WriteFileEx function

Writes data to the specified file or input/output (I/O) device. It reports its completion status asynchronously, calling the specified completion routine when writing is completed or canceled and the calling thread is in an alertable wait state.

To write data to a file or device synchronously, use the **WriteFile** function.

Syntax

C++

```
BOOL WINAPI WriteFileEx(
  _In_     HANDLE                          hFile,
  _In_opt_ LPCVOID                         lpBuffer,
  _In_     DWORD                           nNumberOfBytesToWrite,
  _Inout_  LPOVERLAPPED                    lpOverlapped,
  _In_     LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

hFile [in]

A handle to the file or I/O device (for example, a file, file stream, physical disk, volume, console buffer, tape drive, socket, communications resource, mailslot, or pipe).

This parameter can be any handle opened with the **FILE_FLAG_OVERLAPPED** flag by the **CreateFile** function, or a socket handle returned by the **socket** or **accept** function.

Do not associate an I/O completion port with this handle. For more information, see the Remarks section.

This handle also must have the **GENERIC_WRITE** access right. For more information on access rights, see **File Security and Access Rights**.

lpBuffer [in, optional]

A pointer to the buffer containing the data to be written to the file or device.

This buffer must remain valid for the duration of the write operation. The caller must not use this buffer until the write operation is completed.

nNumberOfBytesToWrite [in]

The number of bytes to be written to the file or device.

A value of zero specifies a null write operation. The behavior of a null write operation depends on the underlying file system.

Pipe write operations across a network are limited to 65,535 bytes per write. For more information regarding pipes, see the Remarks section.

lpOverlapped [in, out]
A pointer to an OVERLAPPED data structure that supplies data to be used during the overlapped (asynchronous) write operation.

For files that support byte offsets, you must specify a byte offset at which to start writing to the file. You specify this offset by setting the **Offset** and **OffsetHigh** members of the OVERLAPPED structure. For files or devices that do not support byte offsets, **Offset** and **OffsetHigh** are ignored.

To write to the end of file, specify both the **Offset** and **OffsetHigh** members of the OVERLAPPED structure as 0xFFFFFFFF. This is functionally equivalent to previously calling the CreateFile function to open *hFile* using **FILE_APPEND_DATA** access.

The **WriteFileEx** function ignores the OVERLAPPED structure's **hEvent** member. An application is free to use that member for its own purposes in the context of a **WriteFileEx** call. **WriteFileEx** signals completion of its writing operation by calling, or queuing a call to, the completion routine pointed to by *lpCompletionRoutine*, so it does not need an event handle.

The **WriteFileEx** function does use the **Internal** and **InternalHigh** members of the OVERLAPPED structure. You should not change the value of these members.

The OVERLAPPED data structure must remain valid for the duration of the write operation. It should not be a variable that can go out of scope while the write operation is pending completion.

lpCompletionRoutine [in]
A pointer to a completion routine to be called when the write operation has been completed and the calling thread is in an alertable wait state. For more information about this completion routine, see FileIOCompletionRoutine.

Return value

If the function succeeds, the return value is nonzero.

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

If the **WriteFileEx** function succeeds, the calling thread has an asynchronous I/O operation pending: the overlapped write operation to the file. When this I/O operation finishes, and the calling thread is blocked in an alertable wait state, the operating system calls the function pointed to by *lpCompletionRoutine*, and the wait completes with a return code of **WAIT_IO_COMPLETION**.

If the function succeeds and the file-writing operation finishes, but the calling thread is not in an alertable wait

state, the system queues the call to **lpCompletionRoutine*, holding the call until the calling thread enters an alertable wait state. For more information about alertable wait states and overlapped input/output operations, see About Synchronization.

Remarks

When using **WriteFileEx** you should check **GetLastError** even when the function returns "success" to check for conditions that are "successes" but have some outcome you might want to know about. For example, a buffer overflow when calling **WriteFileEx** will return **TRUE**, but **GetLastError** will report the overflow with **ERROR_MORE_DATA**. If the function call is successful and there are no warning conditions, **GetLastError** will return **ERROR_SUCCESS**.

The **WriteFileEx** function will fail if the *hFile* parameter is associated with an I/O completion port. To perform writes using this type of handle, use the **WriteFile** function.

The **WriteFileEx** function may fail if there are too many outstanding asynchronous I/O requests. In the event of such a failure, **GetLastError** can return **ERROR_INVALID_USER_BUFFER** or **ERROR_NOT_ENOUGH_MEMORY**.

To cancel all pending asynchronous I/O operations, use either:

- CancelIo—this function only cancels operations issued by the calling thread for the specified file handle.
- CancelIoEx—this function cancels all operations issued by the threads for the specified file handle.

Use CancelSynchronousIo to cancel pending synchronous I/O operations.

I/O operations that are canceled complete with the error **ERROR_OPERATION_ABORTED**.

If part of the file specified by *hFile* is locked by another process, and the specified write operation overlaps the locked portion, **WriteFileEx** fails.

When writing to a file, the last write time is not fully updated until all handles used for writing have been closed. Therefore, to ensure an accurate last write time, close the file handle immediately after writing to the file.

Accessing the output buffer while a write operation is using the buffer may lead to corruption of the data written from that buffer. Applications must not write to, reallocate, or free the output buffer that a write operation is using until the write operation completes.

Note that the time stamps may not be updated correctly for a remote file. To ensure consistent results, use unbuffered I/O.

The system interprets zero bytes to write as specifying a null write operation and **WriteFile** does not truncate or extend the file. To truncate or extend a file, use the SetEndOfFile function.

An application uses the WaitForSingleObjectEx, WaitForMultipleObjectsEx, MsgWaitForMultipleObjectsEx, SignalObjectAndWait, and SleepEx functions to enter an alertable wait state. For more information about alertable wait states and overlapped I/O operations, see About Synchronization.

If you write directly to a volume that has a mounted file system, you must first obtain exclusive access to the

volume. Otherwise, you risk causing data corruption or system instability, because your application's writes may conflict with other changes coming from the file system and leave the contents of the volume in an inconsistent state. To prevent these problems, the following changes have been made in Windows Vista and later:

- A write on a volume handle will succeed if the volume does not have a mounted file system, or if one of the following conditions is true:
  - The sectors to be written to are boot sectors.
  - The sectors to be written to reside outside of file system space.
  - You have explicitly locked or dismounted the volume by using FSCTL_LOCK_VOLUME or FSCTL_DISMOUNT_VOLUME.
  - The volume has no actual file system. (In other words, it has a RAW file system mounted.)

- A write on a disk handle will succeed if one of the following conditions is true:
  - The sectors to be written to do not fall within a volume's extents.
  - The sectors to be written to fall within a mounted volume, but you have explicitly locked or dismounted the volume by using FSCTL_LOCK_VOLUME or FSCTL_DISMOUNT_VOLUME.
  - The sectors to be written to fall within a volume that has no mounted file system other than RAW.

There are strict requirements for successfully working with files opened with **CreateFile** using **FILE_FLAG_NO_BUFFERING**. For details see File Buffering.

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)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Transacted Operations

If there is a transaction bound to the file handle, then the file write is transacted. For more information, see About Transactional NTFS.

Examples

For an example, see Named Pipe Server Using Completion Routines.

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

See also

CancelIo
CancelIoEx
CancelSynchronousIo
CreateFile
File Management Functions
FileIOCompletionRoutine
MsgWaitForMultipleObjectsEx
ReadFileEx
SetEndOfFile
SetErrorMode
SleepEx
SignalObjectAndWait
WaitForMultipleObjectsEx
WaitForSingleObjectEx
WriteFile