ReadFile function

Reads data from the specified file or input/output (I/O) device. Reads occur at the position specified by the file pointer if supported by the device.

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

Syntax

C++

```
BOOL WINAPI ReadFile(
  _In_        HANDLE       hFile,
  _Out_       LPVOID       lpBuffer,
  _In_        DWORD        nNumberOfBytesToRead,
  _Out_opt_   LPDWORD      lpNumberOfBytesRead,
  _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

Parameters

hFile [in]

A handle to the 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 read access. For more information, see **Generic Access Rights** and **File Security and Access Rights**.

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

lpBuffer [out]

A pointer to the buffer that receives the data read from a file or device.

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

nNumberOfBytesToRead [in]

The maximum number of bytes to be read.

lpNumberOfBytesRead [out, optional]

A pointer to the variable that receives the number of bytes read when using a synchronous *hFile* parameter. **ReadFile** 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 it can be **NULL**.

If *hFile* is opened with **FILE_FLAG_OVERLAPPED**, the *lpOverlapped* parameter must point to a valid and unique OVERLAPPED structure, otherwise the function can incorrectly report that the read operation is complete.

For an *hFile* that supports byte offsets, if you use this parameter you must specify a byte offset at which to start reading from 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.

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 read operation is pending completion asynchronously. For more information, see Remarks.

Remarks

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

- The number of bytes requested is read.
- A write operation completes on the write end of the pipe.
- An asynchronous handle is being used and the read is occurring asynchronously.
- An error occurs.

The **ReadFile** 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 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**.

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

If part of a file is locked by another process and the read operation overlaps the locked portion, this function fails.

Accessing the input buffer while a read operation is using the buffer may lead to corruption of the data read into that buffer. Applications must not read from, write to, reallocate, or free the input buffer that a read operation is using until the read operation completes. This can be particularly problematic when using an asynchronous file handle. Additional information regarding synchronous versus asynchronous file handles can be found in the Synchronization and File Position section and in the **CreateFile** reference topic.

Characters can be read from the console input buffer by using **ReadFile** with a handle to console input. The console mode determines the exact behavior of the **ReadFile** function. By default, the console mode is **ENABLE_LINE_INPUT**, which indicates that **ReadFile** should read until it reaches a carriage return. If you press Ctrl+C, the call succeeds, but **GetLastError** returns **ERROR_OPERATION_ABORTED**. For more information, see **CreateFile**.

When reading from a communications device, the behavior of **ReadFile** 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.

If **ReadFile** attempts to read from a mailslot that has a buffer that is too small, the function returns **FALSE** and **GetLastError** returns **ERROR_INSUFFICIENT_BUFFER**.

There are strict requirements for successfully working with files opened with **CreateFile** using the **FILE_FLAG_NO_BUFFERING** flag. 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 read operation is complete.
- The *lpNumberOfBytesRead* parameter should be set to **NULL**. Use the GetOverlappedResult function to get the actual number of bytes read. If the *hFile* parameter is associated with an I/O completion port, you can also get the number of bytes read by calling the **GetQueuedCompletionStatus** function.

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 **ReadFile** 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:

- **ReadFile** may return before the read operation is complete. In this scenario, **ReadFile** returns **FALSE** and the GetLastError function returns **ERROR_IO_PENDING**, which allows the calling process to continue while the system completes the read 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.
  - **ReadFile** 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 read operation is complete; until that time, the read operation is considered pending.
  - Because the read operation starts at the offset that is specified in the OVERLAPPED structure, and **ReadFile** may return before the system-level read operation is complete (read 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 read completes).
  - If end-of-file (EOF) is detected during asynchronous operations, the call to GetOverlappedResult for that operation returns **FALSE** and GetLastError returns **ERROR_HANDLE_EOF**.

Considerations for working with synchronous file handles:

- If *lpOverlapped* is **NULL**, the read operation starts at the current file position and **ReadFile** does not return until the operation is complete, and the system updates the file pointer before **ReadFile** returns.
- If *lpOverlapped* is not **NULL**, the read operation starts at the offset that is specified in the OVERLAPPED structure and **ReadFile** does not return until the read operation is complete. The system updates the **OVERLAPPED** offset before **ReadFile** returns.
- When a synchronous read operation reaches the end of a file, **ReadFile** returns **TRUE** and sets `*lpNumberOfBytesRead` to zero.

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

Pipes

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

If a named pipe is being read in message mode and the next message is longer than the *nNumberOfBytesToRead* parameter specifies, **ReadFile** returns **FALSE** and GetLastError returns **ERROR_MORE_DATA**. The remainder of the message can be read by a subsequent call to the **ReadFile** or PeekNamedPipefunction.

If the *lpNumberOfBytesRead* parameter is zero when **ReadFile** returns **TRUE** on a pipe, the other end of the pipe called the **WriteFile** function with *nNumberOfBytesToWrite* set to zero.

For more information about pipes, see Pipes.

Transacted Operations

If there is a transaction bound to the file handle, then the function returns data from the transacted view of the file. A transacted read handle is guaranteed to show the same view of a file for the duration of the handle. For more information, see About Transactional NTFS.

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

Examples

For a code example that shows you how to test for end-of-file, see Testing for the End of a File. For other examples, see Creating and Using a Temporary File and Opening a File for Reading or Writing.

Requirements

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

server	
Minimum supported phone	Windows Phone 8
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
GetCommTimeouts
GetOverlappedResult
GetQueuedCompletionStatus
OVERLAPPED
PeekNamedPipe
ReadFileEx
SetCommTimeouts
SetErrorMode
WriteFile