A

Windows NT System Services

For various reasons, Microsoft has not documented the native Windows NT I/O services provided by the Windows NT I/O Manager. Application developers are instead expected to use either the Win32 subsystem APIs, or the APIs provided by one of the other supported subsystems, e.g., the POSIX subsystem.

This appendix contains a list of most of the exported, native Windows NT I/O-Manager-provided system services. As was mentioned earlier in the book, the Windows NT system services are quite powerful and comprehensive, and allow the caller to more easily request certain operations that would often otherwise require multiple Win32 API calls. The majority of the structure types and flag definitions required to use the various system services described in this appendix are provided in the Windows NT DDK. Those definitions that are not provided in the DDK can be obtained from the header files supplied with the Windows NT IPS kit. Many such undefined types are described here as well.

NT System Services

The Windows NT system services allow the caller to request normal file stream manipulation operations. These include requests to create a new file or open an existing file stream, requests to perform I/O on the file, get and set file attributes, map a file into a process virtual address space, and requests to close a file handle. Nearly all of the services provided by the native system services can also be requested using Win32 API calls or any one of the various APIs provided by the supported subsystems. However, system and application software developers may sometimes require functionality that may not be easily (or efficiently) provided by any one subsystem. As an example, creating a link to an existing file cannot be easily accomplished (if at all) using the Win32 subsystem. This functionality, however, is more easily requested if an application were to use the POSIX subsystem instead.* In such situations where you may need otherwise hard-to-request functionality, requesting file system services by using the native system service calls provided by the I/O Manager can be quite useful.

Kernel-mode file system and filter driver developers may also wish to scan through the system services documented here to get a good sense of how the I/O Manager translates user requests into corresponding file system dispatch routine invocations, and also how user-specified arguments are eventually passed on to the file system implementation. Descriptions of certain system services also include comments on the responsibilities of an FSD processing such a request.

NtCreateFile()

Parameters

`FileHandle`

Returned handle (created by the I/O Manager) if call succeeds.

`DesiredAccess`

Desired access flags can be one or more of the following:

DELETE

Required if FILE_DELETE_ON_CLOSE is set in CreateOptions below. File can be deleted by caller.

FILE_READ_DATA

Caller can request to read data.

`FILE_WRITE_DATA`

Caller may write file data. The caller is also allowed to append to the file.

`FILE_READ_ATTRIBUTES`

File attributes flags can be read.

`FILE_WRITE_ATTRIBUTES`

The caller can change file attribute flag values.

`FILE_APPEND_DATA`

The caller can only append data to the file.† This access value is not allowed in conjunction with the FILE_NO_INTERMEDIATE_BUFFERING CreateOptions flag.

`READ_CONTROL`

ACL and ownership information for the file stream can be read.

---

* Multiple (hard) links to a file stream are currently supported only by the NTFS driver, out of all of the native file system implementations provided by Microsoft for the Windows NT platform.

† Any byte offset specified in a write operation will be ignored.

WRITE_DAC

Discretionary ACL associated with the file can be written.

WRITE_OWNER

Ownership information can be written.

FILE_LIST_DIRECTORY

Caller can list files contained within the directory. Not valid for data files.

FILE_TRAVERSE

The opened directory can be in the pathname of a file. Not valid for data files.

FILE_READ_EA

Caller can read extended attributes associated with the file.

FILE_WRITE_EA

Required if EaBuffer is not null. Caller may write extended attributes to the file.

SYNCHRONIZE

Caller can wait for the returned file handle for completion of asynchronous I/O requests. Required if either FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT flags in CreateOptions have been set. If this flag is not specified, I/O completion for asynchronous I/O requests must be synchronized by either using an event or an APC routine.

FILE_EXECUTE

File stream is an executable image. If FILE_EXECUTE is set but neither FILE_READ_DATA nor FILE_WRITE_DATA are set, then I/O can only be performed by mapping the file into the process virtual address space.

ObjectAttributes

The caller must allocate memory for this structure of type OBJECT_ATTRIBUTES. Fields in the structure are initialized as follows:

Length

Size, in bytes, of the structure.

ObjectName

A Unicode string specifying the name of file. The name can be either a relative name (RootDirectory is nonnull) or an absolute name (RootDirectory is NULL).

RootDirectory (optional)

The previously opened handle for a directory; ObjectName will be considered relative to this directory (if specified).

SecurityDescriptor (optional)

If nonnull, the specified ACLs will be applied only if the file is created. If the SecurityDescriptor is NULL and if the file is created, the FSD determines which (if any) ACLs will be associated with the file (typically, a default ACL associated with the parent directory is propagated to the created file).

SecurityQualityOfService (optional)

Specifies the access a server should be given to a client's security context. Only nonnull when a connection is being established to a protected server.

Attributes

Combination of OBJ_INHERIT (child processes inherit open handle) and OBJ_CASE_INSENSITIVE (lookups should be processed in a case-insensitive fashion).

loStatusBlock

Caller-supplied structure to receive results of create/open request.

AllocationSize (optional)

The initial allocation size of file. Only used when the file is initially created, overwritten, or superseded. If the FSD cannot allocate the requested disk space for the file, the create/open request will fail.

FileAttributes

Attributes are only applied if file is newly created, superseded, or overwritten. Any combination is allowed but all flag values override the FILE_ATTRIBUTE_NORMAL flag. Attributes can be one or more of the following:

FILE_ATTRIBUTE_NORMAL

A normal file should be created.

FILE_ATTRIBUTE_READONLY

A read-only file should be created.

FILE_ATTRIBUTE_HIDDEN

A hidden file should be created.

FILE_ATTRIBUTE_SYSTEM

The created file should be marked as a system file.

FILE_ATTRIBUTE_ARCHIVE

Mark the file to-be-archived.

FILE_ATTRIBUTE_TEMPORARY

The file to-be-created is marked as a temporary file. Note that modified cached data for the file is often not flushed to secondary storage for temporary files by the Cache Manager.

FILE_ATTRIBUTE_COMPRESSED

The file to be created is a compressed file.

`ShareAccess`

The type of share access requested by the caller. The share access can be a combination of the following:

`FILE_SHARE_READ`

The file can be concurrently opened for read access by other threads.

`FILE_SHARE_WRITE`

Other file open operations requesting write access should be allowed.

FILE_SHARE_DELETE

Other file open operations requesting delete access should be allowed.

Note that the share access flags allow the requester to control how the file can be shared by separate threads and processes. If none of the share values are specified, no other subsequent open operation will be allowed to proceed until the file handle is closed (and an IRP_MJ_CLEANUP issued to the FSD).

`CreateDisposition`

The disposition specified by the caller determines the actions performed by an FSD if a file does or does not exist. Any one of the following values can be specified:

`FILE_SUPERSEDE`

It the file exists, it should be superseded; if the file does not exist, it should be created.

`FILE_CREATE`

If the file does not exist, it should be created; if the file exists, an error should be returned (typically STATUS_OBJECT_NAME_COLLISION is returned).

FILE_OPEN

If the file exists, it should be opened; if the file does not exist, an error should be returned (often STATUS_OBJECT_NAME_NOT_FOUND is returned).

`FILE_OPEN_IF`

Open the file if it exists, create the file if it does not already exist.

`FILE_OVERWRITE`

If the file exists, it should be opened and overwritten. If it does not exist, the create operation should fail (often STATUS_OBJECT_NAME_NOT_FOUND is returned).

`FILE_NO_EA_KNOWLEDGE`

The caller does not understand how to handle extended attributes. If extended attributes are associated with the file being opened, the FSD must fail the open operation.

`FILE_DELETE_ON_CLOSE`

The directory entry for the file being opened should be deleted when the last handle to the file stream has been closed.

FILE_OPEN_BY_FILE_ID

The file name is actually a LARGE_INTEGER-type identifier that should be used to locate and open the target file (see Chapter 9, *Writing a File System Driver I,* for details).

FILE_OPEN_FOR_BACKUP_INTENT

The file is being opened for backup purposes, and the FSD should initiate a check for the appropriate privileges and determine whether the open should be allowed to proceed or be denied.

`FILE_NO_COMPRESSION`

The file cannot be compressed.

`EaBuffer` (optional)

A caller-allocated buffer containing a list of extended attributes to be set on the file only if the file is being created. Must be set to NULL if the file is only being opened. The FILE_FULL_EA_INFORMATION structure defines the format of the extended attributes in `EaBuffer`. Each extended attribute entry must be longword aligned. The `NextEntryOffset` field in the structures specifies the number of bytes between the current entry and the next. For the last entry, the `NextEntryOffset` field is zero.

If extended attributes are specified and if the extended attributes for the newly created file cannot be successfully created, the create/open request will fail. Therefore, creation of extended attributes is an atomic operation with respect to creation of the file.

`EaLength`

Value should be 0 if `EaBuffer` is set to NULL. Otherwise, it contains the length (in bytes) of the EAs listed in `EaBuffer`.

Return code

STATUS_SUCCESS indicates that the operation succeeded and a valid handle is being returned; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD, while STATUS_REPARSE indicates that the name should be parsed again by the object manager (e.g., a new volume has been mounted).

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_OBJECT_TYPE_MISNATCH
- STATUS_NO_SUCH_DEVICE
- STATUS_ACCESS_DENIED (a commonly used error code value)
- STATUS_FILE_IS_A_DIRECTORY
- STATUS_NOT_A_DIRECTORY
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_OBJECT_NAME_INVALID
- STATUS_DELETE_PENDING
- STATUS_SHARING_VIOLATION
- STATUS_INVALID_PARAMETER

IRP

Overlay.Allocations!ze
Set to the caller-supplied AllocationSize value (if any).

Associatedlrp.SystemBuffer
The EaBuffer supplied by the caller (if any).

Flags
The IRP_CREATE_OPERATION, IRP_SYNCHRONOUS_API, and IRP_DEFER_IO_COMPLETION flag values are set.

I/O stack location

MajorFunction
IRP_MJ_CREATE

MinorFunction
None.

Flags
One or more of SL_CASE_SENSITIVE, SL_FORCE_ACCESS_CHECK, SL_OPEN_PAGING_FILE, and SL_OPEN_TARGET_DIRECTORY.

Control
Irrelevant from the FSD's perspective.

Parameters.Create.SecurityContext
Points to an IO_SECURITY_CONTEXT structure (allocated by the I/O Manager) containing the AccessState and DesiredAccess (specified by

FILE_NO_EA_KNOWLEDGE

The caller does not understand how to handle extended attributes. If extended attributes are associated with the file being opened, the FSD must fail the open operation.

FILE_DELETE_ON_CLOSE

The directory entry for the file being opened should be deleted when the last handle to the file stream has been closed.

FILE_OPEN_BY_FILE_ID

The file name is actually a LARGE_INTEGER-type identifier that should be used to locate and open the target file (see Chapter 9, *Writing a File System Driver I,* for details).

FILE_OPEN_FOR_BACKUP_INTENT

The file is being opened for backup purposes, and the FSD should initiate a check for the appropriate privileges and determine whether the open should be allowed to proceed or be denied.

FILE_NO_COMPRESSION

The file cannot be compressed.

EaBuffer (optional)

A caller-allocated buffer containing a list of extended attributes to be set on the file only if the file is being created. Must be set to NULL if the file is only being opened. The FILE_FULL_EA_INFORMATION structure defines the format of the extended attributes in EaBuffer. Each extended attribute entry must be longword aligned. The NextEntryOffset field in the structures specifies the number of bytes between the current entry and the next. For the last entry, the NextEntryOffset field is zero.

If extended attributes are specified and if the extended attributes for the newly created file cannot be successfully created, the create/open request will fail. Therefore, creation of extended attributes is an atomic operation with respect to creation of the file.

EaLength

Value should be 0 if EaBuffer is set to NULL. Otherwise, it contains the length (in bytes) of the EAs listed in EaBuffer.

Return code

STATUS_SUCCESS indicates that the operation succeeded and a valid handle is being returned; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD, while STATUS_REPARSE indicates that the name should be parsed again by the object manager (e.g., a new volume has been mounted).

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_OBJECT_TYPE_MISMATCH
- STATUS_NO_SUCH_DEVICE
- STATUS_ACCESS_DENIED (a commonly used error code value)
- STATUS_FILE_IS_A_DIRECTORY
- STATUS_NOT_A_DIRECTORY
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_OBJECT_NAME_INVALID
- STATUS_DELETE_PENDING
- STATUS_SHARING_VIOLATION
- STATUS_INVALID_PARAMETER

IRP

Overlay.AllocationSize
Set to the caller-supplied AllocationSize value (if any).

AssociatedIrp.SystemBuffer
The EaBuffer supplied by the caller (if any).

Flags
The IRP_CREATE_OPERATION, IRP_SYNCHRONOUS_API, and IRP_DEFER_IO_COMPLETION flag values are set.

I/O stack location

MajorFunction
IRP_MJ_CREATE

MinorFunction
None.

Flags
One or more of SL_CASE_SENSITIVE, SL_FORCE_ACCESS_CHECK, SL_OPEN_PAGING_FILE, and SL_OPEN_TARGET_DIRECTORY.

Control
Irrelevant from the FSD's perspective.

Parameters.Create.SecurityContext
Points to an IO_SECURITY_CONTEXT structure (allocated by the I/O Manager) containing the AccessState and DesiredAccess (specified by

the caller). The FSD can validate the access requested by the caller using the help of the security subsystem (if the FSD supports access checking).

`Parameters.Create.Options`
Bits 0 to 15 contain the caller-specified `CreateOptions`; bits 16 through 23 are reserved by the I/O Manager; and bits 24 through 31 specify the `CreateDisposition`.

`Parameters.Create.FileAttributes`
`FileAttributes` specified by the caller.

`Parameters.Create.ShareAccess`
ShareAccess specified by the caller.

`Parameters.Create.EaLength`
EaLength specified by the caller (the buffer supplied—if any—is pointed to by the `Associatedlrp. SystemBuffer` field in the IRP).

`DeviceObject`
Points to the FSD-created device object representing either the FSD itself or the mounted logical volume.

`FileObject`
A file object structure allocated by the I/O Manager for this particular create/open request.

Notes

Create or open requests are inherently synchronous requests. Therefore, the I/O Manager will block the calling thread until the request has been processed by the FSD (even if STATUS_PENDING is returned by the FSD) and the IRP_DEFER_IO_COMPLETION flag will be set in the `Irp->Flags` field.

The following flags are set in the `FileObject->Flags` field:

`FO_SYNCHRONOUS_IO`
Set by the I/O Manager if either FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT have been specified by the caller.

`FO_ALERTABLE_IO`
Set by the I/O Manager if FILE_SYNCHRONOUS_IO_ALERT is specified by the caller.

`FO_NO_INTERMEDIATE_BUFFERING`
Set by the I/O Manager and by FSDs if FILE_NO_INTERMEDIATE_BUFFERING is specified by the caller.

`FO_WRITE_THROUGH`
Set by the I/O Manager and by FSDs if FILE_WRITE_THROUGH is specified by the caller.

`FO_SEQUENTIAL_ONLY`
Set by the I/O Manager if FILE_SEQUENTIAL_ONLY is specified by the caller.

`FO_TEMPORARY_FILE`
Set by the FSD if FILE_ATTRIBUTE_TEMPORARY is specified by the caller.

FO_FILE_FAST_I0_READ
Set by the FSD if the file is successfully opened for EXECUTE access; also set by the FSD and by the FSRTL package whenever a cached read operation completes, indicating that time stamps for the file (directory entry) should be updated when all handles have been closed.*

NtOpenFile()

```
NTSTATUS NtOpenFile(
    OUT PHANDLE              FileHandle,
    IN ACCESS_MASK           DesiredAccess,
    IN POBJECT_ATTRIBUTES    ObjectAttributes,
    OUT PIO_STATUS_BLOCK     loStatusBlock,
    IN ULONG                 ShareAccess,
    IN ULONG                 OpenOptions,
);
```

Parameters

FileHandle
Returned handle (created by the I/O Manager) if the call succeeds.

DesiredAccess
See the description of this argument for the NtCreateFile () system call described above.

ObjectAttributes
The caller must allocate memory for this structure of type OBJECT_ATTRIBUTES. Fields in the structure are initialized as follows:

Length
The size, in bytes, of the structure.

`ObjectName`
A Unicode string specifying name of file. The name can be a either a relative name (RootDirectory is nonnull) or an absolute name (RootDirectory is NULL).

---

* The FO_FILE_MODIFIED flag is set by the FSRTL package to indicate that time stamps should be updated due to a fast I/O write request.

RootDirectory (optional)

The previously opened handle for a directory; ObjectName will be considered relative to this directory (if specified).

SecurityDescriptor (optional)

NULL pointer.

SecurityQualityOfService (optional)

NULL pointer.

`Attributes`

A combination of OBJ_INHERIT (child processes inherit open handle) and OBJ_CASE_INSENSITIVE (lookups should be processed in a case-insensitive fashion).

`loStatusBlock`

A caller-supplied structure to receive results of create/open request.

`ShareAccess`

The type of share access requested by the caller. The share access can be a combination of the following:

`FILE_SHARE_READ`

The file can be concurrently opened for read access by other threads.

`FILE_SHARE_WRITE`

Other file open operations requesting write access should be allowed.

`FILE_SHARE_DELETE`

Other file open operations requesting delete access should be allowed.

Note that the share access flags allow the requester to control how the file can be shared by separate threads and processes. If none of the share values are specified, no other subsequent open operation will be allowed to proceed until the file handle is closed (and an IRP_MJ_CLEANUP is issued to the FSD).

`OpenOptions`

Options used when the file is opened. See the description for NtCreateFileO for more details.

Return code

STATUS_SUCCESS indicates that the operation succeeded and a valid handle is being returned, STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD, while STATUS_REPARSE indicates that the name should be parsed again by the object manager (e.g., a new volume has been mounted).

In the case of an error, an appropriate error code is returned. See the description for NtCreateFile ( ) for more details.

IRP/I/O stack location

The IRP and I/O stack location for an open request are set up in essentially the same manner as that for a NtCreateFile () system call.

Notes

Time stamps for the file are not affected when an open request is received by the FSD.

NtReadFile()

```
NTSTATUS NtReadFile(
    IN HANDLE               FileHandle,
    IN HANDLE               Event OPTIONAL,
    IN PIO_APC_ROUTINE      ApcRoutine OPTIONAL,
    IN PVOID                ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK    loStatusBlock,
    OUT PVOID               Buffer,
    IN ULONG                Length,
    IN PLARGE_INTEGER       ByteOffset OPTIONAL,
    IN PULONG               Key OPTIONAL
);
```

Parameters

`FileHandle`

Returned to the caller from a previous successful NtCreateFile () or NtOpenFile() invocation.

Event (optional)

The caller can wait for the supplied event object (created by the caller) for completion of the asynchronous read request. The event will be signaled by the I/O Manager when the read operation is completed.

ApcRoutine (optional)

An optional, caller-supplied APC routine invoked by the I/O Manager when the read operation completes.

ApcContext (optional)

The caller-determined context to be passed in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

`loStatusBlock`

The caller must supply this argument to receive the results of the read operation. The Information field in the loStatusBlock is set to the number of bytes actually read by the FSD.

RootDirectory (optional)

The previously opened handle for a directory; ObjectName will be considered relative to this directory (if specified).

SecurityDescriptor (optional)

NULL pointer.

SecurityQualityOfService (optional)

NULL pointer.

Attributes

A combination of OBJ_INHERIT (child processes inherit open handle) and OBJ_CASE_INSENSITIVE (lookups should be processed in a case-insensitive fashion).

IoStatusBlock

A caller-supplied structure to receive results of create/open request.

ShareAccess

The type of share access requested by the caller. The share access can be a combination of the following:

FILE_SHARE_READ

The file can be concurrently opened for read access by other threads.

FILE_SHARE_WRITE

Other file open operations requesting write access should be allowed.

FILE_SHARE_DELETE

Other file open operations requesting delete access should be allowed.

Note that the share access flags allow the requester to control how the file can be shared by separate threads and processes. If none of the share values are specified, no other subsequent open operation will be allowed to proceed until the file handle is closed (and an IRP_MJ_CLEANUP is issued to the FSD).

OpenOptions

Options used when the file is opened. See the description for NtCreateFile () for more details.

Return code

STATUS_SUCCESS indicates that the operation succeeded and a valid handle is being returned, STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD, while STATUS_REPARSE indicates that the name should be parsed again by the object manager (e.g., a new volume has been mounted).

In the case of an error, an appropriate error code is returned. See the description for `NtCreateFile()` for more details.

IRP/I/O stack location

The IRP and I/O stack location for an open request are set up in essentially the same manner as that for a `NtCreateFile()` system call.

Notes

Time stamps for the file are not affected when an open request is received by the FSD.

NtReadFile()

```
NTSTATUS NtReadFile(
    IN HANDLE               FileHandle,
    IN HANDLE               Event OPTIONAL,
    IN PIO_APC_ROUTINE      ApcRoutine OPTIONAL,
    IN PVOID                ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK    loStatusBlock,
    OUT PVOID               Buffer,
    IN ULONG                Length,
    IN PLARGE_INTEGER       ByteOffset OPTIONAL,
    IN PULONG               Key OPTIONAL
);
```

Parameters

`FileHandle`

Returned to the caller from a previous successful `NtCreateFile()` or `NtOpenFile()` invocation.

Event (optional)

The caller can wait for the supplied event object (created by the caller) for completion of the asynchronous read request. The event will be signaled by the I/O Manager when the read operation is completed.

ApcRoutine (optional)

An optional, caller-supplied APC routine invoked by the I/O Manager when the read operation completes.

ApcContext (optional)

The caller-determined context to be passed in to the ApcRoutine. This argument should be NULL if `ApcRoutine` is NULL.

`loStatusBlock`

The caller must supply this argument to receive the results of the read operation. The `Information` field in the `loStatusBlock` is set to the number of bytes actually read by the FSD.

Buffer

A caller-allocated buffer to receive data read from secondary storage.

Length

The size, in bytes, of the Buffer supplied by the caller.

ByteOffset

The starting byte offset where the read begins. Caller can specify FILE_USE_FILE_POINTER_POSITION rather than an explicit byte offset or pass NULL; in either case the FSD will perform the read from the current file pointer position. The I/O Manager maintains the file pointer position whenever the file stream is opened for synchronous I/O, and therefore, specifying a byte offset effectively results in an atomic seek-and-read service for the caller.

Key (optional)

If the byte range is locked, a matching Key value (if supplied by the caller) will result in the FSD allowing the read to proceed. This can be used to selectively share data between threads belonging to the same process.

Return code

STATUS_SUCCESS indicates that the operation succeeded and some subset of the range requested by the caller is being returned by the FSD; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_END_OF_FILE
- STATUS_FILE_LOCK_CONFLICT

IRP

MdlAddress

Any MDL created by the I/O Manager (or by some other kernel-mode component) describing the buffer in which data should be returned by the FSD.

UserBuffer

A pointer to the user-supplied buffer. This field is effectively overridden by the presence of any MDL pointer in the MdlAddress field.*

Flags

One or both of IRP_PAGING_IO and IRP_NOCACHE may be set. IRP_PAGING_IO is only set by the I/O Manager if the I/O request is a result of a synchronous or an asynchronous paging I/O operation requested by the Virtual Memory Manager.

I/O stack location

`MajorFunction`

`IRP_MJ_READ`

`MinorFunction`

One or more of the following:

IRP_MN_DPC

The IRP was dispatched at a high IRQL.

IRP_MN_MDL

The caller wants an MDL returned containing the requested data.

`IRP_MN_COMPLETE`

The caller has finished with the MDL returned from a previous call (with IRP_MN_MDL specified).

IRP_MN_COMPRESSED

The caller does not want any compressed data decompressed.

Flags

One or more of SL_KEY_SPECIFIED and SL_OVERRIDE_VERIFY_VOLUME.

Parameters.Read.Length

The read Length specified by the caller.

Parameters.Read.Key

The Key specified by the caller.

Parameters.Read.ByteOffset

The ByteOffset specified by the caller.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

---

* See Chapter 9 for details. The FSD will check for the presence of an MDL first and will use any MDL pointed to by the MdlAddress field. If MdlAddress is set to NULL, the FSD will use the UserBuffer pointer directly (since typically, FSDs prefer to neither specify DO_DIRECT_IO nor DO_BUFFERED_IO for handling user buffers).

`FileObject`

The file object representing the open instance of the file to be read.

Notes

The LastAccessTime for the file stream being read is typically updated by the FSD upon completion of the read request.

NtWriteFile()

```
NTSTATUS NtWriteFile(
    IN HANDLE                FileHandle,
    IN HANDLE                Event OPTIONAL,
    IN PIO_APC_ROUTINE       ApcRoutine OPTIONAL,
    IN PVOID                 ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK     loStatusBlock,
    OUT PVOID                Buffer,
    IN ULONG                 Length,
    IN PLARGE_INTEGER        ByteOffset OPTIONAL,
    IN PULONG                Key OPTIONAL
);
```

Parameters

`FileHandle`

Returned to the caller from a previous successful NtCreateFile() or NtOpenFile() invocation.

Event (optional)

The caller can wait for the supplied event object (created by the caller) for completion of the asynchronous write request. The event will be signaled by the I/O Manager when the write operation is completed.

ApcRoutine (optional)

The optional, caller-supplied APC routine invoked by the I/O Manager when the write operation completes.

ApcContext (optional)

The caller-determined context to be passed in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

`loStatusBlock`

The caller must supply this argument to receive the results of the write operation. The Information field in the loStatusBlock is set to the number of bytes actually written by the FSD.

Buffer

A caller-allocated buffer containing data to be written to secondary storage.

Length

The size, in bytes, of the `Buffer` supplied by the caller.

`ByteOffset`

The starting byte offset where the write begins. Caller can specify FILE_USE_FILE_POINTER_POSITION rather than an explicit byte offset or pass in NULL; in either case the FSD will perform the write from the current file pointer position. The I/O Manager maintains the file pointer position whenever the file stream is opened for synchronous I/O and therefore specifying a byte offset effectively results in an atomic seek-and-write service for the caller (the file pointer is updated appropriately according to the starting offset from where the write begins and the number of bytes written).

In order to simply write to the current end-of-file, the caller can specify FILE_WRITE_TO_END_OF_FILE in the `ByteOffset` argument.

If the file was opened for FILE_APPEND_DATA, any caller-supplied byte offset is ignored.

Key (optional)

If the byte range is locked, a matching Key value (if supplied by the caller) will result in the FSD allowing the write to proceed. This can be used to selectively allow file modification between threads belonging to the same process.

Return code

STATUS_SUCCESS indicates that the operation succeeded and some subset of the range requested by the caller was written by the FSD; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_FILE_LOCK_CONFLICT

IRP

`MdlAddress`

Any MDL created by the I/O Manager (or by some other kernel-mode component) describing the buffer containing data to be written. This could also be a MDL returned from a previous write request with MinorFunction set to

IRP_MN_MDL, in which case the MDL will eventually be freed by the Cache Manager.

UserBuffer

Pointer to the user-supplied buffer. This field is effectively overridden by the presence of any MDL pointer in the MdlAddress field.

Flags

One or both of IRP_PAGING_IO and IRP_NOCACHE may be set.

I/O stack location

`MajorFunction`

`IRP_MJ_WRITE`

`MinorFunction`

One or more of the following:

IRP_MN_DPC

The IRP was dispatched at a high IRQL.

IRP_MN_MDL

The caller wants an MDL returned, which will eventually be filled with modified data (by the caller).

`IRP_MN_COMPLETE`

The caller has finished with the MDL returned from a previous call (with IRP_MN_MDL specified).

`IRP_MN_COMPRESSED`

The caller is sending compressed data to the FSD.

Flags

One or more of SL_KEY_SPECIFIED and SL_WRITE_THROUGH.

`Parameters.Write.Length`

The number of bytes to be written specified by the caller.

`Parameters.Write.Key`

The Key specified by the caller.

`Parameters.Write.ByteOffset`

The starting ByteOffset specified by the caller.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of the file to be written.

Notes

The LastWriteTime for the file stream being written is typically updated by the FSD upon completion of the write request. The FSD should set the SL_FT_SEQUENTIAL_WRITE flag before forwarding a write-through write request to the next driver in the calling hierarchy.

NtQueryDirectoryFile ()

```
NTSTATUS NtQueryDirectoryFile(
    IN HANDLE                      FileHandle,
    IN HANDLE                      Event OPTIONAL,
    IN PIO_APC_ROUTINE             ApcRoutine OPTIONAL,
    IN PVOID                       ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK           IoStatusBlock,
    OUT PVOID                      FileInformation,
    IN ULONG                       Length,
    IN FILE_INFORMATION_CLASS      FileInformationClass,
    IN BOOLEAN                     ReturnSingleEntry,
    IN PUNICODE_STRING             FileName OPTIONAL,
    IN BOOLEAN                     RestartScan
);
```

Parameters

FileHandle

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile() invocation.

Event (optional)

The caller can wait for the supplied event object (created by the caller) for completion of the asynchronous query directory request. The event will be signaled by the I/O Manager when the query directory IRP is completed by the FSD.

ApcRoutine (optional)

The optional, caller-supplied APC routine invoked by the I/O Manager when the query directory operation completes.

ApcContext (optional)

The caller-determined context to be passed-in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

IoStatusBlock

The caller must supply this argument to receive the results of the query directory operation. The Information field in the IoStatusBlock is set to the number of bytes returned by the FSD (in the buffer pointed to by the FileInformation argument).

Filelnformation

A caller-allocated buffer to receive information about files contained in the directory. Alignment requirements for the buffer and the contents of the buffer (returned by the FSD) are determined by the Filelnformation-Class of the argument.

Note that the buffer passed to the FSD in the query directory IRP is an I/O Manager-allocated system buffer. Copying data from the system buffer to the actual caller-allocated buffer (pointed to by the Filelnformation argument) is performed by the I/O Manager upon completion of the IRP.

Length

The size, in bytes, of the buffer supplied by the caller in Filelnformation.

FilelnformationClass

Specifies the kind of information requested by the caller. This can be one of the following:

FileNamelnformation

The supplied buffer must be longword-aligned, as is the returned information. The size of the buffer must at least be equal to sizeof (FILE_NAMES_INFORMATION). The caller expects to receive the long names of file entries contained in the directory in the caller-supplied buffer.

The FILE_NAMES_INFORMATION structure is defined as follows:

```
typedef struct _FILE_NAMES_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    ULONG FileNameLength;
    WCHAR FileName[l];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

FileDirectoryInformation

The supplied buffer must be quadword-aligned, as is the returned information. The size of the buffer must at least be equal to sizeof (FILE_DIRECTORY_INFORMATION). The caller expects to get basic information (such as the filename, file attributes, various time stamps associated with the file, and so on) for the matching directory entries.

Here is the FILE_DIRECTORY_INFORMATION structure:

```
typedef struct _FILE_DIRECTORY_INFORMATION {
    ULONG                           NextEntryOffset;
    ULONG               Filelndex;
    LARGE_INTEGER       CreationTime;
    LARGE_INTEGER       LastAccessTime;
    LARGE_INTEGER       LastWriteTime;
    LARGE_INTEGER       ChangeTime;
    LARGE_INTEGER       EndOfFile;
    LARGE_INTEGER       AllocationSize;
    ULONG             FileAttributes;
    ULONG             FileNameLength;
    WCHAR             FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;
```

FileFullDirectoryInformation

The supplied buffer must be quadword-aligned, as is the returned information. The size of the buffer must at least be equal to sizeof (FILE_FULL_DIR_INFORMATION). The caller expects to get all of the information that could be obtained via the FileDirectoryInformation information class and in addition, expects to get back information about extended attributes associated with the matching directory entries.

This is the FILE_FULL_DIR_INFORMATION structure:

```
typedef struct _FILE_FULL_DIR_INFORMATION {
ULONG NextEntryOffset;
ULONG FileIndex;
LARGE_INTEGER CreationTime;
LARGE_INTEGER LastAccessTime;
LARGE_INTEGER LastWriteTime;
LARGE_INTEGER ChangeTime;
LARGE_INTEGER EndOfFile;
LARGE_INTEGER AllocationSize;
ULONG FileAttributes;
ULONG FileNameLength;
ULONG EaSize;
WCHAR FileName[1];
} FILE_FULL_DIR_INFORMATION, *PFILE_FULL_DIR_INFORMATION;
```

FileBothDirectoryInformation

The supplied buffer must be quadword-aligned, as is the returned information. The size of the buffer must at least be equal to sizeof (FILE_BOTH_DIR_INFORMATION). The caller expects to get all of the information that could be obtained via the FileFullDirectoryInformation information class and in addition, expects to get back 8.3 versions of file names (if such alternate names are supported by the FSD) for matching directory entries.*

Here is the FILE_BOTH_DIR_INFORMATION structure:

```
typedef struct _FILE_BOTH_DIR_INFORMATION {
ULONG NextEntryOffset;
ULONG FileIndex;
LARGE_INTEGER CreationTime;
```

---

* Note that if your FSD does not support alternate/short (8.3) versions of filenames, the information returned by your driver in the FILE_BOTH_DIR_INFORMATION structure for each matching directory entry will essentially be the same as would be returned by your FSD in the FILE_FULL_DIR_INFORMATION structure; the ShortNameLength field must be initialized to 0 for each entry, and the ShortName pointer field must be initialized to NULL.

```
LARGE_INTEGER LastAccessTime;
LARGE_INTEGER LastWriteTime;
LARGE_INTEGER ChangeTime;
LARGE_INTEGER EndOfFile;
LARGE_INTEGER AllocationSize;
ULONG FileAttributes;
ULONG FileNameLength;
ULONG EaSize;
CCHAR ShortNameLength;
WCHAR ShortName[12];
WCHAR FileName[l];
} FILE_BOTH_DIR_INFORMATION, *PFILE_BOTH_DIR_INFORMATION;
```

Once a query directory request for a particular FileInformationClass type is submitted by a thread using a specific file handle, the FileInformationClass type must not change when any subsequent query directory requests are submitted using the same file handle.

`ReturnSingleEntry`

If TRUE, the caller only wants information on a single matching directory entry returned.

FileName (optional)

The search pattern, specified by the user, for the first query directory request, issued using the particular file object (or file handle); the FSD attempts to find matching directory entries based upon this pattern. If no name is supplied, the FSD uses "*", a wildcard that matches any directory entry.

`RestartScan`

Normally, the FSD begins the search for a matching directory entry from the last file pointer position (based upon the previous query directory request); however, this flag allows the caller to indicate whether the search should begin from the starting byte offset in the directory.

Return code

STATUS_SUCCESS indicates that the operation succeeded and information on at least one directory entry is being returned by the FSD; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST

- STATUS_BUFFER_OVERFLOW
- STATUS_INVALID_INFO_CLASS
- STATUS_NO_SUCH_FILE
- STATUS_NO_MORE_FILES

IRP

MdlAddress

Any MDL created by the FSD, if the request is dispatched to a worker thread for asynchronous processing.

UserBuffer

The pointer to the user-supplied buffer. This field is effectively overridden by the presence of any MDL pointer in the MdlAddress field.

I/O stack location

MajorFunction

IRP_MJ_DIRECTORY_CONTROL

MinorFunction

IRP_MN_QUERY_DIRECTORY

Flags

One or more of SL_RESTART_SCAN, SL_RETURN_SINGLE_ENTRY, and SL_INDEX_SPECIFIED.

Parameters.QueryDirectory.Length

The Length specified by the caller for the buffer in which information is received.

Parameters.QueryDirectory.FileName

The search pattern specified by the caller. The FSD must search for matching entries in the target directory using this specified pattern. The user-specified pattern is typically stored by the FSD in the CCB for the target directory for the particular open operation (of the target directory), when the first such query directory request is received. The caller can temporarily override this search pattern in subsequent query directory requests by specifying a different pattern than the one stored by the FSD; however, the behavior of the FSD in response to such query directory requests containing a new search pattern is highly FSD-specific and not well-defined by the I/O subsystem. Some FSDs may honor the new search pattern while others may choose to ignore it.

Parameters.QueryDirectory.FileInformationClass

The type of information requested by the caller.

`Parameters.QueryDirectory.FileIndex`

Any starting index, to begin the scan from, specified by the caller.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

File object representing the open instance of the target directory.

Notes

The query directory request is an inherently synchronous request. Therefore, the I/O Manager will block the requesting thread until the operation has been completed by the FSD.

The FSD returns information on the following directory entries:

- Information about a single matching directory entry is returned if either `ReturnSingleEntry` is TRUE or if the specified search pattern does not contain any wildcards.
- The number of matching files for which information can be returned in the caller-supplied buffer, constrained by the length of the buffer.
- The total number of directory entries (files or directories) in the target directory being queried.

Information on matching directory entries can be returned in any order. Most returned entries are either `quadword`-aligned or longword-aligned. See Chapter 10, *Writing A File System Driver II,* for information on how directory control requests are processed by the FSD. The maximum length of a file name is constrained (on Windows NT platforms) to be less than or equal to FILE_MAXIMUM_FILENAME_LENGTH.

If no matching entry was found for the very first query directory request received by the FSD using the particular file object, an error code of STATUS_NO_SUCH_FILE is returned to the caller; if no match is found for any subsequent query directory request, the STATUS_NO_MORE_FILES error code is returned.

The FSD maintains context about the returned information in the CCB structure associated with the specified file object. Therefore, requests to obtain directory information from different threads sharing the same file handle (and sharing the same file object and correspondingly the same CCB structure) will share (and affect) the same context maintained by the FSD.

NtNotifyChangeDirectoryFile()

```
NTSTATUS NtNotifyChangeDirectoryFile(
    IN HANDLE               FileHandle,
    IN HANDLE               Event OPTIONAL,
    IN PIO__APC_ROUTINE      ApcRoutine OPTIONAL,
    IN PVOID                ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK     loStatusBlock,
    OUT PVOID               Buffer,
    IN ULONG                Length,
    IN ULONG               CompletionFilter,
    IN BOOLEAN              WatchTree
);
```

Parameters

FileHandle

Returned to the caller from a previous successful NtCreateFile() or NtOpenFileO invocation.

Event (optional)

The caller can wait for the supplied event object (created by the caller) for completion of the asynchronous notify change directory request. The event will be signaled by the I/O Manager when the notify change directory IRP is completed by the FSD.

ApcRoutine (optional)

An optional, caller-supplied APC routine invoked by the I/O Manager when the notify change directory operation completes.

ApcContext (optional)

Caller-determined context to be passed-in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

loStatusBlock

The caller must supply this argument to receive the results of the notify change directory operation. The Information field in the loStatusBlock is set to the number of bytes returned by the FSD (in the buffer pointed to by the Filelnformation argument).

If too many changes have occurred and information about such changes cannot be returned by the FSD in the supplied buffer, the FSD will set the Information field to 0 and the STATUS_NOTIFY_ENUM_DIR return code will be returned in the Status field of the loStatusBlock argument.

Buffer

A caller-allocated buffer to receive information about the names of files contained in the target directory that have been affected. The format of returned information is defined by the FILE_NOTIFY_INFORMATION structure, which is defined as follows:

```
typedef struct _FILE_NOTIFY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG Action;*
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NOTIFY_INFORMATION, *PFILE_NOTIFY_INFORMATION;
```

Length

The size, in bytes, of the buffer supplied by the caller.

`CompletionFilter`

Specifies a combination of flags that indicate the changes the caller is interested in monitoring on the target directory.

These flags can be one or more of the following (see Chapter 10 for details on how the FSD processes the notify change directory request):

`FILE_NOTIFY_CHANGE_FILE_NAME`

Some file has been added, deleted, or renamed.

`FILE_NOTIFY_CHANGE_DIR_NAME`

Some subdirectory has been added, deleted, or renamed.

`FILE_NOTIFY_CHANGE_NAME`

A combination of FILE_NOTIFY_CHANGE_FILE_NAME and FILE_NOTIFY_CHANGE_DIR_NAME.

`FILE_NOTIFY_CHANGE_ATTRIBUTES`

Attributes of any directory entry (representing either a file or a directory) have been changed.

`FILE_NOTIFY_CHANGE_SIZE`

Allocation size or end-of-file position have been changed for any directory entry.

`FILE_NOTIFY_CHANGE_LAST_WRITE`

The last write time stamp value for a directory entry has been changed.

`FILE_NOTTFY_CHANGE_LAST_ACCESS`

The last access time stamp value for a directory entry has been changed.

`FILE_NOTIFY_CHANGE_CREATION`

The creation time stamp value for a directory entry has been changed.

---

* The possible values (hit-flags) that can be returned in this field are given in Chapter 10.

`FILE_NOTIFY_CHANGE_EA`

Extended attributes associated with a directory entry (file or directory) have been changed.

`FILE_NOTIFY_CHANGE_SECURITY`

Security attributes associated with a directory entry have been changed.

`FILE_NOTIFY_CHANGE_STREAM_NAME`

Applies to FSDs that support multiple byte streams associated with files. A new file stream may have been added, deleted, or renamed, in which case the caller should be notified.

`FILE_NOTIFY_CHANGE_STREAM_SIZE`

The size of a file stream may have changed.

`FILE_NOTIFY_CHANGE_STREAM_WRITE`

The contents of an alternate stream have been changed (i.e., the stream data was modified).

WatchTree

If TRUE, the caller wants to recursively monitor changes to all subdirectories contained within the target directory.

Return code

STATUS_PENDING indicates that the IRP has been successfully queued by the FSD and will be completed once one or more of the specified changes (being monitored by the caller) have occurred; STATUS_SUCCESS indicates that at least one monitored change had already occurred before the latest notify change directory IRP was even received by the FSD, and the caller is being notified of the fact.

Once STATUS_PENDING is returned by the FSD, the caller must examine the contents of the Status field in the loStatusBlock argument to determine the results of the notify change directory request, once the request has been completed.

In the case of an error (or a buffer overflow condition), an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- `STATUS_ACCESS_DENIED`
- `STATUS_INSUFFICIENT_RESOURCES`
- `STATUS_INVALID_PARAMETER`
- `STATUS_INVALID_DEVICE_REQUEST`
- `STATUS_NOTIFY_ENUM_DIR`

IRP

UserBuffer

A pointer to the user-supplied buffer. This field is effectively overridden by the presence of any MDL pointer in the MdlAddress field. If your FSD supports buffered I/O, then the I/O Manager will have allocated a system buffer for your FSD, and this buffer can be accessed via the AssociatedIrp.SystemBuffer field in the IRP.

I/O ***stack*** *location*

MajorFunction

IRP_MJ_DIRECTORY_CONTROL

MinorFunction

IRP_MN_NOTIFY_CHANGE_DIRECTORY

Flags

Can be set with SL_WATCH_TREE.

Parameters.NotifyDirectory.Length

The Length, specified by the caller, for the buffer in which information is received.

Parameters.NotifyDirectory.CompletionFilter

The type of changes being monitored by the caller.

DeviceObject

Points to the FSD-created device object representing the mounted logical volume.

FileObject

The file object representing the open instance of the target directory being monitored.

Notes

The notify change directory request interprets a return code of STATUS_PENDING to indicate that the IRP has been successfully queued.

NtQueryInformationFile()

```
NTSTATUS NtQueryInformationFile(
    IN HANDLE                     FileHandle,
    OUT PIO_STATUS_BLOCK          IoStatusBlock,
    OUT PVOID                     FileInformation,
    IN ULONG                      Length,
    IN FILE_INFORMATION_CLASS FileInformationClass
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile() invocation.

`loStatusBlock`

The caller must supply this argument to receive the results of the query file information request. The Information field in the loStatusBlock is set to the number of bytes returned by the FSD (in the buffer pointed to by the Filelnformation argument).

`Filelnformation`

A caller-allocated buffer to receive information about the specified file. The format of returned information is defined by the FilelnformationClass argument.

Length

The size, in bytes, of the buffer supplied by the caller.

FilelnformationClass

Used by the caller to specify the type of information requested for the target file. See Chapter 10 for a detailed discussion on the types of information provided by file system drivers and for corresponding structure definitions.

Return code

STATUS_SUCCESS indicates that the operation succeeded; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error (or a buffer overflow condition), an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOTJRCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_BUFFER_OVERFLOW

IRP

`Associatedlrp.SystemBuffer`

A pointer to an I/O Manager-allocated buffer. The I/O Manager always allocates a system buffer to contain information returned by the FSD. Contents of this buffer are copied to the user-supplied buffer by the I/O Manager (before the system buffer is deallocated by the I/O Manager).

Flags

The IRP_BUFFERED_IO, IRP_DEALLOCATE_BUFFER, IRP_INPUT_OPERATION, and IRP_DEFER_IO_COMPLETION flags are set. However, these are only used internally by the I/O Manager.*

I/O stack location

`MajorFunction`

`IRP_MJ_QUERY_INFORMATION`

`MinorFunction`

None.

`Flags`

None.

`Parameters.QueryFile.Length`

The Length, specified by the caller, for the buffer in which information is received.

Parameters.QueryFile.FileInformationClass

The type of information requested by the user.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of the file for which information has been requested.

Notes

The I/O Manager is responsible for filling in information for some of the FileInformationClass values. See Chapter 10 for further details.

NtSetInformationFile()

```
NTSTATUS NtSetInformationFile(
    IN HANDLE                    FileHandle,
    OUT PIO_STATUS_BLOCK         IoStatusBlock,
    OUT PVOID                    FileInformation,
    IN ULONG                     Length,
    IN FILE_INFORMATION_CLASS    FileInformationClass
);
```

---

[*] See Chapter 4, *The NT I/O Manager*, for a discussion on the IRP_DEFER_IO_COMPLETION flag.

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile() invocation.

`IoStatusBlock`

The caller must supply this argument to receive the results of the set file information request. The Information field in the IoStatusBlock is initialized to the number of bytes actually set by the FSD (from the buffer pointed to by the FileInformation argument).

`FileInformation`

A caller-allocated buffer, containing information about the modified attributes of the target file. The format of the supplied information is defined by the FileInformationClass argument.

Length

The size, in bytes, of the buffer supplied by the caller.

`FileInformationClass`

Used by the caller to specify the type of attributes being modified for the target file. See Chapter 10 for a detailed discussion on the types of attributes that can be modified by the caller and for corresponding structure definitions.

Return code

STATUS_SUCCESS indicates that the operation succeeded; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_CANNOT_DELETE
- STATUS_DIRECTORY_NOT_EMPTY

IRP

`AssociatedIrp.SystemBuffer`

Pointer to an I/O Manager-allocated buffer. The I/O Manager always allocates a system buffer to contain a copy of the user-supplied modified attributes for the file stream. This system buffer is deallocated by the I/O Manager after the IRP has been completed.

`Flags`

The IRP_BUFFERED_IO, IRP_DEALLOCATE_BUFFER, and IRP_DEFER_IO_COMPLETION flags are set. However, these are only used internally by the I/O Manager.*

I/O stack location

`MajorFunction`

`IRP_MJ_SET_INFORMATION`

`MinorFunction`

None.

`Flags`

None.

`Parameters.SetFile.Length`

The Length, specified by the caller, for the buffer in which information about modified attributes is supplied.

`Parameters.SetFile.FileInformationClass`

The type of attributes for which modified information has been provided by the user.

`Parameters.SetFile.FileObject`

The file object representing an open instance of the target directory for a rename/link operation.

`Parameters.SetFile.ReplaceIfExists`

Used during rename operations to reflect the value of the `ReplaceIfExists` field in the FILE_RENAME_INFORMATION structure.

`Parameters.SetFile.AdvanceOnly`

This flag is set to TRUE for a special request initiated by the Windows NT Cache Manager to indicate that the `ValidDataLength` for the file stream has been changed.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of the file whose attributes are being modified.

---

* See Chapter 4 for a discussion on the IRP_DEFER_IO_COMPLETION flag.

Notes

Some FileInformationClass types are handled directly by the I/O Manager (e.g., FilePositionInformation). See Chapter 10 for further details on how other FileInformationClass types are supported by file system drivers.

NtQueryEaFile()

```
NTSTATUS NtQueryEaFile(
    IN HANDLE                   FileHandle,
    OUT PIO_STATUS_BLOCK        IoStatusBlock,
    OUT PVOID                   Buffer,
    IN ULONG                    Length,
    IN BOOLEAN                  ReturnSingleEntry,
    IN PVOID                    EaList OPTIONAL,
    IN ULONG                    EaListLength,
    IN PULONG                   EaIndex OPTIONAL,
    IN BOOLEAN                  RestartScan
);
```

Parameters

FileHandle

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile() invocation.

IoStatusBlock

The caller must supply this argument to receive the results of the query extended attributes operation. The Information field in the IoStatusBlock is set to the number of bytes returned by the FSD (in the buffer pointed to by the Buffer argument).

Buffer

A caller-allocated buffer to receive information about extended attributes associated with the target file. Information for each matching extended attribute (returned by the FSD) is longword-aligned and is contained within a FILE_FULL_EA_INFORMATION structure.

Only complete FILE_FULL_EA_INFORMATION structures are returned by the FSD. The NextEntryOffset value in the structure (if nonzero) indicates the relative offset of the next entry in the buffer. Note that the FSD maintains context to determine the next extended attribute for which information must be returned.

Also note that the value of each named extended attribute begins after the end of the EaName (null-terminated) field in the FILE_FULL_EA_INFORMATION structure. The EaNameLength field in the structure does not include the null-terminator for the extended attribute; therefore, the value for each of the named extended attributes can be located by adding (EaNameLength + 1) to the address of EaName.

Length

The size, in bytes, of the buffer supplied by the caller.

`ReturnSingleEntry`

If TRUE, the caller only wants information on a single, matching extended attribute returned.

EaList

This optional buffer can contain a list of named extended attributes for which information must be returned by the FSD. The structure of each entry in this buffer is of type FILE_GET_EA_INFORMATION and is follows:

```
typedef struct _FILE_GET_EA_INFORMATION {
    ULONG NextEntryOffset;
    UCHAR EaNameLength;
    CHAR  EaName[1];
} FILE_GET_EA_INFORMATION, *PFILE_GET_EA_INFORMATION;
```

The I/O Manager checks to ensure that the contents of the EA list are consistent; each of the entries contained in the list must be longword-aligned and each entry must either point to a complete, valid next entry in the list or the NextEntryOffset value must be set to 0. If errors are encountered, the I/O Manager may return a warning code of STATUS_EA_LIST_INCONSISTENT.

`EaListLength`

The length of the EaList buffer if such a buffer is present; this argument should be set to 0 if EaList is set to NULL.

EaIndex

An optional, zero-based index value specified by the caller. The FSD will return information about extended attributes, beginning with the EA identified by this index. If, however, EaList is nonnull, this argument will be ignored.

`RestartScan`

Normally, the FSD begins the scan for extended attributes from the last extended attribute returned (based upon the immediately preceding query extended attributes request); however, this flag allows the caller to indicate whether the scan should begin with the first EA associated with the file stream. This flag is ignored if either EaList or EaIndex are nonnull.

Return code

STATUS_SUCCESS indicates that the operation succeeded and information on at least one extended attribute is being returned by the FSD; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error, an appropriate error code or a warning is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_NO_MORE_EAS
- STATUS_INVALID_EA_NAME
- STATUS_INVALID_EA_FLAG

IRP

`AssociatedIrp.SystemBuffer`

Any system buffer allocated by the I/O Manager to receive information about EAs from the FSD, if the FSD has specified DO_BUFFERED_IO in the device object flags.

`MdlAddress`

Any MDL created by the I/O Manager if the FSD has specified DO_DIRECT_IO in the device object flags.

`UserBuffer`

Pointer to the user-supplied buffer if neither DO_DIRECT_IO nor DO_BUFFERED__IO have been specified by the FSD. This field is effectively overridden by the presence of any MDL pointer in the MdlAddress field.

I/O stack location

`MajorFunction`

IRP_MJ_QUERY_EA

`MinorFunction`

None.

`Flags`

One or more of SL_RESTART_SCAN, SL_RETURN_SINGLE_ENTRY, and SL_INDEX_SPECIFIED.

`Parameters.QueryEa.Length`

The Length specified by the caller for the buffer in which information is received.

`Parameters.QueryEa.EaList`

A list of named EAs supplied by the caller. Note that the actual buffer passed-in to the FSD is a system buffer that was allocated by the Windows NT I/O

Manager. The I/O Manager copies the user-supplied EA list from the caller's buffer to the system buffer before sending the IRP to the FSD.

`Parameters.QueryEa.EaListLength`
The EaListLength specified by the caller to NtQueryEaFile().

`Parameters.QueryEa.Ealndex`
The starting index, to begin the scan from, specified by the caller.

`DeviceObj ect`
Points to the FSD-created device object representing the mounted logical volume.

`FileObject`
File object representing the open instance of the target file stream.

Notes

The NtQueryEaFile() is an inherently synchronous I/O operation. The I/O Manager will block the requesting thread if STATUS_PENDING is received by the FSD.

The FSD returns information on the following number of extended attributes:

- A single extended attribute if either ReturnSingleEntry is TRUE, or if the supplied EaList describes only a single named extended attribute.
- The number of matching extended attributes for which full information can be returned in the caller-supplied buffer, constrained by the length of the buffer.
- The total number of associated extended attributes associated with the target file stream, or the total number of matching extended attributes as described by the caller in the EaList buffer.

If an error was encountered by the FSD (e.g., an invalid character in an EaName), the Information field in the loStatusBlock argument contains the byte offset to the EA entry that caused the failure, otherwise, it contains the number of bytes of extended attributes information returned by the FSD.

NtSetEaFile()

```
NTSTATUS NtSetEaFile(
    IN HANDLE              FileHandle,
    OUT PIO_STATUS_BLOCK   loStatusBlock,
    OUT PVOID              Buffer,
    IN ULONG               Length,
);
```

Parameters

FileHandle

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile() invocation.

loStatusBlock

The caller must supply this argument to receive the results of the set extended attributes operation. The Information field in the loStatusBlock is set to the number of bytes written by the FSD from the buffer pointed to by the Buffer argument.

Buffer

A caller-allocated buffer containing the extended attributes to be associated with the target file. Information about each matching extended attribute must be longword-aligned and must be contained within a FILE_FULL_EA_INFORMATION structure. The NextEntryOffset value in the structure (if nonzero) must indicate the relative offset of the next entry in the buffer.

As in the case of the NtQueryEaFile() function described earlier, the value of each named extended attribute must begin immediately after the end of the EaName (null-terminated) field in the FILE_FULL_EA_INFORMATION structure. The EaNameLength field in the structure should not include the null-terminator for the extended attribute; therefore, the value for each of the named extended attributes can be located by the FSD by adding (EaNameLength + 1) to the address of EaName.

Length

The size, in bytes, of the buffer supplied by the caller.

Return code

STATUS_SUCCESS indicates that the operation succeeded; STATUS_PENDING indicates that the operation will be performed asynchronously by the FSD.

In the case of an error, an appropriate error code or a warning is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_INVALID_EA_NAME
- STATUS_INVALID_EA_FLAG

IRP

`Associatedlrp.SystemBuffer`

Any system buffer, allocated by the I/O Manager, containing a copy of the information about modified/new EAs provided by the caller if the FSD has specified DO_BUFFERED_IO in the device object flags.

MdlAddress

Any MDL created by the I/O Manager if the FSD has specified DO_DIRECT_10 in the device object flags.

UserBuffer

The pointer to the user-supplied buffer if neither DO_DIRECT_IO nor D0_BUFFERED_IO have been specified by the FSD. This field is effectively overridden by the presence of any MDL pointer in the MdlAddress field.

I/O stack location

`MajorFunction`

`IRP_MJ_SET_EA`

`MinorFunction`

`None.`

`Parameters.SetEa.Length`

The Length specified by the caller for the buffer in which information is supplied.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of the target file stream.

Notes

The NtSetEaFile() is an inherently synchronous I/O operation. The I/O Manager will block the requesting thread if STATUS_PENDING is received by the FSD.

The FSD uses the following rules in applying caller-specified EAs to the target file stream:

- If a supplied EA has a unique EaName among the existing EAs associated with the file stream, the FSD adds the new user-supplied EA to the list of EAs associated with the file.

- If the supplied EA has an EaName that matches an existing EA associated with the file stream and if the supplied EaValueLength is nonzero, the FSD will replace the existing EA with the user-supplied extended attribute.
- If the supplied EA has an EaName that matches an existing EA associated with the file stream and if the supplied EaValueLength is zero length, the FSD will delete the existing EA.

If an error was encountered by the FSD (e.g., an invalid character in an EaName), the Information field in the loStatusBlock argument contains the byte offset to the EA entry that caused the failure; otherwise, it contains the number of bytes of extended attributes information applied by the FSD to the file stream.

NtLockFile()

```
NTSTATUS NtLockFile(
    IN HANDLE                FileHandle,
    IN HANDLE                Event OPTIONAL,
    IN PIO_APC_ROUTINE       ApcRoutine OPTIONAL,
    IN PVOID                 ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK     loStatusBlock,
    IN PLARGE_INTEGER        ByteOffset,
    IN PLARGE_INTEGER        Length,
    IN PULONG                Key,
    IN BOOLEAN               FailImmediately,
    IN BOOLEAN               ExclusiveLock
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile () or NtOpenFileO invocation.

Event (optional)

Caller can wait for the supplied event object (created by the caller) for completion of the lock request. The event will be signaled by the I/O Manager when the lock-file operation is completed.

ApcRoutine (optional)

An optional, caller-supplied APC routine invoked by the I/O Manager when the lock-file operation completes.

ApcContext (optional)

A caller-determined context to be passed-in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

loStatusBlock

The caller must supply this argument to receive the results of the lock-file operation. The Information field in the loStatusBlock is set to the number of bytes locked by the FSD.

ByteOffset

The starting byte offset for the byte-range to be locked on behalf of the caller.

Length

The number of bytes to be locked.

Key

The Key is a caller-defined (opaque) value associated with the locked byte range. This value can be used to selectively share data between threads belonging to the same process (if a unique value is chosen by the requesting thread).

FailImmediately

If set to TRUE and if the lock cannot be obtained immediately by the FSD for the caller (e.g., some other thread was previously granted a conflicting lock on an overlapping byte range), the lock request is completed with an appropriate error code. If, however, Faillmmediately is set to FALSE, the request will block indefinitely until the lock can be obtained (all conflicting locks held by other threads on overlapping byte ranges have been released).

ExclusiveLock

Specifies whether an exclusive (write) lock should be acquired or whether a shared (read) lock is sufficient.

Return code

STATUS_SUCCESS indicates that the operation succeeded, and the lock was granted; STATUS_PENDING is returned if the requesting thread wishes to wait for the byte-range lock and the lock cannot be immediately obtained (the IRP is queued by the FSD/FSRTL package).

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

« STATUS_ACCESS_DENIED

- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_LOCK_NOT_GRANTED

I/O stack location

`MajorFunction`

`IRP_MJ_LOCK_CONTROL`

`MinorFunction`

`IRP_MN_LOCK`

`Flags`

One or more of SL_FAIL_IMMEDIATELY and SL_EXCLUSIVE_LOCK.

`Parameters.LockControl.Length`

The byte-range Length specified by the caller.

`Parameters.LockControl.Key`

The Key specified by the caller.

`Parameters.LockControl.ByteOffset`

The starting ByteOffset specified by the caller.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of the file for which a byte-range lock has been requested.

Notes

Byte-range locks obtained by a thread on Windows NT platforms are mandatory locks. Therefore, the FSD is responsible for enforcing the semantics associated with the lock when subsequent I/O requests are received for the target file stream. To check whether an I/O operation should be allowed to proceed for a locked byte range, the FSD uses the following attributes associated with the locked range:

- The starting byte offset for the locked range
- The number of bytes that have been locked
- The process that owns the locked range
- The Key value associated with the locked range

Byte-range locks are owned by processes and are not associated with individual threads within a process. Therefore, to control access to locked byte-ranges by multiple threads within the same process, a unique Key value should be associated with the locked byte range.

Exclusive locks prohibit any read or write access by any other process other than the owning process for the locked byte range. Shared locks allow other processes to continue to read the data contained within the locked range but do not allow other processes to modify such data. Byte-range exclusive locks requested by a process cannot overlap with any other locked range within the file.

Note that callers can request byte-range locks that start or extend beyond the current end-of-file. This allows the requester to control who can extend the file stream.

NtUnlockFile()

```
NTSTATUS NtUnlockFile(
    IN HANDLE               FileHandle,
    OUT PIO_STATUS_BLOCK    loStatusBlock,
    IN PLARGE_INTEGER       ByteOffset,
    IN PLARGE_INTEGER       Length,
    IN PULONG               Key
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful `NtCreateFile()` or `NtOpenFileO` invocation.

`loStatusBlock`

The caller must supply this argument to receive the results of the unlock-file operation. The `Information` field in the `loStatusBlock` is set to the number of bytes unlocked by the FSD.

`ByteOffset`

The starting byte offset for the byte range to be unlocked on behalf of the caller. This value must match exactly the starting `ByteOffset` supplied in a previous `NtLockFile()` request.

`Length`

The number of bytes to be unlocked. This value must match exactly the `Length` supplied in a previous `NtLockFile()` request.

Key

The Key is a caller-defined (opaque) value associated with the locked byte range. This value must match exactly the Key value supplied in a previous `NtLockFile()` request.

***Return** code*

STATUS_SUCCESS indicates that the operation succeeded and the lock was released; STATUS_PENDING is returned if the FSD processes the request asynchronously.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- `STATUS_ACCESS_DENIED`
- `STATUS_INSUFFICIENT_RESOURCES`
- `STATUS_INVALID_PARAMETER`
- `STATUS_INVALID_DEVICE_REQUEST`
- `STATUS_RANGE_NOT_LOCKED`

I/O stack location

`MajorFunction`

`IRP_MJ_LOCK_CONTROL`

`MinorFunction`

One of the following:

IRP_MN_UNLOCK_SINGLE

The single, locked byte range described in the IRP should be unlocked.

`IRP_MN_UNLOCK_ALL`

All previously locked byte ranges owned by the requesting process should be unlocked.

IRP_MN_UNLOCK_ALL_BY_KEY

All previously locked byte-ranges, owned by the requesting process that match the supplied Key value, should be unlocked.

Flags

None.

`Parameters.LockControl.Length`

The byte-range Length specified by the caller. This should be exactly equal to the Length value supplied in a previous request to NtLockFile().

Parameters.LockControl.Key

The Key specified by the caller.

`Parameters.LockControl.ByteOffset`

The starting ByteOffset specified by the caller. This should be exactly equal to the ByteOffset value supplied in a previous request to `NtLockFile()`.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of the file for which an unlock operation has been requested.

Notes

Only the process that owns a particular byte-range lock can successfully request that the lock be released. Whenever a process closes all open handles for a particular file stream, all outstanding byte-range locks owned by the process for the file stream will be released.

NtQuery VolumeInformationFile()

```
NTSTATUS NtQueryVolumelnformationFile(
    IN HANDLE                  FileHandle,
    OUT PIO_STATUS_BLOCK       loStatusBlock,
    OUT PVOID                  Fslnformation,
    IN ULONG                   Length,
    IN FS_INFORMATION_CLASS    FslnformationClass
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile () invocation for any file or directory contained in the target logical volume, or from a successful open request on either the target volume or the underlying device object.

`loStatusBlock`

The caller must supply this argument to receive the results of the query volume information operation. The Information field in the loStatusBlock is set to the number of information bytes returned by the FSD.

`Fslnformation`

A caller-allocated buffer in which volume information is returned. The structure of returned information depends upon the value of the FsInformationClass argument.

Length

The size of the Fslnformation buffer.

`FsInformationClass`

The type of information requested by the user. This can be one of the following:

`FileFsVolumelnformation`

The following structure defines the format of the information returned by the FSD:

```
typedef struct _FILE_FS_VOLUME_INFORMATION {
LARGE_INTEGER    VolumeCreationTime;
ULONG            VolumeSerialNumber;
ULONG            VolumeLabelLength;
BOOLEAN          SupportsObjects;
WCHAR            VolumeLabel[1];
} FILE_FS_VOLUME_INFORMATION, *PFILE_FS_VOLUME_INFORMATION;
```

`FileFsSizeInformation`

The following structure defines the format of the information returned by the FSD:

```
typedef struct _FILE_FS_SIZE_INFORMATION {
LARGE_INTEGER    TotalAllocationUnits;
LARGE_INTEGER    AvailableAllocationUnits;
ULONG            SectorsPerAllocationUnit;
ULONG            BytesPerSector;
} FILE_FS_SIZE_INFORMATION, *PFILE_FS_SIZE_INFORMATION;
```

`FileFsDeviceInformation`

The following structure defines the format of the information returned by the FSD:

```
typedef struct _FILE_FS_DEVICE_INFORMATION {
DEVICE_TYPE       DeviceType;
ULONG            Characteristics;
} FILE_FS_DEVICE_INFORMATION, *PFILE_FS_DEVICE_INFORMATION;
```

`FileFsAttributeInformation`

The following structure defines the format of the information returned by the FSD:

```
typedef struct _FILE_FS_ATTRIBUTE_INFORMATION {
ULONG             FileSystemAttributes;
LONG              MaximumComponentNameLength;
ULONG             FileSystemNameLength;
WCHAR             FileSystemName[1];
} FILE_FS_ATTRIBUTE_INFORMATION, *PFILE_FS_ATTRIBUTE_INFORMATION;
```

Return code

STATUS_SUCCESS indicates that the operation succeeded and the volume information has been returned by the FSD; STATUS_PENDING is returned if the FSD decides to process the request asynchronously.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST
- STATUS_BUFFER_OVERFLOW

IRP

`Associatedlrp.SystemBuffer`

The I/O Manager allocates a system buffer in which the FSD can return the requested volume information. The I/O Manager copies the returned information into the caller's buffer once the IRP is completed by the FSD.

I/O stack location

`MajorFunction`

`IRP_MJ_QUERY_VOLUME_INFORMATION`

`MinorFunction`

None.

`Flags`

None.

`Parameters.QueryVolume.Length`

The Length of the buffer provided by the caller.

`Parameters.QueryVolume.FslnformationClass`

The FsInformationClass value specified by the caller. This determines the type of information returned by the FSD.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

The file object representing the open instance of a file, directory, volume, or device using which a query volume information operation has been requested.

Notes

Regardless of the type of access requested in the open request for a file, directory, device, or volume, the user can always request volume information using the file handle received from the successful open operation.

NtSetVolumeInformationFile()

```
NTSTATUS NtSetVolumelnformationFile(
    IN HANDLE              FileHandle,
    OUT PIO_STATUS_BLOCK   loStatusBlock,
    IN PVOID               Fslnformation,
    IN ULONG               Length,
    IN FS INFORMATION_CLASSFsInformationClass
);
```

Parameters

FileHandle

Returned to the caller from a previous successful NtCreateFile () or NtOpenFile ( ) invocation on the target volume.

loStatusBlock

The caller must supply this argument to receive the results of the set volume information operation. The Information field in the loStatusBlock is set to the number of information bytes written by the FSD.

Fslnformation

A caller-allocated buffer in which volume information is supplied. The structure of supplied information depends upon the value of the FsInformationClass argument.

Length

The size of the Fslnformation buffer.

FsInformationClass

The type of information provided by the user. Currently, this can be the following:

FileFsLabelInformation

The following structure defines the format of the information supplied by the user:

```
typedef struct _FILE_FS_LABEL_INFORMATION {
    ULONG VolumeLabelLength;
    WCHAR VolumeLabel[l];
} FILE_FS_LABEL_INFOKMATION, *PFILE_FS_LABEL_INFORMATION;
```

Return code

STATUS_SUCCESS indicates that the operation succeeded; STATUS_PENDING is returned if the FSD decides to process the request asynchronously.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST

IRP

`AssociatedIrp.SystemBuffer`
The I/O Manager allocates a system buffer into which the caller-provided volume information is copied before the IRP is dispatched to the FSD.

I/O stack location

`MajorFunction`
`IRP_MJ_SET_VOLUME_INFORMATION`

`MinorFunction`
None.

`Flags`
None.

`Parameters.SetVolume.Length`
The `Length` of the buffer provided by the caller.

`Parameters.SetVolume.FsInformationClass`
The FsInformationClass value specified by the caller. This determines the type of attribute to be modified for the logical volume.

`DeviceObject`
Points to the FSD-created device object representing the mounted logical volume.

`FileObject`
The file object representing the open instance of the logical volume on which a set volume information operation has been requested.

Notes

For the `FileFsLabelInformation` `FsInformation` class value, a value of 0 in the VolumeLabelLength field indicates that the current volume label (if any) should be removed. The FSD expects that any new volume label supplied by the caller should be a wide character string.

NtFsControlFile()

```
NTSTATUS NtFsControlFile(
    IN HANDLE              FileHandle,
    IN HANDLE              Event OPTIONAL,
    IN PIO_APC_ROUTINE     ApcRoutine OPTIONAL,
    IN PVOID               ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK   IoStatusBlock,
    IN ULONG               FsControlCode,
    IN PVOID               InputBuffer OPTIONAL,
    IN ULONG               InputBufferLength,
    OUT PVOID              OutputBuffer OPTIONAL,
    IN ULONG               OutputBufferLength
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile () or NtOpenFileO invocation.

Event (optional)

The caller can wait for the supplied event object (created by the caller) for completion of the asynchronous FSCTL request. The event will be signaled by the I/O Manager when the FSCTL operation is completed.

ApcRoutine (optional)

An optional, caller-supplied APC routine invoked by the I/O Manager when the FSCTL operation completes.

ApcContext (optional)

The caller-determined context to be passed-in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

`loStatusBlock`

The caller must supply this argument to receive the results of the FSCTL operation. The Information field in the loStatusBlock is set to the number of bytes returned by the FSD in the OutputBuffer (if any).

`FsControlCode`

The FSCTL code value specifying the type of file system control function requested.

`InputBuffer`

A caller-allocated buffer in which information to be sent to the FSD is supplied.

`InputBufferLength`

The size of the input buffer.

`OutputBuffer`

A caller-allocated buffer in which the FSD returns information to the caller.

`OutputBufferLength`

The size of the output buffer.

Return code

STATUS_SUCCESS indicates that the operation succeeded; STATUS_PENDING is returned if the FSD processes the request asynchronously.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST

IRP

AssociatedIrp.SystemBuffer

If the FSCTL code value specifies METHOD_BUFFERED or METHOD_IN_DIRECT/METHOD_OUT_DIRECT, the I/O Manager initializes this field with a pointer to a system buffer allocated by the I/O Manager. For METHOD_BUFFERED, the size of the allocated system buffer is equal to the size of the larger of the two buffers supplied by the caller (the InputBuffer and the OutputBuffer).* For METHOD_IN_DIRECT/METHOD_OUT_DIRECT, the I/O Manager allocates a system buffer to correspond to any InputBuffer supplied by the caller.

MdlAddress

If the FSCTL code value specifies METHOD_IN_DIRECT/METHOD_OUT_DIRECT and the OutputBuffer argument supplied by the requesting thread is nonnull, the I/O Manager allocates an MDL describing the caller's OutputBuffer and initializes the MdlAddress field with the MDL pointer value. Note that the physical pages backing this MDL are locked into memory by the I/O Manager.

UserBuffer

If the FSCTL code value specifies METHOD_NEITHER, the I/O Manager initializes this field with the OutputBuffer pointer provided by the caller.

Flags

Set to IRP_MOUNT_COMPLETION and IRP_SYNCHRONOUS_PAGING_IO for mount volume and verify volume FSCTL requests.

I/O stack location

MajorFunction

IRP_MJ_FILE_SYSTEM_CONTROL

---

* The I/O Manager copies the contents of the InputBuffer into the system buffer before dispatching the IRP to the FSD. When the IRP is completed and if the caller had provided an OutputBuffer, the I/O Manager copies any information returned by the FSD back into the caller's OutputBuffer.

MinorFunction

One of the following:

`IRP_MN_MOUNT_VOLUME`

A mount request is being issued to the FSD.

`IRP_MN_LOAD_FILE_SYSTEM`

The FSD is being loaded by a mini file system recognizer.

`IRP_MN_VERIFY_VOLUME`

A verify volume request is issued to the FSD.

`IRP_MN_USERLFS_REQUEST`

Set when a user FSCTL request is received by the I/O Manager, via an invocation to NtFsControlFile (), for either a private FSCTL request or for one of the set of public FSCTL requests supported by most FSDs and/or network redirectors.

Flags

Set to SL_ALLOW_RAW_MOUNT if a target volume is opened for direct access when MinorFunction is initialized to IRP_MN_VERIFY_VOLUME.

Mount requests

`Parameters.MountVolume.Vpb`

The VPB associated with the physical, virtual, or logical "real" device object representing the media on which the logical volume should be mounted.

`Parameters.MountVolume.DeviceObject`

Pointer to the device object representing the partition on the device object on which the logical volume should be mounted. Note that the pointer may refer to some intermediate (filter driver) device object structure that has been attached to the target device object.

`DeviceObject`

Points to the FSD-created device object representing the file system driver (or to the highest-layered filter device object attached to the FSD device object).

`FileObject`

Initialized to NULL.

Load FSD request

`DeviceObject`

Points to the file system recognizer driver-created device object representing the file system recognizer driver.

`FileObject`

Initialized to NULL.

Verify volume requests

`Parameters.VerifyVolume.Vpb`
The VPB associated with the physical, virtual, or logical "real" device object representing the media on which the mounted logical volume should be verified.

`Parameters.VerifyVolume.DeviceObject`
Pointer to the device object representing the media containing the mounted logical volume to be verified.

`DeviceObject`
Points to the FSD-created device object representing the mounted volume to be verified.

`FileObject`
Initialized to NULL.

User FSCTL requests

`Parameters.FileSystemControl.OutputBufferLength`
The OutputBufferLength specified by the caller.

`Parameters.FileSystemControl.InputBufferLength`
The InputBufferLength specified by the caller.

`Parameters.FileSystemControl.FsControlCode`
The FsControlCode specified by the caller.

`Parameters.FileSystemControl.Type3InputBuffer`
Used when the FSCTL code value specifies METHOD_NEITHER for handling user buffers, this field contains a pointer to the user-supplied InputBuffer.

`DeviceObject`
Points to the FSD-created device object representing the mounted volume.

`FileObject`
Initialized to the file object instance representing an open file/directory or volume.

Notes

When dispatching any I/O read request to a lower-level driver while processing a verify volume request itself, the FSD must set the SL_OVERRIDE_VERIFY_VOLUME flag in the next I/O stack location before forwarding the IRP. See Chapter 11 for a detailed discussion on how FSDs process FSCTL requests.

NtDeviceloControlFileO

```
NTSTATUS NtDeviceIoControlFile(
    IN HANDLE              FileHandle,
    IN HANDLE              Event OPTIONAL,
    IN PIO_APC_ROUTINE     ApcRoutine OPTIONAL,
    IN PVOID               ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK   loStatusBlock,
    IN ULONG               loControlCode,
    IN PVOID               InputBuffer OPTIONAL,
    IN ULONG               InputBufferLength,
    OUT PVOID              OutputBuffer OPTIONAL,
    IN ULONG               OutputBufferLength
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile () or NtOpenFile () invocation. The target file or device must have been opened for Direct Access Storage Device (DASD) access.

Event (optional)

The caller can wait for the supplied event object (created by the caller), for completion of the asynchronous IOCTL request. The event will be signaled by the I/O Manager when the IOCTL operation is completed.

ApcRoutine (optional)

An optional, caller-supplied APC routine invoked by the I/O Manager when the IOCTL operation completes.

ApcContext (optional)

A caller-determined context to be passed-in to the ApcRoutine. This argument should be NULL if ApcRoutine is NULL.

`loStatusBlock`

The caller must supply this argument to receive the results of the IOCTL operation. The Information field in the loStatusBlock is set to the number of bytes returned by the FSD in the OutputBuffer (if any).

`FsControlCode`

The IOCTL code value specifying the type of device I/O control function requested.

InputBuffer

A caller-allocated buffer in which information to be sent to the FSD is supplied.

`InputBufferLength`

The size of the input buffer.

`OutputBuffer`
A caller-allocated buffer in which the FSD returns information to the caller.

`OutputBufferLength`
The size of the output buffer.

Return code

STATUS_SUCCESS indicates that the operation succeeded; STATUS_PENDING is returned if the FSD processes the request asynchronously.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST

IRP

`AssociatedIrp.SystemBuffer`
If the IOCTL code value specifies METHOD_BUFFERED or METHOD_IN_DIRECT/METHOD_OUT_DIRECT, the I/O Manager initializes this field with a pointer to a system buffer allocated by the I/O Manager. For METHOD_BUFFERED, the size of the allocated system buffer is equal to the size of the larger of the two buffers supplied by the caller (the InputBuffer and the OutputBuffer).* For METHOD_IN_DIRECT/METHOD_OUT_DIRECT, the I/O Manager allocates a system buffer to correspond to any InputBuffer supplied by the caller.

MdlAddress
If the IOCTL code value specifies METHOD_IN_DIRECT/METHOD_OUT_DIRECT and the OutputBuffer argument supplied by the requesting thread is nonnull, the I/O Manager allocates an MDL describing the caller's OutputBuffer and initializes the MdlAddress field with the MDL pointer value. Note that the physical pages backing this MDL are locked into memory by the I/O Manager.

UserBuffer
If the IOCTL code value specifies METHOD_NEITHER, the I/O Manager initializes this field with the OutputBuffer pointer provided by the caller.

---

* The I/O Manager copies the contents of the InputBuffer into the system buffer before dispatching the IRP to the FSD. When the IRP is completed and if the caller had provided an OutputBuffer, the I/O Manager copies any information returned by the FSD back into the caller's OutputBuffer.

I/O stack location

`MajorFunction`
: `IRP_MJ_DEVICE_CONTROL` or `IRP_MJ_INTERNAL_DEVICE_CONTROL`

`MinorFunction`
: None.

`Flags`
: Can be set to SL_OVERRIDE_VERIFY_VOLUME by the FSD when requesting I/O operations from the lower-level driver while processing verify-volume requests.

`Parameters.DeviceIoControl.OutputBufferLength`
: The OutputBufferLength specified by the caller.

`Parameters.DeviceIoControl.InputBufferLength`
: The InputBufferLength specified by the caller.

`Parameters.DeviceIoControl.FsControlCode`
: The FsControlCode specified by the caller.

`Parameters.DeviceIoControl.Type3InputBuffer`
: Used when the IOCTL code value specifies METHOD_NEITHER for handling user buffers, this field contains a pointer to the user-supplied InputBuffer.

`DeviceObject`
: Points to the FSD-created device object representing the mounted logical volume or target device.

`FileObject`
: Initialized to the file object instance representing an open file or device.

Notes

Most device IOCTL requests are forwarded by the FSD to lower-level device drivers managing the physical/virtual/logical device on which the volume has been mounted. See Chapter 11 for a detailed discussion on how FSDs process IOCTL requests.

Note that the IRP_MJ_SCSI IOCTL code has been defined to *be* the same as IRP_MJ_INTERNAL_DEVICE_CONTROL control code value.

NtDeleteFile()

```
NTSTATUS NtDeleteFile(
    IN POBJECT_ATTRIBUTES ObjectAttributes
);
```

This system call is functionally equivalent to invoking NtSetInformationFile() with FileInformationClass set to FileDispositionInformation.

NtFlushBuffersFile()

```
NTSTATUS NtFlushBuffersFile(
    IN HANDLE               FileHandle,
    OUT PIO_STATUS_BLOCK    loStatusBlock,
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful `NtCreateFile()` or `NtOpenFile()` invocation.

If the supplied handle represents an open instance of either the mounted logical volume or the root directory on the mounted logical volume, all cached data for open files belonging to the mounted logical volume will be flushed by the FSD. If, however, the handle refers to an instance of any other open directory on the volume, no data will be flushed to disk.

If the handle represents an open instance of a specific file, the FSD will write the cached data for the file to secondary storage by the FSD.

`loStatusBlock`

The caller must supply this argument to receive the results of the flush buffers operation. The `Information` field in the `loStatusBlock` is set to the number of bytes flushed to secondary storage by the FSD.

Return code

STATUS_SUCCESS indicates that the operation succeeded.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INSUFFICIENT_RESOURCES
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST

I/O stack location

`MajorFunction`

IRP_MJ_FLUSH_BUFFERS

`MinorFunction`

None.

`DeviceObject`

Points to the FSD-created device object representing the mounted logical volume.

`FileObject`

Initialized to the file object instance representing an open file, directory, or volume.

Notes

Chapter 11 discusses how the flush file buffers IRP is handled by the FSD.

NtCancelIoFile()

```
NTSTATUS NtCancelIoFile(
    IN HANDLE              FileHandle,
    OUT PIO_STATUS_BLOCK    loStatusBlock,
);
```

Parameters

`FileHandle`

Returned to the caller from a previous, successful NtCreateFile() or NtOpenFile() invocation.

`loStatusBlock`

The caller must supply this argument to receive the results of the flush buffers operation.

Return code

STATUS_SUCCESS indicates that the operation succeeded.

In the case of an error, an appropriate error code is returned. This includes (but is not limited to) the following return code values:

- STATUS_ACCESS_DENIED
- STATUS_INVALID_PARAMETER
- STATUS_INVALID_DEVICE_REQUEST

Notes

This system call will not return control back to the caller until all pending I/O requests initiated by the requesting thread using the particular file handle, have been either canceled or completed.

Requests initiated by other threads belonging to the same process or by the same thread but using different file handles will not be affected.

This appendix has listed some of the Windows NT I/O-Manager-provided system services that you can use either from a user-space application or from within a kernel-mode driver. There is a cost, however, associated with using such routines directly. This cost (especially for user-space applications) is the potential loss of portability that your software will suffer if and when these system services are changed and/or made obsolete by Microsoft. The benefit is that certain functionality becomes easier to request by using such Windows NT system services directly.