File System Behavior in the Microsoft Windows Environment

This document contains a compilation of file system specific topics. The audience for these topics is developers interested in creating file systems that are functionally compatible with existing Windows file systems.

This document covers the native windows files systems:

- NTFS
- FAT
- EXFAT
- CDFS
- UDF

Differences in functionality between the above file systems will be explicitly noted. The term 'file system' will be used to refer to all the file systems for functionality that does not differ between them.

Revision History

Date	Changes
June 2008	• Initial Revision
April 2009	• Differences between various windows file systems • Windows 7 Oplock changes • File Stream naming conventions
May 2009	• IRP return codes • Timestamps • Wildcards • Corrections to stream naming information

Table of Contents

1 File Streams

A file stream is a sequence of bytes. File operations such as read and write operate on streams. Historically file systems have typically had only one stream per file that holds the files data and thus had no need to distinguish between the concept of a file and a stream. However some modern file systems like NTFS allow multiple data streams per file so in Windows it's necessary to distinguish between a file and a stream.

In windows all files systems have one default data stream. The default data stream is associated with a file handle when opening a file without specifying a stream name: (e.g. CreateFile("SomeFile",…)) This is the only method supported for the FAT, CDFS, and EXFAT file systems.

The NTFS and UDFS file systems can have alternate named data streams. To open a named data stream use the name syntax *filename*:*stream_name. For* example to open a stream named 'streamX' on file 'SomeFile' : CreateFile("SomeFile:streamX", …).

2 Oplock Semantics

2.1 Overview

Opportunistic locks, or oplocks, provide a mechanism that allows file server clients (such as those utilizing the SMB and SMB2 protocols) to dynamically alter buffering strategy for a given file or stream in a consistent manner to increase performance and to reduce network use. To increase the network performance for remote file operations, a client can locally buffer file data, which reduces or eliminates the need to send and receive network packets. For example, a client may not have to write information into a file on a remote server if the client knows that no other process is accessing the data. Likewise, the client may buffer read-ahead data from the remote file if the client knows that no other process is writing data to the remote file. Applications and drivers can also use oplocks to transparently access files without affecting other applications that might need to use those files.

Oplocks are granted on stream handles, meaning that the operations apply to the given open stream of a file and, in general, operations on one stream do not affect oplocks on a different stream. There are exceptions to this, which are explicitly listed below.

There are currently eight different types of oplocks:

- A **Level 2** (or shared) oplock indicates that there are multiple readers of a stream and no writers. This supports client read caching.
- A **Level 1** (or exclusive) oplock allows a client to open a stream for exclusive access and allows the client to perform arbitrary buffering. This supports client read caching and write caching.
- A **Batch** oplock (also exclusive) allows a client to keep a stream open on the server even though the local accessor on the client machine has closed the stream. This supports scenarios where the client needs to repeatedly open and close the same file, such as during batch script execution. This supports client read caching, write caching, and handle caching.
- A **Filter** oplock (also exclusive) allows applications and file system filters (including minifilters), which open and read stream data, a way to “back out” when other applications, clients, or both try to access the same stream. This supports client read caching and write caching.
- A **Read** (R) oplock (shared) indicates that there are multiple readers of a stream and no writers. This supports client read caching.
- A **Read-Handle** (RH) oplock (shared) indicates that there are multiple readers of a stream, no writers, and that a client can keep a stream open on the server even though the local accessor on the client machine has closed the stream. This supports client read caching and handle caching.
- A **Read-Write** (RW) oplock (exclusive) allows a client to open a stream for exclusive access and allows the client to perform arbitrary buffering. This supports client read caching and write caching.
- A **Read-Write-Handle** (RWH) oplock (exclusive) allows a client to keep a stream open on the server even though the local accessor on the client machine has closed the stream. This supports client read caching, write caching, and handle caching.

Level 1, Level 2, and Batch oplocks were implemented in Windows NT 3.1. The Filter oplock was added in Windows 2000. R, RH, RW, and RWH oplocks have been added in Windows 7.

Some oplocks seem quite similar. In particular, R seems similar to Level 2, RW seems similar to Level 1, and RWH seems similar to Batch. The R, RH, RW, and RWH oplocks (hereinafter referred to collectively as “Windows 7 oplocks”) have been added to Windows 7 to provide greater flexibility for the caller to express caching intentions, and to allow oplock breaks and upgrades (that is, modification of the oplock state from one level to a level of greater caching; for example, upgrading a Read oplock to a Read-Write oplock). This flexibility is not achievable with the Level 2, Level 1, Batch, and Filter oplocks (hereinafter referred to collectively as “legacy oplocks”). Differences between the Windows 7 oplocks and the legacy oplocks are discussed later in this documentation.

The core functionality of the oplock package is implemented in the kernel (primarily through **FsRtl***Xxx* routines). File systems call into this package to implement the oplock functionality in their file system. This document describes how windows file systems interoperate with the kernel oplock package. There may be some differences in behavior between the various windows file systems and this will be explicitly noted.

Oplocks are granted on stream handles. This means an oplock is granted for a given open of a stream. Starting with Windows 7, the stream handle can be associated with an *oplock key*, that is, a GUID value that is used to identify multiple handles that belong to the same client cache view [1]. The oplock key can be explicitly provided (to **IoCreateFileEx**) when the handle is created. If an oplock key is not explicitly specified when the handle is created, the system will treat the handle as having a unique oplock key associated with it, such that its key will differ from any other key on any other handle. If a file operation is received on a handle other than the one on which the oplock was granted, and the oplock key that is associated with the oplock’s handle differs from the key that is associated with the operation’s handle, and that operation is not compatible with the currently granted oplock, then that oplock is broken. The oplock breaks even if it is the same process or thread performing the incompatible operation. For example, if a process opens a stream for which an exclusive oplock is granted and the same process then opens the same stream again, using a different (or no) oplock key, the exclusive oplock is broken immediately.

Remember that oplock keys exist on handles, and they are "put on" the handle when the handle is created. You can associate a handle with an oplock key even if no oplocks are granted.

2.2 Granting Oplocks

Oplocks are requested through FSCTLs. The following list shows the FSCTLs for the different oplock types (which user-mode applications and kernel-mode drivers can issue):

- FSCTL_REQUEST_OPLOCK_LEVEL_1
- FSCTL_REQUEST_OPLOCK_LEVEL_2
- FSCTL_REQUEST_BATCH_OPLOCK

[1] It is more accurate to say that the oplock key is associated with the FILE_OBJECT structure that the stream handle refers to. This distinction is important when the handle is duplicated, such as with DuplicateHandle. Each of the duplicate handles refers to the same underlying FILE_OBJECT structure.

- FSCTL_REQUEST_FILTER_OPLOCK

- FSCTL_REQUEST_OPLOCK

The first four FSCTLs in the list are used to request legacy oplocks. The last FSCTL is used to request Windows 7 oplocks with the REQUEST_OPLOCK_INPUT_FLAG_REQUEST flag specified in the **Flags** member of the REQUEST_OPLOCK_INPUT_BUFFER structure, passed as the **lpInBuffer** parameter of **DeviceIOControl**. In a similar manner, **FltFsControlFile** and **ZwFsControlFile** can be used to request Windows 7 oplocks from kernel mode. To specify which of the four Windows 7 oplocks is required, one or more of the flags OPLOCK_LEVEL_CACHE_READ, OPLOCK_LEVEL_CACHE_HANDLE, or OPLOCK_LEVEL_CACHE_WRITE is set in the **RequestedOplockLevel** member of the REQUEST_OPLOCK_INPUT_BUFFER structure. For more information, see FSCTL_REQUEST_OPLOCK.

When a request is made for an oplock and the oplock can be granted, the file system returns STATUS_PENDING (because of this, oplocks are never granted for synchronous I/O). The FSCTL IRP does not complete until the oplock is broken. If the oplock cannot be granted, an appropriate error code is returned. The most commonly returned error codes are STATUS_OPLOCK_NOT_GRANTED and STATUS_INVALID_PARAMETER (and their equivalent user-mode analogs).

As mentioned previously, the Filter oplock allows an application to "back out" when other applications/clients try to access the same stream. This mechanism allows an application to access a stream without causing other accessors of the stream to receive sharing violations when attempting to open the stream. To avoid sharing violations, a special three-step procedure should be used to request a Filter oplock (FSCTL_REQUEST_FILTER_OPLOCK):

1. Open the file with a required access of FILE_READ_ATTRIBUTES and a share mode of FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE.
2. Request a Filter oplock on the handle from step 1.
3. Open the file again for read access.

The handle opened in step 1 will *not* cause other applications to receive sharing violations, since it is open only for attribute access (FILE_READ_ATTRIBUTES), and not data access (FILE_READ_DATA). This handle is suitable for requesting the Filter oplock, but not for performing actual I/O on the data stream. The handle opened in step 3 allows the holder of the oplock to perform I/O on the stream, while the oplock granted in step 2 allows the holder of the oplock to "get out of the way" without causing a sharing violation to another application that attempts to access the stream.

The NTFS file system provides an optimization for this procedure through the FILE_RESERVE_OPFILTER create option flag. If this flag is specified in step 1 of the previous procedure, it allows the file system to fail the create request with STATUS_OPLOCK_NOT_GRANTED if the file system can determine that step 2 will fail. Be aware that if step 1 succeeds, there is no guarantee that step 2 will succeed, even if FILE_RESERVE_OPFILTER was specified for the create request. Only the NTFS file system supports the FILE_RESERVE_OPFILTER flag.

Windows 7 introduces a more general way to get an oplock without causing other applications to get sharing violations between the time an application opens a handle and requests its oplock. This new

method allows an application to request any oplock type (not just a Filter oplock). To use this method, an application calls **NtCreateFile** and specifies FILE_OPEN_REQURING_OPLOCK in the **CreateOptions** parameter. If an oplock already exists on the file, this call will fail with STATUS_CANNOT_BREAK_OPLOCK. If the call succeeds, the application must then request an oplock. It may request an oplock of any type. It is possible for the oplock request to fail if, before the request is made, a third party opens the file using the FILE_COMPLETE_IF_OPLOCKED create option. In that case the application is advised to close its handle.

The following table identifies the required conditions necessary to grant an oplock.

Request type	Conditions

Level 1
 	
Filter
 	
Batch
	Granted only if all of the following conditions are true:
	• The request is for a given stream of a file.
	  ○ If a directory, STATUS_INVALID_PARAMETER is returned.
	• The stream is opened for ASYNCHRONOUS access.
	  ○ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned (oplocks are not granted for synchronous I/O requests).
	• There are no TxF transactions on any stream of the file.
	  ○ Else STATUS_OPLOCK_NOT_GRANTED is returned.
	• There are no other opens on the stream (even by the same thread).
	  ○ Else STATUS_OPLOCK_NOT_GRANTED is returned.
	Be aware that if the current oplock state is:
	• No Oplock: The request is granted.
	• Level 2: The original Level 2 request is broken with FILE_OPLOCK_BROKEN_TO_NONE. The requested exclusive oplock is then granted.
	• Level 1, Batch, Filter, Read, Read-Handle, Read-Write, or Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.

Level 2	Granted only if all of the following conditions are true:
	• The request is for a given stream of a file.
	  ○ If a directory, STATUS_INVALID_PARAMETER is returned.
	• The stream is opened for ASYNCHRONOUS access.
	  ○ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned.
	• There are no TxF transactions on the file.
	  ○ Else STATUS_OPLOCK_NOT_GRANTED is returned.
	• There are no current Byte Range Locks on the stream.
	  ○ Else STATUS_OPLOCK_NOT_GRANTED is returned.
	  ○ Be aware that prior to Windows 7, the operating system verifies if a byte range lock ever existed on the stream since the last time it was opened, and fails the request if so.
	Be aware that if the current oplock state is:
	• No Oplock: The request is granted.
	• Level 2 and/or Read: The request is granted. You can have multiple Level 2/Read oplocks granted on the same stream at the same time. Multiple Level 2 (but not Read) oplocks can even exist on the same

	handle. ○ If a Read oplock is requested on a handle that already has a Read oplock granted to it, the first Read oplock's IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE before the second Read oplock is granted. • Level 1, Batch, Filter, Read-Handle, Read-Write, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.
Read	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ○ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ○ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. • There are no current Byte Range Locks on the stream. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: The request is granted. • Level 2 and/or Read: The request is granted. You can have multiple Level 2/Read oplocks granted on the same stream at the same time. ○ Additionally, if an existing oplock has the same oplock key as the new request, its IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE. • Read-Handle *and* the existing oplock has a different oplock key from the new request: The request is granted. Multiple Read and Read-Handle oplocks can coexist on the same stream (see the note following this table). ○ Else (oplock keys are the same) STATUS_OPLOCK_NOT_GRANTED is returned. • Level 1, Batch, Filter, Read-Write, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.
Read-Handle	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ○ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ○ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. • There are no current Byte Range Locks on the stream. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: the request is granted. • Read: the request is granted.

	○ If an existing Read oplock has the same oplock key as the new request, its IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE. This means that the oplock is upgraded from Read to Read-Handle. ○ Any existing Read oplock that does not have the same oplock key as the new request remains unchanged. • Level 2, Level 1, Batch, Filter, Read-Write, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.
Read-Write	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ○ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ○ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. • If there are other opens on the stream (even by the same thread) they must have the same oplock key. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: the request is granted. • Read or Read-Write and the existing oplock has the same oplock key as the request: the existing oplock's IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE, the request is granted. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. • Level 2, Level 1, Batch, Filter, Read-Handle, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.
Read-Write-Handle	Granted only if all of the following are true: • The request is for a given stream of a file. ○ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ○ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. • If there are other open requests on the stream (even by the same thread) they must have the same oplock key. ○ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: the request is granted. • Read, Read-Handle, Read-Write, or Read-Write-Handle and the existing oplock has the same oplock key as the request: the existing oplock's IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE, the request is granted.

	○ Else STATUS_OPLOCK_NOT_GRANTED is returned. • Level 2, Level 1, Batch, Filter: STATUS_OPLOCK_NOT_GRANTED is returned.

Note: Read and Level 2 oplocks may coexist on the same stream, and Read and Read-Handle oplocks may coexist, but Level 2 and Read-Handle oplocks may not coexist.

2.3 Breaking Oplocks

After an oplock is granted, the owner of that oplock has access to the stream (based on the type of oplock that was requested). If the operation received is not compatible with the current oplock, the oplock is broken.

When an oplock is granted, the requesting IRP is pended. When an oplock is broken, the pended oplock request IRP is completed with STATUS_SUCCESS. For Level 1, Batch, and Filter oplocks the **IoStatus.Information** member of the IRP is set to indicate the level to which the oplock is breaking. These levels are:

- FILE_OPLOCK_BROKEN_TO_NONE – The oplock was broken and there is no current oplock on the stream. The oplock is said to have been "broken to None".
- FILE_OPLOCK_BROKEN_TO_LEVEL_2 – The current oplock (Level 1 or Batch) was converted to a Level 2 oplock. Note that Filter oplocks never break to Level 2, they always break to None.

For Read-Handle, Read-Write, and Read-Write-Handle oplocks, the level to which the oplock is breaking is described as a combination of zero or more of the flags OPLOCK_LEVEL_CACHE_READ, OPLOCK_LEVEL_CACHE_HANDLE, or OPLOCK_LEVEL_CACHE_WRITE in the **NewOplockLevel** member of the REQUEST_OPLOCK_OUTPUT_BUFFER structure passed as the *lpOutBuffer* parameter of **DeviceIOControl** In a similar manner, **FltFsControlFile** and **ZwFsControlFile** can be used to request Windows 7 oplocks from kernel mode. For more information, see FSCTL_REQUEST_OPLOCK.

When breaking a Level 1, Batch, Filter, Read-Handle, Read-Write, or Read-Write-Handle oplock, the pended oplock request IRP is completed by the oplock package and the operation that caused the oplock break is itself pended (note that if the operation is issued on a synchronous handle, or it is an IRP_MJ_CREATE, which is always synchronous, the I/O manager causes the operation to block, rather than return STATUS_PENDING), waiting for an acknowledgement from the owner of the oplock to tell the oplock package that they have finished their processing and it is safe for the pended operation to proceed. The purpose of this delay is to allow the owner of the oplock to put the stream back into a consistent state before the current operation proceeds. The system waits forever to receive the acknowledgement as there is no timeout. It is therefore incumbent on the owner of the oplock to acknowledge the break in a timely manner. The pended operation's IRP is set into a cancelable state. If the application or driver performing the wait terminates, the oploack package immediately completes the IRP with STATUS_CANCELLED.

An IRP_MJ_CREATE IRP may specify the FILE_COMPLETE_IF_OPLOCKED create option to avoid being blocked as part of oplock break acknowledgement. This option tells the oplock package *not* to block the create IRP until the oplock break acknowledgement is received. Instead, the create is allowed to proceed. If a successful create results in an oplock break, the return code is

STATUS_OPLOCK_BREAK_IN_PROGRESS, rather than STATUS_SUCCESS. The FILE_COMPLETE_IF_OPLOCKED flag is typically used to avoid deadlocks. For example, if a client owns an oplock on a stream and the same client subsequently opens the same stream, the client would block waiting for itself to acknowledge the oplock break. In this scenario, use of the FILE_COMPLETE_IF_OPLOCKED flag avoids the deadlock.

Because the file system initiates oplock breaks for Batch and Filter oplocks before checking for sharing violations, it is possible for a create that specified FILE_COMPLETE_IF_OPLOCKED to fail with STATUS_SHARING_VIOLATION but still cause a Batch or Filter oplock to break. In this case, the information member of the IO_STATUS_BLOCK structure is set to FILE_OPBATCH_BREAK_UNDERWAY to allow the caller to detect this case.

For Read-Handle and Read-Write-Handle oplocks, the oplock break is initiated *after* the file system checks for *and detects* a sharing violation. This gives holders of these oplocks the opportunity to close their handles and get out of the way, thus allowing for the possibility of not returning the sharing violation to the user. It also avoids unconditionally breaking the oplock in cases where the handle that the oplock caches does not conflict with the new create.

When Level 2 and Read (shared) oplocks break, the system does not wait for an acknowledgement. This is because there should be no cached state on the stream that needs to be restored to the file before allowing other clients access to it.

There are certain file system operations which check the current oplock state to determine if the oplock needs to be broken. The following sections list each operation and describe what triggers an oplock break, what determines the level to which the oplock breaks, and whether an acknowledgement of the break is required:

- IRP_MJ_CREATE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_CLEANUP
- IRP_MJ_LOCK_CONTROL
- IRP_MJ_SET_INFORMATION
- IRP_MJ_FILE_SYSTEM_CONTROL

A break of a Windows 7 oplock requires an acknowledgement if the REQUEST_OPLOCK_OUTPUT_FLAG_ACK_REQUIRED flag is set in the **Flags** member of the REQUEST_OPLOCK_OUTPUT_BUFFER structure passed as the output parameter of **DeviceIOControl** (*lpOutBuffer*), **FltFsControlFile** (*OutBuffer*) or **ZwFsControlFile** (*OutBuffer*). For more information, see FSCTL_REQUEST_OPLOCK.

2.3.1 IRP_MJ_CREATE

The following only applies when an existing stream of a file is being opened (that is, newly created streams cannot have pre-existing oplocks on them).

Note: When processing IRP_MJ_CREATE for any oplock, if the desired access contains nothing other than FILE_READ_ATTRIBUTES, FILE_WRITE_ATTRIBUTES, or SYNCHRONIZE, the oplock does not break unless FILE_RESERVE_OPFILTER is specified. Specifying FILE_RESERVE_OPFILTER always results in an oplock break if the create succeeds. For brevity and simplicity, the following table omits the foregoing, since it applies to all oplocks.

Request Type	Conditions
Level 1	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ○ The FILE_RESERVE_OPFILTER flag is set
	  OR
	  ○ Any of the following create disposition values are specified:
	  ○ FILE_SUPERSEDE
	  ○ FILE_OVERWRITE
	  ○ FILE_OVERWRITE_IF
	ELSE
	  ○ Break to Level 2.
	  ○ An acknowledgement must be received before the operation continues.

Level 2	Broken on IRP_MJ_CREATE when:
	• Any of the following create disposition values are specified:
	  ○ FILE_SUPERSEDE
	  ○ FILE_OVERWRITE
	  ○ FILE_OVERWRITE_IF

	If the oplock is broken: • Break to None. • No acknowledgement is required, the operation proceeds immediately.
Batch	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ○ The FILE_RESERVE_OPFILTER flag is set.
	  OR
	  ○ Any of the following create disposition values are specified:
	  ○ FILE_SUPERSEDE

	○ FILE_OVERWRITE ○ FILE_OVERWRITE_IF **ELSE** ○ Break to Level 2. ○ An acknowledgement must be received before the operation continues.
Filter	Broken on IRP_MJ_CREATE when: • A "writable" desired access was requested on the stream which was not opened for FILE_SHARE_READ access. Note that "writeable" access is defined as any attribute other than: ○ FILE_READ_ATTRIBUTES ○ FILE_WRITE_ATTRIBUTES ○ FILE_READ_DATA ○ FILE_READ_EA ○ FILE_EXECUTE ○ SYNCHRONIZE ○ READ_CONTROL **AND** • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.
	If the oplock is broken: • Break to None. • An acknowledgement must be received before the operation continues.
Read	Broken on IRP_MJ_CREATE when: • Any of the following create disposition values are specified: ○ FILE_SUPERSEDE ○ FILE_OVERWRITE ○ FILE_OVERWRITE_IF **AND** • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgement is required, the operation proceeds immediately.
Read-Handle	Broken on IRP_MJ_CREATE when: • The current open conflicts with an existing open such that a sharing violation would occur. **OR** • Any of the following create disposition values are specified:

	○ FILE_SUPERSEDE ○ FILE_OVERWRITE ○ FILE_OVERWRITE_IF **AND (for either of the above two conditions)** • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.
	If the oplock is broken: • Break to None **IF**: ○ The FILE_RESERVE_OPFILTER flag is set. **OR** ○ Any of the following create disposition values are specified: ○ FILE_SUPERSEDE ○ FILE_OVERWRITE ○ FILE_OVERWRITE_IF **ELSE** ○ Break to Read. ○ An acknowledgement must be received before the operation continues.
Read-Write	Broken on IRP_MJ_CREATE when: • Any of the following create disposition values are specified: ○ FILE_SUPERSEDE ○ FILE_OVERWRITE ○ FILE_OVERWRITE_IF **AND** • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.
	If the oplock is broken: • Break to None **IF**: ○ The FILE_RESERVE_OPFILTER flag is set. **OR** ○ Any of the following create disposition values are specified: ○ FILE_SUPERSEDE ○ FILE_OVERWRITE ○ FILE_OVERWRITE_IF **ELSE** ○ Break to Read. ○ An acknowledgement must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_CREATE when: • The current open conflicts with an existing open such that a sharing violation would occur.

	OR
	• Any of the following create disposition values are specified:
	  ○ FILE_SUPERSEDE
	  ○ FILE_OVERWRITE
	  ○ FILE_OVERWRITE_IF
	AND (for either of the above two conditions)
	• The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ○ The FILE_RESERVE_OPFILTER flag is set.
	  OR
	  ○ Any of the following create disposition values are specified:
	  ○ FILE_SUPERSEDE
	  ○ FILE_OVERWRITE
	  ○ FILE_OVERWRITE_IF
	ELSE
	  ○ Break to Read-Write if the current open conflicts with an existing open such that a sharing violation would occur. Otherwise, break to Read-Handle.
	  ○ An acknowledgement must be received before the operation continues.

The file system performs additional checks for Batch and Filter oplocks (rather than the oplock package itself) when processing an IRP_MJ_CREATE operation, which impact whether the file system asks the oplock package to perform oplock break processing. This is a case where operations on one data stream can impact the oplocks on other data streams of the same file (that is,the last two list items of the following criteria list). If one or more of the following criteria are met, the file system asks the oplock package to perform oplock break processing:

Request a break if this is a network query open *and* a KTM transaction is present. Otherwise, do not request a break on network query open.

If a SUPERSEDE, OVERWRITE or OVERWRITE_IF operation is performed on an alternate data stream and FILE_SHARE_DELETE is not specified and there is a Batch or Filter oplock on the primary data stream, request a break of the Batch or Filter oplock on the primary data stream.

If a SUPERSEDE, OVERWRITE or OVERWRITE_IF operation is performed on the primary data stream and DELETE access has been requested and there are Batch or Filter oplocks on any alternate data stream, request a break of the Batch or Filter oplocks on all alternate data streams that have them.

When the file system decides to ask the oplock package to perform oplock break processing, the rules laid out in the preceding table apply.

The check to break Batch and Filter oplocks occurs before the share access checks are made. This means the Batch or Filter oplock is broken even if the open request ultimately fails due to a sharing violation.

2.3.2 IRP_MJ_READ

The following only applies when a *stream* is being read. If a TxF transacted reader performs the read, this check is not made since a transacted reader excludes a writer (that is, a writer holding an oplock cannot be present at all).

Request Type	Conditions
Level 1 Batch	Broken on IRP_MJ_READ when: • The read operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Level 2. • An acknowledgement must be received before the operation continues.
Read-Write	Broken on IRP_MJ_READ when: • The read operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read. • An acknowledgement must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_READ when: • The read operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read-Handle. • An acknowledgement must be received before the operation continues.
Level 2 Filter Read Read-Handle	• The oplock is not broken, no acknowledgement is required, and the operation proceeds immediately.

2.3.3 IRP_MJ_WRITE

The following only applies when a *stream* is being written and the write is not a paging I/O.

Request Type	Conditions

Level 1 Batch	Broken on IRP_MJ_WRITE when: • The write operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
Filter Read-Handle Read-Write Read-Write-Handle	If the oplock is broken: • Break to None. • An acknowledgement must be received before the operation continues.
Read	Broken on IRP_MJ_WRITE when: • The write operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgement is required, the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgement is required, the operation proceeds immediately.

2.3.4 IRP_MJ_CLEANUP

The following only applies when a *stream* is being closed.

Request Type	Conditions
Level 1 Batch Filter Read-Handle Read-Write Read-Write-	• Always break to None. • No acknowledgement is required, the operation proceeds immediately. Note that any I/O operations (IRPs) waiting for an acknowledgement from a pending break request are completed immediately.

Handle	
Level 2 Read	• Always break to None. Note that other Level 2 or Read oplocks on the same stream are not affected; only the Level 2 or Read oplock associated with this FILE_OBJECT is broken. • No acknowledgement is required, the operation proceeds immediately.

2.3.5 IRP_MJ_LOCK_CONTROL

The following applies on every byte range lock operation on the given stream.

Request Type	Conditions
Level 1 Batch	Broken on IRP_MJ_LOCK_CONTROL when: • The lock operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
Read-Handle Read-Write Read-Write-Handle	If the oplock is broken: • Break to None. • An acknowledgement must be received before the operation continues.
Read	Broken on IIRP_MJ_LOCK_CONTROL when: • The lock operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgement is required, the operation proceeds immediately.
Filter	• The oplock is not broken, no acknowledgement is required, and the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgement is required, the operation proceeds immediately.

2.3.6 IRP_MJ_SET_INFORMATION

Certain IRP_MJ_SET_INFORMATION operations check oplock state. The following six operations perform this check:

2.3.6.1 FileEndOfFileInformation and FileAllocationInformation

This information applies when the following operations are being performed on a file or stream:

- A caller attempts to change the logical size of the stream. Note that when the cache manager's lazy writer thread attempts to set a new end of file, no oplock check is made. This is because the check is made previously when the real write request is received.
- A caller attempts to change the allocated size of the stream.

Request Type	Conditions
Level 1 Batch	Broken on IRP_MJ_SET_INFORMATION (for FileEndOfFileInformtion and FileAllocationInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
Filter Read-Handle Read-Write Read-Write-Handle	If the oplock is broken: • Break to None. • An acknowledgement must be received before the operation continues.
Read	Broken on IRP_MJ_SET_INFORMATION (for FileEndOfFileInformtion and FileAllocationInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgement is required, the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgement is required, the operation proceeds immediately.

2.3.6.2 FileRenameInformation, FileShortNameInformation, and FileLinkInformation

This information applies when the following operations are being performed on a file or stream:

- The file or stream is being renamed.
- A short name is being set for the file.
- A hard link is being created for the file. This affects oplock state if the new hard link is superseding an existing link to a different file, and the oplock exists on the link being superseded.
- An ancestor directory of the stream on which the oplock exists is being renamed, or the ancestor directory's short name is being set.

Request Type	Conditions

Batch Filter	Broken on IRP_MJ_SET_INFORMATION (for FileRenameInformation, FileShortNameInformation, and FileLinkInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • An acknowledgement must be received before the operation continues.
Read-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileRenameInformation, FileShortNameInformation, and FileLinkInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read. • An acknowledgement must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileRenameInformation, FileShortNameInformation, and FileLinkInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read-Write. • An acknowledgement must be received before the operation continues.
Level 1 Level 2 Read Read-Write	• The oplock is not broken, no acknowledgement is required, and the operation proceeds immediately.

2.3.6.3 FileDispositionInformation

This information applies when a caller tries to delete the file.

Request Type	Conditions
Read-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileDispositionInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.

	AND • FILE_DISPOSITION_INFORMATION.DeleteFile is TRUE.
	If the oplock is broken: • Break to Read. • An acknowledgement must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileDispositionInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock. **AND** • FILE_DISPOSITION_INFORMATION.DeleteFile is TRUE.
	If the oplock is broken: • Break to Read-Write. • An acknowledgement must be received before the operation continues.

2.3.7 IRP_MJ_FILE_SYSTEM_CONTROL

The FSCTL_SET_ZERO_DATA file system control code operation checks oplock state:

2.3.7.1 FSCTL_SET_ZERO_DATA

This information applies when a caller wants to zero the current contents of the given stream.

Request Type	Conditions
Level 1 Batch	Broken on IRP_MJ_FILE_SYSTEM_CONTROL (for FSCTL_SET_ZERO_DATA) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
Filter Read-Handle Read-Write Read-Write-Handle	If the oplock is broken: • Break to None. • An acknowledgement must be received before the operation continues.
Read	Broken on IRP_MJ_FILE_SYSTEM_CONTROL (for FSCTL_SET_ZERO_DATA) when: • The operation occurs on a FILE_OBJECT with a different oplock key

	from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgement is required, the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgement is required, the operation proceeds immediately.

2.4 Acknowledging Oplock Breaks

There are different types of acknowledgements that the owner of an oplock can return. These acknowledgements are sent as file system control codes (that is, FSCTLs). They are:

- FSCTL_OPLOCK_BREAK_ACKNOWLEDGE
  - This FSCTL indicates that the oplock owner has completed stream synchronization and they accept the level to which the oplock was broken (either Level 2 or None).
- FSCTL_OPLOCK_BREAK_ACK_NO_2
  - This FSCTL indicates that the oplock owner has completed stream synchronization but does not want a Level 2 oplock. Instead, the oplock should be broken to None (that is, the oplock is to be relinquished entirely).
- FSCTL_OPBATCH_ACK_CLOSE_PENDING
  - For a Level 1 oplock, this FSCTL indicates that the oplock owner has completed stream synchronization and is relinquishing the oplock entirely (no Level 2 oplock may result from this acknowledgement).
  - For a Batch or Filter oplock, this FSCTL indicates that the oplock owner intends to close the stream handle on which the oplock was granted. Operations blocked, awaiting acknowledgement of the oplock break, continue to wait until the oplock owner's handle is closed.
- FSCTL_REQUEST_OPLOCK
  - By specifying REQUEST_OPLOCK_INPUT_FLAG_ACK in the **Flags** member of the REQUEST_OPLOCK_INPUT_BUFFER structure passed as the **lpInBuffer** parameter of **DeviceIOControl**, this FSCTL is used to acknowledge breaks of Windows 7 oplocks. The acknowledgement is required only if the REQUEST_OPLOCK_OUTPUT_FLAG_ACK_REQUIRED flag is set in the **Flags** member of the REQUEST_OPLOCK_OUTPUT_BUFFER structure passed as the *lpOutBuffer* parameter of **DeviceIoControl**. In a similar manner, **FltFsControlFile** and **ZwFsControlFile** can be used to acknowledge Windows 7 oplocks from kernel-mode. For more information, see FSCTL_REQUEST_OPLOCK.

A related FSCTL code is FSCTL_OPLOCK_BREAK_NOTIFY. This code is used when the caller wants to be notified when an oplock break on the given stream completes. This call may block. When the FSCTL_OPLOCK_BREAK_NOTIFY call returns STATUS_SUCCESS, this signifies one of the following:

- No oplock granted.
- No oplock break was in progress at the time of the call.
- Any oplock break that was in progress is now complete.

To send an acknowledgement when no acknowledgement is expected is an error and the acknowledgement FSCTL IRP is failed with STATUS_INVALID_OPLOCK_PROTOCOL.

2.5 Differences between the various file systems in Windows:

1. NTFS supports all the oplock semantics discussed above
2. FAT/exFAT/UDF and CDFS file systems support the legacy Level 1, Level 2, Batch, and Filter oplock semantics
3. FAT supports the new FILE_OPEN_REQUIRING_OPLOCK **NtCreateFile** create option, and the R, RH, RW and RWH oplock levels in Windows 7.
4. exFAT only supports the new FILE_OPEN_REQUIRING_OPLOCK **NtCreateFile** create option.

3 Byte Range Lock Semantics

3.1 Overview

Byte range locks are mechanisms that allow a user-mode application to lock a region (or range) of a file's data stream, allowing multiple user-mode processes to have synchronized access to those regions. They are useful for applications that require granular locking mechanisms in a single file, like some database applications.

It is important to note that for windows file systems Byte Range Locks operate on a data stream. The concept of byte range locks in Windows is very similar to the one found in the POSIX standard, and, therefore, the UNIX operating system as well. There is however some differences in the way they work in Windows compared to POSIX.

3.2 Functional Description

Ranges in a stream are defined by an offset, length pair. Offset and length are 64-bit unsigned integers. This way of defining byte range locks is used both in the Win32 and the native NT APIs (but in the Win32 APIs each of the 64-bit integers are split into 2 32-bit ones).

There are two types of byte range locks: shared and exclusive. Shared locks are also known as 'read' locks, whereas exclusive locks are also known as 'write' locks. This illustrates their purpose: shared locks can be acquired by more than one different process at the same range and prevent write operations on that range (no process can write on it, not even the first process to acquire a lock on the region, even if it's the single process locking the region), allowing all processes (not only locking processes) to read it consistently. On the other hand, exclusive locks are only granted to a single process at a time for a given range, disallowing other processes reads and writes in that range, allowing the owner to write it or update it with the guarantee no inconsistent data is being read from or written to it.

One key aspect of Windows byte range locks is the fact they are mandatory instead of only advisory. This means they are enforced by file systems during read and write operations, preventing those operations from occurring on ranges where they conflict with a lock that's in place[2]. However, byte range locks are not enforced when using file mapping APIs.

Locking operations can either wait or fail immediately. If the operation is requested to fail immediately, the API call will return `STATUS_LOCK_NOT_GRANTED` (or the equivalent Win32 API error code) when it conflicts with a range previously locked by another process. However, in case a wait is desired, the call will cause the thread to block, waiting for the conflicting range to be unlocked. If the stream handle was opened for asynchronous access, the call actually returns immediately with `STATUS_PENDING` (or Win32 `ERROR_IO_PENDING`) so the application has to explicitly check for operation completion later or explicitly wait for its completion.

[2] Contrast that with POSIX byte range locks, which are advisory: this means file systems allow any operation in a file range even if there's a lock in place. Therefore, applications need to be polite and explicitly check for existing locks in a range before reading it or writing to it.

Byte range locks are not persistent, which means all information about which ranges are locked and which are not stays exclusively in memory and is never recorded on disk by file systems. They are also logical, which means it is possible to lock a stream in a range that extends past the end of file or even a range that exists completely outside of the stream's actual size.

Byte range locks are removed by either an explicit unlock operation or when the last handle to a given stream is closed. All open handles are automatically closed by the system when an application terminates. This means any outstanding byte range locks will also be cleared.

There are three types of unlock operations. The first unlocks a single range of the stream (`IRP_MN_UNLOCK_SINGLE`), and it's the only one that's exposed via the Win32 and native NT APIs. It takes a range as argument, and this range must exactly match a locked range, so it's not possible to unlock two adjacent ranges by issuing a single unlock operation that spans both, for instance. It is also not possible to unlock only part of a range that was previously locked with a single operation[3]. The second operation unlocks all locks of a stream owned by a given process (`IRP_MN_UNLOCK_ALL`), and the third unlocks all locks of a stream owned by a given process and having a given key (`IRP_MN_UNLOCK_ALL_BY_KEY`). These later two operations are used internally by drivers and the I/O manager to do close and cleanup work.

Ownership of byte range locks is determined by a unique combination of process ID, file object (or handle) and lock key. This has a few important consequences: first, a process will have no access to regions locked by itself if it tries to access them via a handle different than the one used to lock those regions. Also, child processes inheriting a file handle will have no access to regions locked by the parent process (which also happens in POSIX).

3.2.1 Range Semantics

Ranges used in Windows are defined by an offset, length pair, with absolute offsets, which means they start at the beginning of the stream, which is almost like in POSIX, except for the fact that in POSIX offsets can be relative to a file cursor as well (and Windows has no file cursors in its stream handles). Windows ranges also differ from POSIX in at least one very important way, which must be noted because it is a departure from a very common use of POSIX byte range locks, Windows[4] supports zero-length ranges.

Zero-length byte range locks actually affect no single byte of the stream, meaning they can still be accessed even with such locks in place. However, they do conflict with other ranges in the following manner: given a definition of a range as R = {O, L}, where O is the offset and L the length, a zero-length range R_0 = {N, 0} will conflict with range R_1 = {X, Y} if and only if X < N and X+Y > N.

[3] This, however, works with POSIX locks.

[4] In POSIX, a length of zero is treated very differently: it actually means the range extends all the way to the end of file, which means in practice POSIX has no zero-length ranges. This is worth mentioning because using a length of zero with POSIX byte range locks is actually a very common practice, which obviously wouldn't work at all in Windows.

That is perhaps better understood by the following verbose rule: “a conflict will only exist if the positive-length range contains the zero-length range’s offset but doesn’t start at it.” As an example, the ranges R_2 = {5, 10} and R_3 = {6, 0} will conflict with each other, because byte 6 is contained within R_2, and R_2 does not start at byte 6. For another example, range R_4 = {6, 10} will not conflict with R_3 because even though R_4 contains byte 6, it starts at it.

The description of zero-length ranges might give the idea that it may be possible to request a lock with range {0, 0} and it won’t conflict with any lock (as no range can have an offset less than zero). However, this is not true and the use of {0, 0} locks is extremely discouraged because Windows is not prepared to deal with them, leading to unexpected behavior[5].

Similarly, ranges that extend past the 64 bit limit (that is, adding offset + length overflows 64 bits) must not be used as Windows is not prepared to deal with them and they lead to highly unexpected and inconsistent behavior[6].

3.3 Processing Details

The most common way of locking a range in a stream is a call to the Win32 API functions `LockFile()` or `LockFileEx()`. Those functions actually execute the lock request by calling the native NT API function, `NtLockFile()`, with the correct parameters, one of them being a key value of zero. Key values are just 32-bit integers that are used, together with the file object and the process ID, to uniquely identify a lock owner. They are not exposed by the Win32 API and are mostly used internally by remote file systems to differentiate between multiple remote clients.

The lock request is then passed down to the file system. If the file system defines a fast I/O callback, the fast I/O path is used for this operation; otherwise the traditional IRP path is used. If a lock request made via the fast I/O path can’t be immediately satisfied and is not set to fail immediately, then the fast I/O call fails and the request is retried via the IRP path since an IRP can be pended and queued for later processing.

With unlocking operations the mechanism is similar but simpler: fast I/O if the file system driver defines the callback for the corresponding operation type, with the IRP path being used either if there is no such callback or if the fast I/O call fails. The simplicity comes from the fact that those requests always return immediately, that is, they are never set to a pending state.

3.3.1 Lock Operation IRPs

The IRP major and minor function values for all byte range lock operations are illustrated on the following table:

#	Operation	Major Function	Minor Function
1	Lock	`IRP_MJ_LOCK_CONTROL`	`IRP_MN_LOCK`
2	Unlock Single Range		`IRP_MN_UNLOCK_SINGLE`

[5] This is fixed in Windows 7, but it doesn’t make {0, 0} locks any more useful so their use is still discouraged.
[6] Also fixed in Windows 7, which will ensure locks in those ranges always fail, with `STATUS_INVALID_LOCK_RANGE`.

3	Unlock All Ranges	IRP_MN_UNLOCK_ALL
4	Unlock All Ranges By Key	IRP_MN_UNLOCK_ALL_BY_KEY

The next table shows the IRP and IO_STACK_LOCATION fields containing the parameters for all the lock operations:

Parameter	Field in IO_STACK_LOCATION	Used by
File Object	FileObject	1, 2, 3, 4
Offset	Parameters.LockControl.ByteOffset	1, 2
Length	Parameters.LockControl.Length	1, 2
Process	There is no field. The process is obtained from the IRP by IoGetRequestorProcess(Irp)	1, 2, 3, 4
Key	Parameters.LockControl.Key	1, 2, 4
Fail Immediately?	SL_FAIL_IMMEDIATELY set on Flags	1
Exclusive Lock?	SL_EXCLUSIVE_LOCK set on Flags	1

3.3.2 Lock Operation Fast I/O Callbacks

Every lock operation has a fast I/O path that can be implemented by the file system (NTFS implements them). The table below shows the callback fields in the driver object corresponding to every lock operation:

#	Operation	Callback in DRIVER_OBJECT
1	Lock	FastIoDispatch->FastIoLock
2	Unlock Single Range	FastIoDispatch->FastIoUnlockSingle
3	Unlock All Ranges	FastIoDispatch->FastIoUnlockAll
4	Unlock All Ranges By Key	FastIoDispatch->FastIoUnlockAllByKey

Parameters for those operations are the exact same as the ones used for them in the IRP path, except they are passed as function arguments.

3.4 Side Effects of Byte Range Locks

The presence of byte range longs on a given stream impact how other operations operate on that stream. The operations affected are:

- Read (IRP_MJ_READ)
- Write (IRP_MJ_WRITE)
- Byte Range Locks
- Oplocks
- Handle Close (IRP_MJ_CLEANUP)

3.4.1 Reads and Writes

The obvious effect of byte range locks on read and write operations is the prevention of those operations in case they happen in a range that conflicts with a lock (since Windows byte range locks are

mandatory). In this case, the read (not issued by the owner and falling inside a range locked exclusive) or write operation (that either falls inside a range locked shared or falls inside an exclusive range that the process doesn't own) will fail with `STATUS_FILE_LOCK_CONFLICT`.

There is a less obvious implication of byte range locks on these operations: the existence of byte range locks on a stream will mark it for questionable fast I/O (if not already marked as questionable or even flagged for fast I/O not possible). This will cause further reads and writes on that stream to always go through the IRP path if they conflict with a lock in place (as fast reads and writes don't execute if fast I/O is questionable and the range conflicts with a lock).

3.4.2 Byte Range Locks

Existing byte range locks on a stream will affect further lock operations if they conflict with existing locks. It is not possible to lock ranges for exclusive access if they conflict with existing locks. In these cases, the operations will fail with `STATUS_LOCK_NOT_GRANTED` or, depending on the parameters (the `FailImmediately` argument from `NtLockFile()` or the `LOCKFILE_FAIL_IMMEDIATELY` flag on the Win32 API `LockFileEx()`), queued for later processing when the conflicting locks are unlocked, releasing those queued locks. Locking a range for shared access will succeed if the range only overlaps the ranges of other shared locks, or if it overlaps a range locked for exclusive access with the same ownership. This means a process can lock a range for exclusive access, then, using the same file handle (same ownership), can lock the same range or portions of it for shared access, and that shared lock will be enforced, meaning the process cannot write to them, not even using the same handle, even though it still has an exclusive lock on them.

An attempt to unlock a byte range that's not locked (meaning it doesn't match exactly a locked range), or one the current process does not own will fail with STATUS_RANGE_NOT_LOCKED. In the case that a range is locked for both exclusive and shared access by the same owner, the unlock operation will unlock the exclusive lock. A second unlock operation will then unlock the shared lock.

3.4.3 Oplocks

The acquisition of a byte range lock breaks Level1 and Batch oplocks to NONE if they occur on a handle different than the handle owning the oplock. Level 2 (or shared) oplocks are always broken to NONE. Filter oplocks are never broken by specifying a byte range lock.

Once a byte range lock is granted for a given stream, that stream will fail all subsequent Level 2 oplock requests until all existing handles for that stream are closed. Batch and Level 1 oplocks will be granted if requested using the handle that owns the byte range lock.

3.4.4 Handle Close

When the last handle to a stream for a given process is closed, but the stream is still open by other processes, all byte range locks on that stream that are owned by that process are unlocked by the I/O Manager sending an IRP_MN_UNLOCK_ALL request to the file system. No IRP_MJ_CLEANUP operation is generated at this time.

When the last handle to a stream for the entire system is closed, the I/O Manager sends an IRP_MJ_CLEANUP operation to the file system. The file system must remove any remaining byte range locks on that stream as part of its IRP_MJ_CLEANUP processing.

4 File Deletion Semantics

4.1 Overview:

Some Windows file systems, including NTFS, support hard links, which associate multiple names in the directory hierarchy with the same file data. Additionally, some file systems support multiple data streams in a single file. For the purpose of this document the terms "link" and "stream" will be used respectively to refer a hard link and a named data stream. For file systems, such as FAT and CDFS, that do not support hardlinks and alternate data streams, there is one and only one link to the file and a single data stream which is also the default data stream.

Link and stream deletion in Windows are implemented using a delete on close semantic. Broadly this means that deletion occurs when all handles to a link or stream are closed and not when the link or stream is initially marked for deletion.

4.1.1 Definition of Terms

- **Delete-on-close state**: A handle reaches this state if it opened the link or stream with the DELETE_ON_CLOSE flag, but has not yet been closed. Additional handles to the link or stream can be opened during this state as long as they specify FILE_SHARE_DELETE access. We transition the link or stream to the **delete-pending state** when the handle is closed.
- **Delete-pending state**: We reach this state in one of two ways:
    - A handle was closed that specified the DELETE_ON_CLOSE flag at open/create time.
    - A set file information call was made using the *FileDispositionInformation* information class with the *DeleteFile* field of the FILE_DISPOSITION_INFORMATION structure set to TRUE. While in this state attempts to open additional handles to the link or stream fail with the error STATUS_DELETE_PENDING.
- **Open-by-ID:** In addition to opening a file by its path and name, file systems like NTFS supports the ability to open files by an ID number. The system supports two different ID number values:
    - FileID – this is a unique 64-bit number generated by the file system.
    - ObjectID – this is a unique 128-bit number (or GUID) which can be assigned to a file (FSCTL_SET_OBJECT_ID). This is guaranteed to be unique on the given volume. This capability is supported by NTFS only.

4.2 Summary

There are two ways to mark a link or stream for deletion:

- Set the DELETE_ON_CLOSE flag when creating or opening the file (IRP_MJ_CREATE)
- Set the *DeleteFile* field in the FILE_DISPOSITION_INFORMATION structure to TRUE when invoking the *FileDispositionInformation* file information class (IRP_MJ_SET_INFORMATION)

There is one way to unmark a link or stream for deletion:

- Set the *DeleteFile* field in the FILE_DISPOSITION_INFORMATION structure to FALSE when invoking the *FileDispositionInformation* file information class (IRP_MJ_SET_INFORMATION)

There are two ways to query whether deletion is pending on a link or stream

- Examine the *DeletePending* field in the FILE_STANDARD_INFORMATION structure after invoking the *FileStandardInformation* file information class (IRP_MJ_QUERY_INFORMATION)
- Examine the *DeletePending* field in the FILE_STANDARD_INFORMATION structure after invoking the *FileAllInformation* file information class (IRP_MJ_QUERY_INFORMATION)

When all handles to a file object for a link or stream are closed, a cleanup IRP (IRP_MJ_CLEANUP) is sent to the file system which, in certain cases (detailed below), will trigger the deletion of the file data, link or stream.

4.3 Detailed Description

4.3.1 Set Delete-on-close during Open/Create (IRP_MJ_CREATE)

The delete on close flag can be specified when creating or opening a link or stream. While the handle remains open, the handle is in the **delete-on-close state**. This means that additional handles can be opened to the file or stream provided they specify FILE_SHARE_DELETE access. After receiving the cleanup IRP for a handle that specified DELETE_ON_CLOSE, the handle's corresponding link or stream enters the **delete-pending state**. While in this state any attempt to open the link or stream will fail with STATUS_DELETE_PENDING.

Note that if the DELETE_ON_CLOSE flag is specified on a directory with child files or directories, the create operation will succeed, but the delete on close flag will be silently ignored when processing the cleanup IRP and the directory will not be deleted.

The create operation can fail marking the file delete on close for any of the following reasons:

- The file is marked read-only – STATUS_CANNOT_DELETE
- The volume is marked read-only – STATUS_CANNOT_DELETE
- The file backs an image section – STATUS_CANNOT_DELETE
- The link or stream is already in the **delete-pending state** – STATUS_DELETE_PENDING

4.3.2 Set Delete-on-close using *FileDispositionInformation* Information Class (IRP_MJ_SET_INFORMATION)

The FileDispositionInformation information class can be used to mark a link or stream for deletion and transition it to the **delete-pending state**. In this state attempts to open the link or stream will fail with STATUS_DELETE_PENDING.

This operation can fail for any of the following reasons:

- The handle was opened by ID – STATUS_INVALID_PARAMTER
- The file is an internal file system metadata file – STATUS_CANNOT_DELETE

- The file is marked read-only – STATUS_CANNOT_DELETE
- The volume is marked read-only _- STATUS_MEDIA_WRITE_PROTECTED or STATUS_ACCESS_DENIED (on CDFS & UDF file systems when the file system is read-only)
- The handle is to a directory that still has files or directories in it – STATUS_DIRECTORY_NOT_EMTPY

4.3.3 Clear Delete-on-close using *FileDispositionInformation* Information Class (IRP_MJ_SET_INFORMATION)

This operation will remove the delete on close flag from any link or stream that is in the **delete-pending state**. This will cause the link or stream to no longer be deleted when its last handle is closed.

Note that if a handle is marked delete on close during create and the handle has not been closed, clearing the delete on close flag will have no effect. The link or stream will still be set to the **delete-pending state** when the handle is closed and thus ultimately deleted.

This operation can fail for the following reason:

- The handle was opened by ID – STATUS_INVALID_PARAMETER

4.3.4 Query Delete-pending using *FileStandardInformation* or *FileAllInformation* information classes (IRP_MJ_QUERY_INFORMATION)

This will return TRUE in the DeletePending field if the link or stream is in the **delete-pending state** and FALSE otherwise.

Note that the link or stream will be in the **delete-on-close state**, and DeletePending will be set to FALSE, if no handle that was opened with the delete on close flag specified has gone through cleanup (provided the **delete-pending state** has not been explicitly set through IRP_MJ_SET_INFORMATION).

4.3.5 Closing a Handle (IRP_MJ_CLEANUP)

When a cleanup operation is sent by the IO Manager, the file system will do the following to process link and stream deletions:

If the handle is in the **delete-on-close state** the link or stream will be transitioned to the **delete-pending state.**

If the link or stream is in the **delete-pending state** (including the case where it was just transitioned to this state) decisions will be made using the following decision tree.

- If the handle was opened via the file or default data stream name then the file system first checks if any other handles have been opened to that link or any of its data streams.
    - If there are other open handles, the file system does not delete the link or file data.
    - If this is the final handle to the link or stream:
        - If there are no other remaining links and there are no handles opened to the file then it deletes both the link from the namespace and the file's data contents

        - If there are no other remaining links, but there are handles to the file (**open-by-ID** handles in this case), then neither the link nor file data is deleted. The file system will delete the link and file data when the final handle is closed
        - If there are additional links (possible on NTFS & UDF) then the file system removes this name from the namespace and decrements the link count, but retains the file data
- If the handle was opened to a named data stream then the files system checks if there are any open handles to the given stream:
    - If there are then it does not delete the stream
    - If there aren't then it deletes the stream

5 IRP Return Codes

This section lists the common error codes for file system related IRP's. MS-ERREF contains the definitive list of system return codes and their descriptions.

This section is scoped to the NTFS file system at this time. A future version of this document will note differences for the other file systems.

5.1 IRP_MJ_CREATE

Return Code
STATUS_OBJECT_NAME_INVALID
STATUS_INVALID_PARAMETER
STATUS_DIRECTORY_IS_A_REPARSE_POINT
STATUS_VOLUME_DISMOUNTED
STATUS_VOLUME_NOT_UPGRADED
STATUS_CANCELLED
STATUS_SHARING_VIOLATION
STATUS_INVALID_HANDLE
STATUS_FILE_LOCK_CONFLICT
STATUS_INTERNAL_ERROR
STATUS_NOT_A_DIRECTORY
STATUS_OPLOCK_BREAK_IN_PROGRESS
STATUS_INVALID_OWNER
STATUS_MEDIA_WRITE_PROTECTED
STATUS_WAIT_FOR_OPLOCK
STATUS_PENDING
STATUS_UNABLE_TO_DELETE_SECTION
STATUS_SUCCESS
STATUS_OBJECT_PATH_NOT_FOUND
STATUS_UNEXPECTED_IO_ERROR
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_NOT_FOUND
STATUS_MFT_TOO_FRAGMENTED
STATUS_END_OF_FILE
STATUS_ACCESS_DENIED
STATUS_OPLOCK_NOT_GRANTED
STATUS_OBJECT_NAME_COLLISION
STATUS_DISK_CORRUPT_ERROR
STATUS_DISK_FULL
STATUS_EAS_NOT_SUPPORTED
STATUS_IO_REPARSE_DATA_INVALID
STATUS_INSUFFICIENT_RESOURCES
STATUS_FILE_IS_A_DIRECTORY

STATUS_FILE_DELETED
STATUS_DELETE_PENDING
STATUS_USER_MAPPED_FILE
STATUS_CANNOT_DELETE
STATUS_CANT_BREAK_TRANSACTIONAL_DEPENDENCY
STATUS_FILE_CORRUPT_ERROR
STATUS_FILE_SYSTEM_LIMITATION
STATUS_REPARSE

5.2 IRP_MJ_CLOSE

Return Code
STATUS_SUCCESS

5.3 IRP_MJ_READ

Return Code
STATUS_ACCESS_DENIED
STATUS_CANCELLED
STATUS_DISK_CORRUPT_ERROR
STATUS_DISK_FULL
STATUS_END_OF_FILE
STATUS_FILE_CLOSED
STATUS_FILE_CORRUPT_ERROR
STATUS_FILE_DELETED
STATUS_FILE_LOCK_CONFLICT
STATUS_FLOATED_SECTION
STATUS_INSUFFICIENT_RESOURCES
STATUS_INVALID_DEVICE_REQUEST
STATUS_INVALID_HANDLE
STATUS_INVALID_PARAMETER
STATUS_INVALID_USER_BUFFER
STATUS_PENDING
STATUS_REPARSE
STATUS_SHARING_VIOLATION
STATUS_SUCCESS
STATUS_STREAM_MINIVERSION_NOT_VALID
STATUS_UNEXPECTED_IO_ERROR
STATUS_UNSUCCESSFUL
STATUS_THREAD_IS_TERMINATING
STATUS_VOLUME_DISMOUNTED

5.4 IRP_MJ_WRITE

Return Code
STATUS_FILE_DELETED
STATUS_FILE_LOCK_CONFLICT

STATUS_INSUFFICIENT_RESOURCES
STATUS_INVALID_DEVICE_REQUEST
STATUS_PENDING
STATUS_REPARSE
STATUS_SUCCESS
STATUS_INVALID_PARAMETER
STATUS_FLOATED_SECTION
STATUS_FILE_CLOSED
STATUS_VOLUME_DISMOUNTED
STATUS_DISK_FULL
STATUS_UNEXPECTED_IO_ERROR
STATUS_INVALID_HANDLE
STATUS_FILE_CORRUPT_ERROR
STATUS_STREAM_MINIVERSION_NOT_VALID
STATUS_END_OF_FILE
STATUS_FILE_INVALID
STATUS_SHARING_VIOLATION
STATUS_ACCESS_DENIED
STATUS_VERIFY_REQUIRED
STATUS_UNEXPECTED_IO_ERROR
STATUS_BUFFER_TOO_SMALL
STATUS_DEVICE_DATA_ERROR
STATUS_MFT_TOO_FRAGMENTED
STATUS_CANCELLED
STATUS_THREAD_IS_TERMINATING
STATUS_TOO_LATE
STATUS_MEDIA_WRITE_PROTECTED
STATUS_TRANSACTIONAL_CONFLICT
STATUS_USER_MAPPED_FILE
STATUS_WAIT_FOR_OPLOCK
STATUS_NOT_IMPLEMENTED
STATUS_INVALID_USER_BUFFER

5.5 IRP_MJ_QUERY_INFORMATION

The table below lists the return codes that apply to queries of all information classes. In the subsections that follow information class specific return codes are also listed.

Return Code
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_HANDLE
STATUS_SUCCESS
STATUS_INSUFFICIENT_RESOURCES
STATUS_FILE_DELETED

STATUS_INVALID_DEVICE_REQUEST
STATUS_BUFFER_OVERFLOW

5.5.1 FileAccessInformation

Implemented locally by the I/O Manager, does not flow over the wire.

5.5.2 FileAlignmentInformation

Implemented locally by the I/O Manager, does not flow over the wire.

5.5.3 FileAllInformation

Return Code
STATUS_ACCESS_DENIED
STATUS_BUFFER_TOO_SMALL

5.5.4 FileAlternateNameInformation

Return Code
STATUS_OBJECT_NAME_NOT_FOUND

5.5.5 FileAttributeTagInformation

No additional information class specific return codes.

5.5.6 FileBasicInformation

No additional information class specific return codes.

5.5.7 FileCompressionInformation

Return Code
STATUS_FILE_CORRUPT_ERROR

5.5.8 FileEaInformation

(see IRP_MJ_QUERY_EA)

5.5.9 FileFullEaInformation

(not implemented by NTFS, SRV implements it, see SMB server documentation)

5.5.10 FileHardLinkInformation

Return Code
STATUS_INVALID_USER_BUFFER
STATUS_BUFFER_TOO_SMALL
STATUS_ACCESS_DENIED

5.5.11 FileInternalInformation

No additional information class specific return codes.

5.5.12 FileNameInformation

Return Code
STATUS_ACCESS_DENIED
STATUS_BUFFER_TOO_SMALL

5.5.13 FileNetworkOpenInformation

No additional information class specific return codes.

5.5.14 FilePositionInformation

No additional information class specific return codes.

5.5.15 FileSfioReserveInformation

Return Code
STATUS_BUFFER_TOO_SMALL

5.5.16 FileStandardInformation

No additional information class specific return codes.

5.5.17 FileStandardLinkInformation

No additional information class specific return codes.

5.5.18 FileStreamInformation

No additional information class specific return codes.

5.6 IRP_MJ_QUERY_VOLUME_INFORMATION

Return Code
STATUS_BUFFER_OVERFLOW
STATUS_INVALID_DEVICE_REQUEST
STATUS_VOLUME_DISMOUNTED

5.6.1 FileFsVolumeInformation

No additional information class specific return codes.

5.6.2 FileFsSizeInformation

No additional information class specific return codes.

5.6.3 FileFsDeviceInformation

No additional information class specific return codes.

5.6.4 FileFsAttributeInformation

No additional information class specific return codes.

5.6.5 FileFsControlInformation

Return Code
STATUS_NO_MATCH
STATUS_NO_MORE_MATCHES

5.6.6 FileFsFullSizeInformation

No additional information class specific return codes.

5.6.7 FileFsObjectIdInformation

Return Code

STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECT_NAME_NOT_FOUND

5.6.8 FileFsDriverPathInformation

Implemented locally by the I/O Manager, does not flow over the wire.

5.7 IRP_MJ_SET_INFORMATION

Return Code
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_HANDLE
STATUS_MEDIA_WRITE_PROTECTED
STATUS_PENDING
STATUS_SUCCESS
STATUS_INSUFFICIENT_RESOURCES
STATUS_CANCELLED
STATUS_OPLOCK_BREAK_IN_PROGRESS
STATUS_FILE_INVALID
STATUS_ACCESS_DENIED

5.7.1 FileAllocationInformation

Return Code
STATUS_USER_MAPPED_FILE

5.7.2 FileBasicInformation

No additional information class specific return codes.

5.7.3 FileDispositionInformation

Return Code
STATUS_CANNOT_DELETE
STATUS_DIRECTORY_NOT_EMPTY

5.7.4 FileEndOfFileInformation

Return Code
STATUS_USER_MAPPED_FILE

5.7.5 FileFullEaInformation

(not implemented by NTFS, SRV implements it, see SMB server documentation)

5.7.6 FileLinkInformation

Return Code
STATUS_OBJECT_NAME_INVALID
STATUS_OBJECT_NAME_COLLISION
STATUS_TOO_MANY_LINKS
STATUS_FILE_IS_A_DIRECTORY
STATUS_CANT_BREAK_TRANSACTIONAL_DEPENDENCY

STATUS_DELETE_PENDING

5.7.7 FilePositionInformation

No additional information class specific return codes.

5.7.8 FileRenameInformation

Return Code
STATUS_OBJECT_NAME_INVALID
STATUS_OBJECT_NAME_COLLISION
STATUS_OBJECT_TYPE_MISMATCH
STATUS_CANT_BREAK_TRANSACTIONAL_DEPENDENCY
STATUS_FILE_CORRUPT_ERROR
STATUS_DELETE_PENDING
STATUS_OBJECT_NAME_NOT_FOUND

5.7.9 FileSfioReserveInformation

Return Code
STATUS_NOT_SUPPORTED

5.7.10 FileShortNameInformation

Return Code
STATUS_PRIVILEGE_NOT_HELD
STATUS_SHORT_NAMES_NOT_ENABLED_ON_VOLUME
STATUS_OBJECT_NAME_COLLISION
STATUS_FILE_CORRUPT_ERROR
STATUS_CANT_BREAK_TRANSACTIONAL_DEPENDENCY

5.7.11 FileValidDataLengthInformation

Return Code
STATUS_NOT_SUPPORTED
STATUS_PRIVILEGE_NOT_HELD
STATUS_USER_MAPPED_FILE

5.8 IRP_MJ_SET_VOLUME_INFORMATION

5.8.1 FileFsLabelInformation

Return Code
STATUS_INVALID_VOLUME_LABEL
STATUS_FILE_INVALID

5.8.2 FileFsControlInformation

Return Code
STATUS_FILE_INVALID
STATUS_INVALID_PARAMETER

5.8.3 FileFsObjectIdInformation

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECTID_NOT_FOUND
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_OBJECT_NAME_INVALID
STATUS_INVALID_ADDRESS
STATUS_OBJECT_NAME_COLLISION
STATUS_FILE_INVALID
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED

5.9 IRP_MJ_QUERY_EA

Return Code
STATUS_INVALID_PARAMETER
STATUS_NONEXISTENT_EA_ENTRY
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_USER_BUFFER
STATUS_INVALID_EA_NAME
STATUS_EA_CORRUPT_ERROR
STATUS_EA_TOO_LARGE
STATUS_SUCCESS
STATUS_EAS_NOT_SUPPORTED
STATUS_NO_EAS_ON_FILE
STATUS_BUFFER_OVERFLOW
STATUS_BUFFER_TOO_SMALL
STATUS_NO_MORE_EAS
STATUS_EA_LIST_INCONSISTENT

5.10 IRP_MJ_SET_EA

Return Code
STATUS_MEDIA_WRITE_PROTECTED
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_USER_BUFFER
STATUS_INVALID_EA_NAME
STATUS_EA_TOO_LARGE
STATUS_SUCCESS
STATUS_EAS_NOT_SUPPORTED
STATUS_PENDING
STATUS_EA_LIST_INCONSISTENT

5.11 IRP_MJ_FLUSH_BUFFERS

Return Code

STATUS_MEDIA_WRITE_PROTECTED
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_DEVICE_REQUEST
STATUS_UNABLE_TO_DELETE_SECTION
STATUS_SUCCESS
STATUS_UNEXPECTED_IO_ERROR
STATUS_INVALID_HANDLE
STATUS_FILE_CORRUPT_ERROR

5.12 IRP_MJ_DIRECTORY_CONTROL

5.12.1 IRP_MN_NOTIFY_CHANGE_DIRECTORY

Return Code
STATUS_INVALID_PARAMETER
STATUS_PENDING
STATUS_INVALID_DEVICE_REQUEST
STATUS_DELETE_PENDING
STATUS_SUCCESS
STATUS_INVALID_HANDLE
STATUS_NOTIFY_CLEANUP
STATUS_NOTIFY_ENUM_DIR
STATUS_CANCELLED
STATUS_INSUFFICIENT_RESOURCES

5.12.2 IRP_MN_QUERY_DIRECTORY

Return Code
STATUS_OBJECT_NAME_INVALID
STATUS_NO_SUCH_FILE
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_USER_BUFFER
STATUS_INVALID_DEVICE_REQUEST
STATUS_NO_MORE_FILES
STATUS_SUCCESS
STATUS_INVALID_HANDLE
STATUS_FILE_CORRUPT_ERROR
STATUS_BUFFER_OVERFLOW
STATUS_INVALID_INFO_CLASS
STATUS_INSUFFICIENT_RESOURCES
STATUS_PENDING

5.12.2.1 FileBothDirectoryInformation

No additional information class specific return codes.

5.12.2.2 FileDirectoryInformation

No additional information class specific return codes.

5.12.2.3 FileFullDirectoryInformation

No additional information class specific return codes.

5.12.2.4 FileIdBothDirectoryInformation

No additional information class specific return codes.

5.12.2.5 FileIdFullDirectoryInformation

No additional information class specific return codes.

5.12.2.6 FileNamesInformation

No additional information class specific return codes.

5.12.2.7 FileIdGlobalTxDirectoryInformation

No additional information class specific return codes.

5.12.2.8 FileObjectIdInformation

No additional information class specific return codes.

5.12.2.9 FileQuotaInformation

No additional information class specific return codes.

5.12.2.10 FileReparsePointInformation

No additional information class specific return codes.

5.13 IRP_MJ_FILE_SYSTEM_CONTROL

Return Code
STATUS_INVALID_PARAMETER
STATUS_SUCCESS
STATUS_INSUFFICIENT_RESOURCES
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_USER_BUFFER
STATUS_MEDIA_WRITE_PROTECTED
STATUS_INVALID_DEVICE_REQUEST
STATUS_BUFFER_TOO_SMALL
STATUS_ACCESS_DENIED
STATUS_SYSTEM_SHUTDOWN
STATUS_DISK_CORRUPT_ERROR
STATUS_WRONG_VOLUME
STATUS_UNRECOGNIZED_VOLUME
STATUS_BAD_DEVICE_TYPE
STATUS_FILE_CORRUPT_ERROR
STATUS_INVALID_HANDLE

5.13.1 FSCTL_ALLOW_EXTENDED_DASD_IO

No additional FSCTL specific return codes.

5.13.2 FSCTL_CREATE_OR_GET_OBJECT_ID

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_DUPLICATE_NAME

5.13.3 FSCTL_DELETE_OBJECT_ID

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_OBJECTID_NOT_FOUND

5.13.4 FSCTL_DELETE_REPARSE_POINT

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_INVALID_BUFFER_SIZE
STATUS_IO_REPARSE_TAG_MISMATCH
STATUS_REPARSE_ATTRIBUTE_CONFLICT
STATUS_NOT_A_REPARSE_POINT
STATUS_IO_REPARSE_TAG_INVALID
STATUS_IO_REPARSE_DATA_INVALID
STATUS_OBJECT_NAME_INVALID
STATUS_INVALID_ADDRESS
STATUS_OBJECT_NAME_COLLISION

5.13.5 FSCTL_ENCRYPTION_FSCTL_IO

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECT_NAME_COLLISION
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_EFS_ALG_BLOB_TOO_BIG

5.13.6 FSCTL_FILESYSTEM_GET_STATISTICS

Return Code
STATUS_BUFFER_OVERFLOW

5.13.7 FSCTL_FIND_FILES_BY_SID

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_NO_QUOTAS_FOR_ACCOUNT
STATUS_CANCELLED
STATUS_NO_MORE_FILES
STATUS_OBJECT_PATH_NOT_FOUND
STATUS_NAME_TOO_LONG

STATUS_TXF_USE_NEW_PARENT_DIR
STATUS_BUFFER_OVERFLOW

5.13.8 FSCTL_GET_COMPRESSION

No additional FSCTL specific return codes.

5.13.9 FSCTL_GET_NTFS_VOLUME_DATA

No additional FSCTL specific return codes.

5.13.10 FSCTL_GET_OBJECT_ID

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECTID_NOT_FOUND
STATUS_OBJECT_NAME_NOT_FOUND

5.13.11 FSCTL_GET_REPARSE_POINT

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_NOT_A_REPARSE_POINT
STATUS_BUFFER_OVERFLOW
STATUS_IO_REPARSE_DATA_INVALID
STATUS_IO_REPARSE_TAG_INVALID

5.13.12 FSCTL_GET_RETRIEVAL_POINTERS

Return Code
STATUS_END_OF_FILE
STATUS_BUFFER_OVERFLOW
STATUS_INVALID_HANDLE

5.13.13 FSCTL_IS_PATHNAME_VALID

No additional FSCTL specific return codes.

5.13.14 FSCTL_IS_VOLUME_DIRTY

No additional FSCTL specific return codes.

5.13.15 FSCTL_LMR_GET_LINK_TRACKING_INFORMATION

Not implemented by local file systems.

5.13.16 FSCTL_LMR_SET_LINK_TRACKING_INFORMATION

Not implemented by local file systems.

5.13.17 FSCTL_QUERY_FAT_BPB

Obsolete, not supported by NTFS.

5.13.18 FSCTL_QUERY_ALLOCATED_RANGES

Return Code
STATUS_UNEXPECTED_IO_ERROR

STATUS_BUFFER_OVERFLOW

5.13.19 FSCTL_QUERY_SPARING_INFO

(not implemented by NTFS)

5.13.20 FSCTL_READ_FILE_USN_DATA

Return Code
STATUS_BUFFER_OVERFLOW

5.13.21 FSCTL_READ_RAW_ENCRYPTED

Return Code
STATUS_INVALID_HANDLE
STATUS_NOT_IMPLEMENTED
STATUS_UNEXPECTED_IO_ERROR
STATUS_END_OF_FILE

5.13.22 FSCTL_RECALL_FILE

Not implemented by local file systems.

5.13.23 FSCTL_SET_COMPRESSION

Return Code
STATUS_UNSUCCESSFUL
STATUS_COMPRESSION_DISABLED
STATUS_UNEXPECTED_IO_ERROR
STATUS_DISK_FULL

5.13.24 FSCTL_SET_DEFECT_MANAGEMENT

(not implemented by NTFS)

5.13.25 FSCTL_SET_ENCRYPTION

Return Code
STATUS_VOLUME_NOT_UPGRADED

5.13.26 FSCTL_SET_OBJECT_ID

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_OBJECTID_NOT_FOUND
STATUS_OBJECT_NAME_INVALID
STATUS_INVALID_ADDRESS
STATUS_OBJECT_NAME_COLLISION

5.13.27 FSCTL_SET_OBJECT_ID_EXTENDED

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_OBJECTID_NOT_FOUND

STATUS_OBJECT_NAME_INVALID

5.13.28 FSCTL_SET_REPARSE_POINT

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_INVALID_BUFFER_SIZE
STATUS_NOT_A_DIRECTORY
STATUS_REPARSE_ATTRIBUTE_CONFLICT
STATUS_PENDING
STATUS_EAS_NOT_SUPPORTED
STATUS_IO_REPARSE_DATA_INVALID
STATUS_DIRECTORY_NOT_EMPTY
STATUS_IO_REPARSE_TAG_MISMATCH
STATUS_IO_REPARSE_TAG_INVALID

5.13.29 FSCTL_SET_SHORT_NAME_BEHAVIOR

No additional FSCTL specific return codes.

5.13.30 FSCTL_SET_SPARSE

Return Code
STATUS_VOLUME_NOT_UPGRADED
STATUS_UNSUCCESSFUL

5.13.31 FSCTL_SET_ZERO_DATA

Return Code
STATUS_PENDING
STATUS_UNABLE_TO_DELETE_SECTION
STATUS_USER_MAPPED_FILE
STATUS_DISK_FULL
STATUS_FILE_LOCK_CONFLICT
STATUS_OPLOCK_BREAK_IN_PROGRESS
STATUS_FILE_DELETED

5.13.32 FSCTL_SET_ZERO_ON_DEALLOCATION

No additional FSCTL specific return codes.

5.13.33 FSCTL_SIS_COPYFILE

Not Implemented by local file systems.

5.13.34 FSCTL_WRITE_RAW_ENCRYPTED

Return Code
STATUS_UNABLE_TO_DELETE_SECTION
STATUS_UNEXPECTED_IO_ERROR

5.13.35 FSCTL_WRITE_USN_CLOSE_RECORD

Return Code

STATUS_JOURNAL_DELETE_IN_PROGRESS
STATUS_JOURNAL_NOT_ACTIVE

5.14 IRP_MJ_LOCK_CONTROL

Return Code
STATUS_INVALID_PARAMETER
STATUS_PENDING
STATUS_INVALID_DEVICE_REQUEST
STATUS_SUCCESS
STATUS_RANGE_NOT_LOCKED
STATUS_CANCELLED
STATUS_INSUFFICIENT_RESOURCES
STATUS_INVALID_LOCK_RANGE
STATUS_LOCK_NOT_GRANTED
STATUS_INVALID_HANDLE
STATUS_VOLUME_DISMOUNTED

5.15 IRP_MJ_CLEANUP

Return Code
STATUS_SUCCESS
STATUS_PENDING

5.16 IRP_MJ_QUERY_SECURITY

Return Code
STATUS_INVALID_PARAMETER
STATUS_BUFFER_OVERFLOW
STATUS_VOLUME_DISMOUNTED
STATUS_SUCCESS
STATUS_INVALID_USER_BUFFER
STATUS_INVALID_SECURITY_DESCR
STATUS_INSUFFICIENT_RESOURCES
STATUS_INVALID_SID
STATUS_INVALID_ACL
STATUS_UNKNOWN_REVISION
STATUS_INVALID_REVISION
STATUS_BUFFER_TOO_SMALL

5.17 IRP_MJ_SET_SECURITY

Return Code
STATUS_MEDIA_WRITE_PROTECTED
STATUS_INVALID_PARAMETER
STATUS_BUFFER_OVERFLOW
STATUS_VOLUME_DISMOUNTED
STATUS_SUCCESS

STATUS_UNSUCCESSFUL
STATUS_SECURITY_STREAM_IS_INCONSISTENT
STATUS_ACCESS_DENIED
STATUS_INVALID_SECURITY_DESCR
STATUS_NO_INHERITANCE
STATUS_INVALID_OWNER
STATUS_PRIVILEGE_NOT_HELD
STATUS_UNKNOWN_REVISION
STATUS_BAD_INHERITANCE_ACL
STATUS_NO_SECURITY_ON_OBJECT

5.18 IRP_MJ_QUERY_QUOTA

Return Code
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_USER_BUFFER
STATUS_BUFFER_OVERFLOW
STATUS_NO_MORE_ENTRIES
STATUS_INSUFFICIENT_RESOURCES
STATUS_QUOTA_LIST_INCONSISTENT
STATUS_INVALID_DEVICE_REQUEST
STATUS_BUFFER_TOO_SMALL
STATUS_SUCCESS

5.19 IRP_MJ_SET_QUOTA

Return Code
STATUS_INVALID_PARAMETER
STATUS_VOLUME_DISMOUNTED
STATUS_INVALID_USER_BUFFER
STATUS_NO_MORE_ENTRIES
STATUS_INSUFFICIENT_RESOURCES
STATUS_MEDIA_WRITE_PROTECTED
STATUS_QUOTA_LIST_INCONSISTENT
STATUS_INVALID_DEVICE_REQUEST
STATUS_ACCESS_DENIED
STATUS_CANNOT_DELETE
STATUS_PENDING
STATUS_EA_LIST_INCONSISTENT
STATUS_DATATYPE_MISALIGNMENT
STATUS_SUCCESS

6 Time stamps

Windows has 4 types of timestamps as follows:

- CreationTime – when the file was created
- LastAccessTime – when the file was last accessed
- ChangeTime – when the file's metadata or contents were last changed.
- LastWriteTime – when the files contents were last modified.

Some file systems may implement only a subset of these timestamps.

Timestamps are queried and set via the information class 'FileBasicInformation'. See [MS-FSCC] for more details.

Timestamp update may be suppressed by calling IRP_MJ_SET_INFORMATION (FileBasicInformation class) with -1 for the time fields you do not want changed. This will suppress all timestamp updates for the file handle.

6.1 NTFS

6.1.1 Supported TimeStamps

Timestamp	When Set or Changed
CreationTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. Note: By default, the creation time is tunneled if a file is deleted, and a file with the same name is created within 15 seconds. (See KB172190 http://support.microsoft.com/?kbid&id=172190) • Set when IRP_MJ_SET_INFORMATION is processed for the following information classes: ○ FileBasicInformation ○ FileRenameInformation (if the file is being tunneled)
LastAccessTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Noted when the file is accessed, set to the current time when the handle is closed. (See note about LastAccessTime below.)
ChangeTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Set when IRP_MJ_SET_EA is processed. • Set when the following FSCTLs are processed: ○ FSCTL_SET_REPARSE_POINT ○ FSCTL_DELETE_REPARSE_POINT ○ FSCTL_SET_ENCRYPTION ○ FSCTL_SET_OBJECT_ID

	○ FSCTL_SET_OBJECT_ID_EXTENDED ○ FSCTL_CREATE_OR_GET_OBJECT_ID ○ FSCTL_DELETE_OBJECT_ID • Set when IRP_MJ_SET_SECURITY is processed. • Set when IRP_MJ_SET_INFORMATION is processed for the following information classes: ○ FileAllocationInformation ○ FileBasicInformation ○ FileEndOfFileInformation ○ FileLinkInformation ○ FileRenameInformation ○ FileShortNameInformation • Set when IRP_MJ_FLUSH_BUFFERS is processed. • Noted when a write is made to the file, set to the current time when the file handle is closed.
LastWriteTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Set to the current time on a supersede/overwrite open, or on a stream rename. • Set when IRP_MJ_SET_INFORMATION is processed for the following information classes: ○ FileAllocationInformation ○ FileBasicInformation ○ FileEndOfFileInformation • Set when IRP_MJ_FLUSH_BUFFERS is processed. • Noted when a write is made to the file, set to the current time when the file handle is closed.

6.1.2 Time Format

- Stored in UTC time
- Resolution is 100 nanoseconds

6.1.3 Remarks

- LastAccessTime is updated at a 60 minute granularity.
- In Vista/Server08 updates to LastAccessTime are disabled by default and are updated only when the file is closed.

6.2 UDF

6.2.1 Supported TimeStamps

Timestamp	When Set or Changed
CreationTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. Note: UDFS does not support the tunnel cache.

	• Set when IRP_MJ_SET_INFORMATION is processed for the following information classes: ○ FileBasicInformation
LastAccessTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Noted when the file is read, set to the current time when the handle is closed, but only if another timestamp has changed as well.

ChangeTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE.
	• Set when IRP_MJ_SET_INFORMATION is processed for the following information classes:
	  ○ FileAllocationInformation
	  ○ FileBasicInformation
	  ○ FileEndOfFileInformation
	• Otherwise, UDFS updates ChangeTime at the same time as LastWriteTime.
	• Set when a write is made to the file, set to the current time when the file handle is closed.

LastWriteTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE.
	• Set when IRP_MJ_SET_INFORMATION is processed for the following information classes:
	  ○ FileAllocationInformation
	  ○ FileBasicInformation
	  ○ FileEndOfFileInformation
	• Otherwise, UDFS updates ChangeTime at the same time as LastWriteTime.
	• Set when a write is made to the file, set to the current time when the file handle is closed.

6.2.2 Time Format

- Stored in UTC time if time zone is available. Stored as local time if time zone is not available.
- Resolution is 1 microsecond

6.3 FAT

6.3.1 Supported TimeStamps

Timestamp	When Set or Changed
CreationTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. Note: By default, the creation time is tunneled if a file is deleted, and a file with the same name is created within 15 seconds. (See KB172190 http://support.microsoft.com/?kbid&id=172190) • Set when IRP_MJ_SET_INFORMATION is processed for the following information classes:

	○ FileBasicInformation ○ FileRenameInformation (if the file is being tunneled)
LastAccessTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Noted when the file is accessed, set to the current time when the handle is closed.
LastWriteTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Set to the current time on a supersede/overwrite open, or a new stream is created. • Set when IRP_MJ_SET_INFORMATION is processed for the following information classes: ○ FileAllocationInformation ○ FileBasicInformation ○ FileEndOfFileInformation ○ FileValidDataLengthInformation • Set when IRP_MJ_FLUSH_BUFFERS is processed. • Noted when a write is made to the file, set to the current time when the file handle is closed.

6.3.2 Time Format

- Stored in local time
- Resolution for CreationTime is 10 milliseconds
- Resolution for LastWriteTime is 2 seconds
- Resolution for LastAccessTime is 1 day.

6.4 exFAT

6.4.1 Supported TimeStamps

Timestamp	When Set or Changed
CreationTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. Note: By default, the creation time is tunneled if a file is deleted, and a file with the same name is created within 15 seconds. (See KB172190 http://support.microsoft.com/?kbid&id=172190) • Set when IRP_MJ_SET_INFORMATION is processed for the following information classes: ○ FileBasicInformation ○ FileRenameInformation (if the file is being tunneled)
LastAccessTime	• Set to the current time when the file is created during processing of IRP_MJ_CREATE. • Noted when the file is accessed, set when the handle is closed.
LastWriteTime	• Set to the current time when the file is created.

	• Set to the current time on a supersede/overwrite open, or a new stream is created.
	• Set when IRP_MJ_SET_INFORMATION is processed for the following information classes:
	  ○ FileAllocationInformation
	  ○ FileBasicInformation
	  ○ FileEndOfFileInformation
	  ○ FileValidDataLengthInformation
	• Set when IRP_MJ_FLUSH_BUFFERS is processed.
	• Noted when a write is made to the file, set to the current time when the file handle is closed.

6.4.2 Time Format

- Stored in UTC time if time zone is available. Stored as local time if time zone is not available.
- Resolution for CreationTime is 10 milliseconds
- Resolution for LastWriteTime is 10 milliseconds
- Resolution for LastAccessTime is 2 seconds.

7 Wild Cards

IRP_MJ_DIRECTORY_CONTROL can specify wild cards characters as part of the name query. This section outlines these wildcard sets and their processing by IRP_MJ_DIRECTORY_CONTROL within the file system in Windows.

7.1 Wild Card Characters

Characters 0x2A '*' and 0x3F '?' are treated as wild card characters in Windows. These characters can be combined together with other wild card or normal characters to form an expression that can specified as part of a directory query.

7.2 Wild Card Matching

The wild card matching is performed by enumerating all the files in a given directory and matching each file name with the name expression that was specified as part of the directory query. The expression analysis is carried out for each character of the name until the expression stops matching. Since the file names are enumerated from the disk they cannot contain any wildcards. If the directory open was a insensitive open the file name characters and corresponding name expression needs to be upcased using the on disk upcase table before the name – expression comparison is performed.

These wildcard matches are performed as below:

~* is DOS_STAR, ~? is DOS_QM, and ~. is DOS_DOT

* - matches 0 or more characters.

? - matches exactly 1 character.

DOS_STAR - matches 0 or more characters until encountering and matching the final '.' in the name.

DOS_QM - matches any single character, or upon encountering a period or end of name string, advances the expression to the end of the set of contiguous DOS_QMs.

DOS_DOT - matches either a '.' or zero characters beyond name string.

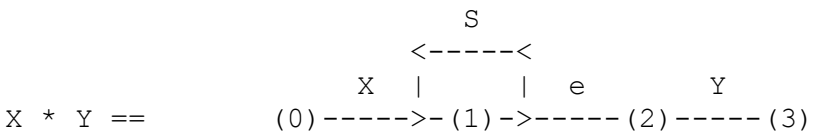
7.2.1 Expression Evaluation

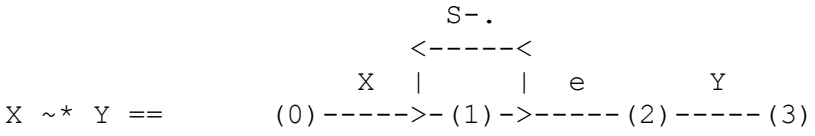
First and foremost the file system starts off name matching with the expression "*". If the expression contains a single wild card character '*' all matches are satisfied immediately. This is the most common wild card character used in Windows and expression evaluation is optimized by looking for this character first.

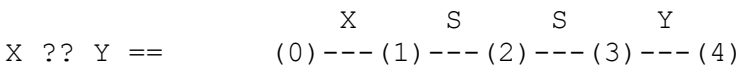
Subsequently evaluation of the "*X" expression is performed. This is a case where the expression starts off with a wild card character and contains some non-wild card characters towards the tail end of the

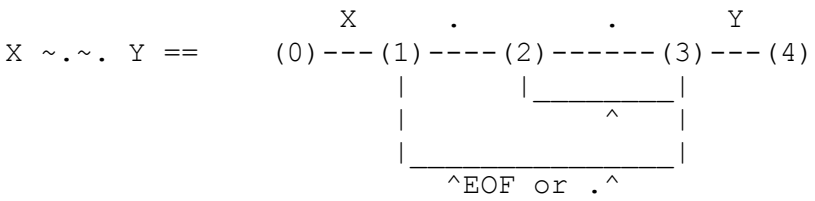
name. This is evaluated by making sure the expression starts off with the character '*' and does not contain any wildcards in the latter part of the expression. The tail part of the expression beyond the first character '*' is matched against the file name at the end upcasing each character if necessary during the comparison.

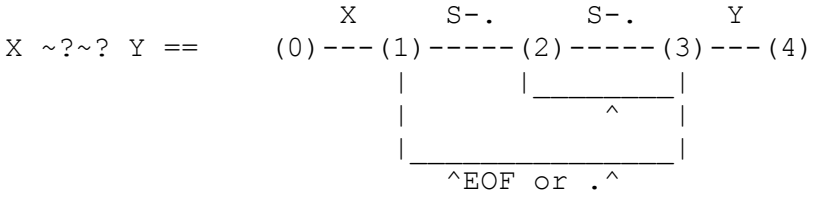
The remaining expressions are evaluated in a non deterministic finite order as listed below:











where S is any single character

S-. is any single character except the final '.'

e is a null character transition

EOF is the end of the name string