CreateFile function

Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified.

To perform this operation as a transacted operation, which results in a handle that can be used for transacted I/O, use the **CreateFileTransacted** function.

Syntax

C++

```
HANDLE WINAPI CreateFile(
  _In_     LPCTSTR               lpFileName,
  _In_     DWORD                 dwDesiredAccess,
  _In_     DWORD                 dwShareMode,
  _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  _In_     DWORD                 dwCreationDisposition,
  _In_     DWORD                 dwFlagsAndAttributes,
  _In_opt_ HANDLE                hTemplateFile
);
```

Parameters

lpFileName [in]

The name of the file or device to be created or opened. You may use either forward slashes (/) or backslashes (\) in this name.

In the ANSI version of this function, the name is limited to **MAX_PATH** characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\\?\" to the path. For more information, see **Naming Files, Paths, and Namespaces**.

For information on special device names, see **Defining an MS-DOS Device Name**.

To create a file stream, specify the name of the file, a colon, and then the name of the stream. For more information, see **File Streams**.

dwDesiredAccess [in]

The requested access to the file or device, which can be summarized as read, write, both or neither zero).

The most commonly used values are **GENERIC_READ**, **GENERIC_WRITE**, or both (`GENERIC_READ | GENERIC_WRITE`). For more information, see Generic Access Rights, File Security and Access Rights, File Access Rights Constants, and ACCESS_MASK.

If this parameter is zero, the application can query certain metadata such as file, directory, or device attributes without accessing that file or device, even if **GENERIC_READ** access would have been denied.

You cannot request an access mode that conflicts with the sharing mode that is specified by the *dwShareMode* parameter in an open request that already has an open handle.

For more information, see the Remarks section of this topic and Creating and Opening Files.

dwShareMode [in]

The requested sharing mode of the file or device, which can be read, write, both, delete, all of these, or none (refer to the following table). Access requests to attributes or extended attributes are not affected by this flag.

If this parameter is zero and **CreateFile** succeeds, the file or device cannot be shared and cannot be opened again until the handle to the file or device is closed. For more information, see the Remarks section.

You cannot request a sharing mode that conflicts with the access mode that is specified in an existing request that has an open handle. **CreateFile** would fail and the GetLastError function would return **ERROR_SHARING_VIOLATION**.

To enable a process to share a file or device while another process has the file or device open, use a compatible combination of one or more of the following values. For more information about valid combinations of this parameter with the *dwDesiredAccess* parameter, see Creating and Opening Files.

Note The sharing options for each open handle remain in effect until that handle is closed, regardless of process context.

Value	Meaning
0 0x00000000	Prevents other processes from opening a file or device if they request delete, read, or write access.
FILE_SHARE_DELETE 0x00000004	Enables subsequent open operations on a file or device to request delete access. Otherwise, other processes cannot open the file or device if they request delete access. If this flag is not specified, but the file or device has been opened for delete access, the

	function fails. **Note** Delete access allows both delete and rename operations.
FILE_SHARE_READ 0x00000001	Enables subsequent open operations on a file or device to request read access. Otherwise, other processes cannot open the file or device if they request read access. If this flag is not specified, but the file or device has been opened for read access, the function fails.
FILE_SHARE_WRITE 0x00000002	Enables subsequent open operations on a file or device to request write access. Otherwise, other processes cannot open the file or device if they request write access. If this flag is not specified, but the file or device has been opened for write access or has a file mapping with write access, the function fails.

lpSecurityAttributes [in, optional]

A pointer to a SECURITY_ATTRIBUTES structure that contains two separate but related data members: an optional security descriptor, and a Boolean value that determines whether the returned handle can be inherited by child processes.

This parameter can be **NULL**.

If this parameter is **NULL**, the handle returned by **CreateFile** cannot be inherited by any child processes the application may create and the file or device associated with the returned handle gets a default security descriptor.

The **lpSecurityDescriptor** member of the structure specifies a SECURITY_DESCRIPTOR for a file or device. If this member is **NULL**, the file or device associated with the returned handle is assigned a default security descriptor.

CreateFile ignores the **lpSecurityDescriptor** member when opening an existing file or device, but continues to use the **bInheritHandle** member.

The **bInheritHandle**member of the structure specifies whether the returned handle can be inherited.

For more information, see the Remarks section.

dwCreationDisposition [in]

An action to take on a file or device that exists or does not exist.

For devices other than files, this parameter is usually set to **OPEN_EXISTING**.

For more information, see the Remarks section.

This parameter must be one of the following values, which cannot be combined:

Value	Meaning
CREATE_ALWAYS 2	Creates a new file, always. If the specified file exists and is writable, the function overwrites the file, the function succeeds, and last-error code is set to **ERROR_ALREADY_EXISTS** (183). If the specified file does not exist and is a valid path, a new file is created, the function succeeds, and the last-error code is set to zero. For more information, see the Remarks section of this topic.
CREATE_NEW 1	Creates a new file, only if it does not already exist. If the specified file exists, the function fails and the last-error code is set to **ERROR_FILE_EXISTS** (80). If the specified file does not exist and is a valid path to a writable location, a new file is created.
OPEN_ALWAYS 4	Opens a file, always. If the specified file exists, the function succeeds and the last-error code is set to **ERROR_ALREADY_EXISTS** (183). If the specified file does not exist and is a valid path to a writable location, the function creates a file and the last-error code is set to zero.
OPEN_EXISTING 3	Opens a file or device, only if it exists. If the specified file or device does not exist, the function fails and the last-error code is set to **ERROR_FILE_NOT_FOUND** (2). For more information about devices, see the Remarks section.
TRUNCATE_EXISTING 5	Opens a file and truncates it so that its size is zero bytes, only if it exists. If the specified file does not exist, the function fails and the last-error code is set to **ERROR_FILE_NOT_FOUND** (2). The calling process must open the file with the **GENERIC_WRITE** bit set as part of the

	dwDesiredAccess parameter.

dwFlagsAndAttributes [in]

The file or device attributes and flags, **FILE_ATTRIBUTE_NORMAL** being the most common default value for files.

This parameter can include any combination of the available file attributes (**FILE_ATTRIBUTE_***). All other file attributes override **FILE_ATTRIBUTE_NORMAL**.

This parameter can also contain combinations of flags (**FILE_FLAG_***) for control of file or device caching behavior, access modes, and other special-purpose flags. These combine with any **FILE_ATTRIBUTE_*** values.

This parameter can also contain Security Quality of Service (SQOS) information by specifying the **SECURITY_SQOS_PRESENT** flag. Additional SQOS-related flags information is presented in the table following the attributes and flags tables.

Note When **CreateFile** opens an existing file, it generally combines the file flags with the file attributes of the existing file, and ignores any file attributes supplied as part of *dwFlagsAndAttributes*. Special cases are detailed in Creating and Opening Files.

Some of the following file attributes and flags may only apply to files and not necessarily all other types of devices that **CreateFile** can open. For additional information, see the Remarks section of this topic and Creating and Opening Files.

For more advanced access to file attributes, see SetFileAttributes. For a complete list of all file attributes with their values and descriptions, see File Attribute Constants.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE 32 (0x20)	The file should be archived. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_ENCRYPTED 16384	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories. For more information, see File Encryption.

(0x4000)	This flag has no effect if **FILE_ATTRIBUTE_SYSTEM** is also specified. This flag is not supported on Home, Home Premium, Starter, or ARM editions of Windows.
FILE_ATTRIBUTE_HIDDEN 2 (0x2)	The file is hidden. Do not include it in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL 128 (0x80)	The file does not have other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE 4096 (0x1000)	The data of a file is not immediately available. This attribute indicates that file data is physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY 1 (0x1)	The file is read only. Applications can read the file, but cannot write to or delete it.
FILE_ATTRIBUTE_SYSTEM 4 (0x4)	The file is part of or used exclusively by an operating system.
FILE_ATTRIBUTE_TEMPORARY 256 (0x100)	The file is being used for temporary storage. For more information, see the Caching Behavior section of this topic.

Flag	Meaning
FILE_FLAG_BACKUP_SEMANTICS 0x02000000	The file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks when the process has **SE_BACKUP_NAME** and **SE_RESTORE_NAME** privileges. For more information, see Changing Privileges in a Token. You must set this flag to obtain a handle to a directory. A directory handle can be passed to some functions instead of a file handle. For more information, see the Remarks section.
FILE_FLAG_DELETE_ON_CLOSE 0x04000000	The file is to be deleted immediately after all of its handles are closed, which includes the specified handle and any other open or duplicated handles. If there are existing open handles to a file, the call fails unless they were all opened with the **FILE_SHARE_DELETE** share mode. Subsequent open requests for the file fail, unless the **FILE_SHARE_DELETE** share mode is specified.
FILE_FLAG_NO_BUFFERING 0x20000000	The file or device is being opened with no system caching for data reads and writes. This flag does not affect hard disk caching or memory mapped files. There are strict requirements for successfully working with files opened with **CreateFile** using the **FILE_FLAG_NO_BUFFERING** flag, for details see File Buffering.
FILE_FLAG_OPEN_NO_RECALL 0x00100000	The file data is requested, but it should continue to be located in remote storage. It should not be transported back to local storage. This flag is for use by remote storage systems.
FILE_FLAG_OPEN_REPARSE_POINT 0x00200000	Normal reparse point processing will not occur; **CreateFile** will attempt to open the reparse point. When a file is opened, a file handle is returned, whether or not the filter that controls the reparse point is operational. This flag cannot be used with the **CREATE_ALWAYS** flag. If the file is not a reparse point, then this flag is ignored. For more information, see the Remarks section.

FILE_FLAG_OVERLAPPED 0x40000000	The file or device is being opened or created for asynchronous I/O. When subsequent I/O operations are completed on this handle, the event specified in the **OVERLAPPED** structure will be set to the signaled state. If this flag is specified, the file can be used for simultaneous read and write operations. If this flag is not specified, then I/O operations are serialized, even if the calls to the read and write functions specify an **OVERLAPPED** structure. For information about considerations when using a file handle created with this flag, see the Synchronous and Asynchronous I/O Handles section of this topic.
FILE_FLAG_POSIX_SEMANTICS 0x0100000	Access will occur according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support that naming. Use care when using this option, because files created with this flag may not be accessible by applications that are written for MS-DOS or 16-bit Windows.
FILE_FLAG_RANDOM_ACCESS 0x10000000	Access is intended to be random. The system can use this as a hint to optimize file caching. This flag has no effect if the file system does not support cached I/O and **FILE_FLAG_NO_BUFFERING**. For more information, see the Caching Behavior section of this topic.
FILE_FLAG_SESSION_AWARE 0x00800000	The file or device is being opened with session awareness. If this flag is not specified, then per-session devices (such as a device using RemoteFX USB Redirection) cannot be opened by processes running in session 0. This flag has no effect for callers not in session 0. This flag is supported only on server editions of Windows. **Windows Server 2008 R2 and Windows Server 2008:** This flag is not supported before Windows Server 2012.
FILE_FLAG_SEQUENTIAL_SCAN 0x08000000	Access is intended to be sequential from beginning to end. The system can use this as a hint to optimize file caching. This flag should not be used if read-behind (that is, reverse scans) will be used. This flag has no effect if the file system does not support cached I/O and **FILE_FLAG_NO_BUFFERING**. For more information, see the Caching Behavior section of this topic.
FILE_FLAG_WRITE_THR	Write operations will not go through any intermediate cache, they will go directly to disk.

OUGH 0x80000000	For additional information, see the Caching Behavior section of this topic.

The *dwFlagsAndAttributes*parameter can also specify SQOS information. For more information, see Impersonation Levels. When the calling application specifies the **SECURITY_SQOS_PRESENT** flag as part of *dwFlagsAndAttributes*, it can also contain one or more of the following values.

Security flag	Meaning
SECURITY_ANONYMOUS	Impersonates a client at the Anonymous impersonation level.
SECURITY_CONTEXT_TRACKING	The security tracking mode is dynamic. If this flag is not specified, the security tracking mode is static.
SECURITY_DELEGATION	Impersonates a client at the Delegation impersonation level.
SECURITY_EFFECTIVE_ONLY	Only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available. This allows the client to limit the groups and privileges that a server can use while impersonating the client.
SECURITY_IDENTIFICATION	Impersonates a client at the Identification impersonation level.
SECURITY_IMPERSONATION	Impersonate a client at the impersonation level. This is the default behavior if no other flags are specified along with the **SECURITY_SQOS_PRESENT** flag.

hTemplateFile [in, optional]

A valid handle to a template file with the **GENERIC_READ** access right. The template file supplies file attributes and extended attributes for the file that is being created.

This parameter can be **NULL**.

When opening an existing file, **CreateFile** ignores this parameter.

When opening a new encrypted file, the file inherits the discretionary access control list from its parent directory. For additional information, see File Encryption.

Return value

If the function succeeds, the return value is an open handle to the specified file, device, named pipe, or mail slot.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call GetLastError.

Remarks

CreateFile was originally developed specifically for file interaction but has since been expanded and enhanced to include most other types of I/O devices and mechanisms available to Windows developers. This section attempts to cover the varied issues developers may experience when using **CreateFile** in different contexts and with different I/O types. The text attempts to use the word *file* only when referring specifically to data stored in an actual file on a file system. However, some uses of *file* may be referring more generally to an I/O object that supports file-like mechanisms. This liberal use of the term *file* is particularly prevalent in constant names and parameter names because of the previously mentioned historical reasons.

When an application is finished using the object handle returned by **CreateFile**, use the CloseHandle function to close the handle. This not only frees up system resources, but can have wider influence on things like sharing the file or device and committing data to disk. Specifics are noted within this topic as appropriate.

Windows Server 2003 and Windows XP: A sharing violation occurs if an attempt is made to open a file or directory for deletion on a remote computer when the value of the *dwDesiredAccess* parameter is the **DELETE** access flag (0x00010000) **OR**'ed with any other access flag, and the remote file or directory has not been opened with **FILE_SHARE_DELETE**. To avoid the sharing violation in this scenario, open the remote file or directory with the **DELETE** access right only, or call DeleteFile without first opening the file or directory for deletion.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes that have a mounted file system with this support, a new file inherits the compression and encryption attributes of its directory.

You cannot use **CreateFile** to control compression, decompression, or decryption on a file or directory. For more information, see Creating and Opening Files, File Compression and Decompression, and File Encryption.

Windows Server 2003 and Windows XP: For backward compatibility purposes, **CreateFile** does not apply inheritance rules when you specify a security descriptor in *lpSecurityAttributes*. To support inheritance, functions that later query the security descriptor of this file may heuristically determine and report that inheritance is in effect. For more information, see Automatic Propagation of Inheritable ACEs.

As stated previously, if the *lpSecurityAttributes* parameter is **NULL**, the handle returned by **CreateFile** cannot be inherited by any child processes your application may create. The following information regarding this parameter also applies:

- If the **bInheritHandle** member variable is not **FALSE**, which is any nonzero value, then the handle can be inherited. Therefore it is critical this structure member be properly initialized to **FALSE** if you do not intend the handle to be inheritable.
- The access control lists (ACL) in the default security descriptor for a file or directory are inherited from its parent directory.
- The target file system must support security on files and directories for the **lpSecurityDescriptor** member to have an effect on them, which can be determined by using GetVolumeInformation.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	See remarks
SMB 3.0 with Scale-out File Shares (SO)	See remarks
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Note that **CreateFile** with supersede disposition will fail if performed on a file where there is already an open alternate data stream.

Symbolic Link Behavior

If the call to this function creates a file, there is no change in behavior. Also, consider the following information regarding **FILE_FLAG_OPEN_REPARSE_POINT**:

- If **FILE_FLAG_OPEN_REPARSE_POINT** is specified:
  - If an existing file is opened and it is a symbolic link, the handle returned is a handle to the symbolic link.
  - If **TRUNCATE_EXISTING** or **FILE_FLAG_DELETE_ON_CLOSE** are specified, the file affected is a symbolic link.

- If **FILE_FLAG_OPEN_REPARSE_POINT** is not specified:
  - If an existing file is opened and it is a symbolic link, the handle returned is a handle to the target.
  - If **CREATE_ALWAYS**, **TRUNCATE_EXISTING**, or **FILE_FLAG_DELETE_ON_CLOSE** are specified, the file affected is the target.

Caching Behavior

Several of the possible values for the *dwFlagsAndAttributes* parameter are used by **CreateFile** to control or affect how the data associated with the handle is cached by the system. They are:

- **FILE_FLAG_NO_BUFFERING**
- **FILE_FLAG_RANDOM_ACCESS**
- **FILE_FLAG_SEQUENTIAL_SCAN**
- **FILE_FLAG_WRITE_THROUGH**
- **FILE_ATTRIBUTE_TEMPORARY**

If none of these flags is specified, the system uses a default general-purpose caching scheme. Otherwise, the system caching behaves as specified for each flag.

Some of these flags should not be combined. For instance, combining **FILE_FLAG_RANDOM_ACCESS** with **FILE_FLAG_SEQUENTIAL_SCAN** is self-defeating.

Specifying the **FILE_FLAG_SEQUENTIAL_SCAN** flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip forward over small ranges of bytes. If an application moves the file pointer for random access, optimum caching performance most likely will not occur. However, correct operation is still guaranteed.

The flags **FILE_FLAG_WRITE_THROUGH** and **FILE_FLAG_NO_BUFFERING** are independent and may be combined.

If **FILE_FLAG_WRITE_THROUGH** is used but **FILE_FLAG_NO_BUFFERING** is not also specified, so that system caching is in effect, then the data is written to the system cache but is flushed to disk without delay.

If **FILE_FLAG_WRITE_THROUGH** and **FILE_FLAG_NO_BUFFERING** are both specified, so that system caching is not in effect, then the data is immediately flushed to disk without going through the Windows system cache. The operating system also requests a write-through of the hard disk's local hardware cache to persistent media.

Note Not all hard disk hardware supports this write-through capability.

Proper use of the **FILE_FLAG_NO_BUFFERING** flag requires special application considerations. For more information, see File Buffering.

A write-through request via **FILE_FLAG_WRITE_THROUGH** also causes NTFS to flush any metadata changes, such as a time stamp update or a rename operation, that result from processing the request. For this reason, the

FILE_FLAG_WRITE_THROUGH flag is often used with the **FILE_FLAG_NO_BUFFERING** flag as a replacement for calling the FlushFileBuffers function after each write, which can cause unnecessary performance penalties. Using these flags together avoids those penalties. For general information about the caching of files and metadata, see File Caching.

When **FILE_FLAG_NO_BUFFERING** is combined with **FILE_FLAG_OVERLAPPED**, the flags give maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations take more time, because data is not being held in the cache. Also, the file metadata may still be cached (for example, when creating an empty file). To ensure that the metadata is flushed to disk, use the FlushFileBuffers function.

Specifying the **FILE_ATTRIBUTE_TEMPORARY** attribute causes file systems to avoid writing data back to mass storage if sufficient cache memory is available, because an application deletes a temporary file after a handle is closed. In that case, the system can entirely avoid writing the data. Although it does not directly control data caching in the same way as the previously mentioned flags, the **FILE_ATTRIBUTE_TEMPORARY** attribute does tell the system to hold as much as possible in the system cache without writing and therefore may be of concern for certain applications.

Files

If you rename or delete a file and then restore it shortly afterward, the system searches the cache for file information to restore. Cached information includes its short/long name pair and creation time.

If you call **CreateFile** on a file that is pending deletion as a result of a previous call to DeleteFile, the function fails. The operating system delays file deletion until all handles to the file are closed. GetLastError returns **ERROR_ACCESS_DENIED**.

The *dwDesiredAccess* parameter can be zero, allowing the application to query file attributes without accessing the file if the application is running with adequate security settings. This is useful to test for the existence of a file without opening it for read and/or write access, or to obtain other statistics about the file or directory. See Obtaining and Setting File Information and GetFileInformationByHandle.

If **CREATE_ALWAYS** and **FILE_ATTRIBUTE_NORMAL** are specified, **CreateFile** fails and sets the last error to **ERROR_ACCESS_DENIED** if the file exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_SYSTEM** attribute. To avoid the error, specify the same attributes as the existing file.

When an application creates a file across a network, it is better to use `GENERIC_READ | GENERIC_WRITE` for *dwDesiredAccess* than to use **GENERIC_WRITE** alone. The resulting code is faster, because the redirector can use the cache manager and send fewer SMBs with more data. This combination also avoids an issue where writing to a file across a network can occasionally return **ERROR_ACCESS_DENIED**.

For more information, see Creating and Opening Files.

Synchronous and Asynchronous I/O Handles

CreateFile provides for creating a file or device handle that is either synchronous or asynchronous. A synchronous handle behaves such that I/O function calls using that handle are blocked until they complete, while an asynchronous file handle makes it possible for the system to return immediately from I/O function calls, whether they completed the I/O operation or not. As stated previously, this synchronous versus asynchronous behavior is determined by specifying **FILE_FLAG_OVERLAPPED** within the *dwFlagsAndAttributes* parameter. There are several complexities and potential pitfalls when using asynchronous I/O; for more information, see Synchronous and Asynchronous I/O.

File Streams

On NTFS file systems, you can use **CreateFile** to create separate streams within a file. For more information, see File Streams.

Directories

An application cannot create a directory by using **CreateFile**, therefore only the **OPEN_EXISTING** value is valid for *dwCreationDisposition* for this use case. To create a directory, the application must call CreateDirectory or CreateDirectoryEx.

To open a directory using **CreateFile**, specify the **FILE_FLAG_BACKUP_SEMANTICS** flag as part of *dwFlagsAndAttributes*. Appropriate security checks still apply when this flag is used without **SE_BACKUP_NAME** and **SE_RESTORE_NAME** privileges.

When using **CreateFile** to open a directory during defragmentation of a FAT or FAT32 file system volume, do not specify the **MAXIMUM_ALLOWED** access right. Access to the directory is denied if this is done. Specify the **GENERIC_READ** access right instead.

For more information, see About Directory Management.

Physical Disks and Volumes

Direct access to the disk or to a volume is restricted. For more information, see "Changes to the file system and to the storage stack to restrict direct disk access and direct volume access in Windows Vista and in Windows Server 2008" in the Help and Support Knowledge Base at http://support.microsoft.com/kb/942448.

Windows Server 2003 and Windows XP: Direct access to the disk or to a volume is not restricted in this manner.

You can use the **CreateFile** function to open a physical disk drive or a volume, which returns a direct access storage device (DASD) handle that can be used with the DeviceIoControl function. This enables you to access the disk or volume directly, for example such disk metadata as the partition table. However, this type of access also exposes the disk drive or volume to potential data loss, because an incorrect write to a disk using this mechanism could make its contents inaccessible to the operating system. To ensure data integrity, be sure to become familiar with **DeviceIoControl** and how other APIs behave differently with a direct access handle as opposed to a file system handle.

The following requirements must be met for such a call to succeed:

- The caller must have administrative privileges. For more information, see Running with Special Privileges.
- The *dwCreationDisposition* parameter must have the **OPEN_EXISTING**flag.
- When opening a volume or floppy disk, the *dwShareMode* parameter must have the **FILE_SHARE_WRITE**flag.

Note The *dwDesiredAccess* parameter can be zero, allowing the application to query device attributes without accessing a device. This is useful for an application to determine the size of a floppy disk drive and the formats it supports without requiring a floppy disk in a drive, for instance. It can also be used for reading statistics without requiring higher-level data read/write permission.

When opening a physical drive *x*:, the *lpFileName* string should be the following form: "\\.\PhysicalDrive*X*". Hard disk numbers start at zero. The following table shows some examples of physical drive strings.

String	Meaning
"\\.\PhysicalDrive0"	Opens the first physical drive.
"\\.\PhysicalDrive2"	Opens the third physical drive.

To obtain the physical drive identifier for a volume, open a handle to the volume and call the **DeviceIoControl** function with **IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS**. This control code returns the disk number and offset for each of the volume's one or more extents; a volume can span multiple physical disks.

For an example of opening a physical drive, see **Calling DeviceIoControl**.

When opening a volume or removable media drive (for example, a floppy disk drive or flash memory thumb drive), the *lpFileName* string should be the following form: "\\.*X*:". Do not use a trailing backslash (\), which indicates the root directory of a drive. The following table shows some examples of drive strings.

String	Meaning
"\\.\A:"	Opens floppy disk drive A.
"\\.\C:"	Opens the C: volume.
"\\.\C:\"	Opens the file system of the C: volume.

You can also open a volume by referring to its volume name. For more information, see **Naming a Volume**.

A volume contains one or more mounted file systems. Volume handles can be opened as noncached at the discretion of the particular file system, even when the noncached option is not specified in **CreateFile**. You should assume that all Microsoft file systems open volume handles as noncached. The restrictions on noncached I/O for files also apply to volumes.

A file system may or may not require buffer alignment even though the data is noncached. However, if the noncached option is specified when opening a volume, buffer alignment is enforced regardless of the file system on the volume. It is recommended on all file systems that you open volume handles as noncached, and follow the noncached I/O restrictions.

> **Note** To read or write to the last few sectors of the volume, you must call DeviceIoControl and specify FSCTL_ALLOW_EXTENDED_DASD_IO. This signals the file system driver not to perform any I/O boundary checks on partition read or write calls. Instead, boundary checks are performed by the device driver.

Changer Device

The **IOCTL_CHANGER_*** control codes for DeviceIoControl accept a handle to a changer device. To open a changer device, use a file name of the following form: "\\.\Changer*x*" where *x* is a number that indicates which device to open, starting with zero. To open changer device zero in an application that is written in C or C++, use the following file name: "\\\\.\\Changer0".

Tape Drives

You can open tape drives by using a file name of the following form: "\\.\TAPE*x*" where *x* is a number that indicates which drive to open, starting with tape drive zero. To open tape drive zero in an application that is written in C or C++, use the following file name: "\\\\.\\TAPE0".

For more information, see Backup.

Communications Resources

The **CreateFile** function can create a handle to a communications resource, such as the serial port COM1. For communications resources, the *dwCreationDisposition* parameter must be **OPEN_EXISTING**, the *dwShareMode* parameter must be zero (exclusive access), and the *hTemplateFile* parameter must be **NULL**. Read, write, or read/write access can be specified, and the handle can be opened for overlapped I/O.

To specify a COM port number greater than 9, use the following syntax: "\\.\COM10". This syntax works for all port numbers and hardware that allows COM port numbers to be specified.

For more information about communications, see Communications.

Consoles

The **CreateFile** function can create a handle to console input (CONIN$). If the process has an open handle to it as a result of inheritance or duplication, it can also create a handle to the active screen buffer (CONOUT$). The calling process must be attached to an inherited console or one allocated by the AllocConsole function. For console handles, set the **CreateFile** parameters as follows.

Parameters	Value
lpFileName	Use the CONIN$ value to specify console input.

	Use the CONOUT$ value to specify console output. CONIN$ gets a handle to the console input buffer, even if the SetStdHandle function redirects the standard input handle. To get the standard input handle, use the GetStdHandlefunction. CONOUT$ gets a handle to the active screen buffer, even if SetStdHandleredirects the standard output handle. To get the standard output handle, use GetStdHandle.
dwDesiredAccess	`GENERIC_READ | GENERIC_WRITE` is preferred, but either one can limit access.
dwShareMode	When opening CONIN$, specify **FILE_SHARE_READ**. When opening CONOUT$, specify **FILE_SHARE_WRITE**. If the calling process inherits the console, or if a child process should be able to access the console, this parameter must be `FILE_SHARE_READ | FILE_SHARE_WRITE`.
lpSecurityAttributes	If you want the console to be inherited, the **bInheritHandle** member of the SECURITY_ATTRIBUTES structure must be **TRUE**.
dwCreationDisposition	You should specify **OPEN_EXISTING** when using **CreateFile** to open the console.
dwFlagsAndAttributes	Ignored.
hTemplateFile	Ignored.

The following table shows various settings of *dwDesiredAccess* and *lpFileName*.

lpFileName	***dwDesiredAccess***	**Result**
"CON"	**GENERIC_READ**	Opens console for input.
"CON"	**GENERIC_WRITE**	Opens console for output.

"CON"	`GENERIC_READ |`
	`GENERIC_WRITE`
		Causes **CreateFile** to fail; GetLastError returns **ERROR_FILE_NOT_FOUND**.

Mailslots

If **CreateFile**opens the client end of a mailslot, the function returns **INVALID_HANDLE_VALUE** if the mailslot client attempts to open a local mailslot before the mailslot server has created it with the CreateMailSlot function.

For more information, see Mailslots.

Pipes

If **CreateFile** opens the client end of a named pipe, the function uses any instance of the named pipe that is in the listening state. The opening process can duplicate the handle as many times as required, but after it is opened, the named pipe instance cannot be opened by another client. The access that is specified when a pipe is opened must be compatible with the access that is specified in the *dwOpenMode*parameter of the CreateNamedPipe function.

If the CreateNamedPipe function was not successfully called on the server prior to this operation, a pipe will not exist and **CreateFile** will fail with **ERROR_FILE_NOT_FOUND**.

If there is at least one active pipe instance but there are no available listener pipes on the server, which means all pipe instances are currently connected, **CreateFile** fails with **ERROR_PIPE_BUSY**.

For more information, see Pipes.

Examples

Example file operations are shown in the following topics:

- Appending One File to Another File
- Canceling Pending I/O Operations
- Creating a Child Process with Redirected Input and Output
- Creating and Using a Temporary File
- FSCTL_RECALL_FILE
- GetFinalPathNameByHandle
- Locking and Unlocking Byte Ranges in Files
- Obtaining a File Name From a File Handle
- Obtaining File System Recognition Information
- Opening a File for Reading or Writing
- Retrieving the Last-Write Time
- SetFileInformationByHandle
- Testing for the End of a File
- Using Fibers
- Using Streams
- Walking a Buffer of Change Journal Records
- Wow64DisableWow64FsRedirection
- Wow64EnableWow64FsRedirection

Physical device I/O is demonstrated in the following topics:

- Calling DeviceIoControl
- Configuring a Communications Resource
- Monitoring Communications Events
- Processing a Request to Remove a Device

An example using named pipes is located at Named Pipe Client.

Working with a mailslot is shown in Writing to a Mailslot.

A tape backup code snippet can found at Creating a Backup Application.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	FileAPI.h (include Windows.h); WinBase.h on Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, and Windows XP (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Unicode and ANSI names	**CreateFileW** (Unicode) and **CreateFileA** (ANSI)

See also

Overview Topics
About Directory Management
About Volume Management
Backup

Communications
Creating, Deleting, and Maintaining Files
Device Input and Output Control (IOCTL)
File Compression and Decompression
File Encryption
File Management Functions
File Security and Access Rights
File Streams
I/O Completion Ports
I/O Concepts
Mailslots
Obtaining and Setting File Information
Pipes
Running with Special Privileges

Functions

CloseHandle
CreateDirectory
CreateDirectoryEx
CreateFileTransacted
CreateMailSlot
CreateNamedPipe
DeleteFile
DeviceIoControl
GetLastError
ReadFile
ReadFileEx
SetFileAttributes
WriteFile
WriteFileEx