes`. By default, both of these variables are set to `TRUE`.

If `NtfsProtectSystemAttributes` is set to `FALSE` (by a debugger, for example), the system attributes (such as the standard information attribute) can be opened, using the names of the form "`filename::$STANDARD_INFORMATION`."

If `NtfsProtectSystemFiles` is set to `FALSE`, then the special files can be opened. There are, however, some drawbacks associated with attempting to do this: Because many of the special files are opened in a special way when mounting the volume, they are not prepared to handle the `IRP_MJ_READ` requests resulting from a call to **`ZwReadFile`**, and the system crashes if such a request is received. These special files can be read by mapping the special file with **`ZwCreateSection`** and **`ZwMapViewOfSection`** and then reading the mapped data. A further problem is that a few of the special files are not prepared to handle the `IRP_MJ_CLEANUP` request that is generated when the last handle to a file object is closed, and the system crashes if such a request is received. The only option is to duplicate the open handle to the special file into a process that never terminates (such as the system process).

Recovering Data from Deleted Files

Example E.1 demonstrates how to recover data from the unnamed data attribute of a file identified by drive letter and MFT entry index—even if the MFT entry represents a deleted file. It can also display a list of the deleted files on the volume. MFT entries are allocated on a first-free basis, so the entries for deleted files are normally quickly reused. Therefore, the example is of little practical use for recovering deleted files, but it can be used to make copies of the unnamed data attributes of the special files.

If the file to be recovered is compressed, the recovered data remains compressed and can be decompressed by a separate utility; Example E.2 shows one way in which this can be done.

Example E.1: Recovering Data from a File

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "ntfs.h"

ULONG BytesPerFileRecord;
HANDLE hVolume;
BOOT_BLOCK bootb;
PFILE_RECORD_HEADER MFT;

template <class T1, class T2> inline
T1* Padd(T1* p, T2 n) { return (T1*)((char *)p + n); }

ULONG RunLength(PUCHAR run)
{
    return (*run & 0xf) + ((*run >> 4) & 0xf) + 1;
}

LONGLONG RunLCN(PUCHAR run)
{
    UCHAR n1 = *run & 0xf;
    UCHAR n2 = (*run >> 4) & 0xf;
    LONGLONG lcn = n2 == 0 ? 0 : CHAR(run[n1 + n2]);

    for (LONG i = n1 + n2 - 1; i > n1; i—)
        lcn = (lcn << 8) + run[i];
    return lcn;
}
```

```
ULONGLONG RunCount(PUCHAR run)
{
    UCHAR n = *run & 0xf;
    ULONGLONG count = 0;

    for (ULONG i = n; i > 0; i—)
        count = (count << 8) + run[i];
    return count;
}

BOOL FindRun(PNONRESIDENT_ATTRIBUTE attr, ULONGLONG vcn,
             PULONGLONG lcn, PULONGLONG count)
{
    if (vcn < attr->LowVcn ¦¦ vcn > attr->HighVcn) return FALSE;

    *lcn = 0;
    ULONGLONG base = attr->LowVcn;

    for (PUCHAR run = PUCHAR(Padd(attr, attr->RunArrayOffset));
         *run != 0;
         run += RunLength(run)) {

        *lcn += RunLCN(run);
        *count = RunCount(run);

        if (base <= vcn && vcn < base + *count) {
            *lcn = RunLCN(run) == 0 ? 0 : *lcn + vcn - base;
            *count -= ULONG(vcn - base);

            return TRUE;
        }
        else
            base += *count;
    }

    return FALSE;
}

PATTRIBUTE FindAttribute(PFILE_RECORD_HEADER file,
                         ATTRIBUTE_TYPE type, PWSTR name)
{
    for (PATTRIBUTE attr = PATTRIBUTE(Padd(file, file->AttributesOffset));
         attr->AttributeType != -1;
         attr = Padd(attr, attr->Length)) {

        if (attr->AttributeType == type) {
            if (name == 0 && attr->NameLength == 0) return attr;

            if (name != 0 && wcslen(name) == attr->NameLength
                && _wcsicmp(name, PWSTR(Padd(attr, attr->NameOffset))) == 0)
                return attr;
        }
    }

    return 0;
}

VOID FixupUpdateSequenceArray(PFILE_RECORD_HEADER file)
{
    PUSHORT usa = PUSHORT(Padd(file, file->Ntfs.UsaOffset));
    PUSHORT sector = PUSHORT(file);
```

```
    for (ULONG i = 1; i < file->Ntfs.UsaCount; i++) {
        sector[255] = usa[i];
        sector += 256;
    }
}

VOID ReadSector(ULONGLONG sector, ULONG count, PVOID buffer)
{
    ULARGE_INTEGER offset;
    OVERLAPPED overlap = {0};
    ULONG n;

    offset.QuadPart = sector * bootb.BytesPerSector;
    overlap.Offset = offset.LowPart; overlap.OffsetHigh = offset.HighPart;

    ReadFile(hVolume, buffer, count * bootb.BytesPerSector, &n, &overlap);
}

VOID ReadLCN(ULONGLONG lcn, ULONG count, PVOID buffer)
{
    ReadSector(lcn * bootb.SectorsPerCluster,
               count * bootb.SectorsPerCluster, buffer);
}

VOID ReadExternalAttribute(PNONRESIDENT_ATTRIBUTE attr,
                           ULONGLONG vcn, ULONG count, PVOID buffer)
{
    ULONGLONG lcn, runcount;
    ULONG readcount, left;
    PUCHAR bytes = PUCHAR(buffer);

    for (left = count; left > 0; left -= readcount) {
        FindRun(attr, vcn, &lcn, &runcount);

        readcount = ULONG(min(runcount, left));

        ULONG n = readcount * bootb.BytesPerSector * bootb.SectorsPerCluster;

        if (lcn == 0)
            memset(bytes, 0, n);
        else
            ReadLCN(lcn, readcount, bytes);

        vcn += readcount;
        bytes += n;
    }
}

ULONG AttributeLength(PATTRIBUTE attr)
{
    return attr->Nonresident == FALSE
        ? PRESIDENT_ATTRIBUTE(attr)->ValueLength
        : ULONG(PNONRESIDENT_ATTRIBUTE(attr)->DataSize);
}

ULONG AttributeLengthAllocated(PATTRIBUTE attr)
{
    return attr->Nonresident == FALSE
        ? PRESIDENT_ATTRIBUTE(attr)->ValueLength
        : ULONG(PNONRESIDENT_ATTRIBUTE(attr)->AllocatedSize);
}
```

```
VOID ReadAttribute(PATTRIBUTE attr, PVOID buffer)
{
    if (attr->Nonresident == FALSE) {
        PRESIDENT_ATTRIBUTE rattr = PRESIDENT_ATTRIBUTE(attr);
        memcpy(buffer, Padd(rattr, rattr->ValueOffset), rattr->ValueLength);
    }
    else {
        PNONRESIDENT_ATTRIBUTE nattr = PNONRESIDENT_ATTRIBUTE(attr);
        ReadExternalAttribute(nattr, 0, ULONG(nattr->HighVcn) + 1, buffer);
    }
}

VOID ReadVCN(PFILE_RECORD_HEADER file, ATTRIBUTE_TYPE type,
             ULONGLONG vcn, ULONG count, PVOID buffer)
{
    PNONRESIDENT_ATTRIBUTE attr
        = PNONRESIDENT_ATTRIBUTE(FindAttribute(file, type, 0));

    if (attr == 0 || (vcn < attr->LowVcn || vcn > attr->HighVcn)) {
        // Support for huge files

        PATTRIBUTE attrlist = FindAttribute(file, AttributeAttributeList, 0);

        DebugBreak();
    }

    ReadExternalAttribute(attr, vcn, count, buffer);
}

VOID ReadFileRecord(ULONG index, PFILE_RECORD_HEADER file)
{
    ULONG clusters = bootb.ClustersPerFileRecord;
    if (clusters > 0x80) clusters = 1;

    PUCHAR p = new UCHAR[bootb.BytesPerSector
                         * bootb.SectorsPerCluster * clusters];

    ULONGLONG vcn = ULONGLONG(index) * BytesPerFileRecord
                  / bootb.BytesPerSector / bootb.SectorsPerCluster;

    ReadVCN(MFT, AttributeData, vcn, clusters, p);

    LONG m = (bootb.SectorsPerCluster * bootb.BytesPerSector
              / BytesPerFileRecord) - 1;

    ULONG n = m > 0 ? (index & m) : 0;

    memcpy(file, p + n * BytesPerFileRecord, BytesPerFileRecord);

    delete [] p;

    FixupUpdateSequenceArray(file);
}

VOID LoadMFT()
{
    BytesPerFileRecord = bootb.ClustersPerFileRecord < 0x80
                       ? bootb.ClustersPerFileRecord
                           * bootb.SectorsPerCluster
                           * bootb.BytesPerSector
                       : 1 << (0x100 - bootb.ClustersPerFileRecord);
```

```
    MFT = PFILE_RECORD_HEADER(new UCHAR[BytesPerFileRecord]);

    ReadSector(bootb.MftStartLcn * bootb.SectorsPerCluster,
               BytesPerFileRecord / bootb.BytesPerSector, MFT);

    FixupUpdateSequenceArray(MFT);
}

BOOL bitset(PUCHAR bitmap, ULONG i)
{
    return (bitmap[i >> 3] & (1 << (i & 7))) != 0;
}

VOID FindDeleted()
{
    PATTRIBUTE attr = FindAttribute(MFT, AttributeBitmap, 0);
    PUCHAR bitmap = new UCHAR[AttributeLengthAllocated(attr)];

    ReadAttribute(attr, bitmap);

    ULONG n = AttributeLength(FindAttribute(MFT, AttributeData, 0))
            / BytesPerFileRecord;

    PFILE_RECORD_HEADER file
        = PFILE_RECORD_HEADER(new UCHAR[BytesPerFileRecord]);

    for (ULONG i = 0; i < n; i++) {
        if (bitset(bitmap, i)) continue;

        ReadFileRecord(i, file);

        if (file->Ntfs.Type == 'ELIF' && (file->Flags & 1) == 0) {
            attr = FindAttribute(file, AttributeFileName, 0);
            if (attr == 0) continue;

            PFILENAME_ATTRIBUTE name
                = PFILENAME_ATTRIBUTE(Padd(attr,
                                PRESIDENT_ATTRIBUTE(attr)->ValueOffset));

            printf("%8lu %.*ws\n", i, int(name->NameLength), name->Name);
        }
    }
}

VOID DumpData(ULONG index, PCSTR filename)
{
    PFILE_RECORD_HEADER file
        = PFILE_RECORD_HEADER(new UCHAR[BytesPerFileRecord]);
    ULONG n;

    ReadFileRecord(index, file);

    if (file->Ntfs.Type != 'ELIF') return;

    PATTRIBUTE attr = FindAttribute(file, AttributeData, 0);
    if (attr == 0) return;

    PUCHAR buf = new UCHAR[AttributeLengthAllocated(attr)];

    ReadAttribute(attr, buf);

    HANDLE hFile = CreateFile(filename, GENERIC_WRITE, 0, 0,
                              CREATE_ALWAYS, 0, 0);
```

```
    WriteFile(hFile, buf, AttributeLength(attr), &n, 0);

    CloseHandle(hFile);

    delete [] buf;
}

int main(int argc, char *argv[])
{
    CHAR drive[] = "\\\\.\\C:";
    ULONG n;

    if (argc < 2) return 0;

    drive[4] = argv[1][0];

    hVolume = CreateFile(drive, GENERIC_READ,
                         FILE_SHARE_READ ¦ FILE_SHARE_WRITE, 0,
                         OPEN_EXISTING, 0, 0);

    ReadFile(hVolume, &bootb, sizeof bootb, &n, 0);

    LoadMFT();

    if (argc == 2) FindDeleted();
    if (argc == 4) DumpData(strtoul(argv[2], 0, 0), argv[3]);

    CloseHandle(hVolume);

    return 0;
}
```

Example E.2: Decompressing Recovered Data

```
#include <windows.h>

typedef ULONG NTSTATUS;

extern "C"
NTSTATUS
NTAPI
RtlDecompressBuffer(
    USHORT CompressionFormat,
    PVOID OutputBuffer,
    ULONG OutputBufferLength,
    PVOID InputBuffer,
    ULONG InputBufferLength,
    PULONG ReturnLength
    );

int main(int argc, char *argv[])
{
    if (argc != 3) return 0;

    HANDLE hFile1 = CreateFile(argv[1], GENERIC_READ,
                               FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0);
    HANDLE hFile2 = CreateFile(argv[2], GENERIC_READ ¦ GENERIC_WRITE,
                               FILE_SHARE_READ, 0, CREATE_ALWAYS, 0, 0);
```

```
    ULONG n = GetFileSize(hFile1, 0);

    HANDLE hMapping1 = CreateFileMapping(hFile1, 0, PAGE_READONLY, 0, 0, 0);
    HANDLE hMapping2 = CreateFileMapping(hFile2, 0, PAGE_READWRITE, 0, n, 0);

    PCHAR p = PCHAR(MapViewOfFileEx(hMapping1, FILE_MAP_READ, 0, 0, 0, 0));
    PCHAR q = PCHAR(MapViewOfFileEx(hMapping2, FILE_MAP_WRITE, 0, 0, 0, 0));

    for (ULONG m, i = 0; i < n; i += m)
        RtlDecompressBuffer(COMPRESSION_FORMAT_LZNT1,
                            q + i, n - i, p + i, n - i, &m);

    return 0;
}
```