CreateFile2 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.

When called from a Windows Store app, **CreateFile2** is simplified. You can open only files or directories inside the ApplicationData.LocalFolder or Package.InstalledLocation directories. You can't open named pipes or mailslots or create encrypted files (**FILE_ATTRIBUTE_ENCRYPTED**).

Note We refer here to the app's local folder and the package's installed location, not additional packages in the package graph, like resource packages. **CreateFile2** doesn't support opening files in additional packages in the package graph. For example, suppose an app has a dependency on WinJS. The app can call **CreateFile2** to open a file in its package but not in the **WinJS** package.

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 CreateFile2(
  _In_     LPCWSTR                           lpFileName,
  _In_     DWORD                             dwDesiredAccess,
  _In_     DWORD                             dwShareMode,
  _In_     DWORD                             dwCreationDisposition,
  _In_opt_ LPCREATEFILE2_EXTENDED_PARAMETERS pCreateExParams
);
```

Parameters

lpFileName [in]

The name of the file or device to be created or opened.

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 **CreateFile2** 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. **CreateFile2** 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. Exclusive access to a file or directory is only granted if the application has write access to the file.

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. If a file or directory is being opened and this flag is not specified, and the caller does not have write access to the file or directory, 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.

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.

pCreateExParams [in, optional]

Pointer to an optional CREATEFILE2_EXTENDED_PARAMETERS structure.

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

To compile an application that uses the **CreateFile2** function, define the **_WIN32_WINNT** macro as 0x0602 or later. For more information, see Using the Windows Headers.

CreateFile2 supports file interaction and 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 **CreateFile2** 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 **CreateFile2**, 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.

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 **CreateFile2** 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.

If the **lpSecurityAttributes** member of the CREATEFILE2_EXTENDED_PARAMETERS structure passed in the *pCreateExParams* parameter is **NULL**, the handle returned by **CreateFile2** cannot be inherited by any child processes your application may create. The following information regarding this member 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)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

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** flag for the **dwFileFlags** member of the CREATEFILE2_EXTENDED_PARAMETERS structure passed in the *pCreateExParams* parameter:

- 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.

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 **CreateFile2** 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.

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.

File Streams

On NTFS file systems, you can use **CreateFile2** to create separate streams within a file. For more information, see File Streams.

Directories

An application cannot create a directory by using **CreateFile2**, 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 **CreateFile2**, specify the **FILE_FLAG_BACKUP_SEMANTICS** flag as part of **dwFileFlags** member of the CREATEFILE2_EXTENDED_PARAMETERS structure passed in the *pCreateExParams* parameter. Appropriate security checks still apply when this flag is used without **SE_BACKUP_NAME** and **SE_RESTORE_NAME** privileges.

When using **CreateFile2** 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.

You can use the **CreateFile2** 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 **CreateFile2**. 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 **CreateFile2** 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 **CreateFile2** 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 **CreateFile2** 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`.
dwCreationDisposition	You should specify **OPEN_EXISTING** when using **CreateFile2** to open the console.

Set the members of the CREATEFILE2_EXTENDED_PARAMETERS structure passed in the *pCreateExParams* parameter as follows.

Members	Value
lpSecurityAttributes	If you want the console to be inherited, the **bInheritHandle** member of the SECURITY_ATTRIBUTES structure must be **TRUE**.
dwFileAttributes **dwFileFlags** **dwSecurityQosFlags** **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 **CreateFile2** to fail; GetLastError returns **ERROR_FILE_NOT_FOUND**.

Mailslots

If **CreateFile2** 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 **CreateFile2** 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 **CreateFile2** 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, **CreateFile2** fails with **ERROR_PIPE_BUSY**.

For more information, see Pipes.

Requirements

Minimum supported client	Windows 8 [desktop apps | Windows Store apps]
Minimum supported server	Windows Server 2012 [desktop apps | Windows Store apps]
Minimum supported phone	Windows Phone 8
Header	FileAPI.h (include Windows.h)
Library	Kernel32.lib

DLL	Kernel32.dll

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
CreateFile
CreateFileTransacted
CreateMailSlot
CreateNamedPipe
DeleteFile
DeviceIoControl
GetLastError
ReadFile
ReadFileEx
SetFileAttributes
WriteFile
WriteFileEx