Portable Systems Group

Windows NT I/O System Specification

Author: *Darryl E. Havens*

Revision 1.7, May 1, 1995

1. Introduction

This specification describes the basic overall API for the I/O system of the **Windows NT** operating system. The I/O system is responsible for the management of all input and output operations in the system and for presenting the remainder of the system with a uniform and device-independent view of the various devices connected to the system.

The I/O system provides an interface for the user to perform I/O to various devices attached to the machine. The I/O operations in this API provide the user with a rich set of primitives to manipulate files and devices in such a way as to hide most of the particulars of how the device actually works.

The I/O system also provides system programmers with the ability to write their own device drivers for those devices that **Windows NT** does not support as part of its regular SDK. This part of the I/O system is documented in the *Windows NT Driver Model Specification* and is beyond the scope of this specification.

This specification does not attempt to exhaustively enumerate all error conditions that occur on all paths or indicate the errors that can occur after calling an API.

2. Overview

The user interface model that **Windows NT** uses for I/O consists of several different routines that perform such operations as Open, Read, Write, Close, etc. For other operations that are not included in the general set of routines, there is an **NtDeviceIoControlFile** service. This service allows device-dependent information to be passed to and from the device in a well structured manner. Likewise, the **NtFsControlFile** service which allows file-system-dependent information to be passed to and from the file system in a well structured manner.

The I/O system is designed to support both OS/2 and POSIX I/O operations easily to provide source code compatibility with those standards. This allows users familiar with those systems to continue to program using those interfaces without having to learn a new I/O programming model. The OS/2 and POSIX subsystems emulate the I/O services on top of the **Windows NT** services.

To perform I/O operations in **Windows NT**, a *file handle* must be specified. File handles are obtained by calling the **NtCreateFile** or **NtOpenFile** services. These services either create or open a file and return a handle to it. Alternatively, they may open a device directly and return a handle to the device. In each case the handle is still referred to as a "file handle" throughout the description of the APIs in this specification.

From the point of view of the object management system, a file is a *persistent object*. That is, a *file object* is treated like any other object in the system except that it remains intact across system boots. Handles to file objects, and therefore devices (depending on how the "file" was opened) are usable in the object system.

Some of the I/O interfaces in **Windows NT** are synchronous and others are asynchronous. For the latter type, it is up to the caller to wait for the I/O operation to complete. This may be done in either an alertable or a non-alertable manner. A file object in **Windows NT** is a waitable object and can therefore be used to synchronize completion of an I/O operation on the file. When a request is made to perform an operation on a file, the file object is set to the Not-Signaled state. When the operation completes, the file object is set to the Signaled state.

Each asynchronous I/O service also optionally accepts an event and/or the address of an *Asynchronous Procedure Call* **(APC)** to be executed when the operation completes. If an event is specified, the system sets it to the Not-Signaled state when the I/O operation is requested and sets it to the Signaled state when the I/O operation completes. The system will not normally set both the File object and the event to the Signaled state. That is, if an event is specified, then the event should be used for I/O completion synchronization; otherwise the file object handle should be used.

If an APC is specified, the procedure is invoked when the I/O completes with a parameter that is also supplied to the service. The procedure is also passed the address of the I/O status block discussed below.

Likewise, it is also possible to synchronize the completion of I/O operations through the use of *I/O Completion objects*. An I/O Completion object may be associated with a file such that a pool of threads may wait on the completion of all I/O associated with the object.

All service calls include the address of an *I/O status block*. This variable contains information about the success or failure of the operation once the operation has been completed. This allows the caller to determine the status of the operation once the file object or the event has been set to the Signaled state, or the APC routine has been invoked. Upon completion of the I/O operation the variable may also contain more information that is service-dependent.

It should be noted that performing multiple operations on a file at the same time requires that each operation be synchronized. That is, requesting two asynchronous reads from a file and then waiting on the file object will not guarantee that both operations have completed. In the same manner, using the same event to synchronize these two operations will not work either. Each operation must have its own event associated with it, or the caller must set up an APC which will be able to distinguish between the completion of each request.

Using an I/O system design whose primary data movement operations can be totally asynchronous makes writing faster programs easier. It frees the programmer from inventing methods of passing I/O requests to another thread to gain parallelism. This means that the main loop need not be blocked or concerned with the completion of I/O operations until it absolutely requires the requested data.

This particular design also allows servers and network servers to be written so that it is not necessary to dedicate a thread in the server to each request or to each client. Because the APC routine can be executed any time the server thread is ready for it, a single server thread can potentially perform I/O for an unlimited number of clients using very few system resources.

Since all potentially long I/O operations are asynchronous, a thread that is waiting on an I/O operation in an alertable manner may fall out of the wait. This allows programs to be written so that rundown and cleanup are much easier to control. Likewise, because the user has a choice, programs can still be written to block in a non-alertable manner and simply wait for the I/O operation to complete. More information on alerts can be found in the *Windows NT Process Structure* specification.

The **Windows NT** I/O system provides one optimization that can be used to save extraneous system calls. If the request for an operation is successfully queued to a driver for completion later, then the return status from the service is *STATUS_PENDING*. However, if the operation successfully completes before the service returns because the driver immediately completed the operation, then a status of *STATUS_SUCCESS* is returned.

It is also possible to write an application that ignores the fact that the **Windows NT** I/O system is asynchronous by specifying that all I/O calls for a particular file object be performed synchronously. Further, the I/O operations are selectively alertable or non-alertable. This option is requested when the file is opened or created. If the I/O is being performed with alerts enabled, then it is possible for the I/O operation to be interrupted by an alert to the thread. It is also possible to specify that no alerts may be taken during the I/O operation.

If an application is performing I/O to a file in an alertable manner, then it must be written to be prepared for the I/O to fail because an alert occurred or an APC was delivered. In either case the I/O operation must be restarted by invoking the API again.

When the I/O system is performing synchronous I/O on a file object, it also maintains a current file pointer context for the file. This file pointer may be read or written using APIs provided by the I/O system. Furthermore, they are automatically updated whenever the file is read or written according to the number of bytes transferred. It is also possible to set the file pointer context on the read or write operation.

Performing synchronous I/O on a file object also means that the I/O to the file is serialized. That is, if Thread A has issued an I/O operation on a file and Thread B issues an I/O operation using the same file object, then Thread B will wait (alertable or non-alertable, depending on how the file was opened) until Thread A's I/O completes.

All of these features help the user deal with the system and use it to perform I/O the way that he wants to work. He can still take advantage of APC routines, for example, even if he is performing synchronous I/O. However, he doesn't have to if that isn't what he needs.

In order to access a file or a device, the caller must have permission to access the device in the requested manner. For example, some devices are considered single user devices. This is accomplished through the object management system in **Windows NT**. The object that represents a device is called a *device object*. Device objects may be created by device drivers using the *exclusive attribute*. This attribute indicates that only one process may open the object. Any other attempt to open a device from a process other than the "owning" process will fail. This implies that it is possible for a process to "own" a device. Of course, since handles can be inherited by child processes, then children of the owning process may share the device with the parent process.

A file or a device may specify an *Access Control List* **(ACL)**. An ACL is a list of *Access Control Entries* **(ACEs)** that specify what access rights a user has to the file or device. The user must have the requested access in order to successfully perform operations on the object.

Windows NT also provides file sharing among threads within a process and between processes. Because of the object architecture design used in **Windows NT**, it is possible for all of the threads within a process to access a file that one of the threads "opened" by using the returned file handle. Furthermore, a process that is created by one of the threads may also have access to the file if the file object is opened so that its handle is inheritable.

Finally, **Windows NT** provides file sharing by allowing multiple processes to open the same file. A file can be opened so that other processes may read, write, or perform both or neither operation on the file.

3. User APIs

The following sections present the user interface to the I/O system.

3.1 Create/Open File/Device Services

When a user wishes to access a file or a device, he must create or open it. This causes a handle to be returned that can then be used to manipulate the file or device in subsequent calls.

File handles are closed via the generic **NtClose** service. This service is discussed elsewhere in the **Windows NT** documentation. It should be noted that, just like all other system objects, a file is not actually deleted until all of the valid handles to it are closed and no referenced pointers remain.

The user APIs that supports creating and opening files and opening devices is as follows:

NtCreateFile - Create or open a file and return a file handle.
NtOpenFile - Open a file and return a file handle.

3.1.1 Creating and Opening Files

A file can be created or opened using the **NtCreateFile** service:

NTSTATUS
NtCreateFile(
OUT PHANDLE *FileHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes,*
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN PLARGE_INTEGER *AllocationSize* **OPTIONAL**,
IN ULONG *FileAttributes,*
IN ULONG *ShareAccess,*
IN ULONG *CreateDisposition,*
IN ULONG *CreateOptions,*
IN PVOID *EaBuffer* **OPTIONAL,**
IN ULONG *EaLength*
);

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the file.

DesiredAccess Flags

SYNCHRONIZE - The file handle may be waited on to synchronize with the completion of the I/O operation.

DELETE - The file may be deleted.

READ_CONTROL - The ACL and ownership information associated with the file may be read.

WRITE_DAC - The Discretionary ACL associated with the file may be written.

WRITE_OWNER - Ownership information associated with the file may be written.

FILE_READ_DATA - Data may be read from the file.

FILE_WRITE_DATA - Data may be written to the file.

FILE_EXECUTE - Data may be faulted into memory from the file via paging I/O.

FILE_APPEND_DATA - Data may only be appended to the file.

FILE_READ_ATTRIBUTES - File attributes flags may be read.

FILE_WRITE_ATTRIBUTES - File attributes flags may be written.

FILE_READ_EA - Extended attributes associated with the file may be read.

FILE_WRITE_EA - Extended attributes associated with the file may be written.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES,* and *FILE_READ_EA.*

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA,* and *FILE_APPEND_DATA*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE, SYNCHRONIZE,* and *FILE_EXECUTE*.

For more information about the standard rights accesses, see the *Windows NT Local Security Specification*.

If the file being created or opened is a directory file, as specified in the *CreateOptions* argument, then the following types of access may be requested:

FILE_LIST_DIRECTORY - Files in the directory may be listed.

FILE_TRAVERSE - The directory may be traversed. That is, it may be in the pathname of a file.

FILE_READ_DATA, FILE_WRITE_DATA, FILE_EXECUTE, and *FILE_APPEND_DATA* accesses are not valid when creating or opening a directory file.

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, a security descriptor, a quality of service descriptor, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be created or opened. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

PSECURITY_DESCRIPTOR *SecurityDescriptor* - Optionally specifies the security descriptor that should be applied to the file. The ACLs specified by the security descriptor are only applied to the file if it is created. If not supplied and the file is created, then the ACL placed on the file is file-system-dependent, but most file systems propagate some part of the ACL from the parent directory file combined with the caller's default ACL.

PSECURITY_QUALITY_OF_SERVICE *SecurityQualityOfService* - Specifies the access a server should be given to the client's security context. This field is only used when a connection to a protected server is established. It allows the caller to control which parts of his security context are made available to the server and whether or not the server may impersonate the caller.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the file is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The actual action taken by the system is written to the *Information* field of this variable.

AllocationSize - Optionally specifies the initial allocation size of the file in bytes. The size has no effect unless the file is created, overwritten, or superseded.

FileAttributes - Specifies the file attributes for the file. Any combination of flags is acceptable except that all other flags override the normal file attribute, *FILE_ATTRIBUTE_NORMAL*. File attributes are only applied to the file if it is created, superseded, or, in some cases, overwritten. See the description in the text below for more details.

FileAttributes Flags

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 - A system file should be created.

FILE_ATTRIBUTE_ARCHIVE - The file should be marked so that it will be archived.

FILE_ATTRIBUTE_TEMPORARY - A temporary should be created.

FILE_ATTRIBUTE_COMPRESSED - A compressed file should be created.

FILE_ATTRIBUTE_OFFLINE - An off-line file should be created.

ShareAccess - Specifies the type of share access that the caller would like to the file.

ShareAccess Flags

FILE_SHARE_READ - Other open operations may be performed on the file for read access.

FILE_SHARE_WRITE - Other open operations may be performed on the file for write access.

FILE_SHARE_DELETE - Other open operations may be performed on the file for delete access.

CreateDisposition - Specifies the actions to be taken if the file does or does not already exist.

CreateDisposition Values

FILE_SUPERSEDE - Indicates that if the file already exists then it should be superseded by the specified file. If it does not already exist then it should be created.

FILE_CREATE - Indicates that if the file already exists then the operation should fail. If the file does not already exist then it should be created.

FILE_OPEN - Indicates that if the file already exists it should be opened rather than creating a new file. If the file does not already exist then the operation should fail.

FILE_OPEN_IF - Indicates that if the file already exists, it should be opened. If the file does not already exist then it should be created.

FILE_OVERWRITE - Indicates that if the file already exists it should be opened and overwritten. If the file does not already exist then the operation should fail.

FILE_OVERWRITE_IF - Indicates that if the file already exists it should be opened and overwritten. If the file does not already exist then it should be created.

CreateOptions - Specifies the options that should be used when creating or opening the file.

CreateOptions Flags

FILE_DIRECTORY_FILE - Indicates that the file being created or opened is a directory file. The *CreateDisposition* parameter must be set to one of *FILE_CREATE, FILE_OPEN,* or *FILE_OPEN_IF*.

FILE_NON_DIRECTORY_FILE - Indicate that the file being opened may not be a directory file.

FILE_WRITE_THROUGH - Indicates that services that write data to the file must actually write the data to the file before the operation is considered to be complete.

FILE_SEQUENTIAL_ONLY - Indicates that the file will only be accessed sequentially.

FILE_RANDOM_ACCESS - Indicates that the file will be accessed randomly so no sequential read ahead operations should be performed on the file.

FILE_NO_INTERMEDIATE_BUFFERING - Indicates that no caching or intermediate buffering is performed for the file.

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the file are performed synchronously. Any wait being performed on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the file are performed synchronously. Waits in the system to synchronize I/O queueing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context.

FILE_CREATE_TREE_CONNECTION - Indicates that a tree connection is to be created.

FILE_COMPLETE_IF_OPLOCKED - Indicates that the operation should complete immediately with an alternate success code if the target file is oplocked rather than blocking the caller's thread.

FILE_NO_EA_KNOWLEDGE - Indicates the if the EAs on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, then the file open should fail because the caller does not understand how to deal with EAs.

FILE_DELETE_ON_CLOSE - Indicates that the file should be deleted when the last handle to it is closed.

FILE_OPEN_BY_FILE_ID - Indicates that the file name contains the name of the device, and a 64-bit ID that is to be used to open the file.

FILE_OPEN_FOR_BACKUP_INTENT - Indicates that the file is being opened for backup intent, hence, the system should check for **SeBackupPrivilege** or **SeRestorePrivilege** and grant the caller the appropriate accesses to the file before checking the *DesiredAccess* against the file's security descriptor.

FILE_TRANSACTED_MODE - Indicates that the file is to be opened in transacted mode. This specifies that no changes to the file should be visible to other openers of the file until the transaction is committed.

FILE_RESERVE_OPFILTER - Indicates that a filter oplock should be reserved on the file if possible. The first I/O operation on the file must be an oplock request so that the caller can determine whether or not the oplock was granted.

FILE_OPEN_OFFLINE_FILE - Indicates that if the target file has been moved from primary storage and the target file is an off-line file, then the marker itself is to be opened rather than retrieving the actual file.

FILE_STORAGE_TYPE_SPECIFIED - Indicates that this *CreateOptions* parameter specifies a storage type field.

FILE_STORAGE_TYPE_DEFAULT - Create/open a file of default storage type.

FILE_STORAGE_TYPE_DIRECTORY - Create/open an enumerable directory file.

FILE_STORAGE_TYPE_FILE - Create/open normal data file.

FILE_STORAGE_TYPE_DOCFILE - Create/open a document file.

FILE_STORAGE_TYPE_JUNCTION_POINT - Create/open a junction point.

FILE_STORAGE_TYPE_CATALOG - Create/open a summary catalogue.

FILE_STORAGE_TYPE_STRUCTURED_STORAGE - Create/open structured storage.

FILE_STORAGE_TYPE_EMBEDDING - Create/open an embedding.

FILE_STORAGE_TYPE_STREAM - Create/open an alternate data stream on a file.

EaBuffer - Optionally specifies a list of EAs that should be set on the file if it is created. This is done as an atomic operation. That is, if an error occurs setting the EAs on the file, then the file will not be created.

EaLength - Supplies the length of the *EaBuffer*. If no buffer is supplied then this value should be zero.

The I/O status block specified by the *IoStatusBlock* parameter has the following type definition:

```
typedef struct _IO_STATUS_BLOCK {
        NTSTATUS Status;
        ULONG Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

Field	Description
Status	Final status of the operation
Information	Additional information about the operation

The **NtCreateFile** service either causes a new file (or directory) to be created, or it opens an existing file or device. The action taken is dependent on the name of the object being opened, whether the object already existed, and the specified create disposition value. A file handle is returned that can be used by subsequent service calls to manipulate the file itself or the data within the file.

There are two basic ways to specify the name of the file that is to be created/opened:

o - A fully qualified pathname. This method simply supplies the full file specification for the file. This is done using the *ObjectName* field of the *ObjectAttributes* structure. No *RootDirectory* handle may be specified.

o - A relative pathname. This method supplies the name of the file as a relative pathname. The path is relative to the directory file represented by the handle in the *RootDirectory* field of the *ObjectAttributes* structure.

Once the I/O operation is complete, the *Information* field of the I/O status block contains information about the action actually taken by the system. That is, one of *FILE_SUPERSEDED*, *FILE_CREATED*, *FILE_OPENED*, or *FILE_OVERWRITTEN*, is returned in this field.

The *SYNCHRONIZE* desired-access flag must be set in order for the caller to wait on the file handle to synchronize I/O completion. If this desired access is not specified, then I/O completion must be synchronized through the use of an event or an APC routine.

If *FILE_EXECUTE* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller cannot directly read or write any data in the file using the returned file handle. All operations on the file occur through the system pager in response to instruction and data accesses.

If *FILE_APPEND_DATA* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller can only write to the end of the file. Any offset information on writes to the file is ignored. The file will automatically be extended as necessary for these types of write operations.

Specifying the *FILE_WRITE_DATA* desired-access flag for a file also allows writes beyond the end of the file to occur. The file is also automatically extended for these types of writes as well.

Files may be shared among threads within a process, or among a family of processes through inheritance, by simply opening or creating the file. The file handle can then be used to access the same file. Note that the OBJ_INHERIT object attribute flag must be specified in the *ObjectAttributes* parameter in order for sharing to occur between parent and child processes through use of the file handle.

Access to a file may be shared among separate cooperating processes or threads by requesting that the file system open the file for shared access. This is accomplished through the flags in the *ShareAccess* mode parameter. Provided that both file openers have the privilege to access the file in the specified manner, the file can be successfully opened and shared. If the caller does not specify *FILE_SHARE_READ*, *FILE_SHARE_WRITE*, or *FILE_SHARE_DELETE*, then no other open operations may be performed on the file.

In order for the file to be successfully opened, the requested access mode to the file must be compatible with the way in which other opens to the file have been made. That is, the desired access mode to the file must not conflict with the accesses that other openers of the file have disallowed.

The *FILE_SUPERSEDE* disposition value specifies that if the file does not already exist, it is to be created. If the file already exists, then it should be superseded. Superseding a file requires that the accessor have delete access to the existing file. That is, the existing file is effectively deleted and then recreated. This implies that if someone else already has the file open, they have specified that the file may be deleted by another file opener. This is done by specifying a *ShareAccess* parameter with the *FILE_SHARE_DELETE* flag set. This type of disposition is consistent with the Unix style of overwriting files.

The *FILE_OVERWRITE_IF* disposition value is much like the *FILE_SUPERSEDE* disposition value. If the file exists, then it will be overwritten; if it does not already exist then it will be created. Overwriting a file is semantically equivalent to a supersede operation except that it requires write access to the file rather than delete access. That is, the requestor must have write access to the file and if someone else already has the file open, they must have specified that the file may be written by another file opener. This is done by specifying a *ShareAccess* parameter with the *FILE_SHARE_WRITE* flag set. Another difference between an overwrite and a supersede is that the specified file attributes are logically OR'd with those already on the file. That is, the caller may not turn off any flags already set in the attributes but may turn others on. This style of overwriting files is consistent with DOS and OS/2.

The *FILE_OVERWRITE* disposition value performs exactly the same operation as a *FILE_OVERWRITE_IF*, except that if the file does not already exist the operation will fail.

The *FILE_DIRECTORY_FILE* option specifies that the file to be created or opened is a directory file. If this option is specified, then the *CreateDisposition* parameter must be set to one of *FILE_CREATE*, *FILE_OPEN*, or *FILE_OPEN_IF*. Likewise, the only create options that may be specified are *FILE_SYNCHRONOUS_IO_ALERT*, *FILE_SYNCHRONOUS_IO_NONALERT*, *FILE_WRITE_THROUGH*, *FILE_OPEN_FOR_BACKUP_INTENT*, and *FILE_OPEN_BY_FILE_ID*. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the file being opened is not a directory file, then the API will fail.

Conversely, the *FILE_NON_DIRECTORY_FILE* option specifies that the target file being opened may not be a directory file. It must be a data file, device, volume, etc., or the API is to fail.

It is also possible to further control the type of file, directory, structured storage, etc. that one wishes to create or open by providing the *FILE_STORAGE_TYPE_SPECIFIED* flag. This flag indicates that one of the *FILE_STORAGE_TYPE_xxx* values has been supplied. Note that specifying *FILE_DIRECTORY_FILE* is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_DIRECTORY*. Likewise, specifying

FILE_NON_DIRECTORY_FILE is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_FILE.*

The *FILE_NO_INTERMEDIATE_BUFFER* option specifies that the file system should not perform any intermediate buffering on behalf of the caller. This causes several restrictions to be placed on the caller's parameters to various service calls.

o - The byte offset parameter to read and write operations must be an integral number of 512-byte blocks.

o - The length of the read or write operation must be an integral number of 512-byte blocks. Note that specifying a read operation to a buffer whose length is 512 bytes may result in a smaller number of significant bytes being transferred to the buffer because the end of the file was reached, however, the driver may still be able to transfer a whole sector of data directly to the buffer.

o - Buffers must be aligned to that of the device. The device alignment requirement can be determined by querying the file.

o - Files opened for this type of access may not be opened for *FILE_APPEND_DATA* access.

o - The *FILE_WRITE_THROUGH* option is automatically set when intermediate buffering is disabled.

o - Calls to set the file position pointer for files opened in this manner may only specify offsets wto 512-byte sector boundaries.

The *FILE_SYNCHRONOUS_IO_ALERT* and *FILE_SYNCHRONOUS_IO_NONALERT* create options allow the caller to specify that all I/O operations on this file are to be performed synchronously as long as they occur through the file object referred to by the returned handle. The system also maintains the current "file pointer context" for the file when the file is opened/created with either of these options. Likewise, all I/O on the file will be serialized across all threads and processes using the returned handle or an inherited copy of the handle. The *SYNCHRONIZE* desired-access flag must also be specified so that the I/O system can use the file object as a synchronization object. Of course, these two options are mutually exclusive.

These two options also imply that the I/O system maintain an internal current file position pointer. This pointer can be used by the read and write services. It can also be set or read by other APIs described later in this document.

The *FILE_CREATE_TREE_CONNECTION* option specifies that a tree connection to a remote node is to be created. For more information, see the *Windows NT LAN Manager Software* specification.

The *FILE_COMPLETE_IF_OPLOCKED* option specifies that if the target file is currently oplocked by another accessor of the file, that the operation should complete immediately anyway without waiting for the oplock break operation to be completed. The call to **NtCreateFile** completes once the oplock break operation has been started, rather than blocking the caller's thread waiting for the break to complete. An alternate success code is returned to the caller if an oplock break is in progress when the service completes. This flag is mutually exclusive with the *FILE_RESERVE_OPFILTER* flag. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

Setting the *FILE_TRANSACTED_MODE* option indicates that the file system and Transaction Manager should work together to only allow other openers of the file to see changes to the file when they are fully committed. This means that other openers will not normally see any writes to the file unless the data has actually been committed.

The *FILE_RESERVE_OPFILTER* option indicates that the caller would like to reserve a filter oplock on the file, if possible. This flag is mutually exclusive with the *FILE_COMPLETE_IF_OPLOCKED* flag. The first I/O request issued on the file must be an oplock *FSCTL* to determine whether or not the oplock was actually reserved. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

A file is considered to have been moved from primary storage and a marker left in its place if the target file's *FILE_ATTRIBUTE_OFFLINE* attribute bit is set. A normal attempt to open such a file causes the HSM(s) in the system to attempt to retrieve the original file. However, the marker itself can be opened by specifying the *FILE_OPEN_OFFLINE_FILE* option.

If a list of EAs is supplied through specifying an *EaBuffer*, then those EAs are applied to the file as an atomic operation. Note that the EAs are only set on the file if the file is created (this also includes supersede and overwrite operations). If setting the EAs on the file incurs an error, then the file is not created, an appropriate error is returned, and the *Information* field of the *IoStatusBlock* variable is set to the offset into the EA buffer of the EA that caused the error.

The type of the contents of the *EaBuffer* is *FILE_FULL_EA_INFORMATION*. This type has the following definition:

```
typedef struct _FILE_FULL_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR Flags;
        UCHAR EaNameLength;
        USHORT EaValueLength;
        CHAR EaName[];
} FILE_FULL_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Flags	Flags to be associated with the EA
EaNameLength	Length of the EA's name field, excluding null termination character
EaValueLength	Length of the EA's value field
EaName	The name of the EA

The flags currently defined for EAs are:

FILE_NEED_EA- This flag indicates that the caller must understand EAs in order to understand the actual meaning or representation of the file. Files who have an EA with this flag set cannot be seen by callers attempting to access the file with the *FILE_NO_EA_KNOWLEDGE CreateOption* set.

The value field begins after the end of the *EaName* field of the structure, including a single null character. The *EaNameLength* field does not include the null character in the count Each entry in the list must be longword aligned. The *NextEntryOffset* field specifies the number of bytes between the current entry and the next entry in the buffer. If there are no more entries following the current entry, then the value of this field is zero.

For more information, refer to the **NtSetEaFile** system service documented elsewhere in this specification.

3.1.2 Opening Files

A file can be opened using the **NtOpenFile** service:

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

Parameters:

FileHandle - A pointer to a variable that receives the file handle value.

DesiredAccess - Specifies the type of access that the caller requires to the file.

DesiredAccess Flags

SYNCHRONIZE - The file handle may be waited on to synchronize with the completion of the I/O operation.

DELETE - The file may be deleted.

READ_CONTROL - The ACL and ownership information associated with the file may be read.

WRITE_DAC - The Discretionary ACL associated with the file may be written.

WRITE_OWNER - Ownership information associated with the file may be written.

FILE_READ_DATA - Data may be read from the file.

FILE_WRITE_DATA - Data may be written to the file.

FILE_EXECUTE - Data may be faulted into memory from the file via paging I/O.

FILE_APPEND_DATA - Data may only be appended to the file.

FILE_READ_ATTRIBUTES - File attributes flags may be read.

FILE_WRITE_ATTRIBUTES - File attributes flags may be written.

FILE_READ_EA - Extended attributes associated with the file may be read.

FILE_WRITE_EA - Extended attributes associated with the file may be written.

FILE_LIST_DIRECTORY - Files in the directory may be listed.

FILE_TRAVERSE - The directory may be traversed. That is, it may be in the pathname of a file.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES,* and *FILE_READ_EA*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA,* and *FILE_APPEND_DATA*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE, SYNCHRONIZE,* and *FILE_EXECUTE*.

For more information about standard rights accesses, see the *Windows NT Local Security Specification*.

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be opened. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the file is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The actual action taken by the system is written to the *Information* field of

this variable. For a more information on this parameter see the **NtCreateFile** system service description.

ShareAccess - Specifies the type of share access that the caller would like to the file.

ShareAccess Flags

FILE_SHARE_READ - Other open operations may be performed on the file for read access.

FILE_SHARE_WRITE - Other open operations may be performed on the file for write access.

FILE_SHARE_DELETE - Other open operations may be performed on the file for delete access.

OpenOptions - Specifies the options that should be used when opening the file.

OpenOptions Flags

FILE_DIRECTORY_FILE - Indicates that the file being opened must be a directory file.

FILE_NON_DIRECTORY_FILE - Indicate that the file being opened may not be a directory file.

FILE_WRITE_THROUGH - Indicates that services that write data to the file must actually write the data to the file before the operation is considered to be complete.

FILE_SEQUENTIAL_ONLY - Indicates that the file will only be accessed sequentially.

FILE_RANDOM_ACCESS - Indicates that the file will be access randomly so no read ahead operations should ever be performed on the file.

FILE_NO_INTERMEDIATE_BUFFERING - Indicates that no caching or intermediate buffering is performed for the file.

FILE_SYNCHRONOUS_IO_ALERT - Indicates that all operations on the file are performed synchronously. Any wait being performed on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context.

FILE_SYNCHRONOUS_IO_NONALERT - Indicates that all operations on the file are performed synchronously. Waits in the system to synchronize I/O queueing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context.

FILE_COMPLETE_IF_OPLOCKED - Indicates that the operation should complete immediately with an alternate success code if the target file is oplocked rather than blocking the caller's thread.

FILE_NO_EA_KNOWLEDGE - Indicates that if the EAs on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, then the file open should fail because the caller does not understand how to deal with EAs.

FILE_DELETE_ON_CLOSE - Indicates that the file should be deleted when the last handle to it is closed.

FILE_OPEN_BY_FILE_ID - Indicates that the file name contains the name of the device, and a 64-bit ID that is to be used to open the file.

FILE_OPEN_FOR_BACKUP_INTENT - Indicates that the file is being opened for backup intent, hence, the system should check for **SeBackupPrivilege** or **SeRestorePrivilege** and grant the caller the appropriate accesses to the file before checking the *DesiredAccess* against the file's security descriptor.

FILE_TRANSACTED_MODE - Indicates that the file is to be opened in transacted mode. This specifies that no changes to the file should be visible to other openers of the file until the transaction is committed.

FILE_RESERVE_OPFILTER - Indicates that a filter oplock should be reserved on the file if possible. The first I/O operation on the file must be an oplock request so that the caller can determine whether or not the oplock was granted.

FILE_OPEN_OFFLINE_FILE - Indicates that if the target file has been moved from primary storage and the target file is an off-line file, then the marker itself is to be opened rather than retrieving the actual file.

FILE_STORAGE_TYPE_SPECIFIED - Indicates that this *CreateOptions* parameter specifies a storage type field.

FILE_STORAGE_TYPE_DEFAULT - Create/open a file of default storage type.

FILE_STORAGE_TYPE_DIRECTORY - Create/open an enumerable directory file.

FILE_STORAGE_TYPE_FILE - Create/open normal data file.

FILE_STORAGE_TYPE_DOCFILE - Create/open a document file.

FILE_STORAGE_TYPE_JUNCTION_POINT - Create/open a junction point.

FILE_STORAGE_TYPE_CATALOG - Create/open a summary catalogue.

FILE_STORAGE_TYPE_STRUCTURED_STORAGE - Create/open structured storage.

FILE_STORAGE_TYPE_EMBEDDING - Create/open an embedding.

FILE_STORAGE_TYPE_STREAM - Create/open an alternate data stream on a file.

The **NtOpenFile** service opens an existing file or device. A file handle is returned that can be used by subsequent service calls to manipulate the file itself or the data within the file.

There are two basic ways to specify the name of the file that is to be opened:

o - A fully qualified pathname. This method simply supplies the full file specification for the file to be opened. This is done using the *ObjectName* field of the *ObjectAttributes* structure. No *RootDirectory* handle may be specified.

o - A relative pathname. This method supplies the name of the file as a relative pathname. The path is relative to the directory file represented by the handle in the *RootDirectory* field of the *ObjectAttributes* structure.

Once the I/O operation is complete, the *Information* field of the I/O status block contains information about the action taken by the system. That is, the *Information* field will contain *FILE_OPENED.*

The *SYNCHRONIZE* desired-access flag must be set in order for the caller to wait on the file handle to synchronize I/O completion. If this desired access is not specified, then I/O completion must be synchronized through the use of an event or an APC routine.

If *FILE_EXECUTE* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller cannot directly read or write any data in the file using the returned file handle. All operations on the file occur through the system pager in response to instruction and data accesses.

If *FILE_APPEND_DATA* is the only desired-access flag specified other than *SYNCHRONIZE*, then the caller can only write to the end of the file. Any offset information on writes to the file is ignored. The file will automatically be extended as necessary for these types of write operations.

Specifying the *FILE_WRITE_DATA* desired-access flag for a file also allows writes beyond the end of the file to occur. The file is also automatically extended for these types of writes as well.

Files may be shared among threads within a process, or among a family of processes through inheritance, by simply opening or creating the file. The file handle can then be used to access the same file. Note that the OBJ_INHERIT object attribute flag must be specified in the *ObjectAttributes* parameter in order for sharing to occur between parent and child processes through use of the file handle.

Access to a file may be shared among separate cooperating processes or threads by requesting that the file system open the file for shared access. This is accomplished through the flags in the *ShareAccess* mode parameter. Provided that both file openers have the privilege to access the file in the specified manner, the file can be successfully opened and shared. If the caller does not specify *FILE_SHARE_READ, FILE_SHARE_WRITE,* or *FILE_SHARE_DELETE*, then no other open operations may be performed on the file.

In order for the file to be successfully opened, the requested access mode to the file must be compatible with the way in which other opens to the file have been made. That is, the desired access mode to the file must not conflict with the accesses that other openers of the file have disallowed.

The *FILE_DIRECTORY_FILE* flag specifies that the file being opened must be a directory file or the service will fail. Likewise, the *FILE_NON_DIRECTORY_FILE* flag specifies that the service will fail if the file being opened is a directory file.

It is also possible to further control the type of file, directory, structured storage, etc. that one wishes to create or open by providing the *FILE_STORAGE_TYPE_SPECIFIED* flag. This flag indicates that one of the *FILE_STORAGE_TYPE_xxx* values has been supplied. Note that specifying *FILE_DIRECTORY_FILE* is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_DIRECTORY*. Likewise, specifying *FILE_NON_DIRECTORY_FILE* is equivalent to specifying *FILE_STORAGE_TYPE_SPECIFIED* and also specifying *FILE_STORAGE_TYPE_FILE*.

The *FILE_NO_INTERMEDIATE_BUFFER* option specifies that the file system should not perform any intermediate buffering on behalf of the caller. This causes several restrictions to be placed on the caller's parameters to various service calls.

o - The byte offset parameter to read and write operations must be an integral number of 512-byte blocks.

o - The length of the read or write operation must be an integral number of 512-byte blocks. Note that specifying a read operation to a buffer whose length is 512 bytes may result in a smaller number of significant bytes being transferred to the buffer because the end of the file was reached, however, the driver may still be able to transfer a whole sector of data directly to the buffer.

o - Buffers must be aligned to that of the device. The device alignment requirement can be determined by querying the file.

o - Files opened for this type of access may not be opened for *FILE_APPEND_DATA* access.

o - The *FILE_WRITE_THROUGH* option is automatically set when intermediate buffering is disabled.

o - Calls to set the file position pointer for files opened in this manner may only specify offsets to 512-byte sector boundaries.

o - All opens of the file must either enable or disable this feature. That is, no mixed opens are permitted.

The *FILE_SYNCHRONOUS_IO_ALERT* and *FILE_SYNCHRONOUS_IO_NONALERT* open options allow the caller to specify that all I/O operations on this file are to be performed synchronously as long as they occur through the file object referred to by the returned handle. The system also maintains the current "file pointer context" for the file when the file is opened with either of these options. Likewise, all I/O on the file will be serialized across all threads and processes using the returned handle or an inherited copy of the handle. The *SYNCHRONIZE* desired access flag must also be specified so that the I/O system can use the file object as a synchronization object.

These two options also imply that the I/O system maintain an internal current file position pointer. This pointer can be used by the read and write services. It can also be set or read by other APIs described later in this document.

The *FILE_COMPLETE_IF_OPLOCKED* option specifies that if the target file is currently oplocked by another accessor of the file, that the operation should complete immediately anyway without waiting for the oplock break operation to be completed. The call to **NtOpenFile** completes once the oplock break operation has been started, rather than blocking the caller's thread waiting for the break to complete. An alternate success code is returned to the caller if an oplock break is in progress when the service completes. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

Setting the *FILE_TRANSACTED_MODE* option indicates that the file system and Transaction Manager should work together to only allow other openers of the file to see changes to the file when they are fully committed. This means that other openers will not normally see any writes to the file unless the data has actually been committed.

The *FILE_RESERVE_OPFILTER* option indicates that the caller would like to reserve a filter oplock on the file, if possible. The first I/O request issued on the file must be an oplock *FSCTL* to determine whether or not the oplock was actually reserved. For more information on oplocks, see the *Windows NT Opportunistic Locking Design Note*.

A file is considered to have been moved from primary storage and a marker left in its place if the target file's *FILE_ATTRIBUTE_OFFLINE* attribute bit is set. A normal attempt to open such a file causes the HSM(s) in the system to attempt to retrieve the original file. However, the marker itself can be opened by specifying the *FILE_OPEN_OFFLINE_FILE* option.

3.2 File Data Services

This section presents those services that read data from and write data to files. They provide the functionality to perform I/O to files according to the options provided in the open/create services.

The APIs that support reading and writing files are as follows:

NtReadFile - Read data from a file into a specified buffer.
NtWriteFile - Write data to a file from a specified buffer.

3.2.1 Reading Files

Data can be read from a file with the **NtReadFile** service:

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

Parameters:

FileHandle - An open file handle to the file to read.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually read from the file is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer to receive the bytes read from the file.

Length - The length of the specified *Buffer* in bytes. This is the number of bytes that are read from the file unless the end of the file is reached.

ByteOffset - Supplies the starting byte offset within the file where the read begins. An error is returned if an attempt is made to start the read beyond the end of the file.

See the note below about the semantics of this parameter if the I/O system is maintaining the current file pointer position.

Key - Optionally specifies a *Key* that is used to indicate the owner of a byte-range lock. If the value of the *Key* and other conditions are met, then the locked range is read.

The routine specified by the *ApcRoutine* parameter has the following type definition:

typedef
VOID
(*PIO_APC_ROUTINE) (
IN PVOID *ApcContext,*
IN PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

ApcContext - This parameter is the value of *ApcContext* in the call to the I/O system service.

IoStatusBlock - This parameter is the pointer *IoStatusBlock* passed in the call to the I/O system service.

The **NtReadFile** service begins reading from the *ByteOffset* byte within the file into the specified *Buffer*. The read terminates under one of the following conditions:

o - The buffer is full. The number of bytes specified by the *Length* parameter has been read. Therefore, no more data can be placed into the buffer without an overflow.

o - During the read operation the end of the file is reached. There is no more data in the file to be placed into the buffer.

If the file was opened or created without intermediate buffering by the file system, there are several restrictions on the parameters supplied to this service. See the descriptions of the **NtCreateFile** and **NtOpenFile** services for more information.

If *FILE_SYNCHRONOUS_IO_ALERT* or *FILE_SYNCHRONOUS_IO_NONALERT* are specified as options when the file is opened/created, then the I/O system maintains the current file position for the file. The caller may specify that the current file pointer position be used instead of a specific byte offset within the file in one of two ways:

o - Specifying a *ByteOffset* parameter whose value is *FILE_USE_FILE_POINTER_POSITION* rather than an actual byte offset within the file.

o - Not specifying the *ByteOffset* parameter at all.

Either of these methods causes the read to occur from the byte offset within the file according to the value of the current file pointer position. Once the read is complete, the pointer position is updated according to the number of bytes that were read from the file.

If the current file position is being maintained by the I/O system, then the caller may still read directly from a location in the file. This automatically changes the current file position to point to that position, performs the read, and then updates the position according to the number of bytes actually read. This gives the caller an atomic "seek and read" service. This is done by supplying the actual byte offset within the file to be read.

The *Key* parameter can optionally be used to specify a key value that is used to determine whether a locked range of bytes can be read by the caller. That is, locked ranges of bytes have a key associated with them using the **NtLockFile** system service. The *Key* parameter is one of the values that must exactly match the key associated with the lock in order to read the locked range of bytes. More information can be found later in this specification on byte range locking.

The **NtReadFile** service is also flexible enough to be invoked for most read functions directly by an RPC stub routine that is emulating a system service on behalf of an emulation subsystem. The OS/2 **DosRead** and POSIX **read** functions, for example, can both be emulated by directly invoking this service.

This service requires **FILE_READ_DATA** access to the file.

Once the data has been read, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.2.2 Writing Files

Data can be written to a file with the **NtWriteFile** service:

NTSTATUS

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

Parameters:

FileHandle - An open file handle to the file to write.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the file is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer containing the data that should be written to the file.

Length - The number of bytes to write to the file from the specified *Buffer*.

ByteOffset - Supplies the starting byte offset within the file where the write begins.

The notes below describe other valid values that this parameter can express.

Key - Optionally specifies a *Key* that it used to indicate the owner of a byte range lock. If the value of the *Key* and other conditions are met, then the locked range is written.

The **NtWriteFile** service begins writing *Length* bytes from the specified *Buffer* to the byte within the file specified by the *ByteOffset* parameter.

If the write occurs to a file beyond the current end of file mark, then the file is automatically extended and the end of file mark is updated. Any bytes not explicitly written between the old end of file mark and the new end of file mark are defined to be zero.

If the file is opened with only **FILE_APPEND_DATA** access, then the *ByteOffset* parameter is ignored. The data contained in the *Buffer*, for *Length* bytes, is written to the current end of the file.

If the file was opened or created without intermediate buffering by the file system, there are several restrictions on the parameters supplied to this service. See the descriptions of the **NtCreateFile** and **NtOpenFile** services for more information.

If *FILE_SYNCHRONOUS_IO_ALERT* or *FILE_SYNCHRONOUS_IO_NONALERT* are specified when the file is opened or created, then the I/O system maintains the current file position pointer. The caller may specify that the current file pointer position be used instead of a specific byte offset within the file in one of two ways:

o - Specifying a *ByteOffset* parameter whose value is *FILE_USE_FILE_POINTER_POSITION* rather than an actual byte offset within the file.

o - Not specifying the *ByteOffset* parameter at all.

Either of these methods causes the write to occur from the byte offset within the file according to the value of the current file pointer position context. Once the write is complete, the pointer position is updated according to the number of bytes that were written to the file.

If the current file position is being maintained by the I/O system, then the caller may still write directly to a location in the file. This automatically changes the current file position to point to that position, performs the write, and then updates the position according to the number of bytes written. This gives the user an atomic "seek and write" service. This is done by supplying the actual byte offset within the file to be written.

It is also possible to cause the write to take place at the current end of file. This can be done regardless of whether the I/O system is maintaining file position information. Specifying a value of *FILE_WRITE_TO_END_OF_FILE* for the *ByteOffset* parameter causes this to occur.

The *Key* parameter can optionally be used to specify a key value that determines whether a locked range of bytes can be written by the caller. That is, locked ranges of bytes have a key associated with them using the **NtLockFile** system service. The *Key* parameter is one of the values that must exactly match the lock specification associated with the lock in order to be able to write the locked range of bytes. More information can be found later in this specification on byte range locking.

The **NtWriteFile** service is also flexible enough to be invoked for most write functions directly by an RPC stub routine executing on behalf of an operating system emulation subsystem. The OS/2 **DosWrite** and POSIX **write** functions, for example, can both be emulated by directly invoking these services.

This service requires either **FILE_WRITE_DATA** or **FILE_APPEND_DATA** access to the file. Note that having only **FILE_APPEND_DATA** access to the file does not allow the caller to write anywhere in the file except at the current end of file mark, while having **FILE_WRITE_DATA** access to a file does not preclude the caller from writing to or beyond the end of the file.

Once the data has been written, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.3 Directory Manipulation Services

This section presents those services that manipulate directories within the file system.

The APIs that permit directory manipulation are as follows:

NtQueryDirectoryFile - Enumerate files within a directory.
NtNotifyChangeDirectoryFile - Monitor directory for modifications.
NtQueryOleDirectoryFile - Enumerate streams and embeddings in the OLE name space.

3.3.1 Enumerating Files in a Directory

The files within a directory can be enumerated using the **NtQueryDirectoryFile** service:

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 - A file handle to an open directory file.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the specified *Buffer* is stored in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer to receive information about the files in the directory. The contents of this buffer are defined by the *FileInformationClass* parameter below.

Length - The length of the specified buffer in bytes.

FileInformationClass - Specifies the type of information that is returned in the *FileInformation* buffer. The type of information in the buffer is defined by the following type codes.

FileInformationClass Values

FileNamesInformation - Specifies that names of files in the directory are written to the *FileInformation* buffer.

FileDirectoryInformation - Specifies that basic directory information about the files is written to the *FileInformation* buffer.

FileFullDirectoryInformation - Specifies that all of the directory information about the files is written to the *FileInformation* buffer.

FileBothDirectoryInformation - Specifies that all of the directory information about the files is written to the *FileInformation* buffer, including both of the file's names.

FileOleDirectoryInformation - Specifies that OLE directory information about the files is written to the *FileInformation* buffer.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single entry should be returned.

FileName - An optional file name within the specified directory. This parameter may only be specified on the first call to the service. It selects the files in the directory that the query calls return. The specification may contain wildcard characters.

RestartScan - A BOOLEAN value that, if TRUE, indicates that the scan should be restarted from the beginning. This causes the directory operation to restart the scan from the beginning of the directory.

The **NtQueryDirectoryFile** function operates on a directory file specified by the *FileHandle* parameter. The service returns information about files in the specified directory. The *ReturnSingleEntry* parameter specifies that only a single entry should be returned rather than filling the buffer. The actual number of files whose information is returned, is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - The number of files whose information fits into the specified buffer.

o - The number of files that exist in the directory according to the wildcard file specification. This defaults to all of the files in the directory.

File systems supported by **Windows NT** return information about files in directories in either random or alphabetically ascending order. It is possible to receive information about a specific file by specifying the name of the file as the *FileName* parameter without using any wildcard characters.

If information about multiple files is returned, then each entry in the buffer will be aligned on a longword or quadword boundary, depending on the type of information being returned. Each type of information class returned begins with the byte offset required to find the next entry in the buffer. If

this value is zero, then there are no more entries following the current entry. Note that there are no entries in the buffer only if the service completes with an error.

The normal operation of this service is to return all of the files in the directory. A wildcard specification may be supplied the first time the service is called to select a subset of the files in the directory. This is done by supplying a wildcard file specification in the *FileName* parameter the first time the service is invoked once the directory file has been opened. Once a wildcard pattern has been supplied, all subsequent **NtQueryDirectoryFile** calls using the same directory handle operate only on those files which match the pattern. That is, restarting the listing will return the first entry in the directory that matches the pattern.

A wildcard file specification may only be supplied the first time that the service is invoked. If no wildcard specification is supplied, the file system assumes all of the files in the directory are selected. Wildcard file specifications must be consistent with those used in **OS/2 V2.0**.

Likewise, the *FileInformationClass* parameter specified the first time indicates the type of information about the files in the directory that is to be returned. Once an information class is established, it may not be changed in subsequent calls to the service. That is, all subsequent calls must pass the same information class as the first call to the service for a given handle.

The information that is returned in the buffer is defined by the following type codes and structures.

FileNamesInformation Format by File Information Class

FileNamesInformation - Data type is *FILE_NAMES_INFORMATION*.

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

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	Index in the directory of this entry
FileNameLength	Length of the file name in bytes
FileName	Name of the file

The information returned for this information class is returned longword aligned, and the *FileInformation* buffer itself must be longword aligned.

FileDirectoryInformation - Data type is *FILE_DIRECTORY_INFORMATION*.

```
typedef struct _FILE_DIRECTORY_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;
        WCHAR FileName[];
} FILE_DIRECTORY_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The file index of this file in the directory
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of all data streams in the file, in bytes
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
FileName	Name of the file

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileFullDirectoryInformation - Data type is *FILE_FULL_DIR_INFORMATION*.

```
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[];
} FILE_FULL_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The file index of this file in the directory
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of all data streams in the file, in bytes
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
EaSize	Size of the EA's associated with the file
FileName	Name of the file

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileBothDirectoryInformation - Data type is *FILE_BOTH_DIR_INFORMATION*.

```
typedef struct _FILE_BOTH_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;
        CCHAR ShortNameLength;
        WCHAR ShortName[12];
        WCHAR FileName[];
} FILE_BOTH_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The file index of this file in the directory
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of all data streams in the file, in bytes
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
EaSize	Size of the EA's associated with the file
ShortNameLength	Length of the 8.3 name of the file
ShortName	8.3 name of the file
FileName	Name of the file

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileOleDirectoryInformation - Data type is *FILE_OLE_DIR_INFORMATION*.

```
typedef struct _FILE_OLE_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;
        FILE_STORAGE_TYPE StorageType;
        GUID OleClassId;
        ULONG OleStateBits;
        BOOLEAN IsExplorable;
        BOOLEAN HasExplorableChildren;
        BOOLEAN ApplicationIsExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
        WCHAR FileName[];
} FILE_OLE_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The index of this file on the volume
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
EndOfFile	Offset to first free byte in the file
AllocationSize	Total allocation size of file, including children
FileAttributes	Attributes of the file
FileNameLength	Length of the name of the file
StorageType	Storage type of the file
OleClassId	OLE class ID
OleStateBits	OLE state bits
IsExplorable	Indicates whether or not object is explorable
HasExplorableChildren	Indicates whether or not object has explorable children
ApplicationHasExplorableChildren	Application-maintained version of above flag
ContentIndexDisable	Indicates whether content indexing has been disabled
InheritContentIndexDisable	Indicates whether content indexing disable is inheritable
FileName	Name of the entry

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE

FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_LIST_DIRECTORY access to the directory is required in order to obtain the above information about files in the specified directory.

As in OS/2 today, users should not depend on any preconceived ideas about the length of file names in **Windows NT**. Because the system supports multiple file system types and will support more in the future, it is difficult to tell just what form file names may take. However, this service guarantees that for **Windows NT V3.1**, a buffer that is large enough to contain at least one *FILE_BOTH_DIR_INFORMATION* structure and has 256 Unicode characters for a file name will be large enough to receive at least one directory entry of any size.

Likewise, a buffer that is large enough to contain at least one name should be at least 256 Unicode characters for the file name itself, plus the size of the remainder of the structure.

Notice that it is legal for the caller to specify the *RestartScan* parameter on a subsequent call to the **NtQueryDirectoryFile** service to have the service restart from the beginning of the directory listing. This causes the scan of the directory to be restarted from the beginning of the list. Notice also that since the file handle may be shared between separate threads within a process, or in threads across processes when the handle is inherited, not all of the directory entries may necessarily be seen by a single thread. That is, the context being maintained to determine which entry should be returned is common among the threads. Therefore, if one thread obtains a directory entry, then the next thread to ask for an entry will obtain the next entry, not the same entry as the first thread.

Once the directory operation has completed, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.3.2 Enumerating Files in an Ole Directory File

The files within an Ole directory file can be enumerate using the **NtQueryOleDirectoryFile** service:

NTSTATUS
NtQueryOleDirectoryFile(
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 - A file handle to an open container about which information is to be returned.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the specified *Buffer* is stored in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer to receive information about the OLE embeddings and streams in the container. The contents of this buffer are defined by the *FileInformationClass* parameter below.

Length - The length of the specified buffer in bytes.

FileInformationClass - Specifies the type of information that is returned in the *FileInformation* buffer. The type of information in the buffer is defined by the following type codes.

FileInformationClass Values

FileDirectoryInformation - Specifies that basic information about the OLE embeddings and streams is written to the *FileInformation* buffer.

FileOleDirectoryInformation - Specifies that comprehensive OLE information about the OLE embeddings and streams is written to the *FileInformation* buffer.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single entry should be returned.

FileName - An optional name within the specified container. This parameter may only be specified on the first call to the service. It selects the embeddings and streams in the container that the query calls return. The specification may contain wildcard characters.

RestartScan - A BOOLEAN value that, if TRUE, indicates that the scan should be restarted from the beginning. This causes the directory operation to restart the scan from the beginning of the container.

The **NtQueryOleDirectoryFile** function operates on a container specified by the *FileHandle* parameter. The service returns information about OLE embeddings and streams in the specified container. The *ReturnSingleEntry* parameter specifies that only a single entry should be returned rather than filling the buffer. The actual number of files whose information is returned, is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - The number of entries whose information fits into the specified buffer.

o - The number of entries that exist in the container according to the wildcard specification. This defaults to all of the entries in the container.

If information about multiple entries is returned, then each entry in the buffer will be aligned on a longword or quadword boundary, depending on the type of information being returned. Each type of information class returned begins with the byte offset required to find the next entry in the buffer. If this value is zero, then there are no more entries following the current entry. Note that there are no entries in the buffer only if the service completes with an error.

The normal operation of this service is to return all of the entries in the container. A wildcard specification may be supplied the first time the service is called to select a subset of the entries in the container. This is done by supplying a wildcard specification in the *FileName* parameter the first time the service is invoked once the container has been opened. Once a wildcard pattern has been supplied, all subsequent **NtQueryOleDirectoryFile** calls using the same handle operate only on those entries which match the pattern. That is, restarting the listing will return the first entry in the container that matches the pattern.

A wildcard specification may only be supplied the first time that the service is invoked. If no wildcard specification is supplied, the file system assumes all of the entries in the container are selected. Wildcard specifications must be consistent with those used in **OS/2 V2.0**.

Likewise, the *FileInformationClass* parameter specified the first time indicates the type of information about the entries in the container that is to be returned. Once an information class is established, it may not be changed in subsequent calls to the service. That is, all subsequent calls must pass the same information class as the first call to the service for a given handle.

The information that is returned in the buffer is defined by the following type codes and structures.

FileNamesInformation Format by File Information Class

FileDirectoryInformation - Data type is *FILE_DIRECTORY_INFORMATION*.

```
typedef struct _FILE_DIRECTORY_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;
        WCHAR FileName[];
} FILE_DIRECTORY_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The index of this entry in the container
CreationTime	Date/time that the entry was created
LastAccessTime	Date/time that the entry was last accessed
LastWriteTime	Date/time that the entry was last written
ChangeTime	Date/time that the entry was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Total allocated size of the OLE embedding or stream in bytes
FileAttributes	Attributes of the OLE embedding or stream
FileNameLength	Length of the name of the entry
FileName	Name of the entry

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags if the object is an embedding. Otherwise, the file attributes field will be zero.

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

FileOleDirectoryInformation - Data type is *FILE_OLE_DIR_INFORMATION*.

```
typedef struct _FILE_OLE_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;
        FILE_STORAGE_TYPE StorageType;
        GUID OleClassId;
        ULONG OleStateBits;
        BOOLEAN IsExplorable;
        BOOLEAN HasExplorableChildren;
        BOOLEAN ApplicationIsExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
        WCHAR FileName[];
} FILE_OLE_DIR_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
FileIndex	The index of this object on the volume
CreationTime	Date/time that the entry was created
LastAccessTime	Date/time that the entry was last accessed
LastWriteTime	Date/time that the entry was last written
ChangeTime	Date/time that the entry was last changed
EndOfFile	Offset to first free byte in the default data stream, in bytes
AllocationSize	Allocated size of the OLE embedding or stream in bytes
FileAttributes	Attributes of the OLE embedding or stream
FileNameLength	Length of the name of the entry
StorageType	Storage type of the entry
OleClassId	OLE class ID
OleStateBits	OLE state bits
IsExplorable	Indicates whether or not object is explorable
HasExplorableChildren	Indicates whether or not object has explorable children
ApplicationHasExplorableChildren	Application-maintained version of above flag
ContentIndexDisable	Indicates whether content indexing has been disabled
InheritContentIndexDisable	Indicates whether CI disable should be inherited
FileName	Name of the entry

The information returned for this information class is returned quadword aligned, and the *FileInformation* buffer itself must be quadword aligned.

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags if the object is an embedding. Otherwise it will be zero.

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

The **FILE_ATTRIBUTE_NORMAL** flag will never be returned in combination with any other flag.

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_LIST_DIRECTORY access to the container is required in order to obtain the above information about OLE embeddings and streams in the specified container.

In the case of the **NtQueryOleDirectoryFile**, users can depend on the maximum length of a file name being 31 Unicode characters, because that is the maximum length defined by OLE. Therefore, the name of any stream, property set, embedding, etc., is guaranteed to be a maximum of 31 Unicode characters because this API only operates on OLE objects.

Likewise, a buffer that is large enough to contain at least one name should be at least 31 Unicode characters for the file name itself, plus the size of the remainder of the structure.

Notice that it is legal for the caller to specify the *RestartScan* parameter on a subsequent call to the **NtQueryOleDirectoryFile** service to have the service restart from the beginning of the listing. This causes the scan of the container to be restarted from the beginning of the list. Notice also that since the file handle may be shared between separate threads within a process, or in threads across processes when the handle is inherited, not all of the entries may necessarily be seen by a single thread. That is, the context being maintained to determine which entry should be returned is common among the threads. Therefore, if one thread obtains an entry, then the next thread to ask for an entry will obtain the next entry, not the same entry as the first thread.

Once the operation has completed, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.3.3 Monitoring Directory Modifications

Directory modifications can be monitored using the **NtNotifyChangeDirectoryFile** service:

NTSTATUS
NtNotifyChangeDirectoryFile(
IN HANDLE *FileHandle,*
IN HANDLE *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
OUT PVOID *Buffer,*
IN ULONG *Length,*
IN ULONG *CompletionFilter,*
IN BOOLEAN *WatchTree*
);

Parameters:

FileHandle - A handle to an open directory file.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A variable to receive the name(s) of the file(s) that changed in the specified target directory.

Length - Specifies the length of the *Buffer*.

CompletionFilter - Specifies a set of flags that indicate the types of operations on the directory or files in the directory that cause the I/O request to complete. The following are the valid flags for this parameter:

CompletionFilter Flags

FILE_NOTIFY_CHANGE_FILE_NAME - Specifies that the I/O operation should be completed if a file is added, deleted, or renamed.

FILE_NOTIFY_CHANGE_DIR_NAME - Specifies that the I/O operation should be completed if a subdirectory is added, deleted, or renamed.

FILE_NOTIFY_CHANGE_NAME - Specifies that the I/O operation should be completed if a file or a subdirectory is added, deleted, or renamed.

FILE_NOTIFY_CHANGE_ATTRIBUTES - Specifies that the I/O operation should be completed if the attributes of a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_SIZE - Specifies that the I/O operation should be completed if the allocation size or end of file for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_LAST_WRITE - Specifies that the I/O operation should be completed if the last write date/time for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_LAST_ACCESS - Specifies that the I/O operation should be completed if the last access date/time for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_CREATION - Specifies that the I/O operation should be completed if the creation date/time for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_EA - Specifies that the I/O operation should be completed if the EAs for a file or subdirectory are changed.

FILE_NOTIFY_CHANGE_SECURITY - Specifies that the I/O operation should be completed if the security information for a file or subdirectory is changed.

FILE_NOTIFY_CHANGE_STREAM_NAME - Specifies that the I/O operation should be completed if the name of an alternate data stream is changed.

FILE_NOTIFY_CHANGE_STREAM_SIZE - Specifies that the I/O operation should be completed if the size of an alternated data stream is changed.

FILE_NOTIFY_CHANGE_STREAM_WRITE - Specifies that the I/O operation should be completed if an alternate data stream is changed due to a write operation.

WatchTree - A BOOLEAN value that, if TRUE, specifies that all changes to files below the directory should also be reported.

The **NtNotifyChangeDirectoryFile** service notifies the caller when files in the directory or directory tree specified by the *FileHandle* are modified. It also returns the name(s) of the file(s) that changed. All names are specified relative to the directory that the handle represents. The service completes once the directory or directory tree has been modified based on the supplied *CompletionFilter*. The service is a "single shot" and therefore needs to be reinvoked to watch the directory for changes again.

The operation of this service begins by opening a directory for **FILE_LIST_DIRECTORY** access. Once the handle is returned, the **NtNotifyChangeDirectoryFile** service may be invoked to begin watching files and subdirectories in the specified target for changes. The first time the service is invoked, the *Length* parameter supplies the size not only of the user's *Buffer*, but also the buffer that will be used by the file system to store names of files that have changed. Likewise, the *CompletionFile*

and *WatchTree* parameters on the first call indicate how notification should operate for all calls using the supplied *FileHandle*. These two parameters are ignored on subsequent calls to the API.

Once a modification is made that should be reported, the system will complete the service. The names of the files that have changed since the last time the service was called will be placed into the caller's output buffer. The *Information* field of the I/O status block indicates the number of bytes that were written to the output buffer. If too many files have changed since the last time the service was called, then zero bytes will be written to the buffer and an alternate status code is returned in the *Status* field of the I/O status block. For the latter case, the application must enumerate the files in the directory or directory tree to note changes.

The format of the data written to the output *Buffer* is defined by the following structure:

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

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Action	Description of what happened to cause this entry
FileNameLength	Length of the file name that changed
FileName	Name of the file that changed

The value of the *Action* field is defined as one of the following:

Value	Description
FILE_ADDED	The file was added to the directory
FILE_REMOVED	The file was removed from the directory
FILE_MODIFIED	The file was modified
FILE_RENAMED_OLD_NAME	The name of the file that was renamed
FILE_RENAMED_NEW_NAME	The new name of the file that was renamed

When a file is renamed within a single directory, then two entries will be placed into the output buffer: the old name of the file and the new name of the file. If the file is renamed from the directory being monitored to another directory, then only a single entry will be placed into the output buffer with an action type of *Removed*.

This service requires **FILE_LIST_DIRECTORY** access to the directory file that was actually modified. If the operation is watching a directory tree, then the caller must have **FILE_TRAVERSE** access to all intervening directories from the grandparent of the modified file, to the directory specified by the *FileHandle* parameter. It is possible to bypass security checks to all directories if the caller has the **SeNotifyChangePrivilege** privilege.

It should be noted that because of the use of both symbolic and hard links within some file systems, the results of changes to directories within a tree may be unpredictable. That is, some changes may only be seen because the *FileHandle* used refers to a point in the tree through which the change was actually made. Changes made to a point lower in the tree may not be seen because the path used to make the change did not traverse the directory referred to by the *FileHandle*.

It should also be noted that this API may not be implemented by some older network servers. In this case, the API will return a status indicating that it is not implemented. Applications using this API should be prepared to enumerate directories or directory trees in this case.

Once a modification is made to the directory or directory tree, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.4 File Services

This section presents those services that control files and obtain and change information about files.

The APIs that perform these functions are as follows:

NtQueryInformationFile - Obtain information about a file.
NtSetInformationFile - Change information on a file.
NtQueryEaFile - Obtain extended attributes for a file.
NtSetEaFile - Set extended attributes for a file.
NtLockFile - Lock a byte range within a file.
NtUnlockFile - Unlock a byte range within a file.

3.4.1 Obtaining Information about a File

Information about a file may be obtained using the **NtQueryInformationFile** service:

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

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The number of bytes actually written to the specified *Buffer* is stored in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer to receive the desired information about the file. The contents of this buffer are defined by the *FileInformationClass* parameter described below.

Length - The length of the *FileInformation* buffer in bytes.

FileInformationClass - Specifies the type of information that should be returned about the file. The information returned in the *FileInformation* buffer is defined by the following type codes:

FileInformationClass Values

FileBasicInformation - Returns basic information about the specified file. **FILE_READ_ATTRIBUTES** access to the file is required. Also see the **NtQueryAttributesFile** service description.

FileStandardInformation - Returns standard information about the specified file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileInternalInformation - Returns file system internal information about the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileEaInformation - Returns the size of the extended attributes structures associated with the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileAccessInformation - Returns the access that the caller has to the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileNameInformation - Returns the volume-relative name of the file. No specific access to the file is required; that is, this information is available as long as the file is open.

FilePositionInformation - Returns the current file position for the file. **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file is required.

FileModeInformation - Returns information about how the file is open for the specified file handle. No specific access to the file is required; that is, this information is available as long as the file is open.

FileAlignmentInformation - Returns information about the alignment requirements for buffers being read or written to the file. This is useful when the file has been opened without intermediate buffering enabled. No specific access to the file is required; that is, this information is available as long as the file is open.

FileAllInformation - Returns all of the above information in one structure. **FILE_READ_ATTRIBUTES** access to the file is required to obtain this information. In order for the file position information to be returned, the accessor must have either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

FileAlternateNameInformation - Returns the DOS format 8.3 alternate name for the file, if it has one.

FileStreamInformation - Returns the names of the alternate data streams for the file, if any exist.

FileCompressionInformation - Returns the compression information about a file. No specific access to the file is required; that is, this information is available as long as the file is open.

FileOleInformation - Returns the OLE-specific information about a file. **FILE_READ_ATTRIBUTES** access to the file is required to obtain this information.

FileOleAllInformation - Returns the all of the OLE-specific information about a file. **FILE_READ_ATTRIBUTES** access to the file is required to obtain this information. In order for the file position information to be returned, the accessor must have either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

The **NtQueryInformationFile** service returns information about the specified file. The information returned in the buffer is defined by the following type codes and structures. Note that the fields that are not supported for a given device or file system are returned as zero. For example, the FAT file system does not support a creation time, so this field is set to zero.

FileInformation Format by File Information Class

FileBasicInformation - Data type is *FILE_BASIC_INFORMATION*.

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

Field	Description
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
FileAttributes	Attributes of the file

All dates and times are returned in the standard **Windows NT** system-time format.

The file attributes field can be a combination of the following flags:

FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_DIRECTORY

FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

Note that the **FILE_ATTRIBUTE_NORMAL** attribute will never be returned in combination with any other attributes, as all other attributes override this attribute. Also see the **NtQueryAttributesFile** service description.

FileStandardInformation - Data type is *FILE_STANDARD_INFORMATION*.

```
typedef struct _FILE_STANDARD_INFORMATION {
        LARGE_INTEGER AllocationSize;
        LARGE_INTEGER EndOfFile;
        DEVICE_TYPE DeviceType;
        ULONG NumberOfLinks;
        BOOLEAN DeletePending;
        BOOLEAN Directory;
} FILE_STANDARD_INFORMATION;
```

Field	Description
AllocationSize	Allocated size of the file in bytes
EndOfFile	Offset to the first free byte in the file
DeviceType	Device type code
NumberOfLinks	Number of hard links to the file
DeletePending	Indicates whether the file is marked for deletion
Directory	Indicates whether the file is a directory

The end of file field specifies the byte offset to the end of the file. Note that because this value is zero-based, it actually refers to the first free byte in the file; that is, it is the offset to the next byte after the last valid byte in the file.

Device types have the following valid values:

FILE_DEVICE_BATTERY
FILE_DEVICE_BEEP
FILE_DEVICE_BUS_EXTENDER
FILE_DEVICE_CD_ROM
FILE_DEVICE_CD_ROM_FILE_SYSTEM
FILE_DEVICE_CONTROLLER
FILE_DEVICE_DATALINK
FILE_DEVICE_DFS
FILE_DEVICE_DISK
FILE_DEVICE_DISK_FILE_SYSTEM
FILE_DEVICE_FILE_SYSTEM
FILE_DEVICE_INPORT_PORT
FILE_DEVICE_KEYBOARD
FILE_DEVICE_MAILSLOT
FILE_DEVICE_MIDI_IN
FILE_DEVICE_MIDI_OUT
FILE_DEVICE_MOUSE
FILE_DEVICE_MULTI_UNC_PROVIDER

FILE_DEVICE_NAMED_PIPE
FILE_DEVICE_NETWORK
FILE_DEVICE_NETWORK_BROWSER
FILE_DEVICE_NETWORK_FILE_SYSTEM
FILE_DEVICE_NETWORK_REDIRECTOR
FILE_DEVICE_NULL
FILE_DEVICE_PARALLEL_PORT
FILE_DEVICE_PHYSICAL_NETCARD
FILE_DEVICE_PRINTER
FILE_DEVICE_SCANNER
FILE_DEVICE_SCREEN
FILE_DEVICE_SERIAL_MOUSE_PORT
FILE_DEVICE_SERIAL_PORT
FILE_DEVICE_SOUND
FILE_DEVICE_STREAMS
FILE_DEVICE_TAPE
FILE_DEVICE_TAPE_FILE_SYSTEM
FILE_DEVICE_TRANSPORT
FILE_DEVICE_UNKNOWN
FILE_DEVICE_VIDEO
FILE_DEVICE_VIRTUAL_DISK
FILE_DEVICE_WAVE_IN
FILE_DEVICE_WAVE_OUT
FILE_DEVICE_8042_PORT

No specific access is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileInternalInformation - Data type is *FILE_INTERNAL_INFORMATION*.

```
typedef struct _FILE_INTERNAL_INFORMATION {
        LARGE_INTEGER IndexNumber;
} FILE_INTERNAL_INFORMATION;
```

Field	Description
IndexNumber	A file system unique file identifier

No specific access to the file is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileEaInformation - Data type is *FILE_EA_INFORMATION*.

```
typedef struct _FILE_EA_INFORMATION {
        ULONG EaSize;
} FILE_EA_INFORMATION;
```

Field	Description
EaSize	Size of file's extended attributes in bytes

No specific access to the file is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileAccessInformation - Data type is *FILE_ACCESS_INFORMATION*.

```
typedef struct _FILE_ACCESS_INFORMATION {
        ACCESS_MASK AccessFlags;
} FILE_ACCESS_INFORMATION;
```

Field	Description
AccessFlags	Access that the caller has to the file

The valid flags that may be set in the **AccessFlags** field are as follows:

SYNCHRONIZE
DELETE
READ_CONTROL
WRITE_DAC
WRITE_OWNER
FILE_READ_EA
FILE_WRITE_EA
FILE_READ_ATTRIBUTES
FILE_WRITE_ATTRIBUTES
FILE_READ_DATA
FILE_WRITE_DATA
FILE_EXECUTE
FILE_APPEND_DATA

If the file is a directory, then the **FILE_READ_DATA** through **FILE_APPEND_DATA** flags are invalid. They are replaced by the following valid values:

FILE_LIST_DIRECTORY
FILE_TRAVERSE

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FileNameInformation - Data type is *FILE_NAME_INFORMATION*.

```
typedef struct _FILE_NAME_INFORMATION {
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_NAME_INFORMATION;
```

Field	Description
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FilePositionInformation - Data type is *FILE_POSITION_INFORMATION*.

```
typedef struct _FILE_POSITION_INFORMATION {
        LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION;
```

Field	Description
CurrentByteOffset	Current byte offset within the file

In order for the information to be valid, the file must have been opened or created specifying synchronous I/O.

FILE_READ_DATA or **FILE_WRITE_DATA** access to the file is required to obtain this information about the file.

FileModeInformation - Data type is *FILE_MODE_INFORMATION*.

```
typedef struct _FILE_MODE_INFORMATION {
        ULONG Mode;
} FILE_MODE_INFORMATION;
```

Field	Description
Mode	Current open mode of file handle to the file

The mode flags that may be returned are as follows:

FILE_WRITE_THROUGH
FILE_SEQUENTIAL_ONLY
FILE_NO_INTERMEDIATE_BUFFERING
FILE_SYNCHRONOUS_IO_ALERT
FILE_SYNCHRONOUS_IO_NONALERT
FILE_DELETE_ON_CLOSE

Note that only one of the synchronous I/O flags will be returned.

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FileAlignmentInformation - Data type is *FILE_ALIGNMENT_INFORMATION*.

```
typedef struct _FILE_ALIGNMENT_INFORMATION {
        ULONG AlignmentRequirement;
} FILE_ALIGNMENT_INFORMATION;
```

Field	Description
AlignmentRequirement	Buffer alignment required by device

The value of this field is one of the following:

FILE_BYTE_ALIGNMENT
FILE_WORD_ALIGNMENT

FILE_LONG_ALIGNMENT
FILE_QUAD_ALIGNMENT
FILE_OCTA_ALIGNMENT
FILE_32_BYTE_ALIGNMENT
FILE_64_BYTE_ALIGNMENT
FILE_128_BYTE_ALIGNMENT
FILE_256_BYTE_ALIGNMENT
FILE_512_BYTE_ALIGNMENT

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open.

FileAllInformation - Data type is *FILE_ALL_INFORMATION*.

```
typedef struct _FILE_ALL_INFORMATION {
        FILE_BASIC_INFORMATION BasicInformation;
        FILE_STANDARD_INFORMATION StandardInformation;
        FILE_INTERNAL_INFORMATION InternalInformation;
        FILE_EA_INFORMATION EaInformation;
        FILE_ACCESS_INFORMATION AccessInformation;
        FILE_POSITION_INFORMATION PositionInformation;
        FILE_MODE_INFORMATION ModeInformation;
        FILE_ALIGNMENT_INFORMATION AlignmentInformation;
        FILE_NAME_INFORMATION NameInformation;
} FILE_ALL_INFORMATION;
```

Field	Description
BasicInformation	Basic information
StandardInformation	Standard information
InternalInformation	Internal information
EaInformation	Extended attributes size information
AccessInformation	Access information
PositionInformation	Current position information
ModeInformation	Mode information
AlignmentInformation	Alignment requirement information
NameInformation	File name information

Notice that the position information will be valid only if the file was opened or created using one of the synchronous I/O options.

FILE_READ_ATTRIBUTES access to the file is required to obtain this information. If the file was opened for synchronous I/O, then the position information will only be valid if the accessor has either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

FileAlternateNameInformation - Data type is *FILE_NAME_INFORMATION*.

```
typedef struct _FILE_NAME_INFORMATION {
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_NAME_INFORMATION;
```

Field	Description
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open. Note that some files do not have alternate names.

FileStreamInformation - Data type is *FILE_STREAM_INFORMATION*.

```
typedef struct _FILE_STREAM_INFORMATION {
        ULONG NextEntryOffset;
        ULONG StreamNameLength;
        LARGE_INTEGER StreamSize;
        LARGE_INTEGER StreamAllocationSize;
        WCHAR StreamName;
} FILE_STREAM_INFORMATION;
```

Field	Description
NextEntryOffset	Offset to the next entry in bytes
StreamNameLength	Length of the name of the stream in bytes
StreamSize	Size of the stream
StreamAllocationSize	Allocation size of the stream
StreamName	Name of the stream

No specific access to the file is required to obtain this information about the file; that is, this information is obtainable as long as the file is open.

FileCompressionInformation - Data type is *FILE_COMPRESSION_INFORMATION*.

```
typedef struct _FILE_COMPRESSION_INFORMATION {
        LARGE_INTEGER CompressedFileSize;
        USHORT CompressionFormat;
} FILE_COMPRESSION_INFORMATION;
```

Field	Description
CompressedFileSize	Size of the compressed file in bytes
CompressionFormat	Compression algorithm code

No specific access to the file is required to obtain this information about the file; that is, this information is available as long as the file is open. Note that if the file is not compressed, then the *CompressionFormat* field is set to zero.

FileOleInformation - Data type is *FILE_OLE_INFORMATION*.

```
typedef struct _FILE_OLE_INFORMATION {
        FILE_OLE_CLASSID_INFORMATION OleClassIdInformation;
        FILE_OBJECTID_INFORMATION ObjectIdInformation;
        FILE_STORAGE_TYPE StorageType;
        ULONG OleStateBits;
        BOOLEAN ApplicationIsExplorable;
```

```
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
} FILE_OLE_INFORMATION;
```

Field	Description
OleClassIdInformation	OLE class ID for the file
ObjectIdInformation	Object ID for the file
OleStateBits	OLE state bits for file
ApplicationIsExplorable	Application-defined notion of explorability
ApplicationHasExplorableChildren	Application-defined notion of children's explorability
ContentIndexDisable	Enable/disable content indexing
InheritContentIndexDisable	Enable/disable content indexing of children

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_READ_ATTRIBUTES access to the file is required to obtain this information.

FileOleAllInformation - Data type is *FILE_OLE_ALL_INFORMATION*.

```
typedef struct _FILE_OLE_ALL_INFORMATION {
        FILE_BASIC_INFORMATION BasicInformation;
        FILE_STANDARD_INFORMATION StandardInformation;
        FILE_INTERNAL_INFORMATION InternalInformation;
        FILE_EA_INFORMATION EaInformation;
        FILE_ACCESS_INFORMATION AccessInformation;
        FILE_POSITION_INFORMATION PositionInformation;
        FILE_MODE_INFORMATION ModeInformation;
        FILE_ALIGNMENT_INFORMATION AlignmentInformation;
        USN Usn;
        FILE_OLE_CLASSID_INFORMATION OleClassIdInformation;
        FILE_OBJECTID_INFORMATION ObjectIdInformation;
        FILE_STORAGE_TYPE StorageType;
        ULONG OleStateBits;
        ULONG OleId;
        ULONG NumberOfStreamReferences;
        ULONG StreamIndex;
        BOOLEAN IsExplorable;
        BOOLEAN HasExplorableChildren;
```

```
        BOOLEAN ApplicationExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
        FILE_NAME_INFORMATION NameInformation;
} FILE_OLE_ALL_INFORMATION;
```

Field	Description
BasicInformation	Basic information
StandardInformation	Standard information
InternalInformation	Internal information
EaInformation	Extended attributes size information
AccessInformation	Access information
PositionInformation	Current position information
ModeInformation	Mode information
AlignmentInformation	Alignment requirement information
Usn	Update sequence number
OleClassIdInformation	OLE Class ID for the file
ObjectIdInformation	Object ID for the file
StorageType	Storage type of the file
OleStateBits	OLE state flags
OleId	OLE ID for the file
NumberOfStreamReferences	Reference count for the stream
StreamIndex	Volume index for this stream
IsExplorable	Indicates whether the file is explorable
HasExplorableChildren	Indicates whether the file has explorable children
ApplicationExplorable	Application version of explorable
ApplicationHasExplorableChildren	Application version of explorable children
ContentIndexDisable	Indicates whether content indexing is disabled
InheritContextIndexDisable	Indicates whether CI disable state is inherited
NameInformation	File name information

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

Notice that the position information will be valid only if the file was opened or created using one of the synchronous I/O options.

FILE_READ_ATTRIBUTES access to the file is required to obtain this information. If the file was opened for synchronous I/O, then the position information will only be valid if the accessor has either **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file.

Once the information about the file has been returned, the caller can determine how much information was actually returned by examining the *Information* field of the *IoStatusBlock* variable.

3.4.2 Changing Information about a File

The information about a file may be changed using the **NtSetInformationFile** service:

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

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

FileInformation - A pointer to a buffer that contains the information about the file to be changed. The contents of this buffer are defined by the *FileInformationClass* parameter described below.

Length - The length of the *FileInformation* buffer in bytes.

FileInformationClass - Specifies the type of information that is contained in the *FileInformation* buffer. The type of information in the buffer is defined by the following type codes.

FileInformationClass Values

FileBasicInformation - Changes the basic information about the specified file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileRenameInformation - Specifies that the name of the file should be changed to a new name. The caller must be able to remove the directory entry for the file in the current directory and therefore **DELETE** access is required to the file. The caller must also be able to write to the new parent directory. See the notes below for further information.

FileLinkInformation - Specifies that a new link be added for the file. The caller must be able to write to the new directory file. See the notes below for further information.

FileDispositionInformation - Specifies that the file should be marked for delete. Once all of the handles to the file have been closed, if the link count for the file is zero, then the file is deleted. Even if the link count is nonzero, at least the directory entry will be deleted. **DELETE** access to the file is required to perform this operation. Also see the **NtDeleteFile** service description.

FilePositionInformation - Specifies a new byte offset as the current position in the file. **FILE_READ_DATA** or **FILE_WRITE_DATA** access to the file is required to perform this operation. The file must also have been opened or created using one of the synchronous I/O options.

FileModeInformation - Specifies that a new mode for the specified handle be set. See the notes below for further information.

FileAllocationInformation - Truncates or extends the allocated size of the file. **FILE_WRITE_DATA** access to the file is required to perform this operation. Note that truncating the allocation size of the file may affect the end of file mark for the file as well.

FileEndOfFileInformation - Truncates or extends the amount of valid data in the file by moving the current end of file. **FILE_WRITE_DATA** access to the file is required to perform this operation.

FileCopyOnWrite - Links two streams together until such time as one is written. No specific access right is required to set this information on the file; that is, it is possible to change this information about the file as long as the caller has a valid handle.

FileCompletionInformation - Associates an I/O completion object with the specified file object. This allows synchronization of I/O request completions through the use of an I/O completion object.

FileMoveClusterInformation - Moves data from one file to the end of another file. **FILE_WRITE_DATA** access to the file is required to peform this operation

FileOleClassIdInformation - Sets the OLE class ID for the file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileOleStateBitsInformation - Sets the OLE state bits for the file. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileApplicationExplorableInformation - Changes the application view of whether or not the object is explorable. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileApplicationExplorableChildrenInformation - Changes the application view of whether or not the object has explorable children.
FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

FileObjectIdInformation - Changes the object ID for the file.
FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

FileContextIndexInformation - Changes whether or not the file is to be content indexed.
FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

FileInheritContentIndexInformation - Changes whether or not the children of this file are to be content indexed. **FILE_WRITE_ATTRIBUTES** access to the file is required to perform this operation.

FileOleInformation - Change the OLE information about the file.
FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

The **NtSetInformationFile** service changes information about a file. The information in the buffer is defined by the following type and structure. Note that the fields that are not supported for a given device or file system are ignored. For example, the FAT file system does not support a creation time, so this field is ignored on an **NtSetInformationFile** service call.

FileInformation Format by File Information Class

FileBasicInformation - Data type is *FILE_BASIC_INFORMATION*.

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

Field	Description
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
FileAttributes	Attributes of the file

All dates and times are specified in the standard **Windows NT** system time format.

The file attributes field can be a combination of the following values:

FILE_ATTRIBUTE_NORMAL

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

Note that the **FILE_ATTRIBUTE_NORMAL** attribute is overridden by all other file attributes flags.

If a field is set to zero, **NtSetInformationFile** does not change the information about the file for that field.

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

FileRenameInformation - Data type is *FILE_RENAME_INFORMATION*.

```
typedef struct _FILE_RENAME_INFORMATION {
        BOOLEAN ReplaceIfExists;
        HANDLE RootDirectory;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_RENAME_INFORMATION;
```

Field	Description
ReplaceIfExists	Replace target file if it exists; else fail
RootDirectory	Root directory of target file name
FileNameLength	Length of the file name in bytes
FileName	Name of the file

This operation requires **DELETE** access to the current file so that the directory entry may be removed from the current parent directory. The caller must also have the appropriate access to create the new entry in the new parent directory file.

The file name may be specified in one of three different ways. No wildcards may ever be specified.

- A simple file name. For this case, the file is simply renamed within the same directory. That is, the name of the file changes but not its location.

- A fully qualified file name. In this case, the file changes not only its name but its location as well.

- A relative file name. In this case, the *RootDirectory* field contains a handle to the target directory for the rename operation. The file name itself must be a simple file name.

FileDispositionInformation - Data type is *FILE_DISPOSITION_INFORMATION*.

```
typedef struct _FILE_DISPOSITION_INFORMATION {
```

```
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION;
```

Field	Description
DeleteFile	Delete the file on close

DELETE access to the file is required to perform this operation.

It should be noted that if the file is deleted, the only legal subsequent operation on the file through the open file handle is to close the file using the NtClose system service.

Also see the **NtDeleteFile** service description.

FileLinkInformation - Data type is *FILE_NAME_INFORMATION*.

```
typedef struct _FILE_NAME_INFORMATION {
    ULONG FileNameLength;
    WCHAR FileName[];
} FILE_NAME_INFORMATION;
```

Field	Description
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to add a link to the file, the file must simply be open. However, the caller must be able to create the new link in the specified target directory.

The file name must be a fully qualified file specification.

FilePositionInformation - Data type is *FILE_POSITION_INFORMATION*.

```
typedef struct _FILE_POSITION_INFORMATION {
    LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION;
```

Field	Description
CurrentByteOffset	Current byte offset within the file

If the file was opened or created with no intermediate buffering, then the new value of the byte offset must be an integral number of 512 bytes.

FILE_READ_DATA or **FILE_WRITE_DATA** access to the file is required to change this information about the file, and the file must be opened for synchronous I/O.

FileModeInformation - Data type is *FILE_MODE_INFORMATION*.

```
typedef struct _FILE_MODE_INFORMATION {
    ULONG Mode;
} FILE_MODE_INFORMATION;
```

Field	Description

Mode Current open mode of file handle to the file

The mode flags that may be changed are as follows:

FILE_WRITE_THROUGH
FILE_SEQUENTIAL_ONLY
FILE_SYNCHRONOUS_IO_ALERT
FILE_SYNCHRONOUS_IO_NONALERT

Note that it is only possible to switch between the two different types of synchronous I/O. It is not possible to either switch to or from synchronous I/O, nor is it possible to specify both types.

If the file has been opened with intermediate buffering disabled, the **FILE_WRITE_THROUGH** flag cannot be turned off. That is, it is forced on by the I/O system. This flag is ignored on a set operation in this case.

Users should be aware that changing this information about the file also changes the access mode for all handles referring to the same file object. That is, all handles referring to the object that are duplicated or inherited are also affected by this access change.

No specific access to the file is required to change this information about the file; that is, this information is available as long as the file is open.

FileAllocationInformation - Data type is *FILE_ALLOCATION_INFORMATION*.

```
typedef struct _FILE_ALLOCATION_INFORMATION {
        LARGE_INTEGER AllocationSize;
} FILE_ALLOCATION_INFORMATION;
```

Field	Description
AllocationSize	The absolute allocation size of the file in bytes

FILE_WRITE_DATA access to the file is required to perform this operation. Setting the allocation size of the file to some number of bytes less than the current end of file mark causes the current end of file mark to be moved to the end of the allocated size of the file.

FileEndOfFileInformation - Data type is *FILE_END_OF_FILE_INFORMATION*.

```
typedef struct _FILE_END_OF_FILE_INFORMATION {
        LARGE_INTEGER EndOfFile;
} FILE_END_OF_FILE_INFORMATION;
```

Field	Description
EndOfFile	The absolute new end of file position

Extending the file beyond the current end of file causes pad bytes of zeroes to be written to the new intermediate bytes.

FILE_WRITE_DATA access to the file is required to perform this operation.

FileCopyOnWrite - Data type is *FILE_COPY_ON_WRITE_INFORMATION*.

```
typedef struct _FILE_COPY_ON_WRITE_INFORMATION {
        BOOLEAN ReplaceIfExists;
        HANDLE RootDirectory;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_COPY_ON_WRITE_INFORMATION;
```

Field	Description
ReplaceIfExists	Replace the target if it exists, else fail
RootDirectory	Root directory of target file name
FileNameLength	Length of the file name in bytes
FileName	Name of the file

No specific access to the file is required to change this information about the file; that is, it is possible to change this information about the file as long as the caller has a valid handle to the file.

FileCompletionInformation - Data type is *FILE_COMPLETION_INFORMATION*.

```
typedef struct _FILE_COMPLETION_INFORMATION {
        HANDLE Port;
        ULONG Key;
} FILE_COMPLETION_INFORMATION;
```

Field	Description
Port	Handle to the I/O completion object to associate with the file
Key	Caller-defined value to be associated with this completion object

No specific access to the file is required to change this information about the file; that is, it is possible to change this information about the file as long as the caller has a valid handle to the file.

FileMoveClusterInformation - Data type is *FILE_MOVE_CLUSTER_INFORMATION*.

```
typedef struct _FILE_MOVE_CLUSTER_INFORMATION {
        ULONG ClusterCount;
        HANDLE RootDirectory;
        ULONG FileNameLength;
        WCHAR FileName[];
} FILE_MOVE_CLUSTER_INFORMATION;
```

Field	Description
ClusterCount	Count of clusters to be moved
RootDirectory	Root directory of target file name
FileNameLength	Length of the file name in bytes
FileName	File name of the target

FILE_WRITE_DATA access to the file is required to perform this operation. Setting the move cluster information on a file causes moves *ClusterCount* clusters to the end of the specified target file.

FileOleClassIdInformation - Data type is *FILE_OLE_CLASS_ID_INFORMATION*.

```
typedef struct _FILE_OLE_CLASS_ID_INFORMATION {
        GUID ClassId;
} FILE_OLE_CLASS_ID_INFORMATION;
```

Field	Description
ClassId	ID of the code that understands this file's format

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the OLE class ID on a file changes the association of application to the file.

FileOleStateBitsInformation - Data type is *FILE_OLE_STATE_BITS_INFORMATION*.

```
typedef struct _FILE_OLE_STATE_BITS_INFORMATION {
        ULONG StateBits;
        ULONG StateBitsMask;
} FILE_OLE_STATE_BITS_INFORMATION;
```

Field	Description
StateBits	OLE state bit information
StateBitsMask	Mask to be applied to state bits

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the OLE state bits on a file changes the value of the file's state bits. The state bits are treated as opaque data to the file system with the exception of the *FILE_ENABLE_DOCFILE_FORMAT* bit which causes a document file to be treated as a single stream rather than as separately addressible streams.

FileApplicationExplorableInformation - Data type is *BOOLEAN*.

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag indicates that the application believes that the object represented by the file handle is explorable.

FileApplicationExplorableChildrenInformation - Data type is *BOOLEAN*.

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag indicates that the application believes that the object represented by the file handle has explorable children.

FileObjectIdInformation - Data type is *FILE_OBJECT_ID_INFORMATION*.

```
typedef struct _FILE_OBJECT_ID_INFORMATION {
        OBJECTID ObjectId;
} FILE_OBJECT_ID_INFORMATION;
```

Field	Description
ObjectId	Object ID for the file

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the Object ID for a file changes the unique ID for the file on the volume.

FileContextIndexInformation - Data type is *BOOLEAN*.

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag disables content indexing for the file.

FileInheritContentIndexInformation - Data type is *BOOLEAN*.

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation. Setting the BOOLEAN flag disables content indexing for the children of the file.

FileOleInformation - Data type is *FILE_OLE_INFORMATION*.

```
typedef struct _FILE_OLE_INFORMATION {
        FILE_OLE_CLASSID_INFORMATION OleClassIdInformation;
        FILE_OBJECTID_INFORMATION ObjectIdInformation;
        FILE_STORAGE_TYPE StorageType;
        ULONG OleStateBits;
        BOOLEAN ApplicationIsExplorable;
        BOOLEAN ApplicationHasExplorableChildren;
        BOOLEAN ContentIndexDisable;
        BOOLEAN InheritContentIndexDisable;
} FILE_OLE_INFORMATION;
```

Field	Description
OleClassIdInformation	OLE class ID for the file
ObjectIdInformation	Object ID for the file
OleStateBits	OLE state bits for file
ApplicationIsExplorable	Application-defined notion of explorability
ApplicationHasExplorableChildren	Application-defined notion of children's explorability
ContentIndexDisable	Enable/disable content indexing
InheritContentIndexDisable	Enable/disable content indexing of children

The possible values for the storage type field are defined by the *FILE_STORAGE_TYPE* enumerated type:

```
typedef enum _FILE_STORAGE_TYPE {
        StorageTypeDirectory,
        StorageTypeFile,
        StorageTypeDocfile,
        StorageTypeJunctionPoint,
        StorageTypeCatalog,
        StorageTypeStructuredStorage,
        StorageTypeEmbedding,
        StorageTypeStream
} FILE_STORAGE_TYPE;
```

FILE_WRITE_ATTRIBUTES access to the file is required to perform this operation.

3.4.3 Obtaining Extended Attributes for a File

The extended attributes for a file may be obtained using the **NtQueryEaFile** service:

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 - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The length, in bytes, that were written to the *Buffer* is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer to receive extended attributes for the file.

Length - The length of the specified buffer in bytes.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single entry should be returned.

EaList - An optional list of extended attributes whose name/value pair is returned in the *Buffer*. If this parameter is supplied, only those EAs matching the names of the EAs in the list are returned.

EaListLength - Supplies the length of the *EaList*, if one was specified. If no *EaList* was specified, this parameter should be zero.

EaIndex - An optional index to an EA whose name/value pair is to be returned. The buffer is filled beginning with the EA associated with the index value.

RestartScan - A BOOLEAN value that indicates, if TRUE, that the scan should be restarted from the beginning. This causes the query operation to restart the scan from the beginning of the extended attributes list.

The **NtQueryEaFile** function obtains extended attributes for the file represented by the file handle. Only complete extended attribute name/value pairs are returned. No partial attribute, such as only the name, is ever written into the buffer. The actual number of EAs returned is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - The number of EAs that fit into the specified buffer.

o - The number of EAs that exist, or the number of EAs that match the list of EAs supplied by the optional *EaList* parameter.

NtQueryEaFile may be invoked multiple times to fill the buffer with EAs from the file. It is possible that the EAs for the file were modified between calls to get more EAs. Due to the sharing semantics defined by OS/2, with which this API is compatible, it is not possible to guarantee that the EAs were not modified.

If the optional *EaList* parameter is specified, then only the information for those EAs specified in the list is returned. Further, if this parameter is specified, then the *EaIndex* parameter is ignored.

The *EaIndex* parameter may optionally be specified to return EAs on the file beginning with an EA other than the first EA in the list.

If multiple EAs are returned, then the structure for each EA in the buffer will be aligned on a longword boundary. Each EA in the list begins with a *NextEntryOffset* field that specifies the number of bytes from the base of the current entry to the start of the next entry. If there are no more entries following the current entry, then the value of this field is zero.

The information that is returned in the *Buffer* is defined by the following structure:

```
typedef struct _FILE_FULL_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR Flags;
        UCHAR EaNameLength;
        USHORT EaValueLength;
        CHAR EaName[];
} FILE_FULL_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Flags	Flags to be associated with the EA
EaNameLength	Length of the EA's name field
EaValueLength	Length of the EA's value field
EaName	The name of the EA

The flags currently defined for EAs are:

FILE_NEED_EA

The value field begins after the end of the *EaName* field of the structure, including a single null character. The null character is not included in the *EaNameLength* field.

The value of the EA can be located then, by adding the length of the EA name to the address of the *EaName* field, and adding one.

The type of the *EaList* parameter is defined by the following structure:

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

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
EaNameLength	Length of the EA's name field
EaName	The name of the EA to be retrieved

The *NextEntryOffset* field, like its *FILE_FULL_EA_INFORMATION* counterpart, is the offset in bytes from the current entry in the list to the start of the next entry, if there is one. If there are no more entries in the list, then the value of this field is zero.

The *EaList* parameter defines the list of the EAs whose information is to be returned. This selects a proper subset of the EAs and only those EAs are returned.

FILE_READ_EA access to the file is required in order to obtain information about the extended attributes associated with the file.

If an error, such as an invalid character is found in an EA name field, is encountered, then the *Information* field in the I/O status block contains the byte offset from the base of the *Buffer* to the offending EA entry that caused the failure.

Once extended attributes for the file have been written to the *Buffer*, the *Information* field of the *IoStatusBlock* variable can be examined to determine how many bytes of extended attributes information were actually returned.

3.4.4 Changing Extended Attributes for a File

The extended attributes associated with a file may be changed using the **NtSetEaFile** service:

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

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer that contains the extended attributes to be applied to the file.

Length - The length of the specified buffer in bytes.

The **NtSetEaFile** service changes the extended attributes on the file using the EAs specified by the *Buffer* parameter.

The information specified by the *Buffer* parameter is defined by the following structure.

```
typedef struct _FILE_FULL_EA_INFORMATION {
        ULONG NextEntryOffset;
        UCHAR Flags;
        UCHAR EaNameLength;
        USHORT EaValueLength;
        CHAR EaName[];
} FILE_FULL_EA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
Flags	Flags to be associated with the EA
EaNameLength	Length of the EA's name field
EaValueLength	Length of the EA's value field
EaName	The name of the EA

The flags currently defined for EAs are:

FILE_NEED_EA

The value field begins after the end of the *EaName* field of the structure, including a single null character. The null character is not included in the *EaNameLength* field.

If multiple EAs are contained in the buffer, then the structure for each entry is longword aligned. The *NextEntryOffset* field contains the byte offset to the start of the next entry in the buffer. If there are no more entries past the current entry, then this field is zero.

EAs are applied to the file such that if the EA does not exist, then it is added. If the EA does exist, it is replaced. An entry whose *EaValueLength* field is zero indicates that the EA whose name matches the entry is to be deleted from the list of EAs on the file.

If an error occurs changing the EAs on the file, then the *Information* field in the I/O status block contains the byte offset from the base of the *Buffer* to the offending EA entry that caused the failure.

FILE_WRITE_EA access to the file is required in order to change the extended attributes associated with the file.

3.4.5 Locking Byte Ranges in Files

A byte range within a file may be locked using the **NtLockFile** service:

```
NTSTATUS
NtLockFile(
        IN HANDLE FileHandle,
```

IN HANDLE *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN PLARGE_INTEGER *ByteOffset,*
IN PLARGE_INTEGER*Length,*
IN ULONG *Key,*
IN BOOLEAN *FailImmediately,*
IN BOOLEAN *ExclusiveLock*
);

Parameters:

FileHandle - A handle to an open file.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

ByteOffset - Specifies the starting byte offset of the file where the lock should begin.

Length - The length of the byte range to lock, in bytes.

Key - A value to be associated with the lock range for further identification.

FailImmediately - A BOOLEAN value that indicates whether the service will return immediately if the lock cannot be obtained (TRUE), or whether the service will wait indefinitely until the lock is acquired (FALSE).

ExclusiveLock - A BOOLEAN value that indicates the type of lock that is applied to the byte range. If the value is TRUE, then the lock is *exclusive*; otherwise, the lock is *shared*.

The **NtLockFile** service is used to lock the specified byte range for the file. The range locked is for the specified file, and is controlled by the following:

o - the *ByteOffset* of the file

o - the *Length* of the byte range

o - the *Key* value associated with the byte range

o - the invoking process

Locks are not inherited by child processes when they are created. They are owned by the process that acquired the lock. Locks may be manipulated and "owned" by separate threads within a process as thread-specific locks by specifying non-zero values for the *Key* parameter in each thread.

There are two types of locks on files, shared and exclusive. A shared lock allows read-only access by any process attempting to read the locked range, including the owning process. Shared locks may also overlap. Exclusive locks allow read/write access by only the owning process and by access to any other process. Exclusive locks may not overlap either shared locks or other exclusive locks.

Locks owned by a given process are unlocked once all of the handles to the specified file have been closed by that process. The locks are not released in any particular order.

It is not an error to specify a range that either spans or even begins after the end of the file. These types of locks can be used to synchronize access to the end of the file or for appending data to the file.

FILE_READ_DATA or **FILE_WRITE_DATA** access is required to the file to request a lock.

3.4.6 Unlocking Byte Ranges in Files

A byte range within a file may be unlocked using the **NtUnlockFile** service:

NTSTATUS
NtUnlockFile(
    IN HANDLE *FileHandle,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN PLARGE_INTEGER *ByteOffset,*
    IN PLARGE_INTEGER *Length,*
    IN ULONG *Key*
    );

Parameters:

FileHandle - A handle to an open file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

ByteOffset - The byte offset of the file whose corresponding lock is released. This value must exactly match the byte offset of the lock.

Length - The length of the locked byte range that is released. This value must exactly match the length of the lock.

Key - The value associated with the lock range for further identification. This value must exactly match the key of the lock.

The **NtUnlockFile** service is used to unlock the specified byte range for the file. The lock parameters must exactly match those of the acquired lock. If the parameters exactly match those of the locked range, then the lock is released.

Only the process that owns the lock may unlock the byte range.

3.5 File System Services

This section presents those services that obtain information about file systems and control them.

The APIs that perform these functions are as follows:

NtQueryVolumeInformationFile - Obtain information about a file system volume.
NtSetVolumeInformationFile - Change information about a file system volume.
NtQueryQuotaInformationFile Obtain quota information about a file system volume.
NtSetQuotaInformationFile - Change quota information about a file system volume.
NtFsControlFile - General file system control interface.

3.5.1 Obtaining Information about a File System Volume

Information about a file system volume may be obtained using the **NtQueryVolumeInformationFile** service:

NTSTATUS
NtQueryVolumeInformationFile(
IN HANDLE *FileHandle,*
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
OUT PVOID *FsInformation,*
IN ULONG *Length,*
IN FS_INFORMATION_CLASS *FsInformationClass*
);

Parameters:

FileHandle - A handle to an open file, device, directory, or volume for which volume information is returned.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The length, in bytes, of the data written to the *FsInformation* buffer is returned in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FsInformation - A pointer to a buffer to receive information about the specified volume. The contents of this buffer are defined by the *FsInformationClass* parameter described below.

Length - The length of the *FsInformation* buffer in bytes.

FsInformationClass - Specifies the type of information that should be returned about the volume. The information in the *FsInformation* buffer is defined by the following type codes.

FsInformationClass Values

FileFsVolumeInformation - Returns information about the volume that is currently "mounted" on the specified device. No specific access to the volume is required to obtain this information.

FileFsSizeInformation - Returns information about the size and the free space on the volume. No specific access to the volume is required to obtain this information.

FileFsDeviceInformation - Returns information about the device upon which the volume is actually mounted, or the device to which the handle directly refers. No specific access to the volume is required to obtain this information.

FileFsAttributeInformation - Returns attribute information about the file system responsible for the volume. No specific access to the volume is required to obtain this information.

FileFsControlInformation - Returns file system control information about the volume. No specific access to the volume is require to obtain this information.

The **NtQueryVolumeInformationFile** service returns information about the volume specified by the *FileHandle* parameter. The information returned in the buffer is defined by the following type codes and structures.

FsInformation Format by Fs Information Class

FileFsVolumeInformation - Data type is *FILE_FS_VOLUME_INFORMATION*.

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

Field	Description
VolumeCreationTime	Date/time the volume was created
VolumeSerialNumber	Serial number of the volume
VolumeLabelLength	Length of the name of the volume
SupportsObjects	File system supports object-oriented file system objects
VolumeLabel	Name of the volume

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsSizeInformation - Data type is *FILE_FS_SIZE_INFORMATION*.

```
typedef struct _FILE_FS_SIZE_INFORMATION {
        LARGE_INTEGER TotalAllocationUnits;
        LARGE_INTEGER AvailableAllocationUnits;
        ULONG SectorsPerAllocationUnit;
```

```
        ULONG BytesPerSector;
} FILE_FS_SIZE_INFORMATION;
```

Field	Description
TotalAllocationUnits	Total allocation units on volume
AvailableAllocationUnits	Free allocation units on volume
SectorsPerAllocationUnit	Number of sectors in each allocation unit
BytesPerSector	Number of bytes in each sector

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsDeviceInformation - Data type is *FILE_FS_DEVICE_INFORMATION*.

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

Field	Description
DeviceType	Type of the target device
Characteristics	Characteristcs of the target device

Device types have the following valid values:

FILE_DEVICE_BATTERY
FILE_DEVICE_BEEP
FILE_DEVICE_BUS_EXTENDER
FILE_DEVICE_CD_ROM
FILE_DEVICE_CD_ROM_FILE_SYSTEM
FILE_DEVICE_CONTROLLER
FILE_DEVICE_DATALINK
FILE_DEVICE_DFS
FILE_DEVICE_DISK
FILE_DEVICE_DISK_FILE_SYSTEM
FILE_DEVICE_FILE_SYSTEM
FILE_DEVICE_INPORT_PORT
FILE_DEVICE_KEYBOARD
FILE_DEVICE_MAILSLOT
FILE_DEVICE_MIDI_IN
FILE_DEVICE_MIDI_OUT
FILE_DEVICE_MOUSE
FILE_DEVICE_MULTI_UNC_PROVIDER
FILE_DEVICE_NAMED_PIPE
FILE_DEVICE_NETWORK
FILE_DEVICE_NETWORK_BROWSER
FILE_DEVICE_NETWORK_FILE_SYSTEM
FILE_DEVICE_NETWORK_REDIRECTOR
FILE_DEVICE_NULL
FILE_DEVICE_PARALLEL_PORT

FILE_DEVICE_PHYSICAL_NETCARD
FILE_DEVICE_PRINTER
FILE_DEVICE_SCANNER
FILE_DEVICE_SCREEN
FILE_DEVICE_SERIAL_MOUSE_PORT
FILE_DEVICE_SERIAL_PORT
FILE_DEVICE_SOUND
FILE_DEVICE_STREAMS
FILE_DEVICE_TAPE
FILE_DEVICE_TAPE_FILE_SYSTEM
FILE_DEVICE_TRANSPORT
FILE_DEVICE_UNKNOWN
FILE_DEVICE_VIDEO
FILE_DEVICE_VIRTUAL_DISK
FILE_DEVICE_WAVE_IN
FILE_DEVICE_WAVE_OUT
FILE_DEVICE_8042_PORT

Device characteristics have the following valid flags:

Flag	Meaning
FILE_REMOVABLE_MEDIA	Device supports removable media
FILE_READ_ONLY_DEVICE	Device is a read-only device
FILE_FLOPPY_DISKETTE	Media in device is a floppy diskette
FILE_WRITE_ONCE_MEDIA	Device supports write once media
FILE_REMOTE_DEVICE	Device is a remote device
FILE_DEVICE_IS_MOUNTED	Device is currently mounted
FILE_VIRTUAL_VOLUME	Device volume is virtual

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsAttributeInformation - Data type is *FILE_FS_ATTRIBUTE_INFORMATION*.

```
typedef struct _FILE_FS_ATTRIBUTE_INFORMATION {
        ULONG FileSystemAttributes;
        LONG MaximumComponentNameLength;
        ULONG FileSystemNameLength;
        WCHAR FileSystemName;
} FILE_FS_ATTRIBUTE_INFORMATION;
```

Field	Description
FileSystemAttributes	Attributes of the volume's owning file system
MaximumComponentNameLength	Maximum length of each file name component
FileSystemNameLength	The length of the file system's name
FileSystemName	The name of the file system

File system attributes have the following valid flags:

Flag	Meaning

FILE_CASE_SENSITIVE_SEARCH	Supports case sensitive searches
FILE_CASE_PRESERVED_NAMES	Supports preserving name case on disk
FILE_UNICODE_ON_DISK	Stores UNICODE characters on disk
FILE_PERSISTENT_ACLS	Stores ACLs on disk
FILE_FILE_COMPRESSION	Supports file compression
FILE_VOLUME_IS_COMPRESSED	Handle refers to a compressed volume

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

FileFsControlInformation - Data type is *FILE_FS_CONTROL_INFORMATION {*

```
typedef struct _FILE_FS_CONTROL_INFORMATION {
        LARGE_INTEGER FreeSpaceStartFiltering;
        LARGE_INTEGER FreeSpaceThreshold;
        LARGE_INTEGER FreeSpaceStopFiltering;
        LARGE_INTEGER DefaultQuotaThreshold;
        LARGE_INTEGER DefaultQuotaLimit;
        LARGE_INTEGER DeletionLogSizeLimit;
        ULONG FileSystemControlFlags;
} FILE_FS_CONTROL_INFORMATION;
```

Field	**Description**
FreeSpaceStartFiltering	Amount of space required to begin content indexing
FreeSpaceThreshold	Amount of space remaining to generate popup
FreeSpaceStopFiltering	Amount of space remaining to stop content indexing
DefaultQuotaThreshold	Default quota threshold for volume
DefaultQuotaLimit	Default quota limit for volume
DeletionLogSizeLimit	Size of deletion file log
FileSystemControlFlags	Flags to control this volume

File system control flags consist of the following valid flag values:

Flag	**Meaning**
FILE_VC_QUOTA_NONE	No quota information maintained
FILE_VC_QUOTA_TRACK	Quotas are being tracked on volume
FILE_VC_QUOTA_ENFORCE	Quotas are being enforced on volume
FILE_VC_QUOTAS_INCOMPLETE	Volume quotas are incomplete
FILE_VC_CONTENT_INDEX_DISABLED	Content indexing disabled
FILE_VC_LOG_QUOTA_THRESHOLD	Log quota threshold reached event
FILE_VC_LOG_QUOTA_LIMIT	Log quota limit reached event
FILE_VC_LOG_VOLUME_THRESHOLD	Log volume free space threshold event
FILE_VC_LOG_VOLUME_LIMIT	Log volume free space limit event

No specific access to the volume is required to obtain this information about the volume; that is, this information is available as long as the volume is accessed through an open handle to the volume or device itself, or to a file or directory on the volume.

Once the information about the volume has been returned, the *Information* field of the *IoStatusBlock* variable can be examined to determine the number of bytes of volume information actually written to the *FsInformation* buffer.

3.5.2 Changing Information about a File System Volume

Information about a file system volume may be changed using the **NtSetVolumeInformationFile** service:

NTSTATUS
NtSetVolumeInformationFile(
IN HANDLE *FileHandle,*
OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN PVOID *FsInformation,*
IN ULONG *Length,*
IN FS_INFORMATION_CLASS *FsInformationClass*
);

Parameters:

FileHandle - A handle to an open volume for which information is changed.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

FsInformation - A pointer to a buffer that contains the information about the file system to be changed. The contents of this buffer are defined by the *FsInformationClass* parameter described below.

Length - The length of the *FsInformation* buffer in bytes.

FsInformationClass - Specifies the type of information that should be changed about the file system. The information in the *FsInformation* buffer is defined by the following type codes.

FsInformationClass Values

FileFsLabelInformation - Changes the volume label on the volume that is currently "mounted" on the specified device. **FILE_WRITE_DATA** access to the device or volume is required.

FileFsControlInformation - Changes the file system control information for the volume that is currently “mounted” on the specified device. **FILE_WRITE_DATA** access to the device or volume is required.

The **NtSetVolumeInformationFile** service changes information about the volume "mounted" on the device specified by the *FileHandle* parameter. The information to be changed is in the *FsInformation* buffer. Its contents are defined by the following type codes and structures.

FsInformation Format by Fs Information Class

FileFsLabelInformation - Data type is *FILE_FS_LABEL_INFORMATION*.

```
typedef struct _FILE_FS_LABEL_INFORMATION {
        ULONG VolumeLabelLength;
        WCHAR VolumeLabel[];
} FILE_FS_LABEL_INFORMATION;
```

Field	Description
VolumeLabelLength	Length of the name of the volume
VolumeLabel	Name of the volume

FILE_WRITE_DATA access to the device or volume is required to change this information.

FileFsControlInformation - Data type is *FILE_FS_CONTROL_INFORMATION {*

```
typedef struct _FILE_FS_CONTROL_INFORMATION {
        LARGE_INTEGER FreeSpaceStartFiltering;
        LARGE_INTEGER FreeSpaceThreshold;
        LARGE_INTEGER FreeSpaceStopFiltering;
        LARGE_INTEGER DefaultQuotaThreshold;
        LARGE_INTEGER DefaultQuotaLimit;
        LARGE_INTEGER DeletionLogSizeLimit;
        ULONG FileSystemControlFlags;
} FILE_FS_CONTROL_INFORMATION;
```

Field	Description
FreeSpaceStartFiltering	Amount of space required to begin content indexing
FreeSpaceThreshold	Amount of space remaining to generate popup
FreeSpaceStopFiltering	Amount of space remaining to stop content indexing
DefaultQuotaThreshold	Default quota threshold for volume
DefaultQuotaLimit	Default quota limit for volume
DeletionLogSizeLimit	Size of deletion file log
FileSystemControlFlags	Flags to control this volume

File system control flags consist of the following valid flag values:

Flag	Meaning
FILE_VC_QUOTA_NONE	No quota information maintained
FILE_VC_QUOTA_TRACK	Quotas are being tracked on volume
FILE_VC_QUOTA_ENFORCE	Quotas are being enforced on volume
FILE_VC_QUOTAS_INCOMPLETE	Volume quotas are incomplete
FILE_VC_CONTENT_INDEX_DISABLED	Content indexing disabled
FILE_VC_LOG_QUOTA_THRESHOLD	Log quota threshold reached event
FILE_VC_LOG_QUOTA_LIMIT	Log quota limit reached event
FILE_VC_LOG_VOLUME_THRESHOLD	Log volume free space threshold event
FILE_VC_LOG_VOLUME_LIMIT	Log volume free space limit event

FILE_WRITE_DATA access to the volume is required in order to change the file system control information.

3.5.3 Obtaining Quota Information about a File System Volume

Quota information about a file system volume may be obtained using the **NtQueryQuotaInformationFile** service:

NTSTATUS
NtQueryQuotaInformationFile(
IN HANDLE *FileHandle*,
OUT PIO_STATUS_BLOCK *IoStatusBlock*,
OUT PVOID *Buffer*,
IN ULONG *Length*,
IN BOOLEAN *ReturnSingleEntry*,
IN PVOID *SidList* **OPTIONAL**,
IN ULONG *SidListLength*,
IN PSID *StartSid* **OPTIONAL**,
IN BOOLEAN *RestartScan*
);

Parameters:

FileHandle - An open handle to an open file, directory, device, or volume whose quota information is to be returned.

IoStatusBlock - A variable to receive the final completion status and information about the operation. Service calls that return information, return the length of the data written to the output buffer in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A pointer to a buffer to receive the requested quota information about the specified volume.

Length - Specifies the length of the *Buffer* parameter in bytes.

ReturnSingleEntry - A BOOLEAN value that, if TRUE, indicates that only a single quota entry should be returned.

SidList - An optional list of entries whose quota entries are returned in the *Buffer*. If this parameter is supplied, only those quota entries matching the SIDs in the list are returned.

SidListLength - Supplies the length of the *SidList*, if one was specified. If no *SidList* was specified, this parameter should be zero.

StartSid - An optional SID that specifies a quota entry to be rewound to during a *RestartScan* operation. The first quota entry returned is the entry following the entry specified by the SID.

RestartScan - A BOOLEAN value that, if TRUE, indicates that the scan should be restarted from the beginning, or alternately from the entry following the *StartSid* entry. This causes the query to restart the scan from the beginning or from the entry following the quota entry for the specified SID.

The **NtQueryQuotaInformationFile** function obtains quota entry information for the volume represented by the file handle. Only complete quota entries are returned. The actual number of quota entries returned is the smallest of the following:

o - One entry, if the *ReturnSingleEntry* parameter is TRUE.

o - One entry, if the only entry visible is the entry for the current thread's SID.

o - The number of quota entries that fit into the specified buffer.

o - The number of quota entries that exist, or the number of entries that match the list of entries supplied by the optional *SidList* parameter.

NtQueryQuotaInformationFile may be invoked multiple times to fill the buffer with quota entries for the volume. It is possible that the quota entries for the volume were modified between calls to get more entries, unless the volume is locked.

If the optional *SidList* parameter is specified, then only the quota information for those SIDs specified in the list is returned. Specifying a SID which has no corresponding quota information on the volume causes an entry to be returned with all zeroes for the quota fields. Futher, if this parameter is specified, then the *StartSid* parameter is ignored. Finally, if a *SidList* is specified, the output buffer will be filled with as many matching entries as possible. If they do not fit, then the caller should invoke the service again, changing the start of the list to the point where the last service left off.

For example, if the caller passed in a *SidList* with entries for SIDs A, B, C, D, and E, and the output buffer was only large enough for the file system to return entries for SIDs A, B, and C, then the caller should invoke the service again specifying SIDs D and E. Because the list is self-describing, this can be easily accomplished by simply changing the starting pointer and adjusting the *SidListLength* parameter.

The *StartSid* parameter may optionally be specified to return quota entries for the volume beginning with an entry other than the first quota entry. If a *StartSid* is specified, and the *RestartScan* parameter is specified, then the quota entries returned will be start with the quota entry for the entry after the one selected by the *StartSid* parameter.

If multiple quota entries are returned, then the structure for each entry in the buffer will be aligned on a longword boundary. Each entry in the list begins with a *NextEntryOffset* field that specifies the number of bytes from the base of the current entry to the start of the next entry. If there are no more entries following the current entry, then the value of this field is zero.

The format of the *SidList* information buffer is defined by the following structure:

```
typedef struct _FILE_GET_QUOTA_INFORMATION {
        ULONG NextEntryOffset;
        ULONG SidLength;
        SID Sid;
} FILE_GET_QUOTA_INFORMATION, *PFILE_GET_QUOTA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list

SidLength Length, in bytes, of the SID
Sid SID of entry to be returned

No special access to the volume is required in order to obtain quota information about the volume. The *FileHandle* may refer to either the volume, or a file or directory anywhere on the volume to which the caller has some access.

Once quota entries for the volume have been written to the *Buffer*, the *Information* field of the *IoStatusBlock* variable can be examined to determine how many bytes of quota information were actually returned.

3.5.4 Changing Quota Information about a File System Volume

Quota information about a file system volume may be changed using the **NtSetQuotaInformationFile** service:

NTSTATUS
NtSetQuotaInformationFile(
IN HANDLE *FileHandle*,
OUT PIO_STATUS_BLOCK *IoStatusBlock*,
IN PVOID *Buffer*,
IN ULONG *Length*
);

Parameters:

FileHandle - A handle to a volume whose quota entries are to be changed.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

Buffer - A opinter to a buffer that contains the quota entry information to be applied to the volume.

Length - The length of the specified buffer in bytes.

The **NtSetQuotaInformationFile** service changes the quota information on a volume using the quota entries specified by the *Buffer* parameter.

The information specified by the *Buffer* parameter is defined by the following structure:

```
typedef struct _FILE_QUOTA_INFORMATION {
        ULONG NextEntryOffset;
        ULONG SidLength;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER QuotaUsed;
        LARGE_INTEGER QuotaThreshold;
        LARGE_INTEGER QuotaLimit;
        SID Sid;
} FILE_QUOTA_INFORMATION, *PFILE_QUOTA_INFORMATION;
```

Field	Description
NextEntryOffset	Offset, in bytes, to the next entry in the list
SidLength	Length, in bytes, of the SID
ChangeTime	Time that the quota entry was last changed
QuotaUsed	Amount of disk space used
QuotaThreshold	Amount of disk space useable without incurring an event
QuotaLimit	Amount of disk space permitted to be used
Sid	SID of this quota entry

If multiple quota entries are contained in the buffer, then the structure for each entry is longword aligned. The *NextEntryOffset* field contains the byte offset to the start of the next entry in the buffer. If there are no more entries past the current entry, then this field is zero.

If an error occurs changing the quotas on the volume, then the *Information* field in the I/O status block contains the byte offset from the vase of the *Buffer* to the offending quota entry that caused the failure.

FILE_WRITE_DATA access to the volume is required in order to change the quota information associated with the volume.

3.5.5 Controlling File Systems

Information may be passed between applications and file systems using the **NtFsControlFile** service:

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 - An open file handle to the file or device to whose file system the control information should be given.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine,* if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. Service calls that return information, return the length of the data written to the output buffer in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

FsControlCode - A code that indicates which file system control function is to be executed.

InputBuffer - An optional pointer to a buffer that contains the information to be given to the target file system. This information is file-system-specific.

InputBufferLength - The length of the *InputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

OutputBuffer - An optional pointer to a buffer that is to receive the file-system-dependent return information from the target file system.

OutputBufferLength - The length of the *OutputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

The **NtFsControlFile** service is a file-system-dependent interface that extends the control that applications have over various components within the system. This API provides a consistent view of the input and output data to the system while still providing the application and file system drivers a file-system-dependent method of specifying a communications interface.

The type of access that the caller needs to the file is dependent on the actual operation being performed.

3.6 Miscellaneous Services

This section presents those service that provide miscellaneous functionality for files and devices.

The APIs that perform these functions are as follows:

NtFlushBuffersFile - Flushes all buffered and cached data out to the file.
NtCancelIoFile - Cancels all I/O operations on a file.
NtDeviceIoControlFile - Miscellaneous device control.

3.6.1 Flushing File Buffers

Buffered data may be flushed out to the file using the **NtFlushBuffersFile** service:

NTSTATUS
NtFlushBuffersFile(
IN HANDLE *FileHandle,*
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

FileHandle - An open file handle to a file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

The **NtFlushBuffersFile** service causes all buffered data to be written to the file.

FILE_WRITE_DATA or **FILE_APPEND_DATA** access to the file is required to perform this service.

3.6.2 Canceling Pending I/O on a File

Pending I/O operations on a file may be canceled using the **NtCancelIoFile** service:

NTSTATUS
NtCancelIoFile(
IN HANDLE *FileHandle,*
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

FileHandle - An open file handle to a file.

IoStatusBlock - A variable to receive the final completion status. For more information about this parameter see the **NtCreateFile** system service description.

The **NtCancelIoFile** service causes all pending I/O for the specified file to be marked as canceled. Most types of operations can be canceled immediately, while others may continue toward completion before they are actually canceled. For example, once a DMA disk drive has begun a transfer, the operation cannot be canceled by a device driver, but to the caller it will appear as if the operation had effectively been canceled.

Only those pending operations that were issued by the current thread using the specified handle are canceled. Any operations issued for the file by any other thread or any other process continues normally.

No specific access to the file is required in order to use this service since the caller is only canceling those operations that he requested in the first place.

All pending I/O operations complete with a status that indicates that the operation was canceled.

3.6.3 Miscellaneous I/O Control

Various operations may be performed on files to control the file, or the device associated with the file, using the **NtDeviceIoControlFile** service:

NTSTATUS
NtDeviceIoControlFile(
IN HANDLE *FileHandle,*
IN HANDLE *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**

OUT PIO_STATUS_BLOCK *IoStatusBlock,*
IN ULONG *IoControlCode,*
IN PVOID *InputBuffer* **OPTIONAL,**
IN ULONG *InputBufferLength,*
OUT PVOID *OutputBuffer* **OPTIONAL,**
IN ULONG *OutputBufferLength*
);

Parameters:

FileHandle - An open file handle to the file or device to which the control information should be given.

Event - An optional handle to an event to be set to the Signaled state when the operation completes.

ApcRoutine - An optional procedure to be invoked once the operation completes. For more information about this parameter see the **NtReadFile** system service description.

ApcContext - A pointer to pass as an argument to the *ApcRoutine*, if one was specified, when the operation completes. This argument is required if an *ApcRoutine* was specified.

IoStatusBlock - A variable to receive the final completion status and information about the operation. Service calls that return information, return the length of the data written to the output buffer in the *Information* field of this variable. For more information about this parameter see the **NtCreateFile** system service description.

IoControlCode - A code that indicates which device I/O control function is to be executed.

InputBuffer - An optional pointer to a buffer that contains the information to be given to the target device. This information is device-dependent.

InputBufferLength - The length of the *InputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

OutputBuffer - An optional pointer to a buffer that is to receive the device-dependent return information from the target device.

OutputBufferLength - The length of the *OutputBuffer* in bytes. If the buffer is not supplied, then this value is ignored.

The **NtDeviceIoControlFile** service is a device-dependent interface that extends the control that applications have over various devices within the system. This API provides a consistent view of the input and output data to the system while still providing the application and the driver a device-dependent method of specifying a communications interface.

The type of access that the caller needs to the file is dependent on the actual operation being performed.

Once the service has completed, the *Event*, if specified, will be set to the Signaled state. If no *Event* parameter was specified, then the file object specified by the *FileHandle* will be set to the Signaled

state. If an *ApcRoutine* was specified, it is invoked with the *ApcContext* and the address of the *IoStatusBlock* as its arguments.

3.6.4 Deleting a File

A file can be deleted using the **NtDeleteFile** service:

NTSTATUS
NtDeleteFile(
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
    );

Parameters:

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be deleted. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

The **NtDeleteFile** service allows the caller to delete a file. **DELETE** access to the target file is required. This service is equivalent to calling **NtOpenFile, NtSetInformationFile** with a file information class of *FileDispositionInformation*, and **NtClose**. However, this service is faster because less ring transitions are made.

3.6.5 Querying the Attributes of a File

The attributes of a file can be queried using the **NtQueryAttributesFile** service:

NTSTATUS
NtQueryAttributesFile(
    IN POBJECT_ATTRIBUTES *ObjectAttributes*,
    OUT PFILE_BASIC_INFORMATION *FileInformation*
    );

Parameters:

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, and a set of file object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the file to be queried. This file specification must be a fully qualified file specification or the name of a device, unless it is a file relative to the directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the file specified by the *ObjectName* field is a file specification relative to the directory file supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

FileInformation - A variable to receive the basic information about the file.

The **NtQueryAttributesFile** service allows the caller to query the basic information about a file. **FILE_READ_ATTRIBUTES** access to the target file is required. This service is equivalent to calling **NtOpenFile, NtQueryInformationFile** with a file information class of *FileBasicInformation*, and **NtClose**. However, this service is faster because less ring transitions are made.

The information that is returned in the *FileInformation* buffer is defined by the following structure:

```
typedef struct _FILE_BASIC_INFORMATION {
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

Field	Description
CreationTime	Date/time that the file was created
LastAccessTime	Date/time that the file was last accessed
LastWriteTime	Date/time that the file was last written
ChangeTime	Date/time that the file was last changed
FileAttributes	Attributes of the file

All dates and times are specified in the standard **Windows NT** system time format.

The file attributes field can be a combination of the following values:

FILE_ATTRIBUTE_NORMAL

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE

3.7 I/O Completion Objects

This section describes the creation and use of completion objects.

3.7.1 Creating/Opening I/O Completion Objects

When a user wishes to synchronize the completion of I/O through the use of completion objects, he must first create or open an I/O completion object. Creating or opening a completion object causes the system to return a handle to the specified object.

I/O completion object handles are closed via the generic **NtClose** service. This service is discussed elsewhere in the **Windows NT** documentation. It should be noted that, just like all other system objects, a completion object is not actually deleted until all of the valid handles to it are closed and no referenced pointers remain.

The user APIs that support creating and opening completion objects are as follows:

NtCreateIoCompletion - Create or open an I/O completion object and return a handle.
NtOpenIoCompletion - Open an existing I/O completion object and return a handle.

3.7.1.1 Create/Open I/O Completion Objects

An I/O completion object can be created or opened using the **NtCreateIoCompletion** service:

NTSTATUS
NtCreateIoCompletion(
OUT PHANDLE *IoCompletionHandle*,
IN ACCESS_MASK *DesiredAccess*,
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
IN ULONG *Count* **OPTIONAL**
);

Parameters:

IoCompletionHandle - A pointer to a variable that receives the I/O completion object handle value.

DesiredAccess - Specifies the type of access that the caller requires to the completion object.

DesiredAccess Flags

SYNCHRONIZE - The completion object handle may be waited.

IO_COMPLETION_QUERY_STATE - The completion object may be queried.

IO_COMPLETION_MODIFY_STATE - The completion object may be modified.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ* and *IO_COMPLETION_QUERY_STATE*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE* and *IO_COMPLETION_MODIFY_STATE*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE* and *SYNCHRONIZE*.

ObjectAttributes - A pointer to a structure that specifies the name of completion object, a root directory, a security descriptor, a quality of service descriptor, and a set of completion object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the completion object to be created or opened. This object name specification must be a fully qualified path, unless it is relative to the object directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the completion object specified by the *ObjectName* field is a path specification relative to the directory object supplied by this handle.

PSECURITY_DESCRIPTOR *SecurityDescriptor* - Optionally specifies the security descriptor that should be applied to the I/O completion object. The ACLs specified by the security descriptor are only applied to the object if it is created. If not supplied and the completion object is created, then the ACL placed on the completion object is formed from a combination of the ACL on the parent directory of the object and the current default ACL for the creating process.

PSECURITY_QUALITY_OF_SERVICE *SecurityQualityOfService* - Specifies the access a server should be given to the client's security context. This field is only used when a connection to a protected server is established. It allows the caller to control which parts of his security context are made available to the server and whether or not the server may impersonate the caller.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the I/O completion object is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

OBJ_EXCLUSIVE - Indicates that the I/O completion object is to be created such that no other opens to the object may be performed.

OBJ_OPENIF - Indicates that if the I/O completion object already exists then it is to be opened; otherwise it is to be created.

Count - An optional value that supplies the maximum number of threads that should be concurrently active. If this parameter is not specified, then the number of processors is used.

The **NtCreateIoCompletion** service either causes a new I/O completion object to be created, or it opens an existing completion object. The action taken is dependent on the name of the object being opened, and whether the object already existed, and the value of the *OBJ_OPENIF ObjectAttributes* flag. If the object is created, then the maximum target concurrent thread count is set to the value specified by the *Count* parameter. A handle to the I/O completion object with the *DesiredAccess* is returned.

Once the caller has established a handle to an I/O completion object, he can then associate the completion object with a file, via the **NtSetInformationFile** system service. As each request for the file is completed, the I/O system stores a completion message in the I/O completion object.

Each completion message consists of a caller-determined key identifying the target file, a caller-supplied *CompletionContext* pointer, which was passed as *ApcContext* to the asynchronous **Nt...File** service when the request was originally issued, and a pointer to the returned I/O status block for the completed request.

3.7.1.2 Open I/O Completion Objects

An I/O completion object can be opened using the **NtOpenIoCompletion** service:

NTSTATUS
NtOpenIoCompletion(
OUT PHANDLE *IoCompletionHandle*,
IN ACCESS_MASK *DesiredAccess*,
IN POBJECT_ATTRIBUTES *ObjectAttributes*
);

Parameters:

IoCompletionHandle - A pointer to a variable that receives the I/O completion object handle value.

DesiredAccess - Specifies the type of access that the caller requires to the completion object.

DesiredAccess Flags

SYNCHRONIZE - The completion object handle may be waited.

IO_COMPLETION_QUERY_STATE - The completion object may be queried.

IO_COMPLETION_MODIFY_STATE - The completion object may be modified.

The three following values are the generic access types that the caller may request. The mapping to specific access rights is given for each:

GENERIC_READ - Maps to *STANDARD_RIGHTS_READ* and *IO_COMPLETION_QUERY_STATE*.

GENERIC_WRITE - Maps to *STANDARD_RIGHTS_WRITE* and *IO_COMPLETION_MODIFY_STATE*.

GENERIC_EXECUTE - Maps to *STANDARD_RIGHTS_EXECUTE* and *SYNCHRONIZE*.

ObjectAttributes - A pointer to a structure that specifies the name of completion object, a root directory, a security descriptor, a quality of service descriptor, and a set of completion object attribute flags.

ObjectAttributes Structure

ULONG *Length* - Specifies the length of the object attributes structure. This field must be equal to the size of an *OBJECT_ATTRIBUTES* structure.

PUNICODE_STRING *ObjectName* - The name of the completion object to be opened. This object name specification must be a fully qualified path, unless it is relative to the object directory specified by the next field.

HANDLE *RootDirectory* - Optionally specifies a handle to a directory. If specified, then the name of the completion object specified by the *ObjectName* field is a path specification relative to the directory object supplied by this handle.

ULONG *Attributes* - A set of flags that controls the file object attributes.

OBJ_INHERIT - Indicates that the handle to the I/O completion object is to be inherited by the new process when an **NtCreateProcess** operation is performed to create a new process.

OBJ_CASE_INSENSITIVE - Indicates that the name lookup should ignore the case of *ObjectName* rather than performing an exact match search.

The **NtOpenIoCompletion** service opens an existing I/O completion object and returns a handle to it through the *IoCompletionHandle* parameter.

As with the **NtCreateIoCompletion** service, once the caller has established a handle to an I/O completion object, he can then associate the completion object with a file, via the **NtSetInformationFile** system service. As each request for the file is completed, the I/O system stores a completion message in the I/O completion object.

Each completion message consists of a caller-determined key identifying the target file, a caller-supplied *CompletionContext* pointer, which was passed as *ApcContext* to the asynchronous **Nt...File** service when the request was originally issued, and a pointer to the returned I/O status block for the completed request.

3.7.2 Operating on I/O Completion Objects

This section presents those services that manipulate I/O completion objects. The APIs that support operations on I/O completion objects are as follows:

NtQueryIoCompletion - Query the state of an I/O completion object.
NtSetIoCompletion - Inserts a message onto an I/O completion object.
NtRemoveIoCompletion - Removes an entry from an I/O completion object.

3.7.2.1 Querying Completion Objects

The state of an I/O completion object can be queried using the **NtQueryIoCompletion** service:

NTSTATUS
NtQueryIoCompletion(
IN HANDLE *IoCompletionHandle*,
IN IO_COMPLETION_INFORMATION_CLASS IoCompletionInformationClass,
OUT PVOID *IoCompletionInformation*,
IN ULONG *IoCompletionInformationLength*,
OUT PULONG *ReturnLength* **OPTIONAL**
);

Parameters:

IoCompletionHandle - Supplies a handle to an open I/O completion object to be queried.

IoCompletionInformationClass - Specifies the type of information that should be returned about the I/O completion object. The information returned in the *IoCompletionInformation* buffer is defined by the following type codes:

IoCompletionInformationClass Values

IoCompletionBasicInformation - Returns basic information about the specified I/O completion object. **IO_COMPLETION_QUERY_STATE** access to the object is required.

IoCompletionInformation - A pointer to a buffer to receive the desired information about the I/O completion object. The contents of this buffer are defined by the *IoCompletionInformationClass* parameter described above.

IoCompletionInformationLength - The length of the *IoCompletionInformation* buffer in bytes.

ReturnLength - An optional pointer to a variable to receive the actual number of bytes of information returned in the *IoCompletionInformation* buffer.

The **NtQueryIoCompletion** service returns information about the specified I/O completion object. The information in the buffer is defined by the following type codes and structures.

IoCompletionInformation Format by I/O Completion Information Class

IoCompletionBasicInformation - Data type is *IO_COMPLETION_BASIC_INFORMATION*.

```
typedef struct _IO_COMPLETION_BASIC_INFORMATION {
        LONG Depth;
} IO_COMPLETION_BASIC_INFORMATION;
```

Field	Description
Depth	Depth, in messages, of the I/O completion object

IO_COMPLETION_QUERY_STATE access to the I/O completion object is required to obtain this information.

Once the information about the object has been returned, the caller can determine how much information was actually returned by examining the variable passed in as the *ReturnLength* parameter, if one was passed.

3.7.2.2 Setting Completion Objects

A completion message can be manually queued to an I/O completion object using the **NtSetIoCompletion** service:

```
NTSTATUS
NtSetIoCompletion(
        IN HANDLE IoCompletionHandle,
        IN ULONG KeyContext,
        IN PVOID ApcContext,
        IN NTSTATUS IoStatus,
        IN ULONG IoStatusInformation
        );
```

Parameters:

IoCompletionHandle - A handle to the I/O completion port.

KeyContext - Supplies the key contex that is returned during a call to **NtRemoveIoCompletion**.

ApcContext - Supplies the APC context that is returned during a call to **NtRemoveIoCompletion**.

IoStatus - Supplies the status data that will be returned in the *Status* field of the I/O status block during a call to **NtRemoveIoCompletion**.

IoStatusInformation - Supplies the information data that will be returned in the *Information* field of the I/O status block during a call to **NtRemoveIoCompletion**.

The **NtSetIoCompletion** service allows the caller to insert an I/O completion message into the completion object manually. This allows threads that are waiting on messages to arrive to be awakened to deal with a particular work item posted by the caller. Note that no I/O was actually performed to cause the completion message to be read by the remover of the item.

3.7.2.3 Removing Messages from Completion Objects

An I/O completion message can be removed from an I/O completion object using the **NtRemoveIoCompletion** service:

NTSTATUS
NtRemoveIoCompletion(
IN HANDLE *IoCompletionHandle*,
OUT PVOID **KeyContext*,
OUT PVOID **ApcContext*,
OUT PIO_STATUS_BLOCK *IoStatusBlock*,
IN PLARGE_INTEGER *Timeout* **OPTIONAL**
);

Parameters:

IoCompletionHandle - A handle to the I/O completion port.

KeyContext - Supplies a pointer to a variable to receive the key contex that was specified when the I/O completion object was associated with a file object.

ApcContext - Supplies a pointer to a variable to receive the context that was specified when the I/O was issued. This value was passed in as the *ApcContext* parameter when the I/O was queued.

IoStatus - Supplies a pointer to a variable that receives the final I/O completion status from the I/O operation.

Timeout - Supplies a pointer to an optional time out value.

The **NtRemoveIoCompletion** service removes a single I/O completion message from the completion object. If an entry is removed, then the *KeyContext*, *ApcContext*, and *IoStatus* variables receive the information about the I/O operation that was completed. The *Status* field of the *IoStatus* variable indicates whether or not the I/O operation was successfully completed. Note that this is separate from the return value from this service, which indicates whether or not a completion message was successfully removed from the completion object.

If there are no entries in the completion object, or if there are already *Count* threads concurrently ready and/or running due to other completion messages having been removed, then the calling thread will wait for another message according to the *Timeout* parameter. This parameter is treated in the normal manner of all time-out values in *Windows NT*.

4. Naming Conventions

Devices in **Windows NT** are named according to a very simple set of rules. There are three general rules:

1) If there can only be one device of the specified type in the system, such as the PC subsystem keyboard, then the name of the device is simply the device type name.

2) If there can be more than one device of the specified type in the system, such as a floppy, then the name of the device is the device type name followed by a decimal number that indicates which device of that type it is.

3) For devices such as disks, which can be partitioned, the name of the partition is the name of the device followed by *\Partition* and a decimal number representing which partition on the disk it is. The first partition on a disk is called *\Partition1*. The name that refers to the entire device for partitioned media is *\Partition0.*

For example, the following are valid **Windows NT** device names.

- *Device\Floppy2*
- *Device\Harddisk1\Partition3*
- *Device\Keyboard*
- *Device\Mouse*

Note that all of the above device names are in a directory called the *\Device* directory. All device names in **Windows NT** reside in this object directory by convention. Any valid object directory operations can be used to determine the names of the devices on the system, provided the caller has the appropriate privileges and access to the object directory.

5. Appendix A - Time Field Changes

This section contains a list of those APIs that implicitly change the various time fields associated with a file.

5.1 Last Access Time

The Last Access Time field for a file is implicitly changed under the following conditions:

- **NtQueryDirectoryFile** - The directory file's time field is updated.
- **NtCreateFile** - The file's time field is set if the file was created.
- **NtReadFile** - The file's time field is updated.

5.2 Last Modify Time

- **NtCreateFile** - If the file was created, superseded, or overwritten, then the file's time field is updated. If the file was created or superseded then the parent directory's time field is also updated.
- **NtSetInformationFile**

- **FileLinkInformation** - The directory file containing the name of the link's time field is updated.

- **FileDispositionInformation** - The time field of the directory that contains the file is updated.

- **FileRenameInformation** - The old and the new parent directory file's times are updated.

o - **NtWriteFile** - The file's time field is updated.

5.3 Last Change Time

o - **NtCreateFile** - If the file was created, superseded, or overwritten, then the file's time field is updated. If the file was created or superseded then the parent directory's time field is also updated.

o - **NtSetInformationFile**

- **FileLinkInformation** - The time field of both the file and the directory containing the name of the link are updated.

- **FileDispositionInformation** - The time field of both the file and the directory containing the file are updated.

- **FileRenameInformation** - The time field of both the old and the new parent directories are updated.

- **FileAllocationInformation** - The file's time field is updated.

- **FileEndOfFileInformation** - The file's time field is updated.

o - **NtWriteFile** - The file's time field is updated.

o - **NtSetSecurityObject** - The file's time field is updated.

6. Revision History

Original Draft 1.0, March 21, 1989

Revision Draft 1.1, March 31, 1989

- Fixed spelling, grammar and numbering problems.
- Incorporated initial review comments.
- Removed all APIs that didn't use file handles.
- Rewrote overview section dealing with file objects.
- Added access right types to services.
- Redesigned **NtCreateFile** service.
- Removed **NtOpenFile** service.
- Revamped **NtQueryDirectoryFile** service.
- Added more types to **NtQueryInformationFile** service.
- Added more types to **NtSetInformationFile** service.
- Performed general fixup on most other services.
- Added description of DISPATCH_LEVEL driver context.
- Changed device work queues to device queues.
- Redesigned communication region protocol.
- Planned section on volume verification.

Revision Draft 1.2, May 12, 1989

- Allow setting of owner in **NtSetInformationFile.**
- Removed device info from **NtQueryFsInformationFile.**
- Changed **NtQueryFsInformationFile** to **QueryVolume**.
- Changed **NtSetFsInformationFile** to **SetVolume**.
- Added ChangeTime to appropriate structures.
- Added I/O provided time-out functions.
- Remove mount entry point from file systems.
- Fleshed out section on volume verification.
- Wrote section on error logging and handling.
- Wrote section on naming conventions.
- Added "subsystem input" section for terminals.
- Wrote section on network service description.
- Added directory access options.
- Fixed access type names.
- Make all byte offsets block/byte offsets.
    - o Read pointer
    - o Write pointer
    - o File allocation size
    - o End of file marker
- Add new security access types.
- Flesh out Miscellaneous I/O APIs.
- Change FILE_READ and _WRITE back again.

Revision Draft 1.3, October 9, 1989

- Split specification into two separate specs.
- Redo attributes again for security changes (twice).

- Add "names" type to **NtQueryDirectoryFile** since other API was dropped by object manager.
- Change APC parameter to context and make PVOID.
- Add AscendingDirectories flag to volume info.
- Make file objects waitable objects.
- Make block/byte values zero-based.
- Add synchronous I/O.
- Only signal file handle if no event specified.
- Fix FILEINFO and FSINFO to be like all other APIs.
- Remove nonsensical directory desired accesses.
- Return actual action in Information on create/open.
- Add FILE_SHARE_ NO_DELETE and NO_RENAME.
- Drop FILE_CREATE_TREE_CONNECTION. Will be service.
- Drop FILE_EXECUTE desired access restrictions.
- Drop FILE_APPEND desired access restrictions.
- Drop or change name of privileges.
- Added time field changes appendix.

Revision Draft 1.4, January 21, 1990

- Added **NtOpenFile** system service.
- Removed **NtQueryAclFile** and **NtSetAclFile** APIs.
- Removed documentation on FileAclInformation.
- Added **NtLockFile** and **NtUnlockFile** services again.
- Change most services to have synchronous APIs.
- Redo attributes again for security changes.
- Revamped structures around security, especially for directories and subdirectories.
- Added EAs to **NtCreateFile**.
- Redo EA APIs and EA structures.
- Added rewind capabilities to EA and directory services.
- Added optional key parameter to **NtReadFile** and **NtWriteFile**.
- Fixed object attributes structure type name and fields.
- Converted APIs from Block and Byte to LARGE_INTEGER.
- Reversed polarity of shared delete and rename flags.
- Expanded type names out to full names.
- Miscellaneous edits and explanation changes.

Revision Draft 1.5, July 9, 1990

- Add EaListLength parameter to **NtQueryEaFile**.
- Removed FILE_MAPPED_IO option.
- Removed FILE_SHARE_RENAME share access.
- Document file sharing semantics.
- Add FileFsSizeInformation to **NtQueryVolumeInformationFile**.
- Removed FileFsBiosInformation from **NtQueryVolumeInformationFile**.
- Add RemovableMedia and SupportsObjects fields for volumes.
- Add FILE_OVERWRITE, FILE_OVERWRITE_IF to **NtCreateFile**.
- Document directory wildcarding.
- Document deleting a file is last valid I/O operation.
- Add FileAlignmentInformation to **NtQueryInformationFile**.
- Replace OBJ_OPEN_LINK with FILE_OPEN_LINK.

- Add FILE_TRAVERSE as legal directory access.
- Add FILE_OPEN_UNKNOWN_OBJECT option.
- Add FILE_OPENED_UNKNOWN_OBJECT I/O status block value.
- Replace FILE_DISABLE_CACHING with FILE_NO_INTERMEDIATE_BUFFERING and add requirement restrictions description.
- Add FILE_COMPLETE_IF_OPLOCKED option to create and open.
- Add FileRemainingNameInformation query information type.
- Explicitly state that locking beyond EOF is permissible.
- Switch fields in FILE_FULL_EA_INFORMATION to keep compatibility with OS/2.
- Fixed references to IOSB and PIOSB.
- Removed explicit ACL and owner interfaces and converted to the new security semantics.
- Add ability for synchronous I/O locks to be asynchronous.
- Subsumed **NtSetNewSizeFile** functionality in **NtSetInformationFile**.
- Removed FileOwnerInformation from **NtQueryInformationFile**.
- Removed FileOwnerInformation from **NtSetInformationFile**.
- Removed FILE_OWNER_INFORMATION structure type declaration.

Revision Draft 1.6, July 15, 1993

- Removed outdated "++" notation for subsystems.
- Updated system name from NT OS/2 to Windows NT.
- Removed error ports from all appropriate APIs.
- Added new file attribute definitions for FILE_ATTRIBUTE_TEMPORARY, FILE_ATTRIBUTE_ATOMIC_WRITE, and FILE_ATTRIBUTE_XACTION_WRITE.
- Removed all vestiges of "unknown objects" and all related functionality.
- Replaced old style create/open directory manipulation flags (see next).
- Documented all new Create/Open options:
    - FILE_DIRECTORY_FILE
    - FILE_NON_DIRECTORY_FILE
    - FILE_RANDOM_ACCESS
    - FILE_NO_EA_KNOWLEDGE
    - FILE_DELETE_ON_CLOSE
    - FILE_OPEN_BY_FILE_ID
    - FILE_OPEN_FOR_BACKUP_INTENT
- Updated all appropriate CHAR's to WCHAR's in accordance w/Unicode changes.
- Updated all STRING's to UNICODE_STRING's in accordance w/Unicode changes.
- Removed source/target process from **NtReadFile** and **NtWriteFile**.
- Removed **NtReadTerminalFile** API.
- Updated all TIME data types to LARGE_INTEGER's.
- Moved FILE_ATTRIBUTE_DIRECTORY flag into attributes for query operations.
- Added FileBothDirectoryInformation file information class to **NtQueryDirectoryFile**.
- Changed Action field of FILE_NOTIFY_INFORMATION to ULONG.
- Added FileAlternateNameInformation to **NtQueryInformationFile**.
- Added FileStreamInformation to **NtQueryInformationFile**.
- Changed FileNameInformation to FileRenameInformation for **NtSetInformationFile**.
- Updated Length parameter to LARGE_INTEGER from ULONG for locking services.
- Added FileFsDeviceInformation and FileFsAttributeInformation to **NtQueryVolumeInformation**.

Revision Draft 1.7, May 1, 1995

- Added new FILE_OPEN_TRANSACTED and FILE_RESERVE_OPFILTER create/open options.
- Removed FILE_ATTRIBUTE_ATOMIC_WRITE and FILE_ATTRIBUTE_XACTION_WRITE and added FILE_ATTRIBUTE_COMPRESSED and FILE_ATTRIBUTE_OFFLINE..
- Added new STORAGE_TYPE enumerated type as well as new create/open option fields for storage types.
- Added values for FILE_NOTIFY_CHANGE_STREAM_NAME, FILE_NOTIFY_CHANGE_STREAM_SIZE, and FILE_NOTIFY_CHANGE_STREAM_WRITE.
- Added documentation of file system attributes flags, and included new flags FILE_FILE_COMPRESSED and FILE_VOLUME_IS_COMPRESSED for compression.
- Added FILE_VIRTUAL_VOLUME device characteristic flag for virtual volumes.
- Added the following query and set information class information values and their associated structure type definitions:
    - FileCompressionInformation
    - FileCopyOnWriteInformation
    - FileCompletionInformation
    - FileMoveClusterInformation
    - FileOleClassIdInformation
    - FileOleStateBitsInformation
    - FileApplicationExplorableInformation
    - FileApplicationExplorableChildrenInformation
    - FileObjectIdInformation
    - FileOleAllInformation
    - FileContentIndexInformation
    - FileInheritContentIndexInformation
    - FileOleInformation
- Added new **NtQueryOleDirectoryFile** API description.
- Added new FileOleDirectoryInformation directory information class and its associated structure type definition.
- Added new directory query information class for OLE files.
- Added query and set volume information class information values and its associated type definitions for FileFsControlInformation
- Added new **NtQueryQuotaInformationFile** and **NtSetQuotaInformationFile** API descriptions.
- Added new data structure types (FILE_GET_QUOTA_INFORMATION and FILE_QUOTA_INFORMATION) for the above services.
- Added new **NtDeleteFile** API description.
- Added new **NtQueryAttributesFile** API description.
- Added new I/O completion object section for APIs, access rights, information class values, and data structures.
- Removed old **NtDeviceIoControlFile** and **NtFsControlFile** appendicies to alleviate concerns that they weren't filled in (since they never will be populated).
- Added device types for FILE_DEVICE_BATTERY and FILE_DEVICE_BUS_EXTENDER.
- Removed POSIX and OS/2 subsystem API implementation sections