WriteFile function

Writes data to the specified file or input/output (I/O) device.

This function is designed for both synchronous and asynchronous operation. For a similar function designed solely for asynchronous operation, see **WriteFileEx**.

Syntax

C++

```
BOOL WINAPI WriteFile(
  _In_        HANDLE       hFile,
  _In_        LPCVOID      lpBuffer,
  _In_        DWORD        nNumberOfBytesToWrite,
  _Out_opt_   LPDWORD      lpNumberOfBytesWritten,
  _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

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).

The *hFile* parameter must have been created with the write access. For more information, see **Generic Access Rights** and **File Security and Access Rights**.

For asynchronous write operations, *hFile* can be any handle opened with the **CreateFile** function using the **FILE_FLAG_OVERLAPPED** flag or a socket handle returned by the **socket** or **accept** function.

lpBuffer [in]

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 or communications technology.

Windows Server 2003 and Windows XP: Pipe write operations across a network are limited in size per write. The amount varies per platform. For x86 platforms it's 63.97 MB. For x64 platforms it's 31.97 MB. For Itanium it's 63.95 MB. For more information regarding pipes, see the Remarks section.

lpNumberOfBytesWritten [out, optional]

A pointer to the variable that receives the number of bytes written when using a synchronous *hFile* parameter. **WriteFile** sets this value to zero before doing any work or error checking. Use **NULL** for this parameter if this is an asynchronous operation to avoid potentially erroneous results.

This parameter can be **NULL** only when the *lpOverlapped* parameter is not **NULL**.

For more information, see the Remarks section.

lpOverlapped [in, out, optional]

A pointer to an OVERLAPPED structure is required if the *hFile* parameter was opened with **FILE_FLAG_OVERLAPPED**, otherwise this parameter can be **NULL**.

For an *hFile* that supports byte offsets, if you use this parameter you must specify a byte offset at which to start writing to the file or device. This offset is specified by setting the **Offset** and **OffsetHigh** members of the OVERLAPPED structure. For an *hFile* that does 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.

For more information about different combinations of *lpOverlapped* and **FILE_FLAG_OVERLAPPED**, see the Remarks section and the Synchronization and File Position section.

Return value

If the function succeeds, the return value is nonzero (**TRUE**).

If the function fails, or is completing asynchronously, the return value is zero (**FALSE**). To get extended error information, call the GetLastError function.

Note The GetLastError code **ERROR_IO_PENDING** is not a failure; it designates the write operation is pending completion asynchronously. For more information, see Remarks.

Remarks

The **WriteFile** function returns when one of the following conditions occur:

- The number of bytes requested is written.
- A read operation releases buffer space on the read end of the pipe (if the write was blocked). For more information, see the Pipes section.
- An asynchronous handle is being used and the write is occurring asynchronously.
- An error occurs.

The **WriteFile** function may fail with **ERROR_INVALID_USER_BUFFER** or **ERROR_NOT_ENOUGH_MEMORY** whenever there are too many outstanding asynchronous I/O requests.

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

- CancelIo—this function cancels only 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 the CancelSynchronousIo function to cancel pending synchronous I/O operations.

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

The **WriteFile** function may fail with **ERROR_NOT_ENOUGH_QUOTA**, which means the calling process's buffer could not be page-locked. For more information, see SetProcessWorkingSetSize.

If part of the file is locked by another process and the write operation overlaps the locked portion, **WriteFile** 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. This can be particularly problematic when using an asynchronous file handle. Additional information regarding synchronous versus asynchronous file handles can be found later in the Synchronization and File Position section and Synchronous and Asynchronous I/O.

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.

Characters can be written to the screen buffer using **WriteFile** with a handle to console output. The exact behavior of the function is determined by the console mode. The data is written to the current cursor position. The cursor position is updated after the write operation. For more information about console handles, see CreateFile.

When writing to a communications device, the behavior of **WriteFile** is determined by the current communication time-out as set and retrieved by using the SetCommTimeouts and GetCommTimeouts functions. Unpredictable results can occur if you fail to set the time-out values. For more information about communication time-outs, see COMMTIMEOUTS.

Although a single-sector write is atomic, a multi-sector write is not guaranteed to be atomic unless you are using a transaction (that is, the handle created is a transacted handle; for example, a handle created using CreateFileTransacted). Multi-sector writes that are cached may not always be written to the disk right away; therefore, specify **FILE_FLAG_WRITE_THROUGH** in **CreateFile** to ensure that an entire multi-sector write is written to the disk without potential caching delays.

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.

If *hFile* was opened with **FILE_FLAG_OVERLAPPED**, the following conditions are in effect:

- The *lpOverlapped* parameter must point to a valid and unique OVERLAPPED structure, otherwise the function can incorrectly report that the write operation is complete.
- The *lpNumberOfBytesWritten* parameter should be set to **NULL**. To get the number of bytes written, use the GetOverlappedResult function. If the *hFile* parameter is associated with an I/O completion port, you can also get the number of bytes written by calling the GetQueuedCompletionStatus function.

In 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

Synchronization and File Position

If *hFile* is opened with **FILE_FLAG_OVERLAPPED**, it is an asynchronous file handle; otherwise it is synchronous. The rules for using the OVERLAPPED structure are slightly different for each, as previously noted.

> **Note** If a file or device is opened for asynchronous I/O, subsequent calls to functions such as **WriteFile** using that handle generally return immediately, but can also behave synchronously with respect to blocked execution. For more information, see http://support.microsoft.com/kb/156932.

Considerations for working with asynchronous file handles:

- **WriteFile** may return before the write operation is complete. In this scenario, **WriteFile** returns **FALSE** and the GetLastError function returns **ERROR_IO_PENDING**, which allows the calling process to continue while the system completes the write operation.
- The *lpOverlapped* parameter must not be **NULL** and should be used with the following facts in mind:
  - Although the event specified in the OVERLAPPED structure is set and reset automatically by the system, the offset that is specified in the **OVERLAPPED** structure is not automatically updated.
  - **WriteFile** resets the event to a nonsignaled state when it begins the I/O operation.
  - The event specified in the OVERLAPPED structure is set to a signaled state when the write operation is complete; until that time, the write operation is considered pending.
  - Because the write operation starts at the offset that is specified in the OVERLAPPED structure, and **WriteFile** may return before the system-level write operation is complete (write pending), neither the offset nor any other part of the structure should be modified, freed, or reused by the application until the event is signaled (that is, the write completes).

Considerations for working with synchronous file handles:

- If *lpOverlapped* is **NULL**, the write operation starts at the current file position and **WriteFile** does not return until the operation is complete, and the system updates the file pointer before **WriteFile** returns.
- If *lpOverlapped* is not **NULL**, the write operation starts at the offset that is specified in the OVERLAPPED structure and **WriteFile** does not return until the write operation is complete. The system updates the **OVERLAPPED** offset before **WriteFile** returns.

For more information, see CreateFile and Synchronous and Asynchronous I/O.

Pipes

If an anonymous pipe is being used and the read handle has been closed, when **WriteFile** attempts to write using the pipe's corresponding write handle, the function returns **FALSE** and GetLastError returns **ERROR_BROKEN_PIPE**.

If the pipe buffer is full when an application uses the **WriteFile** function to write to a pipe, the write operation may not finish immediately. The write operation will be completed when a read operation (using the ReadFile function) makes more system buffer space available for the pipe.

When writing to a non-blocking, byte-mode pipe handle with insufficient buffer space, **WriteFile** returns **TRUE** with **lpNumberOfBytesWritten* < *nNumberOfBytesToWrite*.

For more information about pipes, see Pipes.

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 some examples, see Creating and Using a Temporary File and Opening a File for Reading or Writing.

The following C++ example shows how to align sectors for unbuffered file writes. The *Size* variable is the size of the original data block you are interested in writing to the file. For additional rules regarding unbuffered file I/O, see File Buffering.

C++

```
#include <windows.h>

#define ROUND_UP_SIZE(Value,Pow2) ((SIZE_T) ((((ULONG)(Value)) + (Pow2) - 1) & (~
(((LONG)(Pow2)) - 1))))

#define ROUND_UP_PTR(Ptr,Pow2)  ((void *) ((((ULONG_PTR)(Ptr)) + (Pow2) - 1) & (~
(((LONG_PTR)(Pow2)) - 1))))


void main()
{
// Function code

    DWORD BytesPerSector = 0; // obtained from the GetFreeDiskSpace function.
    DWORD Size = 0; // buffer size of your data to write

// ... obtain data here
// sample data
    BytesPerSector = 65536;
```

```
     Size = 15536;
  //

     // Ensure you have one more sector than Size would require.
     SIZE_T SizeNeeded = BytesPerSector + ROUND_UP_SIZE(Size, BytesPerSector);

     // Replace this statement with any allocation routine.
     LPBYTE Buffer = (LPBYTE) malloc(SizeNeeded);

     // Error checking of your choice.
     if ( !Buffer )
     {
       goto cleanup;
     }

     // Actual alignment happens here.
     void * BufferAligned = ROUND_UP_PTR(Buffer, BytesPerSector);

     // Add code using BufferAligned here.


  cleanup:

     if ( Buffer )
     {
        // Replace with corresponding free routine.
        free(Buffer);
     }

  }
```

Requirements

Minimum supported client	Windows XP [desktop apps | Windows Store apps]
Minimum supported server	Windows Server 2003 [desktop apps | Windows Store apps]
Minimum	Windows Phone 8

supported phone	
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
CreateFileTransacted
File Management Functions
GetLastError
GetOverlappedResult
GetQueuedCompletionStatus
ReadFile
SetEndOfFile
WriteFileEx