# CreatePipe function (namedpipeapi.h)

12/05/2018 • 2 minutes to read

**In this article**



Creates an anonymous pipe, and returns handles to the read and write ends of the pipe.

## Syntax

C++ Copy

```
BOOL CreatePipe(
  PHANDLE                hReadPipe,
  PHANDLE                hWritePipe,
  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  DWORD                  nSize
);
```

## Parameters

`hReadPipe`

A pointer to a variable that receives the read handle for the pipe.

`hWritePipe`

A pointer to a variable that receives the write handle for the pipe.

`lpPipeAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpPipeAttributes* is **NULL**, the handle cannot

be inherited.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new pipe. If *lpPipeAttributes* is **NULL**, the pipe gets a default security descriptor. The ACLs in the default security descriptor for a pipe come from the primary or impersonation token of the creator.

`nSize`

The size of the buffer for the pipe, in bytes. The size is only a suggestion; the system uses the value to calculate an appropriate buffering mechanism. If this parameter is zero, the system uses the default buffer size.

# 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.

# Remarks

**CreatePipe** creates the pipe, assigning the specified pipe size to the storage buffer. **CreatePipe** also creates handles that the process uses to read from and write to the buffer in subsequent calls to the ReadFile and WriteFile functions.

To read from the pipe, a process uses the read handle in a call to the ReadFile function. **ReadFile** returns when one of the following is true: a write operation completes on the write end of the pipe, the number of bytes requested has been read, or an error occurs.

When a process uses WriteFile to write to an anonymous pipe, the write operation is not completed until all bytes are written. If the pipe buffer is full before all bytes are written, **WriteFile** does not return until another process or thread uses ReadFile to make more buffer space available.

Anonymous pipes are implemented using a named pipe with a unique name. Therefore, you can often pass a handle to an anonymous pipe to a function that requires a handle to a named pipe.

If **CreatePipe** fails, the contents of the output parameters are indeterminate. No assumptions should be made about their contents in this event.

To free resources used by a pipe, the application should always close handles when they are no longer needed, which is accomplished either by calling the CloseHandle function or when the process associated with the instance handles ends. Note that an instance of a pipe may have more than one handle associated with it. An instance of a pipe is always deleted when the last handle to the instance of the named pipe is closed.

## Examples

For an example, see Creating a Child Process with Redirected Input and Output.

# Requirements

| | |
|---|---|
| **Minimum supported client** | Windows 2000 Professional [desktop apps \| UWP apps] |
| **Minimum supported server** | Windows 2000 Server [desktop apps \| UWP apps] |
| **Target Platform** | Windows |
| **Header** | namedpipeapi.h |
| **Library** | Kernel32.lib |
| **DLL** | Kernel32.dll |

# See also

Pipe Functions

Pipes Overview

ReadFile

SECURITY_ATTRIBUTES

WriteFile

**Is this page helpful?**

# CreateFileMappingA function (winbase.h)

12/05/2018 • 11 minutes to read

**In this article**



Creates or opens a named or unnamed file mapping object for a specified file.

To specify the NUMA node for the physical memory, see CreateFileMappingNuma.

## Syntax

C++ Copy

```
HANDLE CreateFileMappingA(
  HANDLE                hFile,
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
  DWORD                 flProtect,
  DWORD                 dwMaximumSizeHigh,
  DWORD                 dwMaximumSizeLow,
  LPCSTR                lpName
);
```

## Parameters

`hFile`

A handle to the file from which to create a file mapping object.

The file must be opened with access rights that are compatible with the protection flags that the *flProtect* parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access. For more information, see File Security and Access Rights.

If *hFile* is **INVALID_HANDLE_VALUE**, the calling process must also specify a size for the file mapping object in the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters. In this scenario, **CreateFileMapping** creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

`lpFileMappingAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that determines whether a returned handle can be inherited by child processes. The **lpSecurityDescriptor** member of the **SECURITY_ATTRIBUTES** structure specifies a security descriptor for a new file mapping object.

If *lpFileMappingAttributes* is **NULL**, the handle cannot be inherited and the file mapping object gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a file mapping object come from the primary or impersonation token of the creator. For more information, see File Mapping Security and Access Rights.

`flProtect`

Specifies the page protection of the file mapping object. All mapped views of the object must be compatible with this protection.

This parameter can be one of the following values.

| Value | Meaning |
| --- | --- |
| **PAGE_EXECUTE_READ**<br>0x20 | Allows views to be mapped for read-only, copy-on-write, or execute access.<br>The file handle specified by the *hFile* parameter must be created with the **GENERIC_READ** and **GENERIC_EXECUTE** access rights.<br><br>**Windows Server 2003 and Windows XP:** This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1. |
| **PAGE_EXECUTE_READWRITE**<br>0x40 | Allows views to be mapped for read-only, copy-on-write, read/write, or execute access.<br>The file handle that the *hFile* parameter specifies must be created with the **GENERIC_READ**, **GENERIC_WRITE**, and **GENERIC_EXECUTE** access rights.<br><br>**Windows Server 2003 and Windows XP:** This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1. |

| | |
|---|---|
| **PAGE_EXECUTE_WRITECOPY**<br>0x80 | Allows views to be mapped for read-only, copy-on-write, or execute access. This value is equivalent to **PAGE_EXECUTE_READ**.<br><br>The file handle that the *hFile* parameter specifies must be created with the **GENERIC_READ** and **GENERIC_EXECUTE** access rights.<br><br>**Windows Vista:** This value is not available until Windows Vista with SP1.<br><br>**Windows Server 2003 and Windows XP:** This value is not supported. |
| **PAGE_READONLY**<br>0x02 | Allows views to be mapped for read-only or copy-on-write access. An attempt to write to a specific region results in an access violation.<br>The file handle that the *hFile* parameter specifies must be created with the **GENERIC_READ** access right. |
| **PAGE_READWRITE**<br>0x04 | Allows views to be mapped for read-only, copy-on-write, or read/write access.<br>The file handle that the *hFile* parameter specifies must be created with the **GENERIC_READ** and **GENERIC_WRITE** access rights. |
| **PAGE_WRITECOPY**<br>0x08 | Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to **PAGE_READONLY**.<br><br>The file handle that the *hFile* parameter specifies must be created with the **GENERIC_READ** access right. |

An application can specify one or more of the following attributes for the file mapping object by combining them with one of the preceding page protection values.

| Value | Meaning |
|---|---|
| **SEC_COMMIT**<br>0x8000000 | If the file mapping object is backed by the operating system paging file (the *hfile* parameter is **INVALID_HANDLE_VALUE**), specifies that when a view of the file is mapped into a process address space, the entire range of pages is committed rather than reserved. The system must have enough committable pages to |

| | |
|---|---|
| | hold the entire mapping. Otherwise, **CreateFileMapping** fails.<br><br>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the *hfile* parameter is a handle to a file).<br><br>**SEC_COMMIT** cannot be combined with **SEC_RESERVE**.<br><br>If no attribute is specified, **SEC_COMMIT** is assumed. |
| **SEC_IMAGE**<br>0x1000000 | Specifies that the file that the *hFile* parameter specifies is an executable image file.<br><br>The **SEC_IMAGE** attribute must be combined with a page protection value such as **PAGE_READONLY**. However, this page protection value has no effect on views of the executable image file. Page protection for views of an executable image file is determined by the executable file itself.<br><br>No other attributes are valid with **SEC_IMAGE**. |
| **SEC_IMAGE_NO_EXECUTE**<br>0x11000000 | Specifies that the file that the *hFile* parameter specifies is an executable image file that will not be executed and the loaded image file will have no forced integrity checks run. Additionally, mapping a view of a file mapping object created with the **SEC_IMAGE_NO_EXECUTE** attribute will not invoke driver callbacks registered using the PsSetLoadImageNotifyRoutine kernel API.<br><br>The **SEC_IMAGE_NO_EXECUTE** attribute must be combined with the **PAGE_READONLY** page protection value. No other attributes are valid with **SEC_IMAGE_NO_EXECUTE**.<br><br>**Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This value is not supported before Windows Server 2012 and Windows 8. |
| **SEC_LARGE_PAGES**<br>0x80000000 | Enables large pages to be used for file mapping objects that are backed by the operating system paging file (the *hfile* parameter is **INVALID_HANDLE_VALUE**). This attribute is not supported for file mapping objects that are backed by executable image files or data files (the |

| | |
|---|---|
| | *hFile* parameter is a handle to an executable image or data file).<br><br>The maximum size of the file mapping object must be a multiple of the minimum size of a large page returned by the GetLargePageMinimum function. If it is not, **CreateFileMapping** fails. When mapping a view of a file mapping object created with **SEC_LARGE_PAGES**, the base address and view size must also be multiples of the minimum large page size.<br><br>**SEC_LARGE_PAGES** requires the SeLockMemoryPrivilege privilege to be enabled in the caller's token.<br><br>If **SEC_LARGE_PAGES** is specified, **SEC_COMMIT** must also be specified.<br><br>**Windows Server 2003:** This value is not supported until Windows Server 2003 with SP1.<br><br>**Windows XP:** This value is not supported. |
| **SEC_NOCACHE**<br>0x10000000 | Sets all pages to be non-cachable.<br>Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with **SEC_NOCACHE** can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception.<br><br>**SEC_NOCACHE** requires either the **SEC_RESERVE** or **SEC_COMMIT** attribute to be set. |
| **SEC_RESERVE**<br>0x4000000 | If the file mapping object is backed by the operating system paging file (the *hfile* parameter is **INVALID_HANDLE_VALUE**), specifies that when a view of the file is mapped into a process address space, the entire range of pages is reserved for later use by the process rather than committed.<br><br>Reserved pages can be committed in subsequent calls to the VirtualAlloc function. After the pages are committed, they cannot be freed or decommitted with the VirtualFree function.<br><br>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the *hfile* parameter is a handle to a file). |

| | |
|---|---|
| | **SEC_RESERVE** cannot be combined with **SEC_COMMIT**. |
| **SEC_WRITECOMBINE**<br>0x40000000 | Sets all pages to be write-combined.<br>Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with **SEC_WRITECOMBINE** can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception.<br><br>**SEC_WRITECOMBINE** requires either the **SEC_RESERVE** or **SEC_COMMIT** attribute to be set.<br><br>**Windows Server 2003 and Windows XP:** This flag is not supported until Windows Vista. |

`dwMaximumSizeHigh`

The high-order **DWORD** of the maximum size of the file mapping object.

`dwMaximumSizeLow`

The low-order **DWORD** of the maximum size of the file mapping object.

If this parameter and *dwMaximumSizeHigh* are 0 (zero), the maximum size of the file mapping object is equal to the current size of the file that *hFile* identifies.

An attempt to map a file with a length of 0 (zero) fails with an error code of **ERROR_FILE_INVALID**. Applications should test for files with a length of 0 (zero) and reject those files.

`lpName`

The name of the file mapping object.

If this parameter matches the name of an existing mapping object, the function requests access to the object with the protection that *flProtect* specifies.

If this parameter is **NULL**, the file mapping object is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or job object, the function fails, and the GetLastError function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the

backslash character (). Creating a file mapping object in the global namespace from a session other than session zero requires the SeCreateGlobalPrivilege privilege. For more information, see Kernel Object Namespaces.

Fast user switching is implemented by using Terminal Services sessions. The first user to log on uses session 0 (zero), the next user to log on uses session 1 (one), and so on. Kernel object names must follow the guidelines that are outlined for Terminal Services so that applications can support multiple users.

# Return value

If the function succeeds, the return value is a handle to the newly created file mapping object.

If the object exists before the function call, the function returns a handle to the existing object (with its current size, not the specified size), and GetLastErrorreturns **ERROR_ALREADY_EXISTS**.

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

# Remarks

After a file mapping object is created, the size of the file must not exceed the size of the file mapping object; if it does, not all of the file contents are available for sharing.

If an application specifies a size for the file mapping object that is larger than the size of the actual named file on disk and if the page protection allows write access (that is, the *flProtect* parameter specifies **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE**), then the file on disk is increased to match the specified size of the file mapping object. If the file is extended, the contents of the file between the old end of the file and the new end of the file are not guaranteed to be zero; the behavior is defined by the file system. If the file on disk cannot be increased, **CreateFileMapping** fails and GetLastError returns **ERROR_DISK_FULL**.

The initial contents of the pages in a file mapping object backed by the operating system paging file are 0 (zero).

The handle that **CreateFileMapping** returns has full access to a new file mapping object, and can be used with any function that requires a handle to a file mapping object.

Multiple processes can share a view of the same file by either using a single shared file mapping object or creating separate file mapping objects backed by the same file. A single file mapping object can be shared by multiple processes through inheriting the handle at process creation, duplicating the handle, or opening the file mapping object by name. For more information, see the CreateProcess, DuplicateHandle and OpenFileMapping functions.

Creating a file mapping object does not actually map the view into a process address space. The MapViewOfFile and MapViewOfFileEx functions map a view of a file into a process address space.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **CreateFileMapping** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

A mapped file and a file that is accessed by using the input and output (I/O) functions (ReadFile and WriteFile) are not necessarily coherent.

Mapped views of a file mapping object maintain internal references to the object, and a file mapping object does not close until all references to it are released. Therefore, to fully close a file mapping object, an application must unmap all mapped views of the file mapping object by calling UnmapViewOfFile and close the file mapping object handle by calling CloseHandle. These functions can be called in any order.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use SetFileTime to set the timestamp.

Creating a file mapping object in the global namespace from a session other than session zero requires the SeCreateGlobalPrivilege privilege. Note that this privilege check is limited to the creation of file mapping objects and does not apply to opening existing ones. For example, if a service or the system creates a file mapping object in the global namespace, any process running in any session can access that file mapping object provided that the caller has the required access rights.

**Windows XP:** The requirement described in the previous paragraph was introduced with Windows Server 2003 and Windows XP with SP2

Use structured exception handling to protect any code that writes to or reads from a file view. For more information, see Reading and Writing From a File View.

To have a mapping with executable permissions, an application must call **CreateFileMapping** with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ**, and then call MapViewOfFile with `FILE_MAP_EXECUTE | FILE_MAP_WRITE` or `FILE_MAP_EXECUTE | FILE_MAP_READ`.

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 |

## Examples

For an example, see Creating Named Shared Memory or Creating a File Mapping Using Large Pages.

# Requirements

| | |
|---|---|
| **Minimum supported client** | Windows XP [desktop apps only] |
| **Minimum supported server** | Windows Server 2003 [desktop apps only] |
| **Target Platform** | Windows |

| | |
|---|---|
| **Header** | winbase.h (include Windows.h, Memoryapi.h) |
| **Library** | Kernel32.lib |
| **DLL** | Kernel32.dll |

# See also

CloseHandle

CreateFileMappingNuma

Creating a File Mapping Object

DuplicateHandle

File Mapping Functions

MapViewOfFile

MapViewOfFileEx

Memory Management Functions

OpenFileMapping

ReadFile

SECURITY_ATTRIBUTES

UnmapViewOfFile

VirtualAlloc

WriteFile

# OpenFileMappingA function (winbase.h)

12/05/2018 • 2 minutes to read

**In this article**



Opens a named file mapping object.

## Syntax

C++ Copy

```
HANDLE OpenFileMappingA(
  DWORD  dwDesiredAccess,
  BOOL   bInheritHandle,
  LPCSTR lpName
);
```

## Parameters

`dwDesiredAccess`

The access to the file mapping object. This access is checked against any security descriptor on the target file mapping object. For a list of values, see File Mapping Security and Access Rights.

`bInheritHandle`

If this parameter is **TRUE**, a process created by the CreateProcess function can inherit the handle; otherwise, the handle cannot be inherited.

`lpName`

The name of the file mapping object to be opened. If there is an open handle to a file mapping object by this name and the security descriptor on the mapping object does not conflict with the *dwDesiredAccess* parameter, the open operation succeeds. The name can have a "Global\" or "Local\" prefix to explicitly open an object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Namespaces. Fast user switching is implemented using Terminal Services sessions. The first user to log on uses session 0, the next user to log on uses session 1, and so on. Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users.

# Return value

If the function succeeds, the return value is an open handle to the specified file mapping object.

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

# Remarks

The handle that **OpenFileMapping** returns can be used with any function that requires a handle to a file mapping object.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use SetFileTime to set the timestamp.

When it is no longer needed, the caller should call release the handle returned by **OpenFileMapping** with a call to CloseHandle.

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 |

## Examples

For an example, see Creating Named Shared Memory.

# Requirements

| | |
|---|---|
| **Minimum supported client** | Windows XP [desktop apps only] |
| **Minimum supported server** | Windows Server 2003 [desktop apps only] |
| **Target Platform** | Windows |
| **Header** | winbase.h (include Windows.h, Memoryapi.h) |
| **Library** | Kernel32.lib |
| **DLL** | Kernel32.dll |

# See also

CreateFileMapping

File Mapping Functions

Memory Management Functions

Sharing Files and Memory

# MapViewOfFile function (memoryapi.h)

12/05/2018 • 7 minutes to read

**In this article**



Maps a view of a file mapping into the address space of a calling process.

To specify a suggested base address for the view, use the MapViewOfFileEx function. However, this practice is not recommended.

## Syntax

C++ Copy

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD  dwDesiredAccess,
  DWORD  dwFileOffsetHigh,
  DWORD  dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap
);
```

## Parameters

`hFileMappingObject`

A handle to a file mapping object. The CreateFileMapping and OpenFileMapping functions return this handle.

`dwDesiredAccess`

The type of access to a file mapping object, which determines the page protection of the pages. This parameter can be one of the following values, or a bitwise OR combination of multiple values where appropriate.

| Value | Meaning |
|---|---|
| **FILE_MAP_ALL_ACCESS** | A read/write view of the file is mapped. The file mapping object must have been created with **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE** protection.<br><br>When used with the **MapViewOfFile** function, **FILE_MAP_ALL_ACCESS** is equivalent to **FILE_MAP_WRITE**. |
| **FILE_MAP_READ** | A read-only view of the file is mapped. An attempt to write to the file view results in an access violation. The file mapping object must have been created with **PAGE_READONLY**, **PAGE_READWRITE**, **PAGE_EXECUTE_READ**, or **PAGE_EXECUTE_READWRITE** protection. |
| **FILE_MAP_WRITE** | A read/write view of the file is mapped. The file mapping object must have been created with **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE** protection.<br><br>When used with **MapViewOfFile**, (**FILE_MAP_WRITE \| FILE_MAP_READ**) and **FILE_MAP_ALL_ACCESS** are equivalent to **FILE_MAP_WRITE**. |

Using bitwise OR, you can combine the values above with these values.

| Value | Meaning |
|---|---|
| **FILE_MAP_COPY** | A copy-on-write view of the file is mapped. The file mapping object must have been created with **PAGE_READONLY**, **PAGE_READ_EXECUTE**, **PAGE_WRITECOPY**, **PAGE_EXECUTE_WRITECOPY**, **PAGE_READWRITE**, or **PAGE_EXECUTE_READWRITE** protection.<br><br>When a process writes to a copy-on-write page, the system copies the original page to a new page that is private to the process. The new page is backed by the paging file. The protection of the new page changes from copy-on-write to read/write. |

| | |
|---|---|
| | When copy-on-write access is specified, the system and process commit charge taken is for the entire view because the calling process can potentially write to every page in the view, making all pages private. The contents of the new page are never written back to the original file and are lost when the view is unmapped. |
| **FILE_MAP_EXECUTE** | An executable view of the file is mapped (mapped memory can be run as code). The file mapping object must have been created with **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_WRITECOPY**, or **PAGE_EXECUTE_READWRITE** protection.<br><br>**Windows Server 2003 and Windows XP:** This value is available starting with Windows XP with SP2 and Windows Server 2003 with SP1. |
| **FILE_MAP_LARGE_PAGES** | Starting with Windows 10, version 1703, this flag specifies that the view should be mapped using large page support. The size of the view must be a multiple of the size of a large page reported by the GetLargePageMinimum function, and the file-mapping object must have been created using the **SEC_LARGE_PAGES** option. If you provide a non-null value for *lpBaseAddress*, then the value must be a multiple of **GetLargePageMinimum**.<br><br>**Note:** On OS versions before Windows 10, version 1703, the **FILE_MAP_LARGE_PAGES** flag has no effect. On these releases, the view is automatically mapped using large pages if the section was created with the **SEC_LARGE_PAGES** flag set. |
| **FILE_MAP_TARGETS_INVALID** | Sets all the locations in the mapped file as invalid targets for Control Flow Guard (CFG). This flag is similar to **PAGE_TARGETS_INVALID**. Use this flag in combination with the execute access right **FILE_MAP_EXECUTE**. Any indirect call to locations in those pages will fail CFG checks, and the process will be terminated. The default behavior for executable pages allocated is to be marked valid call targets for CFG. |

For file mapping objects created with the **SEC_IMAGE** attribute, the *dwDesiredAccess* parameter has no effect, and should be set to any valid value such as **FILE_MAP_READ**.

For more information about access to file mapping objects, see File Mapping Security and Access Rights.

`dwFileOffsetHigh`

A high-order **DWORD** of the file offset where the view begins.

`dwFileOffsetLow`

A low-order **DWORD** of the file offset where the view is to begin. The combination of the high and low offsets must specify an offset within the file mapping. They must also match the memory allocation granularity of the system. That is, the offset must be a multiple of the allocation granularity. To obtain the memory allocation granularity of the system, use the GetSystemInfo function, which fills in the members of a SYSTEM_INFO structure.

`dwNumberOfBytesToMap`

The number of bytes of a file mapping to map to the view. All bytes must be within the maximum size specified by CreateFileMapping. If this parameter is 0 (zero), the mapping extends from the specified offset to the end of the file mapping.

# Return value

If the function succeeds, the return value is the starting address of the mapped view.

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

# Remarks

Mapping a file makes the specified portion of a file visible in the address space of the calling process.

For files that are larger than the address space, you can only map a small portion of the file data at one time. When the first view is complete, you can unmap it and map a new view.

To obtain the size of a view, use the VirtualQuery function.

Multiple views of a file (or a file mapping object and its mapped file) are *coherent* if they contain identical data at a specified time. This occurs if the file views are derived from any file mapping object that is backed by the same file. A process can duplicate a file mapping

object handle into another process by using the DuplicateHandle function, or another process can open a file mapping object by name by using the OpenFileMapping function.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **MapViewOfFile** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

A mapped view of a file is not guaranteed to be coherent with a file that is being accessed by the ReadFile or WriteFile function.

Do not store pointers in the memory mapped file; store offsets from the base of the file mapping so that the mapping can be used at any address.

To guard against **EXCEPTION_IN_PAGE_ERROR** exceptions, use structured exception handling to protect any code that writes to or reads from a memory mapped view of a file other than the page file. For more information, see Reading and Writing From a File View.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use SetFileTime to set the timestamp.

If a file mapping object is backed by the paging file (CreateFileMapping is called with the *hFile* parameter set to **INVALID_HANDLE_VALUE**), the paging file must be large enough to hold the entire mapping. If it is not, **MapViewOfFile** fails. The initial contents of the pages in a file mapping object backed by the paging file are 0 (zero).

When a file mapping object that is backed by the paging file is created, the caller can specify whether **MapViewOfFile** should reserve and commit pages at the same time (**SEC_COMMIT**) or simply reserve pages (**SEC_RESERVE**). Mapping the file makes the entire mapped virtual address range unavailable to other allocations in the process. After a page from the reserved range is committed, it cannot be freed or decommitted by calling VirtualFree. Reserved and committed pages are released when the view is unmapped and the file mapping object is closed. For details, see the UnmapViewOfFile and CloseHandle functions.

To have a file with executable permissions, an application must call CreateFileMapping with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ**, and then call **MapViewOfFile** with **FILE_MAP_EXECUTE | FILE_MAP_WRITE** or **FILE_MAP_EXECUTE | FILE_MAP_READ**.

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 |

When CsvFs is paused this call might fail with an error indicating that there is a lock conflict.

## Examples

For an example, see Creating Named Shared Memory.

# Requirements

| | |
|---|---|
| **Minimum supported client** | Windows XP [desktop apps only] |
| **Minimum supported server** | Windows Server 2003 [desktop apps only] |
| **Target Platform** | Windows |
| **Header** | memoryapi.h (include Windows.h, Memoryapi.h) |
| **Library** | Kernel32.lib |

| | |
|---|---|
| DLL | Kernel32.dll |

## See also

CreateFileMapping

Creating a File View

DuplicateHandle

GetSystemInfo

MapViewOfFileEx

Memory Management Functions

OpenFileMapping

SYSTEM_INFO

UnmapViewOfFile

# UnmapViewOfFile function (memoryapi.h)

12/05/2018 • 2 minutes to read

**In this article**



Unmaps a mapped view of a file from the calling process's address space.

## Syntax

C++ Copy

```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress
);
```

## Parameters

`lpBaseAddress`

A pointer to the base address of the mapped view of a file that is to be unmapped. This value must be identical to the value returned by a previous call to the MapViewOfFile or MapViewOfFileEx function.

## 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.

# Remarks

Unmapping a mapped view of a file invalidates the range occupied by the view in the address space of the process and makes the range available for other allocations. It removes the working set entry for each unmapped virtual page that was part of the working set of the process and reduces the working set size of the process. It also decrements the share count of the corresponding physical page.

Modified pages in the unmapped view are not written to disk until their share count reaches zero, or in other words, until they are unmapped or trimmed from the working sets of all processes that share the pages. Even then, the modified pages are written "lazily" to disk; that is, modifications may be cached in memory and written to disk at a later time. To minimize the risk of data loss in the event of a power failure or a system crash, applications should explicitly flush modified pages using the FlushViewOfFile function.

Although an application may close the file handle used to create a file mapping object, the system holds the corresponding file open until the last view of the file is unmapped. Files for which the last view has not yet been unmapped are held open with no sharing restrictions.

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 |

## Examples

For an example, see Creating a View Within a File.

# Requirements

| | |
|---|---|
| **Minimum supported client** | Windows XP [desktop apps \| UWP apps] |
| **Minimum supported server** | Windows Server 2003 [desktop apps \| UWP apps] |
| **Target Platform** | Windows |
| **Header** | memoryapi.h (include Windows.h, Memoryapi.h) |
| **Library** | Kernel32.lib |
| **DLL** | Kernel32.dll |

# See also

Closing a File Mapping Object

File Mapping Functions

MapViewOfFile

MapViewOfFileEx

Memory Management Functions