

winbase.h header

Article01/24/2023

This header is used by multiple technologies. For more information, see:

- Application Installation and Servicing
- Application Recovery and Restart
- Backup
- Data Access and Storage
- Data Exchange
- Developer Notes
- eventlogprov
- Hardware Counter Profiling
- Internationalization for Windows Applications
- Menus and Other Resources
- Operation Recorder
- Remote Desktop Services
- Security and Identity
- System Services
- Window Stations and Desktops
- Windows and Messages

winbase.h contains the following programming interfaces:

Functions

[_lclose](#)

The _lclose function closes the specified file so that it is no longer available for reading or writing. This function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should use the CloseHandle function.

[_lcreat](#)

Creates or opens the specified file.

[_lseek](#)

Repositions the file pointer for the specified file.

[_lopen](#)

The _lopen function opens an existing file and sets the file pointer to the beginning of the file. This function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should use the CreateFile function.

[_lread](#)

The _lread function reads data from the specified file. This function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should use the ReadFile function.

[_lwrite](#)

Writes data to the specified file.

[AccessCheckAndAuditAlarmA](#)

Determines whether a security descriptor grants a specified set of access rights to the client being impersonated by the calling thread. (AccessCheckAndAuditAlarmA)

[AccessCheckByTypeAndAuditAlarmA](#)

Determines whether a security descriptor grants a specified set of access rights to the client being impersonated by the calling thread. (AccessCheckByTypeAndAuditAlarmA)

[AccessCheckByTypeResultListAndAuditAlarmA](#)

Determines whether a security descriptor grants a specified set of access rights to the client being impersonated by the calling thread. (AccessCheckByTypeResultListAndAuditAlarmA)

[AccessCheckByTypeResultListAndAuditAlarmByHandleA](#)

The AccessCheckByTypeResultListAndAuditAlarmByHandleA (ANSI) function (winbase.h) determines whether a security descriptor grants a specified set of access rights to the client that the calling thread is impersonating.

[ActivateActCtx](#)

The ActivateActCtx function activates the specified activation context.

[AddAtomA](#)

Adds a character string to the local atom table and returns a unique value (an atom) identifying the string. (ANSI)

[AddAtomW](#)

Adds a character string to the local atom table and returns a unique value (an atom) identifying the string. (Unicode)

AddConditionalAce
Adds a conditional access control entry (ACE) to the specified access control list (ACL).
AddIntegrityLabelToBoundaryDescriptor
Adds a new required security identifier (SID) to the specified boundary descriptor.
AddRefActCtx
The AddRefActCtx function increments the reference count of the specified activation context.
AddSecureMemoryCacheCallback
Registers a callback function to be called when a secured memory range is freed or its protections are changed.
ApplicationRecoveryFinished
Indicates that the calling application has completed its data recovery.
ApplicationRecoveryInProgress
Indicates that the calling application is continuing to recover data.
BackupEventLogA
Saves the specified event log to a backup file. (ANSI)
BackupEventLogW
Saves the specified event log to a backup file. (Unicode)
BackupRead
Back up a file or directory, including the security information.
BackupSeek
Seeks forward in a data stream initially accessed by using the BackupRead or BackupWrite function.
BackupWrite
Restore a file or directory that was backed up using BackupRead.
BeginUpdateResourceA

Retrieves a handle that can be used by the UpdateResource function to add, delete, or replace resources in a binary module. (ANSI)

[BeginUpdateResourceW](#)

Retrieves a handle that can be used by the UpdateResource function to add, delete, or replace resources in a binary module. (Unicode)

[BindIoCompletionCallback](#)

Associates the I/O completion port owned by the thread pool with the specified file handle. On completion of an I/O request involving this file, a non-I/O worker thread will execute the specified callback function.

[BuildCommDCBA](#)

Fills a specified DCB structure with values specified in a device-control string. (ANSI)

[BuildCommDCBAndTimeoutsA](#)

Translates a device-definition string into appropriate device-control block codes and places them into a device control block. (ANSI)

[BuildCommDCBAndTimeoutsW](#)

Translates a device-definition string into appropriate device-control block codes and places them into a device control block. (Unicode)

[BuildCommDCBW](#)

Fills a specified DCB structure with values specified in a device-control string. (Unicode)

[CallNamedPipeA](#)

Connects to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe. (CallNamedPipeA)

[CheckNameLegalDOS8Dot3A](#)

Determines whether the specified name can be used to create a file on a FAT file system. (ANSI)

[CheckNameLegalDOS8Dot3W](#)

Determines whether the specified name can be used to create a file on a FAT file system. (Unicode)

[ClearCommBreak](#)

	Restores character transmission for a specified communications device and places the transmission line in a nonbreak state.
ClearCommError	Retrieves information about a communications error and reports the current status of a communications device.
ClearEventLogA	Clears the specified event log, and optionally saves the current copy of the log to a backup file. (ANSI)
ClearEventLogW	Clears the specified event log, and optionally saves the current copy of the log to a backup file. (Unicode)
CloseEncryptedFileRaw	Closes an encrypted file after a backup or restore operation, and frees associated system resources.
CloseEventLog	Closes the specified event log. (CloseEventLog)
CommConfigDialogA	Displays a driver-supplied configuration dialog box. (ANSI)
CommConfigDialogW	Displays a driver-supplied configuration dialog box. (Unicode)
ConvertFiberToThread	Converts the current fiber into a thread.
ConvertThreadToFiber	Converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers. (ConvertThreadToFiber)
ConvertThreadToFiberEx	Converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers. (ConvertThreadToFiberEx)

[CopyContext](#)

Copies a source context structure (including any XState) onto an initialized destination context structure.

[CopyFile](#)

The CopyFile function (`winbase.h`) copies an existing file to a new file.

[CopyFile2](#)

Copies an existing file to a new file, notifying the application of its progress through a callback function. (`CopyFile2`)

[CopyFileA](#)

Copies an existing file to a new file. (`CopyFileA`)

[CopyFileExA](#)

Copies an existing file to a new file, notifying the application of its progress through a callback function. (`CopyFileExA`)

[CopyFileExW](#)

Copies an existing file to a new file, notifying the application of its progress through a callback function. (`CopyFileExW`)

[CopyFileTransactedA](#)

Copies an existing file to a new file as a transacted operation, notifying the application of its progress through a callback function. (ANSI)

[CopyFileTransactedW](#)

Copies an existing file to a new file as a transacted operation, notifying the application of its progress through a callback function. (Unicode)

[CopyFileW](#)

The `CopyFileW` (Unicode) function (`winbase.h`) copies an existing file to a new file.

[CreateActCtxA](#)

The `CreateActCtx` function creates an activation context. (ANSI)

[CreateActCtxW](#)

The `CreateActCtx` function creates an activation context. (Unicode)

[CreateBoundaryDescriptorA](#)

The CreateBoundaryDescriptorA (ANSI) function (winbase.h) creates a boundary descriptor.

[.CreateDirectory](#)

The CreateDirectory function (winbase.h) creates a new directory.

[.CreateDirectoryExA](#)

Creates a new directory with the attributes of a specified template directory. (ANSI)

[.CreateDirectoryExW](#)

Creates a new directory with the attributes of a specified template directory. (Unicode)

[.CreateDirectoryTransactedA](#)

Creates a new directory as a transacted operation, with the attributes of a specified template directory. (ANSI)

[.CreateDirectoryTransactedW](#)

Creates a new directory as a transacted operation, with the attributes of a specified template directory. (Unicode)

[CreateFiber](#)

Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber. (CreateFiber)

[CreateFiberEx](#)

Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber. (CreateFiberEx)

[CreateFileMappingA](#)

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

[CreateFileMappingNumaA](#)

Creates or opens a named or unnamed file mapping object for a specified file and specifies the NUMA node for the physical memory. (CreateFileMappingNumaA)

[CreateFileTransactedA](#)

Creates or opens a file, file stream, or directory as a transacted operation. (ANSI)

CreateFileTransactedW	Creates or opens a file, file stream, or directory as a transacted operation. (Unicode)
CreateHardLinkA	Establishes a hard link between an existing file and a new file. (ANSI)
CreateHardLinkTransactedA	Establishes a hard link between an existing file and a new file as a transacted operation. (ANSI)
CreateHardLinkTransactedW	Establishes a hard link between an existing file and a new file as a transacted operation. (Unicode)
CreateHardLinkW	Establishes a hard link between an existing file and a new file. (Unicode)
CreateJobObjectA	Creates or opens a job object. (CreateJobObjectA)
CreateMailslotA	Creates a mailslot with the specified name and returns a handle that a mailslot server can use to perform operations on the mailslot. (ANSI)
CreateMailslotW	Creates a mailslot with the specified name and returns a handle that a mailslot server can use to perform operations on the mailslot. (Unicode)
CreateNamedPipeA	The CreateNamedPipeA (ANSI) function (winbase.h) creates an instance of a named pipe and returns a handle for subsequent pipe operations.
CreatePrivateNamespaceA	The CreatePrivateNamespaceA (ANSI) function (winbase.h) creates a private namespace.
CreateProcessWithLogonW	Creates a new process and its primary thread. Then the new process runs the specified executable file in the security context of the specified credentials (user, domain, and password). It can optionally load the user profile for a specified user.

[CreateProcessWithTokenW](#)

Creates a new process and its primary thread. The new process runs in the security context of the specified token. It can optionally load the user profile for the specified user.

[CreateSemaphoreA](#)

Creates or opens a named or unnamed semaphore object. (CreateSemaphoreA)

[CreateSemaphoreExA](#)

Creates or opens a named or unnamed semaphore object and returns a handle to the object. (CreateSemaphoreExA)

[CreateSymbolicLinkA](#)

Creates a symbolic link. (ANSI)

[CreateSymbolicLinkTransactedA](#)

Creates a symbolic link as a transacted operation. (ANSI)

[CreateSymbolicLinkTransactedW](#)

Creates a symbolic link as a transacted operation. (Unicode)

[CreateSymbolicLinkW](#)

Creates a symbolic link. (Unicode)

[CreateTapePartition](#)

Reformats a tape.

[CreateUmsCompletionList](#)

Creates a user-mode scheduling (UMS) completion list.

[CreateUmsThreadContext](#)

Creates a user-mode scheduling (UMS) thread context to represent a UMS worker thread.

[DeactivateActCtx](#)

The DeactivateActCtx function deactivates the activation context corresponding to the specified cookie.

[DebugBreakProcess](#)

Causes a breakpoint exception to occur in the specified process. This allows the calling thread to signal the debugger to handle the exception.

[DebugSetProcessKillOnExit](#)

Sets the action to be performed when the calling thread exits.

[DecryptFileA](#)

Decrypts an encrypted file or directory. (ANSI)

[DecryptFileW](#)

Decrypts an encrypted file or directory. (Unicode)

[DefineDosDeviceA](#)

Defines, redefines, or deletes MS-DOS device names. (DefineDosDeviceA)

[DeleteAtom](#)

Decrements the reference count of a local string atom. If the atom's reference count is reduced to zero, DeleteAtom removes the string associated with the atom from the local atom table.

[DeleteFiber](#)

Deletes an existing fiber.

[DeleteFile](#)

The DeleteFile function (winbase.h) deletes an existing file.

[DeleteFileTransactedA](#)

Deletes an existing file as a transacted operation. (ANSI)

[DeleteFileTransactedW](#)

Deletes an existing file as a transacted operation. (Unicode)

[DeleteUmsCompletionList](#)

Deletes the specified user-mode scheduling (UMS) completion list. The list must be empty.

[DeleteUmsThreadContext](#)

Deletes the specified user-mode scheduling (UMS) thread context. The thread must be terminated.

DeleteVolumeMountPointA	Deletes a drive letter or mounted folder. (DeleteVolumeMountPointA)
DequeueUmsCompletionListItems	Retrieves user-mode scheduling (UMS) worker threads from the specified UMS completion list.
DeregisterEventSource	Closes the specified event log. (DeregisterEventSource)
DestroyThreadpoolEnvironment	Deletes the specified callback environment. Call this function when the callback environment is no longer needed for creating new thread pool objects. (DestroyThreadpoolEnvironment)
DisableThreadProfiling	Disables thread profiling.
DnsHostnameToComputerNameA	Converts a DNS-style host name to a NetBIOS-style computer name. (ANSI)
DnsHostnameToComputerNameW	Converts a DNS-style host name to a NetBIOS-style computer name. (Unicode)
DosDateTimeToFileTime	Converts MS-DOS date and time values to a file time.
EnableProcessOptionalXStateFeatures	The EnableProcessOptionalXStateFeatures function enables a set of optional XState features for the current process.
EnableThreadProfiling	Enables thread profiling on the specified thread.
EncryptFileA	Encrypts a file or directory. (ANSI)
EncryptFileW	Encrypts a file or directory. (Unicode)

EndUpdateResourceA
Commits or discards changes made prior to a call to UpdateResource. (ANSI)
EndUpdateResourceW
Commits or discards changes made prior to a call to UpdateResource. (Unicode)
EnterUmsSchedulingMode
Converts the calling thread into a user-mode scheduling (UMS) scheduler thread.
EnumResourceLanguagesA
Enumerates language-specific resources, of the specified type and name, associated with a binary module. (ANSI)
EnumResourceLanguagesW
Enumerates language-specific resources, of the specified type and name, associated with a binary module. (Unicode)
EnumResourceTypesA
Enumerates resource types within a binary module. (ANSI)
EnumResourceTypesW
Enumerates resource types within a binary module. (Unicode)
EraseTape
Erases all or part of a tape.
EscapeCommFunction
Directs the specified communications device to perform an extended function.
ExecuteUmsThread
Runs the specified UMS worker thread.
FatalExit
Transfers execution control to the debugger. The behavior of the debugger thereafter is specific to the type of debugger used.
FileEncryptionStatusA

Retrieves the encryption status of the specified file. (ANSI)

[FileEncryptionStatusW](#)

Retrieves the encryption status of the specified file. (Unicode)

[FileTimeToDosDateTime](#)

Converts a file time to MS-DOS date and time values.

[FindActCtxSectionGuid](#)

The FindActCtxSectionGuid function retrieves information on a specific GUID in the current activation context and returns a ACTCTX_SECTION_KEYED_DATA structure.

[FindActCtxSectionStringA](#)

The FindActCtxSectionString function retrieves information on a specific string in the current activation context and returns a ACTCTX_SECTION_KEYED_DATA structure. (ANSI)

[FindActCtxSectionStringW](#)

The FindActCtxSectionString function retrieves information on a specific string in the current activation context and returns a ACTCTX_SECTION_KEYED_DATA structure. (Unicode)

[FindAtomA](#)

Searches the local atom table for the specified character string and retrieves the atom associated with that string. (ANSI)

[FindAtomW](#)

Searches the local atom table for the specified character string and retrieves the atom associated with that string. (Unicode)

[FindFirstFileNameTransactedW](#)

Creates an enumeration of all the hard links to the specified file as a transacted operation. The function returns a handle to the enumeration that can be used on subsequent calls to the FindNextFileNameW function.

[FindFirstFileTransactedA](#)

Searches a directory for a file or subdirectory with a name that matches a specific name as a transacted operation. (ANSI)

[FindFirstFileTransactedW](#)

Searches a directory for a file or subdirectory with a name that matches a specific name as a transacted operation. (Unicode)

[FindFirstStreamTransactedW](#)

Enumerates the first stream in the specified file or directory as a transacted operation.

[FindFirstVolumeA](#)

Retrieves the name of a volume on a computer. (FindFirstVolumeA)

[FindFirstVolumeMountPointA](#)

Retrieves the name of a mounted folder on the specified volume. (ANSI)

[FindFirstVolumeMountPointW](#)

Retrieves the name of a mounted folder on the specified volume. (Unicode)

[FindNextVolumeA](#)

Continues a volume search started by a call to the FindFirstVolume function. (FindNextVolumeA)

[FindNextVolumeMountPointA](#)

Continues a mounted folder search started by a call to the FindFirstVolumeMountPoint function. (ANSI)

[FindNextVolumeMountPointW](#)

Continues a mounted folder search started by a call to the FindFirstVolumeMountPoint function. (Unicode)

[FindResourceA](#)

Determines the location of a resource with the specified type and name in the specified module. (FindResourceA)

[FindResourceExA](#)

Determines the location of the resource with the specified type, name, and language in the specified module. (FindResourceExA)

[FindVolumeMountPointClose](#)

Closes the specified mounted folder search handle.

[FormatMessage](#)

The FormatMessage function (winbase.h) formats a message string.

[FormatMessageA](#)

Formats a message string. (FormatMessageA)

[FormatMessageW](#)

The FormatMessageW (Unicode) function (winbase.h) formats a message string.

[GetActiveProcessorCount](#)

Returns the number of active processors in a processor group or in the system.

[GetActiveProcessorGroupCount](#)

Returns the number of active processor groups in the system.

[GetApplicationRecoveryCallback](#)

Retrieves a pointer to the callback routine registered for the specified process. The address returned is in the virtual address space of the process.

[GetApplicationRestartSettings](#)

Retrieves the restart information registered for the specified process.

[GetAtomNameA](#)

Retrieves a copy of the character string associated with the specified local atom. (ANSI)

[GetAtomNameW](#)

Retrieves a copy of the character string associated with the specified local atom. (Unicode)

[GetBinaryTypeA](#)

Determines whether a file is an executable (.exe) file, and if so, which subsystem runs the executable file. (ANSI)

[GetBinaryTypeW](#)

Determines whether a file is an executable (.exe) file, and if so, which subsystem runs the executable file. (Unicode)

[GetCommConfig](#)

Retrieves the current configuration of a communications device.

GetCommMask	Retrieves the value of the event mask for a specified communications device.
GetCommModemStatus	Retrieves the modem control-register values.
GetCommPorts	Gets an array that contains the well-formed COM ports.
GetCommProperties	Retrieves information about the communications properties for a specified communications device.
GetCommState	Retrieves the current control settings for a specified communications device.
GetCommTimeouts	Retrieves the time-out parameters for all read and write operations on a specified communications device.
GetCompressedFileSizeTransactedA	Retrieves the actual number of bytes of disk storage used to store a specified file as a transacted operation. (ANSI)
GetCompressedFileSizeTransactedW	Retrieves the actual number of bytes of disk storage used to store a specified file as a transacted operation. (Unicode)
GetComputerNameA	Retrieves the NetBIOS name of the local computer. This name is established at system startup, when the system reads it from the registry. (ANSI)
GetComputerNameW	Retrieves the NetBIOS name of the local computer. This name is established at system startup, when the system reads it from the registry. (Unicode)
GetCurrentActCtx	

The GetCurrentActCtx function returns the handle to the active activation context of the calling thread.

[GetCurrentDirectory](#)

Retrieves the current directory for the current process.

[GetCurrentHwProfileA](#)

Retrieves information about the current hardware profile for the local computer. (ANSI)

[GetCurrentHwProfileW](#)

Retrieves information about the current hardware profile for the local computer. (Unicode)

[GetCurrentUmsThread](#)

Returns the user-mode scheduling (UMS) thread context of the calling UMS thread.

[GetDefaultCommConfigA](#)

Retrieves the default configuration for the specified communications device. (ANSI)

[GetDefaultCommConfigW](#)

Retrieves the default configuration for the specified communications device. (Unicode)

[GetDevicePowerState](#)

Retrieves the current power state of the specified device.

[GetDllDirectoryA](#)

Retrieves the application-specific portion of the search path used to locate DLLs for the application. (ANSI)

[GetDllDirectoryW](#)

Retrieves the application-specific portion of the search path used to locate DLLs for the application. (Unicode)

[GetEnabledXStateFeatures](#)

Gets a mask of enabled XState features on x86 or x64 processors.

[GetEnvironmentVariable](#)

The GetEnvironmentVariable function (`winbase.h`) retrieves the contents of the specified variable from the environment block of the calling process.

GetEventLogInformation	Retrieves information about the specified event log.
GetFileAttributesTransactedA	Retrieves file system attributes for a specified file or directory as a transacted operation. (ANSI)
GetFileAttributesTransactedW	Retrieves file system attributes for a specified file or directory as a transacted operation. (Unicode)
GetFileBandwidthReservation	Retrieves the bandwidth reservation properties of the volume on which the specified file resides.
GetFileInformationByHandleEx	Retrieves file information for the specified file. (GetFileInformationByHandleEx)
GetFileSecurityA	Obtains specified information about the security of a file or directory. The information obtained is constrained by the caller's access rights and privileges. (GetFileSecurityA)
GetFirmwareEnvironmentVariableA	Retrieves the value of the specified firmware environment variable. (ANSI)
GetFirmwareEnvironmentVariableExA	Retrieves the value of the specified firmware environment variable and its attributes. (ANSI)
GetFirmwareEnvironmentVariableExW	Retrieves the value of the specified firmware environment variable and its attributes. (Unicode)
GetFirmwareEnvironmentVariableW	Retrieves the value of the specified firmware environment variable. (Unicode)
GetFirmwareType	Retrieves the firmware type of the local computer.
GetFullPathNameTransactedA	Retrieves the full path and file name of the specified file as a transacted operation. (ANSI)

[GetFullPathNameTransactedW](#)

Retrieves the full path and file name of the specified file as a transacted operation. (Unicode)

[GetLogicalDriveStringsA](#)

Fills a buffer with strings that specify valid drives in the system. (GetLogicalDriveStringsA)

[GetLongPathNameTransactedA](#)

Converts the specified path to its long form as a transacted operation. (ANSI)

[GetLongPathNameTransactedW](#)

Converts the specified path to its long form as a transacted operation. (Unicode)

[GetMailslotInfo](#)

Retrieves information about the specified mailslot.

[GetMaximumProcessorCount](#)

Returns the maximum number of logical processors that a processor group or the system can have.

[GetMaximumProcessorGroupCount](#)

Returns the maximum number of processor groups that the system can have.

[GetNamedPipeClientComputerNameA](#)

The GetNamedPipeClientComputerNameA (ANSI) function (winbase.h) retrieves the client computer name for the specified named pipe.

[GetNamedPipeClientProcessId](#)

Retrieves the client process identifier for the specified named pipe.

[GetNamedPipeClientSessionId](#)

Retrieves the client session identifier for the specified named pipe.

[GetNamedPipeHandleStateA](#)

The GetNamedPipeHandleStateA (ANSI) function (winbase.h) retrieves information about a specified named pipe.

[GetNamedPipeServerProcessId](#)

	<p>Retrieves the server process identifier for the specified named pipe.</p>
GetNamedPipeServerSessionId	<p>Retrieves the server session identifier for the specified named pipe.</p>
GetNextUmsListItem	<p>Returns the next user-mode scheduling (UMS) thread context in a list of thread contexts.</p>
GetNumaAvailableMemoryNode	<p>Retrieves the amount of memory available in the specified node.</p>
GetNumaAvailableMemoryNodeEx	<p>Retrieves the amount of memory that is available in a node specified as a USHORT value.</p>
GetNumaNodeNumberFromHandle	<p>Retrieves the NUMA node associated with the file or I/O device represented by the specified file handle.</p>
GetNumaNodeProcessorMask	<p>Retrieves the processor mask for the specified node.</p>
GetNumaProcessorNode	<p>Retrieves the node number for the specified processor.</p>
GetNumaProcessorNodeEx	<p>Retrieves the node number as a USHORT value for the specified logical processor.</p>
GetNumaProximityNode	<p>Retrieves the NUMA node number that corresponds to the specified proximity domain identifier.</p>
GetNumberOfEventLogRecords	<p>Retrieves the number of records in the specified event log.</p>
GetOldestEventLogRecord	<p>Retrieves the absolute record number of the oldest record in the specified event log.</p>
GetPrivateProfileInt	

The GetPrivateProfileInt function (winbase.h) retrieves an integer associated with a key in the specified section of an initialization file.

[GetPrivateProfileIntA](#)

Retrieves an integer associated with a key in the specified section of an initialization file.
(GetPrivateProfileIntA)

[GetPrivateProfileIntW](#)

The GetPrivateProfileIntW (Unicode) function (winbase.h) retrieves an integer associated with a key in the specified section of an initialization file.

[GetPrivateProfileSection](#)

The GetPrivateProfileSection function (winbase.h) retrieves all the keys and values for the specified section of an initialization file.

[GetPrivateProfileSectionA](#)

Retrieves all the keys and values for the specified section of an initialization file.
(GetPrivateProfileSectionA)

[GetPrivateProfileSectionNames](#)

The GetPrivateProfileSectionNames function (winbase.h) retrieves the names of all sections in an initialization file.

[GetPrivateProfileSectionNamesA](#)

Retrieves the names of all sections in an initialization file. (GetPrivateProfileSectionNamesA)

[GetPrivateProfileSectionNamesW](#)

The GetPrivateProfileSectionNamesW (Unicode) function (winbase.h) retrieves the names of all sections in an initialization file.

[GetPrivateProfileSectionW](#)

The GetPrivateProfileSectionW (Unicode) function (winbase.h) retrieves all the keys and values for the specified section of an initialization file.

[GetPrivateProfileString](#)

The GetPrivateProfileString function (winbase.h) retrieves a string from the specified section in an initialization file.

[GetPrivateProfileStringA](#)

Retrieves a string from the specified section in an initialization file. (GetPrivateProfileStringA)

[GetPrivateProfileStringW](#)

The GetPrivateProfileStringW (Unicode) function (winbase.h) retrieves a string from the specified section in an initialization file.

[GetPrivateProfileStruct](#)

The GetPrivateProfileStruct function (winbase.h) retrieves the data associated with a key in the specified section of an initialization file.

[GetPrivateProfileStructA](#)

Retrieves the data associated with a key in the specified section of an initialization file.
(GetPrivateProfileStructA)

[GetPrivateProfileStructW](#)

The GetPrivateProfileStructW (Unicode) function (winbase.h) retrieves the data associated with a key in the specified section of an initialization file.

[GetProcessAffinityMask](#)

Retrieves the process affinity mask for the specified process and the system affinity mask for the system.

[GetProcessDEPPolicy](#)

Gets the data execution prevention (DEP) and DEP-ATL thunk emulation settings for the specified 32-bit process. Windows XP with SP3: Gets the DEP and DEP-ATL thunk emulation settings for the current process.

[GetProcessIoCounters](#)

Retrieves accounting information for all I/O operations performed by the specified process.

[GetProfileIntA](#)

Retrieves an integer from a key in the specified section of the Win.ini file. (ANSI)

[GetProfileIntW](#)

Retrieves an integer from a key in the specified section of the Win.ini file. (Unicode)

[GetProfileSectionA](#)

Retrieves all the keys and values for the specified section of the Win.ini file. (ANSI)

[GetProfileSectionW](#)

Retrieves all the keys and values for the specified section of the Win.ini file. (Unicode)

[GetProfileStringA](#)

Retrieves the string associated with a key in the specified section of the Win.ini file. (ANSI)

[GetProfileStringW](#)

Retrieves the string associated with a key in the specified section of the Win.ini file. (Unicode)

[GetShortPathNameA](#)

Retrieves the short path form of the specified path. (GetShortPathNameA)

[GetSystemDEPPolicy](#)

Gets the data execution prevention (DEP) policy setting for the system.

[GetSystemPowerStatus](#)

Retrieves the power status of the system. The status indicates whether the system is running on AC or DC power, whether the battery is currently charging, how much battery life remains, and if battery saver is on or off.

[GetSystemRegistryQuota](#)

Retrieves the current size of the registry and the maximum size that the registry is allowed to attain on the system.

[GetTapeParameters](#)

Retrieves information that describes the tape or the tape drive.

[GetTapePosition](#)

Retrieves the current address of the tape, in logical or absolute blocks.

[GetTapeStatus](#)

Determines whether the tape device is ready to process tape commands.

[GetTempFileName](#)

The GetTempFileName function (winbase.h) creates a name for a temporary file. If a unique file name is generated, an empty file is created and the handle to it is released; otherwise, only a file name is generated.

[GetThreadEnabledXStateFeatures](#)

The GetThreadEnabledXStateFeatures function returns the set of XState features that are currently enabled for the current thread.

[GetThreadSelectorEntry](#)

Retrieves a descriptor table entry for the specified selector and thread.

[GetUmsCompletionListEvent](#)

Retrieves a handle to the event associated with the specified user-mode scheduling (UMS) completion list.

[GetUmsSystemThreadInformation](#)

Queries whether the specified thread is a UMS scheduler thread, a UMS worker thread, or a non-UMS thread.

[GetUserNameA](#)

Retrieves the name of the user associated with the current thread. (ANSI)

[GetUserNameW](#)

Retrieves the name of the user associated with the current thread. (Unicode)

[GetVolumeNameForVolumeMountPointA](#)

Retrieves a volume GUID path for the volume that is associated with the specified volume mount point (drive letter, volume GUID path, or mounted folder).

(GetVolumeNameForVolumeMountPointA)

[GetVolumePathNameA](#)

Retrieves the volume mount point where the specified path is mounted. (GetVolumePathNameA)

[GetVolumePathNamesForVolumeNameA](#)

Retrieves a list of drive letters and mounted folder paths for the specified volume.

(GetVolumePathNamesForVolumeNameA)

[GetXStateFeaturesMask](#)

Returns the mask of XState features set within a CONTEXT structure.

[GlobalAddAtomA](#)

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string. (GlobalAddAtomA)

[GlobalAddAtomExA](#)

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string. (GlobalAddAtomExA)

[GlobalAddAtomExW](#)

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string. (GlobalAddAtomExW)

[GlobalAddAtomW](#)

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string. (GlobalAddAtomW)

[GlobalAlloc](#)

Allocates the specified number of bytes from the heap. (GlobalAlloc)

[GlobalDeleteAtom](#)

Decrements the reference count of a global string atom. If the atom's reference count reaches zero, GlobalDeleteAtom removes the string associated with the atom from the global atom table.

[GlobalDiscard](#)

Discards the specified global memory block.

[GlobalFindAtomA](#)

Searches the global atom table for the specified character string and retrieves the global atom associated with that string. (ANSI)

[GlobalFindAtomW](#)

Searches the global atom table for the specified character string and retrieves the global atom associated with that string. (Unicode)

[GlobalFlags](#)

Retrieves information about the specified global memory object.

[GlobalFree](#)

Frees the specified global memory object and invalidates its handle.

GlobalGetAtomNameA	Retrieves a copy of the character string associated with the specified global atom. (ANSI)
GlobalGetAtomNameW	Retrieves a copy of the character string associated with the specified global atom. (Unicode)
GlobalHandle	Retrieves the handle associated with the specified pointer to a global memory block.
GlobalLock	Locks a global memory object and returns a pointer to the first byte of the object's memory block.
GlobalMemoryStatus	Retrieves information about the system's current usage of both physical and virtual memory. (<code>GlobalMemoryStatus</code>)
GlobalReAlloc	Changes the size or attributes of a specified global memory object. The size can increase or decrease.
GlobalSize	Retrieves the current size of the specified global memory object, in bytes.
GlobalUnlock	Decrement the lock count associated with a memory object that was allocated with <code>GMEM_MOVEABLE</code> .
HasOverlappedIoCompleted	Provides a high performance test operation that can be used to poll for the completion of an outstanding I/O operation.
InitAtomTable	Initializes the local atom table and sets the number of hash buckets to the specified size.
InitializeContext	Initializes a <code>CONTEXT</code> structure inside a buffer with the necessary size and alignment.

InitializeContext2	Initializes a CONTEXT structure inside a buffer with the necessary size and alignment, with the option to specify an XSTATE compaction mask.
InitializeThreadpoolEnvironment	Initializes a callback environment.
InterlockedExchangeSubtract	Performs an atomic subtraction of two values.
IsBadCodePtr	Determines whether the calling process has read access to the memory at the specified address.
IsBadReadPtr	Verifies that the calling process has read access to the specified range of memory. (IsBadReadPtr)
IsBadStringPtrA	Verifies that the calling process has read access to the specified range of memory. (IsBadStringPtrA)
IsBadStringPtrW	Verifies that the calling process has read access to the specified range of memory. (IsBadStringPtrW)
IsBadWritePtr	Verifies that the calling process has write access to the specified range of memory.
IsNativeVhdBoot	Indicates if the OS was booted from a VHD container.
IsSystemResumeAutomatic	Determines the current state of the computer.
IsTextUnicode	Determines if a buffer is likely to contain a form of Unicode text.
LoadModule	

	Loads and executes an application or creates a new instance of an existing application.
LoadPackagedLibrary	Loads the specified packaged module and its dependencies into the address space of the calling process.
LocalAlloc	Allocates the specified number of bytes from the heap. (LocalAlloc)
LocalFlags	Retrieves information about the specified local memory object.
LocalFree	Frees the specified local memory object and invalidates its handle.
LocalHandle	Retrieves the handle associated with the specified pointer to a local memory object.
LocalLock	Locks a local memory object and returns a pointer to the first byte of the object's memory block.
LocalReAlloc	Changes the size or the attributes of a specified local memory object. The size can increase or decrease.
LocalSize	Retrieves the current size of the specified local memory object, in bytes.
LocalUnlock	Decrement the lock count associated with a memory object that was allocated with LMEM_MOVEABLE.
LocateXStateFeature	Retrieves a pointer to the processor state for an XState feature within a CONTEXT structure.
LogonUserA	The Win32 LogonUser function attempts to log a user on to the local computer. LogonUser returns a handle to a user token that you can use to impersonate user. (ANSI)

[LogonUserExA](#)

The LogonUserEx function attempts to log a user on to the local computer. (ANSI)

[LogonUserExW](#)

The LogonUserEx function attempts to log a user on to the local computer. (Unicode)

[LogonUserW](#)

The Win32 LogonUser function attempts to log a user on to the local computer. LogonUser returns a handle to a user token that you can use to impersonate user. (Unicode)

[LookupAccountNameA](#)

Accepts the name of a system and an account as input. It retrieves a security identifier (SID) for the account and the name of the domain on which the account was found. (ANSI)

[LookupAccountNameW](#)

Accepts the name of a system and an account as input. It retrieves a security identifier (SID) for the account and the name of the domain on which the account was found. (Unicode)

[LookupAccountSidA](#)

Accepts a security identifier (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found. (ANSI)

[LookupAccountSidLocalA](#)

Retrieves the name of the account for the specified SID on the local machine. (ANSI)

[LookupAccountSidLocalW](#)

Retrieves the name of the account for the specified SID on the local machine. (Unicode)

[LookupAccountSidW](#)

Accepts a security identifier (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found. (Unicode)

[LookupPrivilegeDisplayNameA](#)

Retrieves the display name that represents a specified privilege. (ANSI)

[LookupPrivilegeDisplayNameW](#)

Retrieves the display name that represents a specified privilege. (Unicode)

[LookupPrivilegeNameA](#)

Retrieves the name that corresponds to the privilege represented on a specific system by a specified locally unique identifier (LUID). (ANSI)

[LookupPrivilegeNameW](#)

Retrieves the name that corresponds to the privilege represented on a specific system by a specified locally unique identifier (LUID). (Unicode)

[LookupPrivilegeValueA](#)

Retrieves the locally unique identifier (LUID) used on a specified system to locally represent the specified privilege name. (ANSI)

[LookupPrivilegeValueW](#)

Retrieves the locally unique identifier (LUID) used on a specified system to locally represent the specified privilege name. (Unicode)

[lstrcatA](#)

Appends one string to another. Warning Do not use. (ANSI)

[lstrcatW](#)

Appends one string to another. Warning Do not use. (Unicode)

[lstrcmpA](#)

Compares two character strings. The comparison is case-sensitive. (ANSI)

[lstrcmpiA](#)

Compares two character strings. The comparison is not case-sensitive. (ANSI)

[lstrcmpiW](#)

Compares two character strings. The comparison is not case-sensitive. (Unicode)

[lstrcmpW](#)

Compares two character strings. The comparison is case-sensitive. (Unicode)

[lstrcpyA](#)

Copies a string to a buffer. (ANSI)

[lstrcpynA](#)

Copies a specified number of characters from a source string into a buffer. Warning Do not use. (ANSI)

[lstrcpynW](#)

Copies a specified number of characters from a source string into a buffer. Warning Do not use. (Unicode)

[lstrcpyW](#)

Copies a string to a buffer. (Unicode)

[lstrlenA](#)

Determines the length of the specified string (not including the terminating null character). (ANSI)

[lstrlenW](#)

Determines the length of the specified string (not including the terminating null character). (Unicode)

[MAKEINTATOM](#)

Converts the specified atom into a string, so it can be passed to functions which accept either atoms or strings.

[MapViewOfFileExScatter](#)

Maps previously allocated physical memory pages at a specified address in an Address Windowing Extensions (AWE) region. (MapViewOfFileExScatter)

[MapViewOfFileExNuma](#)

Maps a view of a file mapping into the address space of a calling process and specifies the NUMA node for the physical memory.

[MoveFile](#)

The MoveFile function (winbase.h) moves an existing file or a directory, including its children.

[MoveFileA](#)

Moves an existing file or a directory, including its children. (MoveFileA)

[MoveFileExA](#)

Moves an existing file or directory, including its children, with various move options. (ANSI)

MoveFileExW	Moves an existing file or directory, including its children, with various move options. (Unicode)
MoveFileTransactedA	Moves an existing file or a directory, including its children, as a transacted operation. (ANSI)
MoveFileTransactedW	Moves an existing file or a directory, including its children, as a transacted operation. (Unicode)
MoveFileW	The MoveFileW (Unicode) function (winbase.h) moves an existing file or a directory, including its children.
MoveFileWithProgressA	Moves a file or directory, including its children. You can provide a callback function that receives progress notifications. (ANSI)
MoveFileWithProgressW	Moves a file or directory, including its children. You can provide a callback function that receives progress notifications. (Unicode)
MulDiv	Multiplies two 32-bit values and then divides the 64-bit result by a third 32-bit value.
NotifyChangeEventLog	Enables an application to receive notification when an event is written to the specified event log.
ObjectCloseAuditAlarmA	Generates an audit message in the security event log when a handle to a private object is deleted. (ObjectCloseAuditAlarmA)
ObjectDeleteAuditAlarmA	The ObjectDeleteAuditAlarmA (ANSI) function (winbase.h) generates audit messages when an object is deleted.
ObjectOpenAuditAlarmA	Generates audit messages when a client application attempts to gain access to an object or to create a new one. (ObjectOpenAuditAlarmA)

[**ObjectPrivilegeAuditAlarmA**](#)

Generates an audit message in the security event log. (ObjectPrivilegeAuditAlarmA)

[**OpenBackupEventLogA**](#)

Opens a handle to a backup event log created by the BackupEventLog function. (ANSI)

[**OpenBackupEventLogW**](#)

Opens a handle to a backup event log created by the BackupEventLog function. (Unicode)

[**OpenCommPort**](#)

Attempts to open a communication device.

[**OpenEncryptedFileRawA**](#)

Opens an encrypted file in order to backup (export) or restore (import) the file. (ANSI)

[**OpenEncryptedFileRawW**](#)

Opens an encrypted file in order to backup (export) or restore (import) the file. (Unicode)

[**OpenEventLogA**](#)

Opens a handle to the specified event log. (ANSI)

[**OpenEventLogW**](#)

Opens a handle to the specified event log. (Unicode)

[**OpenFile**](#)

Creates, opens, reopens, or deletes a file.

[**OpenFileByld**](#)

Opens the file that matches the specified identifier.

[**OpenFileMappingA**](#)

Opens a named file mapping object. (OpenFileMappingA)

[**OpenJobObjectA**](#)

Opens an existing job object. (OpenJobObjectA)

[**OpenPrivateNamespaceA**](#)

	The OpenPrivateNamespaceA (ANSI) function (winbase.h) opens a private namespace.
OperationEnd	Notifies the system that the application is about to end an operation.
OperationStart	Notifies the system that the application is about to start an operation.
PowerClearRequest	Decrements the count of power requests of the specified type for a power request object.
PowerCreateRequest	Creates a new power request object.
PowerSetRequest	Increments the count of power requests of the specified type for a power request object.
PrepareTape	Prepares the tape to be accessed or removed.
PrivilegedServiceAuditAlarmA	Generates an audit message in the security event log. (PrivilegedServiceAuditAlarmA)
PulseEvent	Sets the specified event object to the signaled state and then resets it to the nonsignaled state after releasing the appropriate number of waiting threads.
PurgeComm	Discards all characters from the output or input buffer of a specified communications resource. It can also terminate pending read or write operations on the resource.
QueryActCtxSettingsW	The QueryActCtxSettingsW function specifies the activation context, and the namespace and name of the attribute that is to be queried.
QueryActCtxW	The QueryActCtxW function queries the activation context.

[QueryDosDeviceA](#)

Retrieves information about MS-DOS device names. (QueryDosDeviceA)

[QueryFullProcessImageNameA](#)

Retrieves the full name of the executable image for the specified process. (ANSI)

[QueryFullProcessImageNameW](#)

Retrieves the full name of the executable image for the specified process. (Unicode)

[QueryThreadProfiling](#)

Determines whether thread profiling is enabled for the specified thread.

[QueryUmsThreadInformation](#)

Retrieves information about the specified user-mode scheduling (UMS) worker thread.

[ReadDirectoryChangesExW](#)

Retrieves information that describes the changes within the specified directory, which can include extended information if that information type is specified.

[ReadDirectoryChangesW](#)

Retrieves information that describes the changes within the specified directory.

[ReadEncryptedFileRaw](#)

Backs up (export) encrypted files.

[ReadEventLogA](#)

Reads the specified number of entries from the specified event log. (ANSI)

[ReadEventLogW](#)

Reads the specified number of entries from the specified event log. (Unicode)

[ReadThreadProfilingData](#)

Reads the specified profiling data associated with the thread.

[RegisterApplicationRecoveryCallback](#)

Registers the active instance of an application for recovery.

RegisterApplicationRestart
Registers the active instance of an application for restart.
RegisterEventSourceA
Retrieves a registered handle to the specified event log. (ANSI)
RegisterEventSourceW
Retrieves a registered handle to the specified event log. (Unicode)
RegisterWaitForSingleObject
Directs a wait thread in the thread pool to wait on the object.
ReleaseActCtx
The ReleaseActCtx function decrements the reference count of the specified activation context.
RemoveDirectoryTransactedA
Deletes an existing empty directory as a transacted operation. (ANSI)
RemoveDirectoryTransactedW
Deletes an existing empty directory as a transacted operation. (Unicode)
RemoveSecureMemoryCacheCallback
Unregisters a callback function that was previously registered with the AddSecureMemoryCacheCallback function.
ReOpenFile
Reopens the specified file system object with different access rights, sharing mode, and flags.
ReplaceFileA
Replaces one file with another file, with the option of creating a backup copy of the original file. (ANSI)
ReplaceFileW
Replaces one file with another file, with the option of creating a backup copy of the original file. (Unicode)
ReportEventA

Writes an entry at the end of the specified event log. (ANSI)
ReportEventW
Writes an entry at the end of the specified event log. (Unicode)
RequestWakeupLatency
Has no effect and returns STATUS_NOT_SUPPORTED. This function is provided only for compatibility with earlier versions of Windows. Windows Server 2008 and Windows Vista: Has no effect and always returns success.
SetCommBreak
Suspends character transmission for a specified communications device and places the transmission line in a break state until the ClearCommBreak function is called.
SetCommConfig
Sets the current configuration of a communications device.
SetCommMask
Specifies a set of events to be monitored for a communications device.
SetCommState
Configures a communications device according to the specifications in a device-control block (a DCB structure). The function reinitializes all hardware and control settings, but it does not empty output or input queues.
SetCommTimeouts
Sets the time-out parameters for all read and write operations on a specified communications device.
SetCurrentDirectory
Changes the current directory for the current process.
SetDefaultCommConfigA
Sets the default configuration for a communications device. (ANSI)
SetDefaultCommConfigW
Sets the default configuration for a communications device. (Unicode)
SetDllDirectoryA

Adds a directory to the search path used to locate DLLs for the application. (ANSI)
SetDllDirectoryW Adds a directory to the search path used to locate DLLs for the application. (Unicode)
SetEnvironmentVariable The SetEnvironmentVariable function (winbase.h) sets the contents of the specified environment variable for the current process.
SetFileAttributesTransactedA Sets the attributes for a file or directory as a transacted operation. (ANSI)
SetFileAttributesTransactedW Sets the attributes for a file or directory as a transacted operation. (Unicode)
SetFileBandwidthReservation Requests that bandwidth for the specified file stream be reserved. The reservation is specified as a number of bytes in a period of milliseconds for I/O requests on the specified file handle.
SetFileCompletionNotificationModes Sets the notification modes for a file handle, allowing you to specify how completion notifications work for the specified file.
SetFileSecurityA The SetFileSecurityA (ANSI) function (winbase.h) sets the security of a file or directory object.
SetFileShortNameA Sets the short name for the specified file. (ANSI)
SetFileShortNameW Sets the short name for the specified file. (Unicode)
SetFirmwareEnvironmentVariableA Sets the value of the specified firmware environment variable. (ANSI)
SetFirmwareEnvironmentVariableExA Sets the value of the specified firmware environment variable as the attributes that indicate how this variable is stored and maintained.

SetFirmwareEnvironmentVariableExW	Sets the value of the specified firmware environment variable and the attributes that indicate how this variable is stored and maintained.
SetFirmwareEnvironmentVariableW	Sets the value of the specified firmware environment variable. (Unicode)
SetHandleCount	The SetHandleCount function changes the number of file handles available to a process.
SetMailslotInfo	Sets the time-out value used by the specified mailslot for a read operation.
SetProcessAffinityMask	Sets a processor affinity mask for the threads of the specified process.
SetProcessDEPPolicy	Changes data execution prevention (DEP) and DEP-ATL thunk emulation settings for a 32-bit process.
SetSearchPathMode	Sets the per-process mode that the SearchPath function uses when locating files.
SetSystemPowerState	Suspends the system by shutting power down. Depending on the ForceFlag parameter, the function either suspends operation immediately or requests permission from all applications and device drivers before doing so.
SetTapeParameters	Specifies the block size of a tape or configures the tape device.
SetTapePosition	Sets the tape position on the specified device.
SetThreadAffinityMask	Sets a processor affinity mask for the specified thread.

SetThreadExecutionState
Enables an application to inform the system that it is in use, thereby preventing the system from entering sleep or turning off the display while the application is running.
SetThreadpoolCallbackCleanupGroup
Associates the specified cleanup group with the specified callback environment. (SetThreadpoolCallbackCleanupGroup)
SetThreadpoolCallbackLibrary
Ensures that the specified DLL remains loaded as long as there are outstanding callbacks. (SetThreadpoolCallbackLibrary)
SetThreadpoolCallbackPersistent
Specifies that the callback should run on a persistent thread. (SetThreadpoolCallbackPersistent)
SetThreadpoolCallbackPool
Sets the thread pool to be used when generating callbacks.
SetThreadpoolCallbackPriority
Specifies the priority of a callback function relative to other work items in the same thread pool. (SetThreadpoolCallbackPriority)
SetThreadpoolCallbackRunsLong
Indicates that callbacks associated with this callback environment may not return quickly. (SetThreadpoolCallbackRunsLong)
SetUmsThreadInformation
Sets application-specific context information for the specified user-mode scheduling (UMS) worker thread.
SetupComm
Initializes the communications parameters for a specified communications device.
SetVolumeLabelA
Sets the label of a file system volume. (ANSI)
SetVolumeLabelW
Sets the label of a file system volume. (Unicode)

SetVolumeMountPointA
Associates a volume with a drive letter or a directory on another volume. (ANSI)
SetVolumeMountPointW
Associates a volume with a drive letter or a directory on another volume. (Unicode)
SetXStateFeaturesMask
Sets the mask of XState features set within a CONTEXT structure.
SwitchToFiber
Schedules a fiber. The function must be called on a fiber.
TransmitCommChar
Transmits a specified character ahead of any pending data in the output buffer of the specified communications device.
UmsThreadYield
Yields control to the user-mode scheduling (UMS) scheduler thread on which the calling UMS worker thread is running.
UnregisterApplicationRecoveryCallback
Removes the active instance of an application from the recovery list.
UnregisterApplicationRestart
Removes the active instance of an application from the restart list.
UnregisterWait
Cancels a registered wait operation issued by the RegisterWaitForSingleObject function. (UnregisterWait)
UpdateResourceA
Adds, deletes, or replaces a resource in a portable executable (PE) file. (ANSI)
UpdateResourceW
Adds, deletes, or replaces a resource in a portable executable (PE) file. (Unicode)
VerifyVersionInfoA

Compares a set of operating system version requirements to the corresponding values for the currently running version of the system. (ANSI)

[VerifyVersionInfoW](#)

Compares a set of operating system version requirements to the corresponding values for the currently running version of the system. (Unicode)

[WaitCommEvent](#)

Waits for an event to occur for a specified communications device. The set of events that are monitored by this function is contained in the event mask associated with the device handle.

[WaitNamedPipeA](#)

The `WaitNamedPipeA` (ANSI) function (`winbase.h`) waits until either a time-out interval elapses or an instance of the specified named pipe is available for connection (that is, the pipe's server process has a pending `ConnectNamedPipe` operation on the pipe).

[WinExec](#)

Runs the specified application.

[WinMain](#)

The user-provided entry point for a graphical Windows-based application.

[Wow64GetThreadSelectorEntry](#)

Retrieves a descriptor table entry for the specified selector and WOW64 thread.

[WriteEncryptedFileRaw](#)

Restores (import) encrypted files.

[WritePrivateProfileSectionA](#)

Replaces the keys and values for the specified section in an initialization file. (ANSI)

[WritePrivateProfileSectionW](#)

Replaces the keys and values for the specified section in an initialization file. (Unicode)

[WritePrivateProfileStringA](#)

Copies a string into the specified section of an initialization file. (ANSI)

[WritePrivateProfileStringW](#)

Copies a string into the specified section of an initialization file. (Unicode)
WritePrivateProfileStructA
Copies data into a key in the specified section of an initialization file. As it copies the data, the function calculates a checksum and appends it to the end of the data. (ANSI)
WritePrivateProfileStructW
Copies data into a key in the specified section of an initialization file. As it copies the data, the function calculates a checksum and appends it to the end of the data. (Unicode)
WriteProfileSectionA
Replaces the contents of the specified section in the Win.ini file with specified keys and values. (ANSI)
WriteProfileSectionW
Replaces the contents of the specified section in the Win.ini file with specified keys and values. (Unicode)
WriteProfileStringA
Copies a string into the specified section of the Win.ini file. (ANSI)
WriteProfileStringW
Copies a string into the specified section of the Win.ini file. (Unicode)
WriteTapemark
Writes a specified number of filemarks, setmarks, short filemarks, or long filemarks to a tape device.
WTSGetActiveConsoleSessionId
Retrieves the session identifier of the console session.
ZombifyActCtx
The ZombifyActCtx function deactivates the specified activation context, but does not deallocate it.

Callback functions

LPPROGRESS_ROUTINE

An application-defined callback function used with the CopyFileEx, MoveFileTransacted, and MoveFileWithProgress functions.

PCOPYFILE2_PROGRESS_ROUTINE

An application-defined callback function used with the CopyFile2 function.

PFE_EXPORT_FUNC

An application-defined callback function used with ReadEncryptedFileRaw.

PFE_IMPORT_FUNC

An application-defined callback function used with WriteEncryptedFileRaw. The system calls ImportCallback one or more times, each time to retrieve a portion of a backup file's data.

PFIBER_START_ROUTINE

An application-defined function used with the CreateFiber function. It serves as the starting address for a fiber.

Structures

ACTCTX_SECTION_KEYED_DATA

The ACTCTX_SECTION_KEYED_DATA structure is used by the FindActCtxSectionString and FindActCtxSectionGuid functions to return the activation context information along with either the GUID or 32-bit integer-tagged activation context section.

ACTCTXA

The ACTCTX structure is used by the CreateActCtx function to create the activation context. (ANSI)

ACTCTXW

The ACTCTX structure is used by the CreateActCtx function to create the activation context. (Unicode)

COMMCONFIG

Contains information about the configuration state of a communications device.

COMMPROP
Contains information about a communications driver.
COMMTIMEOUTS
Contains the time-out parameters for a communications device.
COMSTAT
Contains information about a communications device.
COPYFILE2_EXTENDED_PARAMETERS
Contains extended parameters for the CopyFile2 function.
COPYFILE2_EXTENDED_PARAMETERS_V2
Contains updated, additional functionality beyond the COPYFILE2_EXTENDED_PARAMETERS structure for the CopyFile2 function
COPYFILE2_MESSAGE
Passed to the CopyFile2ProgressRoutine callback function with information about a pending copy operation.
DCB
Defines the control setting for a serial communications device.
EVENTLOG_FULL_INFORMATION
Indicates whether the event log is full.
FILE_ALIGNMENT_INFO
Contains alignment information for a file.
FILE_ALLOCATION_INFO
Contains the total number of bytes that should be allocated for a file.
FILE_ATTRIBUTE_TAG_INFO
Receives the requested file attribute information. Used for any handles.
FILE_BASIC_INFO
Contains the basic information for a file. Used for file handles.

[FILE_COMPRESSION_INFO](#)

Receives file compression information.

[FILE_DISPOSITION_INFO](#)

Indicates whether a file should be deleted. Used for any handles.

[FILE_END_OF_FILE_INFO](#)

Contains the specified value to which the end of the file should be set.

[FILE_FULL_DIR_INFO](#)

Contains directory information for a file. (FILE_FULL_DIR_INFO)

[FILE_ID_BOTH_DIR_INFO](#)

Contains information about files in the specified directory.

[FILE_ID_DESCRIPTOR](#)

Specifies the type of ID that is being used.

[FILE_ID_EXTD_DIR_INFO](#)

Contains identification information for a file. (FILE_ID_EXTD_DIR_INFO)

[FILE_ID_INFO](#)

Contains identification information for a file. (FILE_ID_INFO)

[FILE_IO_PRIORITY_HINT_INFO](#)

Specifies the priority hint for a file I/O operation.

[FILE_NAME_INFO](#)

Receives the file name.

[FILE_REMOTE_PROTOCOL_INFO](#)

Contains file remote protocol information.

[FILE_RENAME_INFO](#)

Contains the name to which the file should be renamed.

[FILE_STANDARD_INFO](#)

Receives extended information for the file.
FILE_STORAGE_INFO
Contains directory information for a file. (FILE_STORAGE_INFO)
FILE_STREAM_INFO
Receives file stream information for the specified file.
HW_PROFILE_INFOA
Contains information about a hardware profile. (ANSI)
HW_PROFILE_INFOW
Contains information about a hardware profile. (Unicode)
MEMORYSTATUS
Contains information about the current state of both physical and virtual memory.
OFSTRUCT
Contains information about a file that the OpenFile function opened or attempted to open.
OPERATION_END_PARAMETERS
This structure is used by the OperationEnd function.
OPERATION_START_PARAMETERS
This structure is used by the OperationStart function.
STARTUPINFOEXA
Specifies the window station, desktop, standard handles, and attributes for a new process. It is used with the CreateProcess and CreateProcessAsUser functions. (ANSI)
STARTUPINFOEXW
Specifies the window station, desktop, standard handles, and attributes for a new process. It is used with the CreateProcess and CreateProcessAsUser functions. (Unicode)
SYSTEM_POWER_STATUS
Contains information about the power status of the system.
UMS_SCHEDULER_STARTUP_INFO

Specifies attributes for a user-mode scheduling (UMS) scheduler thread.

[UMS_SYSTEM_THREAD_INFORMATION](#)

Specifies a UMS scheduler thread, UMS worker thread, or non-UMS thread. The GetUmsSystemThreadInformation function uses this structure.

[WIN32_STREAM_ID](#)

Contains stream data.

Enumerations

[COPYFILE2_COPY_PHASE](#)

Indicates the phase of a copy at the time of an error.

[COPYFILE2_MESSAGE_ACTION](#)

Returned by the CopyFile2ProgressRoutine callback function to indicate what action should be taken for the pending copy operation.

[COPYFILE2_MESSAGE_TYPE](#)

Indicates the type of message passed in the COPYFILE2_MESSAGE structure to the CopyFile2ProgressRoutine callback function.

[FILE_ID_TYPE](#)

Discriminator for the union in the FILE_ID_DESCRIPTOR structure.

[PRIORITY_HINT](#)

Defines values that are used with the FILE_IO_PRIORITY_HINT_INFO structure to specify the priority hint for a file I/O operation.

Feedback

Was this page helpful?



_lclose function (winbase.h)

Article04/02/2021

The _lclose function closes the specified file so that it is no longer available for reading or writing. This function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should use the CloseHandle function.

Syntax

C++

```
HFILE _lclose(  
    HFILE hFile  
)
```

Parameters

`hFile`

Identifies the file to be closed. This handle is returned by the function that created or last opened the file.

Return value

Handle to file to close.

Requirements

Target Platform	Windows
Header	winbase.h
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

_lcreat function (winbase.h)

Article04/02/2021

[This function is provided for compatibility with 16-bit versions of Windows. New applications should use the **CreateFile** function.]

Creates or opens the specified file. This documentation is included only for troubleshooting existing code.

Syntax

C++

```
FILE _lcreat(
    LPCSTR lpPathName,
    int     iAttribute
);
```

Parameters

`lpPathName`

The name of the file. The string must consist of characters from the Windows ANSI character set.

`iAttribute`

The attributes of the file.

This parameter must be set to one of the following values.

Value	Meaning
0	Normal. Can be read from or written to without restriction.
1	Read-only. Cannot be opened for write.
2	Hidden. Not found by directory search.
4	System. Not found by directory search.

Return value

If the function succeeds, the return value is a file handle. Otherwise, the return value is HFILE_ERROR. To get extended error information, use the [GetLastError](#) function.

Remarks

If the file does not exist, `_lcreat` creates and opens a new file for writing. If the file does exist, `_lcreat` truncates the file size to zero and opens it for reading and writing.

When the function opens a file, the pointer is set to the beginning of the file.

Use the `_lcreat` function with care. It can open any file, even one already opened by another function.

Requirements

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

_lseek function (winbase.h)

Article 04/02/2021

[This function is provided for compatibility with 16-bit versions of Windows. New applications should use the [SetFilePointer](#) function.]

Repositions the file pointer for the specified file.

Syntax

C++

```
LONG _lseek(
    HFILE hFile,
    LONG lOffset,
    int iOrigin
);
```

Parameters

`hFile`

A handle to an open file. This handle is created by [_lcreat](#).

`lOffset`

The number of bytes that the file pointer is to be moved.

`iOrigin`

The starting point and the direction that the pointer will be moved.

This parameter must be set to one of the following values.

Value	Meaning
0	Moves the pointer from the beginning of the file.
1	Moves the file from its current location.
2	Moves the pointer from the end of the file.

Return value

If the function succeeds, the return value specifies the new offset. Otherwise, the return value is HFILE_ERROR. To get extended error information, use the [GetLastError](#) function.

Remarks

When a file is initially opened, the file pointer is set to the beginning of the file. The `_lseek` function moves the pointer without reading data, which allows random access to the content of the file.

Requirements

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[SetFilePointer](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

_lopen function (winbase.h)

Article04/02/2021

The _lopen function opens an existing file and sets the file pointer to the beginning of the file. This function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should use the CreateFile function.

Syntax

C++

```
HFILE _lopen(
    LPCSTR lpPathName,
    int     iReadWrite
);
```

Parameters

lpPathName

Pointer to a null-terminated string that names the file to open. The string must consist of characters from the Windows ANSI character set.

iReadWrite

Specifies the modes in which to open the file. This parameter consists of one access mode and an optional share mode. The access mode must be one of the following values: OF_READ, OF_READWRITE, OF_WRITE

The share mode can be one of the following values: OF_SHARE_COMPAT, OF_SHARE_DENY_NONE, OF_SHARE_DENY_READ, OF_SHARE_DENY_WRITE, OF_SHARE_EXCLUSIVE

Return value

If the function succeeds, the return value is a file handle.

Requirements

Target Platform	Windows
Header	winbase.h
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

_lread function (winbase.h)

Article 04/02/2021

The _lread function reads data from the specified file. This function is provided for compatibility with 16-bit versions of Windows. Win32-based applications should use the ReadFile function.

Syntax

C++

```
UINT _lread(
    HFILE hFile,
    LPVOID lpBuffer,
    UINT   uBytes
);
```

Parameters

hFile

Identifies the specified file.

lpBuffer

Pointer to a buffer that contains the data read from the file.

uBytes

Specifies the number of bytes to be read from the file.

Return value

The return value indicates the number of bytes actually read from the file. If the number of bytes read is less than uBytes, the function has reached the end of file (EOF) before reading the specified number of bytes.

Requirements

Target Platform	Windows
Header	winbase.h
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

_lwrite function (winbase.h)

Article04/02/2021

[This function is provided for compatibility with 16-bit versions of Windows. New applications should use the [WriteFile](#) function.]

Writes data to the specified file.

Syntax

C++

```
UINT _lwrite(
    HFILE hFile,
    LPCCH lpBuffer,
    UINT   uBytes
);
```

Parameters

`hFile`

A handle to the file that receives the data. This handle is created by [_lcreat](#).

`lpBuffer`

The buffer that contains the data to be added.

`uBytes`

The number of bytes to write to the file.

Return value

If the function succeeds, the return value is the number of bytes written to the file.

Otherwise, the return value is `HFILE_ERROR`. To get extended error information, use the [GetLastError](#) function.

Requirements

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[WriteFile](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

AccessCheckAndAuditAlarmA function (winbase.h)

Article07/27/2022

The **AccessCheckAndAuditAlarm** function determines whether a [security descriptor](#) grants a specified set of access rights to the client being impersonated by the calling thread. If the security descriptor has a SACL with ACEs that apply to the client, the function generates any necessary audit messages in the security event log.

Alarms are not currently supported.

Syntax

C++

```
BOOL AccessCheckAndAuditAlarmA(
    [in]          LPCSTR             SubsystemName,
    [in, optional] LPVOID            HandleId,
    [in]          LPSTR              ObjectTypeName,
    [in, optional] LPSTR              ObjectName,
    [in]          PSECURITY_DESCRIPTOR SecurityDescriptor,
    [in]          DWORD               DesiredAccess,
    [in]          PGENERIC_MAPPING    GenericMapping,
    [in]          BOOL                ObjectCreation,
    [out]         LPDWORD             GrantedAccess,
    [out]         LPBOOL              AccessStatus,
    [out]         LPBOOL              pfGenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string specifying the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in, optional] HandleId

A pointer to a unique value representing the client's handle to the object. If the access is denied, the system ignores this value.

[in] ObjectTypeName

A pointer to a null-terminated string specifying the type of object being created or accessed. This string appears in any audit message that the function generates.

[in, optional] `ObjectName`

A pointer to a null-terminated string specifying the name of the object being created or accessed. This string appears in any audit message that the function generates.

[in] `SecurityDescriptor`

A pointer to the [SECURITY_DESCRIPTOR](#) structure against which access is checked.

[in] `DesiredAccess`

[Access mask](#) that specifies the access rights to check. This mask must have been mapped by the [MapGenericMask](#) function to contain no generic access rights.

If this parameter is `MAXIMUM_ALLOWED`, the function sets the *GrantedAccess* access mask to indicate the maximum access rights the security descriptor allows the client.

[in] `GenericMapping`

A pointer to the [GENERIC_MAPPING](#) structure associated with the object for which access is being checked.

[in] `ObjectCreation`

Specifies a flag that determines whether the calling application will create a new object when access is granted. A value of `TRUE` indicates the application will create a new object. A value of `FALSE` indicates the application will open an existing object.

[out] `GrantedAccess`

A pointer to an [access mask](#) that receives the granted access rights. If *AccessStatus* is set to `FALSE`, the function sets the access mask to zero. If the function fails, it does not set the access mask.

[out] `AccessStatus`

A pointer to a variable that receives the results of the access check. If the security descriptor allows the requested access rights to the client, *AccessStatus* is set to `TRUE`. Otherwise, *AccessStatus* is set to `FALSE`.

[out] `pfGenerateOnClose`

A pointer to a flag set by the audit-generation routine when the function returns. Pass this flag to the [ObjectCloseAuditAlarm](#) function when the object handle is closed.

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

For more information, see the [How AccessCheck Works](#) overview.

The **AccessCheckAndAuditAlarm** function requires the calling [process](#) to have the **SE_AUDIT_NAME** privilege enabled. The test for this privilege is performed against the [primary token](#) of the calling process, not the [impersonation token](#) of the thread.

The **AccessCheckAndAuditAlarm** function fails if the calling thread is not impersonating a client.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AccessCheck](#)

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

GENERIC_MAPPING

[How AccessCheck Works](#)

[MakeAbsoluteSD](#)

[MapGenericMask](#)

[ObjectCloseAuditAlarm](#)

[ObjectOpenAuditAlarm](#)

[ObjectPrivilegeAuditAlarm](#)

[PrivilegeCheck](#)

[PrivilegedServiceAuditAlarm](#)

[SECURITY_DESCRIPTOR](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AccessCheckByTypeAndAuditAlarmA function (winbase.h)

Article07/27/2022

The **AccessCheckByTypeAndAuditAlarm** function determines whether a [security descriptor](#) grants a specified set of access rights to the client being impersonated by the calling thread. The function can check the client's access to a hierarchy of objects, such as an object, its property sets, and properties. The function grants or denies access to the hierarchy as a whole. If the security descriptor has a [system access control list](#) (SACL) with [access control entries](#) (ACEs) that apply to the client, the function generates any necessary audit messages in the security event log.

Alarms are not currently supported.

Syntax

C++

```
BOOL AccessCheckByTypeAndAuditAlarmA(
    [in]           LPCSTR             SubsystemName,
    [in]           LPVOID            HandleId,
    [in]           LPCSTR            ObjectTypeName,
    [in, optional] LPCSTR            ObjectName,
    [in]           PSECURITY_DESCRIPTOR SecurityDescriptor,
    [in, optional] PSID              PrincipalSelfSid,
    [in]           DWORD              DesiredAccess,
    [in]           AUDIT_EVENT_TYPE   AuditType,
    [in]           DWORD              Flags,
    [in, out, optional] POBJECT_TYPE_LIST ObjectTypeNameList,
    [in]           DWORD              ObjectTypeNameListLength,
    [in]           PGENERIC_MAPPING   GenericMapping,
    [in]           BOOL               ObjectCreation,
    [out]          LPDWORD            GrantedAccess,
    [out]          LPBOOL             AccessStatus,
    [out]          LPBOOL             pfGenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string that specifies the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in] HandleId

A pointer to a unique value that represents the client's handle to the object. If the access is denied, the system ignores this value.

[in] ObjectTypeName

A pointer to a null-terminated string that specifies the type of object being created or accessed. This string appears in any audit message that the function generates.

[in, optional] ObjectName

A pointer to a null-terminated string that specifies the name of the object being created or accessed. This string appears in any audit message that the function generates.

[in] SecurityDescriptor

A pointer to a [SECURITY_DESCRIPTOR](#) structure against which access is checked.

[in, optional] PrincipalSelfSid

A pointer to a [security identifier](#) (SID). If the security descriptor is associated with an object that represents a principal (for example, a user object), the *PrincipalSelfSid* parameter should be the SID of the object. When evaluating access, this SID logically replaces the SID in any ACE containing the well-known PRINCIPAL_SELF SID (S-1-5-10). For information about well-known SIDs, see [Well-known SIDs](#).

If the protected object does not represent a principal, set this parameter to **NULL**.

[in] DesiredAccess

An [access mask](#) that specifies the access rights to check. This mask must have been mapped by the [MapGenericMask](#) function to contain no generic access rights.

If this parameter is MAXIMUM_ALLOWED, the function sets the *GrantedAccess* access mask to indicate the maximum access rights the security descriptor allows the client.

[in] AuditType

The type of audit to be generated. This can be one of the values from the [AUDIT_EVENT_TYPE](#) enumeration type.

[in] Flags

A flag that controls the function's behavior if the calling [process](#) does not have the SE_AUDIT_NAME privilege enabled. If the AUDIT_ALLOW_NO_PRIVILEGE flag is set, the

function performs the access check without generating audit messages when the privilege is not enabled. If this parameter is zero, the function fails if the privilege is not enabled.

[in, out, optional] *ObjectTypeList*

A pointer to an array of [OBJECT_TYPE_LIST](#) structures that identify the hierarchy of object types for which to check access. Each element in the array specifies a GUID that identifies the object type and a value that indicates the level of the object type in the hierarchy of object types. The array should not have two elements with the same GUID.

The array must have at least one element. The first element in the array must be at level zero and identify the object itself. The array can have only one level zero element. The second element is a subobject, such as a property set, at level 1. Following each level 1 entry are subordinate entries for the level 2 through 4 subobjects. Thus, the levels for the elements in the array might be {0, 1, 2, 2, 1, 2, 3}. If the object type list is out of order, [AccessCheckByTypeAndAuditAlarm](#) fails and [GetLastError](#) returns [ERROR_INVALID_PARAMETER](#).

[in] *ObjectTypeListLength*

The number of elements in the *ObjectTypeList* array.

[in] *GenericMapping*

A pointer to the [GENERIC_MAPPING](#) structure associated with the object for which access is being checked.

[in] *ObjectCreation*

A flag that determines whether the calling application will create a new object when access is granted. A value of **TRUE** indicates the application will create a new object. A value of **FALSE** indicates the application will open an existing object.

[out] *GrantedAccess*

A pointer to an access mask that receives the granted access rights. If *AccessStatus* is set to **FALSE**, the function sets the access mask to zero. If the function fails, it does not set the access mask.

[out] *AccessStatus*

A pointer to a variable that receives the results of the access check. If the security descriptor allows the requested access rights to the client, *AccessStatus* is set to **TRUE**.

Otherwise, *AccessStatus* is set to **FALSE** and you can call [GetLastError](#) to get extended error information.

[out] *pfGenerateOnClose*

A pointer to a flag set by the audit-generation routine when the function returns. Pass this flag to the [ObjectCloseAuditAlarm](#) function when the object handle is closed.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

For more information, see the [How AccessCheck Works](#) overview.

If the *PrincipalSelfSid* and *ObjectTypeList* parameters are **NULL**, the *AuditType* parameter is *AuditEventObjectAccess*, and the *Flags* parameter is zero,

AccessCheckByTypeAndAuditAlarm performs in the same way as the [AccessCheckAndAuditAlarm](#) function.

The *ObjectTypeList* array does not necessarily represent the entire defined object. Rather, it represents that subset of the object for which to check access. For instance, to check access to two properties in a property set, specify an object type list with four elements: the object itself at level zero, the property set at level 1, and the two properties at level 2.

The **AccessCheckByTypeAndAuditAlarm** function evaluates ACEs that apply to the object itself and object-specific ACEs for the object types listed in the *ObjectTypeList* array. The function ignores object-specific ACEs for object types not listed in the *ObjectTypeList* array. Thus, the results returned in the *AccessStatus* parameter indicate the access allowed to the subset of the object defined by the *ObjectTypeList* parameter, not to the entire object.

For more information about how a hierarchy of ACEs controls access to an object and its subobjects, see [ACEs to Control Access to an Object's Properties](#).

To generate audit messages in the security event log, the calling process must have the **SE_AUDIT_NAME** privilege enabled. The system checks for this privilege in the [primary token](#) of the calling process, not the [impersonation token](#) of the thread. If the *Flags*

parameter includes the AUDIT_ALLOW_NO_PRIVILEGE flag, the function performs the access check without generating audit messages when the privilege is not enabled.

The **AccessCheckByTypeAndAuditAlarm** function fails if the calling thread is not impersonating a client.

If the security descriptor does not contain owner and group SIDs, **AccessCheckByTypeAndAuditAlarm** fails with ERROR_INVALID_SECURITY_DESCR.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AUDIT_EVENT_TYPE](#)

[AccessCheck](#)

[AccessCheckAndAuditAlarm](#)

[AccessCheckByType](#)

[AccessCheckByTypeResultList](#)

[AccessCheckByTypeResultListAndAuditAlarm](#)

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[GENERIC_MAPPING](#)

[How AccessCheck Works](#)

[MakeAbsoluteSD](#)

[MapGenericMask](#)

[OBJECT_TYPE_LIST](#)

[ObjectCloseAuditAlarm](#)

[PRIVILEGE_SET](#)

[SECURITY_DESCRIPTOR](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AccessCheckByTypeResultListAndAuditAlarmA function (winbase.h)

Article07/27/2022

The **AccessCheckByTypeResultListAndAuditAlarm** function determines whether a [security descriptor](#) grants a specified set of access rights to the client being impersonated by the calling thread. The function can check access to a hierarchy of objects, such as an object, its property sets, and properties. The function reports the access rights granted or denied to each object type in the hierarchy. If the security descriptor has a [system access control list](#) (SACL) with [access control entries](#) (ACEs) that apply to the client, the function generates any necessary audit messages in the security event log. Alarms are not currently supported.

Syntax

C++

```
BOOL AccessCheckByTypeResultListAndAuditAlarmA(
    [in]           LPCSTR             SubsystemName,
    [in]           LPVOID            HandleId,
    [in]           LPCSTR             ObjectTypeName,
    [in, optional] LPCSTR             ObjectName,
    [in]           PSECURITY_DESCRIPTOR SecurityDescriptor,
    [in, optional] PSID              PrincipalSelfSid,
    [in]           DWORD              DesiredAccess,
    [in]           AUDIT_EVENT_TYPE   AuditType,
    [in]           DWORD              Flags,
    [in, out, optional] POBJECT_TYPE_LIST ObjectTypeNameList,
    [in]           DWORD              ObjectTypeNameListLength,
    [in]           PGENERIC_MAPPING   GenericMapping,
    [in]           BOOL               ObjectCreation,
    [out]          LPDWORD            GrantedAccess,
    [out]          LPDWORD            AccessStatusList,
    [out]          LPBOOL              pfGenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string that specifies the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in] HandleId

A pointer to a unique value that represents the client's handle to the object. If the access is denied, the system ignores this value.

[in] ObjectTypeName

A pointer to a null-terminated string that specifies the type of object being created or accessed. This string appears in any audit message that the function generates.

[in, optional] ObjectName

A pointer to a null-terminated string that specifies the name of the object being created or accessed. This string appears in any audit message that the function generates.

[in] SecurityDescriptor

A pointer to a [SECURITY_DESCRIPTOR](#) structure against which access is checked.

[in, optional] PrincipalSelfSid

A pointer to a [security identifier](#) (SID). If the security descriptor is associated with an object that represents a principal (for example, a user object), the *PrincipalSelfSid* parameter should be the SID of the object. When evaluating access, this SID logically replaces the SID in any ACE that contains the well-known PRINCIPAL_SELF SID (S-1-5-10). For information about well-known SIDs, see [Well-known SIDs](#).

Set this parameter to **NULL** if the protected object does not represent a principal.

[in] DesiredAccess

An [access mask](#) that specifies the access rights to check. This mask must have been mapped by the [MapGenericMask](#) function so that it contains no generic access rights.

If this parameter is MAXIMUM_ALLOWED, the function sets the access mask in *GrantedAccess* to indicate the maximum access rights the security descriptor allows the client.

[in] AuditType

The type of audit to be generated. This can be one of the values from the [AUDIT_EVENT_TYPE](#) enumeration type.

[in] Flags

A flag that controls the function's behavior if the calling [process](#) does not have the SE_AUDIT_NAME privilege enabled. If the AUDIT_ALLOW_NO_PRIVILEGE flag is set, the function performs the access check without generating audit messages when the privilege is not enabled. If this parameter is zero, the function fails if the privilege is not enabled.

[in, out, optional] ObjectTypeList

A pointer to an array of [OBJECT_TYPE_LIST](#) structures that identify the hierarchy of object types for which to check access. Each element in the array specifies a GUID that identifies the object type and a value that indicates the level of the object type in the hierarchy of object types. The array should not have two elements with the same GUID.

The array must have at least one element. The first element in the array must be at level zero and identify the object itself. The array can have only one level zero element. The second element is a subobject, such as a property set, at level 1. Following each level 1 entry are subordinate entries for the level 2 through 4 subobjects. Thus, the levels for the elements in the array might be {0, 1, 2, 2, 1, 2, 3}. If the object type list is out of order, [AccessCheckByTypeResultListAndAuditAlarm](#) fails, and [GetLastError](#) returns [ERROR_INVALID_PARAMETER](#).

[in] ObjectTypeListLength

The number of elements in the *ObjectTypeList* array.

[in] GenericMapping

A pointer to the [GENERIC_MAPPING](#) structure associated with the object for which access is being checked.

[in] ObjectCreation

A flag that determines whether the calling application will create a new object when access is granted. A value of **TRUE** indicates the application will create a new object. A value of **FALSE** indicates the application will open an existing object.

[out] GrantedAccess

A pointer to an array of access masks. The function sets each access mask to indicate the access rights granted to the corresponding element in the object type list. If the function fails, it does not set the access masks.

[out] AccessStatusList

A pointer to an array of status codes for the corresponding elements in the object type list. The function sets an element to zero to indicate success or to a nonzero value to indicate the specific error during the access check. If the function fails, it does not set any of the elements in the array.

[out] `pfGenerateOnClose`

A pointer to a flag set by the audit-generation routine when the function returns. Pass this flag to the [ObjectCloseAuditAlarm](#) function when the object handle is closed.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

For more information, see the [How AccessCheck Works](#) overview.

The **AccessCheckByTypeResultListAndAuditAlarm** function is a combination of the [AccessCheckByTypeResultList](#) and [AccessCheckAndAuditAlarm](#) functions.

The *ObjectTypeList* array does not necessarily represent the entire defined object. Rather, it represents that subset of the object for which to check access. For instance, to check access to two properties in a property set, specify an object type list with four elements: the object itself at level zero, the property set at level 1, and the two properties at level 2.

The **AccessCheckByTypeResultListAndAuditAlarm** function evaluates ACEs that apply to the object itself and object-specific ACEs for the object types listed in the *ObjectTypeList* array. The function ignores object-specific ACEs for object types not listed in the *ObjectTypeList* array.

For more information about how a hierarchy of ACEs controls access to an object and its subobjects, see [ACEs to Control Access to an Object's Properties](#).

To generate audit messages in the security event log, the calling process must have the SE_AUDIT_NAME privilege enabled. The system checks for this privilege in the [primary token](#) of the calling process, not the [impersonation token](#) of the thread. If the *Flags* parameter includes the AUDIT_ALLOW_NO_PRIVILEGE flag, the function performs the access check without generating audit messages when the privilege is not enabled.

The **AccessCheckByTypeResultListAndAuditAlarm** function fails if the calling thread is not impersonating a client.

If the security descriptor does not contain owner and group SIDs, **AccessCheckByTypeResultListAndAuditAlarm** fails with **ERROR_INVALID_SECURITY_DESCR**.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AUDIT_EVENT_TYPE](#)

[AccessCheck](#)

[AccessCheckAndAuditAlarm](#)

[AccessCheckByType](#)

[AccessCheckByTypeResultList](#)

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[GENERIC_MAPPING](#)

[How AccessCheck Works](#)

[MakeAbsoluteSD](#)

[MapGenericMask](#)

OBJECT_TYPE_LIST

ObjectCloseAuditAlarm

PRIVILEGE_SET

SECURITY_DESCRIPTOR

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

AccessCheckByTypeResultListAndAuditAlarmByHandleA function (winbase.h)

Article08/03/2022

The **AccessCheckByTypeResultListAndAuditAlarmByHandle** function determines whether a [security descriptor](#) grants a specified set of access rights to the client that the calling thread is impersonating. The difference between this function and [AccessCheckByTypeResultListAndAuditAlarm](#) is that this function allows the calling thread to perform the access check before impersonating the client.

The function can check access to a hierarchy of objects, such as an object, its property sets, and properties. The function reports the access rights granted or denied to each object type in the hierarchy. If the security descriptor has a [system access control list](#) (SACL) with [access control entries](#) (ACEs) that apply to the client, the function generates any necessary audit messages in the security event log. Alarms are not currently supported.

Syntax

C++

```
BOOL AccessCheckByTypeResultListAndAuditAlarmByHandleA(
    [in]           LPCSTR             SubsystemName,
    [in]           LPVOID              HandleId,
    [in]           HANDLE              ClientToken,
    [in]           LPCSTR              ObjectTypeName,
    [in, optional] LPCSTR              ObjectName,
    [in]           PSECURITY_DESCRIPTOR SecurityDescriptor,
    [in, optional] PSID               PrincipalSelfSid,
    [in]           DWORD               DesiredAccess,
    [in]           AUDIT_EVENT_TYPE   AuditType,
    [in]           DWORD               Flags,
    [in, out, optional] POBJECT_TYPE_LIST ObjectTypeNameList,
    [in]           DWORD               ObjectTypeNameListLength,
    [in]           PGENERIC_MAPPING    GenericMapping,
    [in]           BOOL                ObjectCreation,
    [out]          LPDWORD             GrantedAccess,
    [out]          LPDWORD             AccessStatusList,
    [out]          LPBOOL              pfGenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string that specifies the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in] HandleId

A pointer to a unique value that represents the client's handle to the object. If the access is denied, the system ignores this value.

[in] ClientToken

A handle to a token object that represents the client that requested the operation. This handle must be obtained through a communication session layer, such as a local named pipe, to prevent possible security policy violations. The caller must have TOKEN_QUERY access for the specified token.

[in] ObjectTypeName

A pointer to a null-terminated string that specifies the type of object being created or accessed. This string appears in any audit message that the function generates.

[in, optional] ObjectName

A pointer to a null-terminated string that specifies the name of the object being created or accessed. This string appears in any audit message that the function generates.

[in] SecurityDescriptor

A pointer to a [SECURITY_DESCRIPTOR](#) structure against which access is checked.

[in, optional] PrincipalSelfSid

A pointer to a SID. If the security descriptor is associated with an object that represents a principal (for example, a user object), the *PrincipalSelfSid* parameter should be the SID of the object. When evaluating access, this SID logically replaces the SID in any ACE containing the well-known PRINCIPAL_SELF SID (S-1-5-10). For information about well-known SIDs, see [Well-known SIDs](#).

Set this parameter to **NULL** if the protected object does not represent a principal.

[in] DesiredAccess

An [access mask](#) that specifies the access rights to check. This mask must have been mapped by the [MapGenericMask](#) function so that it contains no generic access rights.

If this parameter is MAXIMUM_ALLOWED, the function sets the access mask in *GrantedAccess* to indicate the maximum access rights the security descriptor allows the client.

[in] `AuditType`

The type of audit to be generated. This can be one of the values from the [AUDIT_EVENT_TYPE](#) enumeration type.

[in] `Flags`

A flag that controls the function's behavior if the calling [process](#) does not have the SE_AUDIT_NAME privilege enabled. If the AUDIT_ALLOW_NO_PRIVILEGE flag is set, the function performs the access check without generating audit messages when the privilege is not enabled. If this parameter is zero, the function fails if the privilege is not enabled.

[in, out, optional] `ObjectTypeList`

A pointer to an array of [OBJECT_TYPE_LIST](#) structures that identify the hierarchy of object types for which to check access. Each element in the array specifies a GUID that identifies the object type and a value that indicates the level of the object type in the hierarchy of object types. The array should not have two elements with the same GUID.

The array must have at least one element. The first element in the array must be at level zero and identify the object itself. The array can have only one level zero element. The second element is a subobject, such as a property set, at level 1. Following each level 1 entry are subordinate entries for the level 2 through 4 subobjects. Thus, the levels for the elements in the array might be {0, 1, 2, 2, 1, 2, 3}. If the object type list is out of order, [AccessCheckByTypeResultListAndAuditAlarmByHandle](#) fails, and [GetLastError](#) returns ERROR_INVALID_PARAMETER.

[in] `ObjectTypeListLength`

The number of elements in the *ObjectTypeList* array.

[in] `GenericMapping`

A pointer to the [GENERIC_MAPPING](#) structure associated with the object for which access is being checked.

[in] `ObjectCreation`

A flag that determines whether the calling application will create a new object when access is granted. A value of TRUE indicates the application will create a new object. A

value of **FALSE** indicates the application will open an existing object.

[out] GrantedAccess

A pointer to an array of access masks. The function sets each access mask to indicate the access rights granted to the corresponding element in the object type list. If the function fails, it does not set the access masks.

[out] AccessStatusList

A pointer to an array of status codes for the corresponding elements in the object type list. The function sets an element to zero to indicate success or to a nonzero value to indicate the specific error during the access check. If the function fails, it does not set any of the elements in the array.

[out] pfGenerateOnClose

A pointer to a flag set by the audit-generation routine when the function returns. Pass this flag to the [ObjectCloseAuditAlarm](#) function when the object handle is closed.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

For more information, see the [How AccessCheck Works](#) overview.

Like [AccessCheckByTypeResultListAndAuditAlarm](#), the [AccessCheckByTypeResultListAndAuditAlarmByHandle](#) function is a combination of the [AccessCheckByTypeResultList](#) and [AccessCheckAndAuditAlarm](#) functions. However, [AccessCheckByTypeResultListAndAuditAlarmByHandle](#) also requires a client token handle to provide security information on the client.

The *ObjectTypeList* array does not necessarily represent the entire defined object. Rather, it represents that subset of the object for which to check access. For instance, to check access to two properties in a property set, specify an object type list with four elements: the object itself at level zero, the property set at level 1, and the two properties at level 2.

The **AccessCheckByTypeResultListAndAuditAlarmByHandle** function evaluates ACEs that apply to the object itself and object-specific ACEs for the object types listed in the *ObjectTypeList* array. The function ignores object-specific ACEs for object types not listed in the *ObjectTypeList* array.

For more information about how a hierarchy of ACEs controls access to an object and its subobjects, see [ACEs to Control Access to an Object's Properties](#).

To generate audit messages in the security event log, the calling process must have the SE_AUDIT_NAME privilege enabled. The system checks for this privilege in the [primary token](#) of the calling process, not the [impersonation token](#) of the thread. If the *Flags* parameter includes the AUDIT_ALLOW_NO_PRIVILEGE flag, the function performs the access check without generating audit messages when the privilege is not enabled.

The **AccessCheckByTypeResultListAndAuditAlarmByHandle** function fails if the calling thread is not impersonating a client.

If the security descriptor does not contain owner and group SIDs, **AccessCheckByTypeResultListAndAuditAlarmByHandle** fails with ERROR_INVALID_SECURITY_DESCR.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AUDIT_EVENT_TYPE](#)

[AccessCheck](#)

[AccessCheckAndAuditAlarm](#)

[AccessCheckByType](#)

[AccessCheckByTypeResultList](#)

[AccessCheckByTypeResultListAndAuditAlarm](#)

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[GENERIC_MAPPING](#)

[How AccessCheck Works](#)

[MakeAbsoluteSD](#)

[MapGenericMask](#)

[OBJECT_TYPE_LIST](#)

[ObjectCloseAuditAlarm](#)

[PRIVILEGE_SET](#)

[SECURITY_DESCRIPTOR](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ACTCTX_SECTION_KEYED_DATA structure (winbase.h)

Article04/02/2021

The ACTCTX_SECTION_KEYED_DATA structure is used by the [FindActCtxSectionString](#) and [FindActCtxSectionGuid](#) functions to return the activation context information along with either the GUID or 32-bit integer-tagged activation context section.

Syntax

C++

```
typedef struct tagACTCTX_SECTION_KEYED_DATA {
    ULONG                      cbSize;
    ULONG                      ulDataFormatVersion;
    PVOID                      lpData;
    ULONG                      ulLength;
    PVOID                      lpSectionGlobalData;
    ULONG                      ulSectionGlobalDataLength;
    PVOID                      lpSectionBase;
    ULONG                      ulSectionTotalLength;
    HANDLE                     hActCtx;
    ULONG                      ulAssemblyRosterIndex;
    ULONG                      ulFlags;
    ACTCTX_SECTION_KEYED_DATA_ASSEMBLY_METADATA AssemblyMetadata;
} ACTCTX_SECTION_KEYED_DATA, *PACTCTX_SECTION_KEYED_DATA;
```

Members

`cbSize`

The size, in bytes, of the activation context keyed data structure.

`ulDataFormatVersion`

Number that indicates the format of the data in the section where the key was found. Clients should verify that the data format version is as expected rather than trying to interpret the values of unfamiliar data formats. This number is only changed when major non-backward-compatible changes to the section data formats need to be made. The current format version is 1.

`lpData`

Pointer to the redirection data found associated with the section identifier and key.

`ulLength`

Number of bytes in the structure referred to by **IpData**. Note that the data structures grow over time; do not access members in the instance data that extend beyond **ulLength**.

`lpSectionGlobalData`

Returned pointer to a section-specific data structure which is global to the activation context section where the key was found. Its interpretation depends on the section identifier requested.

`ulSectionGlobalDataLength`

Number of bytes in the section global data block referred to by **IpSectionGlobalData**.

Note that the data structures grow over time and you may receive an old format activation context data block; do not access members in the section global data that extend beyond **ulSectionGlobalDataLength**.

`lpSectionBase`

Pointer to the base of the section where the key was found. Some instance data contains offsets relative to the section base address, in which case this pointer value is used.

`ulSectionTotalLength`

Number of bytes for the entire section starting at **IpSectionBase**. May be used to verify that offset/length pairs, which are specified as relative to the section base are wholly contained in the section.

`hActCtx`

Handle to the activation context where the key was found. First, the active activation context for the thread is searched, followed by the process-default activation context and then the system-compatible-default-activation context. This member indicates which activation context contained the section and key requested. This is only returned if the FIND_ACTCTX_SECTION_KEY_RETURN_HACTCTX flag is passed.

Note that when this is returned, the caller must call [ReleaseActCtx\(\)](#) on the activation context handle returned to release system resources when all other references to the activation context have been released.

`ulAssemblyRosterIndex`

Cardinal number of the assembly in the activation context that provided the redirection information found. This value can be presented to [QueryActCtxW](#) for more information about the contributing assembly.

`ulFlags`

`AssemblyMetadata`

Remarks

Callers should initialize the `ACTCTX_SECTION_KEYED_DATA` structure as such:

```
"ACTCTX_SECTION_KEYED_DATA askd = { sizeof(askd) };"
```

which initializes all members to zero/null except the size field which is set correctly.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[ACTCTX](#)

[FindActCtxSectionGuid](#)

[FindActCtxSectionString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ACTCTXA structure (winbase.h)

Article 07/27/2022

The ACTCTX structure is used by the [CreateActCtx](#) function to create the activation context.

Syntax

C++

```
typedef struct tagACTCTXA {
    ULONG    cbSize;
    DWORD    dwFlags;
    LPCSTR   lpSource;
    USHORT   wProcessorArchitecture;
    LANGID   wLangId;
    LPCSTR   lpAssemblyDirectory;
    LPCSTR   lpResourceName;
    LPCSTR   lpApplicationName;
    HMODULE  hModule;
} ACTCTXA, *PACTCTXA;
```

Members

`cbSize`

The size, in bytes, of this structure. This is used to determine the version of this structure.

`dwFlags`

Flags that indicate how the values included in this structure are to be used. Set any undefined bits in `dwFlags` to 0. If any undefined bits are not set to 0, the call to [CreateActCtx](#) that creates the activation context fails and returns an invalid parameter error code.

Bit flag	Meaning
ACTCTX_FLAG_PROCESSOR_ARCHITECTURE_VALID 1	0x001
ACTCTX_FLAG_LANGID_VALID 2	0x002
ACTCTX_FLAG_ASSEMBLY_DIRECTORY_VALID	0x004

4

ACTCTX_FLAG_RESOURCE_NAME_VALID	0x008
8	
ACTCTX_FLAG_SET_PROCESS_DEFAULT	0x010
16	
ACTCTX_FLAG_APPLICATION_NAME_VALID	0x020
32	
ACTCTX_FLAG_HMODULE_VALID	0x080
128	

lpSource

Null-terminated string specifying the path of the manifest file or PE image to be used to create the activation context. If this path refers to an EXE or DLL file, the **lpResourceName** member is required.

wProcessorArchitecture

Identifies the type of processor used. Specifies the system's processor architecture.

This value can be one of the following values:

wLangId

Specifies the language manifest that should be used. The default is the current user's current UI language.

If the requested language cannot be found, an approximation is searched for using the following order:

- The current user's specific language. For example, for US English (1033).
- The current user's primary language. For example, for English (9).
- The current system's specific language.
- The current system's primary language.
- A nonspecific worldwide language. Language neutral (0).

lpAssemblyDirectory

The base directory in which to perform private assembly probing if assemblies in the activation context are not present in the system-wide store.

lpResourceName

Pointer to a null-terminated string that contains the resource name to be loaded from the PE specified in **hModule** or **IpSource**. If the resource name is an integer, set this member using MAKEINTRESOURCE. This member is required if **IpSource** refers to an EXE or DLL.

lpApplicationName

The name of the current application. If the value of this member is set to null, the name of the executable that launched the current process is used.

hModule

Use this member rather than **IpSource** if you have already loaded a DLL and wish to use it to create activation contexts rather than using a path in **IpSource**. See **IpResourceName** for the rules of looking up resources in this module.

Remarks

If the file identified by the value of the **IpSource** member is a PE image file, [CreateActCtx](#) searches for the manifest in the .manifest file located in the same directory and in the first RT_MANIFEST resource located in the PE image file. To find a specific named resource from the image, set the **IpResourceName** to the name of the resource, and add the ACTCTX_FLAG_RESOURCE_NAME_VALID to the **dwFlags** member. Refer to [FindResource](#) for more information on specifying resource names.

In most cases, the caller should not set the ACTCTX_FLAG_PROCESSOR_ARCHITECTURE_VALID and ACTCTX_FLAG_LANGID_VALID flags of the **dwFlags** member. Also, in most cases, the value of the **IpResourceName** member should be set to null.

The values of **IpApplicationName** and **IpAssemblyDirectory** are not set to null when the executable creating the activation context is a host for the application. In this case, the host can set a different name for the application to find configuration files, report errors, and so forth.

Note

The winbase.h header defines ACTCTX as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[ACTCTX_SECTION_KEYED_DATA](#)

[CreateActCtx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ACTCTXW structure (winbase.h)

Article 07/27/2022

The ACTCTX structure is used by the [CreateActCtx](#) function to create the activation context.

Syntax

C++

```
typedef struct tagACTCTXW {
    ULONG    cbSize;
    DWORD    dwFlags;
    LPCWSTR  lpSource;
    USHORT   wProcessorArchitecture;
    LANGID   wLangId;
    LPCWSTR  lpAssemblyDirectory;
    LPCWSTR  lpResourceName;
    LPCWSTR  lpApplicationName;
    HMODULE  hModule;
} ACTCTXW, *PACTCTXW;
```

Members

`cbSize`

The size, in bytes, of this structure. This is used to determine the version of this structure.

`dwFlags`

Flags that indicate how the values included in this structure are to be used. Set any undefined bits in `dwFlags` to 0. If any undefined bits are not set to 0, the call to [CreateActCtx](#) that creates the activation context fails and returns an invalid parameter error code.

Bit flag	Meaning
ACTCTX_FLAG_PROCESSOR_ARCHITECTURE_VALID 1	0x001
ACTCTX_FLAG_LANGID_VALID 2	0x002
ACTCTX_FLAG_ASSEMBLY_DIRECTORY_VALID	0x004

4

ACTCTX_FLAG_RESOURCE_NAME_VALID	0x008
8	
ACTCTX_FLAG_SET_PROCESS_DEFAULT	0x010
16	
ACTCTX_FLAG_APPLICATION_NAME_VALID	0x020
32	
ACTCTX_FLAG_HMODULE_VALID	0x080
128	

lpSource

Null-terminated string specifying the path of the manifest file or PE image to be used to create the activation context. If this path refers to an EXE or DLL file, the **lpResourceName** member is required.

wProcessorArchitecture

Identifies the type of processor used. Specifies the system's processor architecture.

This value can be one of the following values:

wLangId

Specifies the language manifest that should be used. The default is the current user's current UI language.

If the requested language cannot be found, an approximation is searched for using the following order:

- The current user's specific language. For example, for US English (1033).
- The current user's primary language. For example, for English (9).
- The current system's specific language.
- The current system's primary language.
- A nonspecific worldwide language. Language neutral (0).

lpAssemblyDirectory

The base directory in which to perform private assembly probing if assemblies in the activation context are not present in the system-wide store.

lpResourceName

Pointer to a null-terminated string that contains the resource name to be loaded from the PE specified in **hModule** or **IpSource**. If the resource name is an integer, set this member using MAKEINTRESOURCE. This member is required if **IpSource** refers to an EXE or DLL.

lpApplicationName

The name of the current application. If the value of this member is set to null, the name of the executable that launched the current process is used.

hModule

Use this member rather than **IpSource** if you have already loaded a DLL and wish to use it to create activation contexts rather than using a path in **IpSource**. See **IpResourceName** for the rules of looking up resources in this module.

Remarks

If the file identified by the value of the **IpSource** member is a PE image file, [CreateActCtx](#) searches for the manifest in the .manifest file located in the same directory and in the first RT_MANIFEST resource located in the PE image file. To find a specific named resource from the image, set the **IpResourceName** to the name of the resource, and add the ACTCTX_FLAG_RESOURCE_NAME_VALID to the **dwFlags** member. Refer to [FindResource](#) for more information on specifying resource names.

In most cases, the caller should not set the ACTCTX_FLAG_PROCESSOR_ARCHITECTURE_VALID and ACTCTX_FLAG_LANGID_VALID flags of the **dwFlags** member. Also, in most cases, the value of the **IpResourceName** member should be set to null.

The values of **IpApplicationName** and **IpAssemblyDirectory** are not set to null when the executable creating the activation context is a host for the application. In this case, the host can set a different name for the application to find configuration files, report errors, and so forth.

Note

The winbase.h header defines ACTCTX as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[ACTCTX_SECTION_KEYED_DATA](#)

[CreateActCtx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ActivateActCtx function (winbase.h)

Article10/13/2021

The **ActivateActCtx** function activates the specified activation context. It does this by pushing the specified activation context to the top of the activation stack. The specified activation context is thus associated with the current thread and any appropriate side-by-side API functions.

Syntax

C++

```
BOOL ActivateActCtx(
    [in]    HANDLE     hActCtx,
    [out]   ULONG_PTR *lpCookie
);
```

Parameters

[in] hActCtx

Handle to an [ACTCTX](#) structure that contains information on the activation context that is to be made active.

[out] lpCookie

Pointer to a **ULONG_PTR** that functions as a cookie, uniquely identifying a specific, activated activation context.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

The *lpCookie* parameter is later passed to **DeactivateActCtx**, which verifies the pairing of calls to **ActivateActCtx** and **DeactivateActCtx** and ensures that the appropriate activation context is being deactivated. This is done because the deactivation of activation contexts must occur in the reverse order of activation.

The activation of activation contexts can be understood as pushing an activation context onto a stack of activation contexts. The activation context you activate through this function redirects any binding to DLLs, window classes, COM servers, type libraries, and mutexes for any side-by-side APIs you call.

The top item of an activation context stack is the active, default-activation context of the current thread. If a null activation context handle is pushed onto the stack, thereby activating it, the default settings in the original manifest override all activation contexts that are lower on the stack.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

[DeactivateActCtx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AddAtomA function (winbase.h)

Article 02/09/2023

Adds a character string to the local atom table and returns a unique value (an atom) identifying the string.

Syntax

C++

```
ATOM AddAtomA(  
    [in] LPCSTR lpString  
);
```

Parameters

[in] lpString

Type: LPCTSTR

The null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings differing only in case are considered identical. The case of the first string added is preserved and returned by the [GetAtomName](#) function.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

Return value

Type: ATOM

If the function succeeds, the return value is the newly created atom.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The [AddAtom](#) function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom and, in

the case of a string atom, increments the string's reference count.

If *lpString* has the form "#1234", **AddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000 or is greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **AddAtom** returns a string atom.

 **Note**

The winbase.h header defines AddAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see **Conventions for Function Prototypes**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[MAKEINTATOM](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AddAtomW function (winbase.h)

Article 02/09/2023

Adds a character string to the local atom table and returns a unique value (an atom) identifying the string.

Syntax

C++

```
ATOM AddAtomW(
    [in] LPCWSTR lpString
);
```

Parameters

[in] lpString

Type: LPCTSTR

The null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings differing only in case are considered identical. The case of the first string added is preserved and returned by the [GetAtomName](#) function.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

Return value

Type: ATOM

If the function succeeds, the return value is the newly created atom.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The [AddAtom](#) function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom and, in

the case of a string atom, increments the string's reference count.

If *lpString* has the form "#1234", **AddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000 or is greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **AddAtom** returns a string atom.

 **Note**

The winbase.h header defines AddAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see **Conventions for Function Prototypes**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[MAKEINTATOM](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AddConditionalAce function (winbase.h)

Article10/13/2021

The **AddConditionalAce** function adds a conditional [access control entry](#) (ACE) to the specified [access control list](#) (ACL). A conditional ACE specifies a logical condition that is evaluated during access checks.

Syntax

C++

```
BOOL AddConditionalAce(
    [in, out] PACL    pAcl,
    [in]      DWORD   dwAceRevision,
    [in]      DWORD   AceFlags,
    [in]      UCHAR   AceType,
    [in]      DWORD   AccessMask,
    [in]      PSID    pSid,
    [in]      PWCHAR  ConditionStr,
    [out]     DWORD   *ReturnLength
);
```

Parameters

[in, out] pAcl

A pointer to an ACL. This function adds an ACE to this ACL.

The value of this parameter cannot be **NULL**.

[in] dwAceRevision

Specifies the revision level of the ACL being modified. This value can be **ACL_REVISION** or **ACL_REVISION_DS**. Use **ACL_REVISION_DS** if the ACL contains object-specific ACEs.

[in] AceFlags

A set of bit flags that control ACE inheritance. The function sets these flags in the **AceFlags** member of the [ACE_HEADER](#) structure of the new ACE. This parameter can be a combination of the following values.

Value	Meaning
CONTAINER_INHERIT_ACE	The ACE is inherited by container objects.

INHERIT_ONLY_ACE	The ACE does not apply to the object to which the ACL is assigned, but it can be inherited by child objects.
INHERITED_ACE	Indicates an inherited ACE. This flag allows operations that change the security on a tree of objects to modify inherited ACEs while not changing ACEs that were directly applied to the object.
NO_PROPAGATE_INHERIT_ACE	The OBJECT_INHERIT_ACE and CONTAINER_INHERIT_ACE bits are not propagated to an inherited ACE.
OBJECT_INHERIT_ACE	The ACE is inherited by noncontainer objects.

[in] AceType

The type of the ACE.

This can be one of the following values.

Value	Meaning
ACCESS_ALLOWED_CALLBACK_ACE_TYPE 0x9	Access-allowed callback ACE that uses the ACCESS_ALLOWED_CALLBACK_ACE structure.
ACCESS_DENIED_CALLBACK_ACE_TYPE 0xA	Access-denied callback ACE that uses the ACCESS_DENIED_CALLBACK_ACE structure.
SYSTEM_AUDIT_CALLBACK_ACE_TYPE 0xD	System audit callback ACE that uses the SYSTEM_AUDIT_CALLBACK_ACE structure.

[in] AccessMask

Specifies the mask of access rights to be granted to the specified SID.

[in] pSid

A pointer to the SID that represents a user, group, or logon account being granted access.

[in] ConditionStr

A string that specifies the conditional statement to be evaluated for the ACE.

[out] ReturnLength

The size, in bytes, of the ACL. If the buffer specified by the *pACL* parameter is not of sufficient size, the value of this parameter is the required size.

Return value

If the function succeeds, it returns **TRUE**.

If the function fails, it returns **FALSE**. For extended error information, call [GetLastError](#).

The following are possible error values.

Return code	Description
ERROR_INSUFFICIENT_BUFFER	The new ACE does not fit into the <i>pAcl</i> buffer.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AddIntegrityLabelToBoundaryDescriptor function (winbase.h)

Article10/13/2021

Adds a new required security identifier (SID) to the specified boundary descriptor.

Syntax

C++

```
BOOL AddIntegrityLabelToBoundaryDescriptor(
    [in, out] HANDLE *BoundaryDescriptor,
    [in]      PSID   IntegrityLabel
);
```

Parameters

[in, out] BoundaryDescriptor

A handle to the boundary descriptor. The [CreateBoundaryDescriptor](#) function returns this handle.

[in] IntegrityLabel

A pointer to a [SID](#) structure that represents the mandatory integrity level for the namespace. Use one of the following RID values to create the SID:

SECURITY_MANDATORY_UNTRUSTED_RID **SECURITY_MANDATORY_LOW_RID**
SECURITY_MANDATORY_MEDIUM_RID **SECURITY_MANDATORY_SYSTEM_RID**
SECURITY_MANDATORY_PROTECTED_PROCESS_RID For more information, see [Well-Known SIDs](#).

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

A process can create a private namespace only with an integrity level that is equal to or lower than the current integrity level of the process. Therefore, a high integrity-level process can create a high, medium or low integrity-level namespace. A medium integrity-level process can create only a medium or low integrity-level namespace.

A process would usually specify a namespace at the same integrity level as the process for protection against squatting attacks by lower integrity-level processes.

The security descriptor that the creator places on the namespace determines who can open the namespace. So a low or medium integrity-level process could be given permission to open a high integrity level namespace if the security descriptor of the namespace permits it.

To compile an application that uses this function, define _WIN32_WINNT as 0x0601 or later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateBoundaryDescriptor](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

AddRefActCtx function (winbase.h)

Article10/13/2021

The **AddRefActCtx** function increments the reference count of the specified activation context.

Syntax

C++

```
void AddRefActCtx(  
    [in] HANDLE hActCtx  
);
```

Parameters

[in] hActCtx

Handle to an [ACTCTX](#) structure that contains information on the activation context for which the reference count is to be incremented.

Return value

None

Remarks

This function is provided so that multiple clients can access a single activation context.

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)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

AddSecureMemoryCacheCallback function (winbase.h)

Article 10/13/2021

Registers a callback function to be called when a secured memory range is freed or its protections are changed.

Syntax

C++

```
BOOL AddSecureMemoryCacheCallback(
    [in] PSECURE_MEMORY_CACHE_CALLBACK pfnCallBack
);
```

Parameters

[in] `pfnCallBack`

A pointer to the application-defined [SecureMemoryCacheCallback](#) function to register.

Return value

If the function succeeds, it registers the callback function and returns **TRUE**.

If the function fails, it returns **FALSE**. To get extended error information, call the [GetLastError](#) function.

Remarks

An application that performs I/O directly to a high-performance device typically caches a virtual-to-physical memory mapping for the buffer it uses for the I/O. The device's driver typically secures this memory address range by calling the [MmSecureVirtualMemory](#) routine, which prevents the memory range from being freed or its protections changed until the driver unsecures the memory.

An application can use **AddSecureMemoryCacheCallback** to register a callback function that will be called when the memory is freed or its protections are changed, so the

application can invalidate its cached memory mapping. For more information, see [SecureMemoryCacheCallback](#).

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows Vista with SP1 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[RemoveSecureMemoryCacheCallback](#)

[SecureMemoryCacheCallback](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ApplicationRecoveryFinished function (winbase.h)

Article10/13/2021

Indicates that the calling application has completed its data recovery.

Syntax

C++

```
void ApplicationRecoveryFinished(
    [in] BOOL bSuccess
);
```

Parameters

[in] bSuccess

Specify **TRUE** to indicate that the data was successfully recovered; otherwise, **FALSE**.

Return value

None

Remarks

This should be the last call that you make in your callback because your application terminates as soon as this function is called.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ApplicationRecoveryInProgress](#)

[RegisterApplicationRecoveryCallback](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

ApplicationRecoveryInProgress function (winbase.h)

Article 10/13/2021

Indicates that the calling application is continuing to recover data.

Syntax

C++

```
HRESULT ApplicationRecoveryInProgress(
    [out] PBOOL pbCancelled
);
```

Parameters

[out] pbCancelled

Indicates whether the user has canceled the recovery process. Set by WER if the user clicks the Cancel button.

Return value

This function returns **S_OK** on success or one of the following error codes.

Return code	Description
E_FAIL	You can call this function only after Windows Error Reporting has called your recovery callback function.
E_INVALIDARG	The <i>pbCancelled</i> cannot be NULL .

Remarks

The application must call this function within the interval specified when calling the [RegisterApplicationRecoveryCallback](#) function. If the application fails to call this function within the specified interval, WER terminates the application. The recovery process can continue as long as this function is being called.

If the user cancels the recovery process, the application should terminate.

To indicate that the recovery process has been completed, call the [ApplicationRecoveryFinished](#) function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ApplicationRecoveryFinished](#)

[RegisterApplicationRecoveryCallback](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BackupEventLogA function (winbase.h)

Article02/09/2023

Saves the specified event log to a backup file. The function does not clear the event log.

Syntax

C++

```
BOOL BackupEventLogA(
    [in] HANDLE hEventLog,
    [in] LPCSTR lpBackupFileName
);
```

Parameters

[in] hEventLog

A handle to the open event log. The [OpenEventLog](#) function returns this handle.

[in] lpBackupFileName

The absolute or relative path of the backup file.

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

The **BackupEventLog** function fails with the **ERROR_PRIVILEGE_NOT_HELD** error if the user does not have the **SE_BACKUP_NAME** privilege.

Note

The winbase.h header defines **BackupEventLog** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the

UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[OpenBackupEventLog](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BackupEventLogW function (winbase.h)

Article02/09/2023

Saves the specified event log to a backup file. The function does not clear the event log.

Syntax

C++

```
BOOL BackupEventLogW(
    [in] HANDLE hEventLog,
    [in] LPCWSTR lpBackupFileName
);
```

Parameters

[in] hEventLog

A handle to the open event log. The [OpenEventLog](#) function returns this handle.

[in] lpBackupFileName

The absolute or relative path of the backup file.

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

The **BackupEventLog** function fails with the **ERROR_PRIVILEGE_NOT_HELD** error if the user does not have the **SE_BACKUP_NAME** privilege.

Note

The winbase.h header defines **BackupEventLog** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the

UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[OpenBackupEventLog](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BackupRead function (winbase.h)

Article10/13/2021

The **BackupRead** function can be used to back up a file or directory, including the security information. The function reads data associated with a specified file or directory into a buffer, which can then be written to the backup medium using the [WriteFile](#) function.

Syntax

C++

```
BOOL BackupRead(
    [in] HANDLE hFile,
    [out] LPBYTE lpBuffer,
    [in] DWORD nNumberOfBytesToRead,
    [out] LPDWORD lpNumberOfBytesRead,
    [in] BOOL bAbort,
    [in] BOOL bProcessSecurity,
    [out] LPVOID *lpContext
);
```

Parameters

[in] `hFile`

Handle to the file or directory to be backed up. To obtain the handle, call the [CreateFile](#) function. The SACLs are not read unless the file handle was created with the **ACCESS_SYSTEM_SECURITY** access right. For more information, see [File security and access rights](#).

The handle must be synchronous (nonoverlapped). This means that the **FILE_FLAG_OVERLAPPED** flag must not be set when [CreateFile](#) is called. This function does not validate that the handle it receives is synchronous, so it does not return an error code for a synchronous handle, but calling it with an asynchronous (overlapped) handle can result in subtle errors that are very difficult to debug.

The **BackupRead** function may fail if [CreateFile](#) was called with the flag **FILE_FLAG_NO_BUFFERING**. In this case, the [GetLastError](#) function returns the value **ERROR_INVALID_PARAMETER**.

[out] `lpBuffer`

Pointer to a buffer that receives the data.

[in] `nNumberOfBytesToRead`

Length of the buffer, in bytes. The buffer size must be greater than the size of a [WIN32_STREAM_ID](#) structure.

[out] `lpNumberOfBytesRead`

Pointer to a variable that receives the number of bytes read.

If the function returns a nonzero value, and the variable pointed to by *lpNumberOfBytesRead* is zero, then all the data associated with the file handle has been read.

[in] `bAbort`

Indicates whether you have finished using **BackupRead** on the handle. While you are backing up the file, specify this parameter as **FALSE**. Once you are done using **BackupRead**, you must call **BackupRead** one more time specifying **TRUE** for this parameter and passing the appropriate *lpContext*. *lpContext* must be passed when *bAbort* is **TRUE**; all other parameters are ignored.

[in] `bProcessSecurity`

Indicates whether the function will restore the access-control list (ACL) data for the file or directory.

If *bProcessSecurity* is **TRUE**, the ACL data will be backed up.

[out] `lpContext`

Pointer to a variable that receives a pointer to an internal data structure used by **BackupRead** to maintain context information during a backup operation.

You must set the variable pointed to by *lpContext* to **NULL** before the first call to **BackupRead** for the specified file or directory. The function allocates memory for the data structure, and then sets the variable to point to that structure. You must not change *lpContext* or the variable that it points to between calls to **BackupRead**.

To release the memory used by the data structure, call **BackupRead** with the *bAbort* parameter set to **TRUE** when the backup operation is complete.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero, indicating that an I/O error occurred. To get extended error information, call [GetLastError](#).

Remarks

This function is not intended for use in backing up files encrypted under the Encrypted File System. Use [ReadEncryptedFileRaw](#) for that purpose.

If an error occurs while **BackupRead** is reading data, the calling process can skip the bad data by calling the [BackupSeek](#) function.

The file or directory should be restored using the [BackupWrite](#) function.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BackupSeek](#)

[BackupWrite](#)

[Creating a Backup Application](#)

[ReadEncryptedFileRaw](#)

[WIN32_STREAM_ID](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BackupSeek function (winbase.h)

Article10/13/2021

The **BackupSeek** function seeks forward in a data stream initially accessed by using the [BackupRead](#) or [BackupWrite](#) function.

Syntax

C++

```
BOOL BackupSeek(
    [in]  HANDLE  hFile,
    [in]  DWORD   dwLowBytesToSeek,
    [in]  DWORD   dwHighBytesToSeek,
    [out] LPDWORD lpdwLowByteSeeked,
    [out] LPDWORD lpdwHighByteSeeked,
    [in]  LPVOID   *lpContext
);
```

Parameters

[in] `hFile`

Handle to the file or directory. This handle is created by using the [CreateFile](#) function.

The handle must be synchronous (nonoverlapped). This means that the `FILE_FLAG_OVERLAPPED` flag must not be set when [CreateFile](#) is called. This function does not validate that the handle it receives is synchronous, so it does not return an error code for a synchronous handle, but calling it with an asynchronous (overlapped) handle can result in subtle errors that are very difficult to debug.

[in] `dwLowBytesToSeek`

Low-order part of the number of bytes to seek.

[in] `dwHighBytesToSeek`

High-order part of the number of bytes to seek.

[out] `lpdwLowByteSeeked`

Pointer to a variable that receives the low-order bits of the number of bytes the function actually seeks.

[out] *lpdwHighByteSeeked*

Pointer to a variable that receives the high-order bits of the number of bytes the function actually seeks.

[in] *lpContext*

Pointer to an internal data structure used by the function. This structure must be the same structure that was initialized by the [BackupRead](#) or [BackupWrite](#) function. An application must not touch the contents of this structure.

Return value

If the function could seek the requested amount, the function returns a nonzero value.

If the function could not seek the requested amount, the function returns zero. To get extended error information, call [GetLastError](#).

Remarks

Applications use the [BackupSeek](#) function to skip portions of a data stream that cause errors. This function does not seek across stream headers. For example, this function cannot be used to skip the stream name. If an application attempts to seek past the end of a substream, the function fails, the *lpdwLowByteSeeked* and *lpdwHighByteSeeked* parameters indicate the actual number of bytes the function seeks, and the file position is placed at the start of the next stream header.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BackupRead](#)

[BackupWrite](#)

[CreateFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BackupWrite function (winbase.h)

Article10/13/2021

The **BackupWrite** function can be used to restore a file or directory that was backed up using [BackupRead](#). Use the [ReadFile](#) function to get a stream of data from the backup medium, then use **BackupWrite** to write the data to the specified file or directory.

Syntax

C++

```
BOOL BackupWrite(
    [in] HANDLE hFile,
    [in] LPBYTE lpBuffer,
    [in] DWORD nNumberOfBytesToWrite,
    [out] LPDWORD lpNumberOfBytesWritten,
    [in] BOOL bAbort,
    [in] BOOL bProcessSecurity,
    [out] LPVOID *lpContext
);
```

Parameters

[in] hFile

Handle to the file or directory to be restored. To obtain the handle, call the [CreateFile](#) function. The SACLs are not restored unless the file handle was created with the **ACCESS_SYSTEM_SECURITY** access right. To ensure that the integrity ACEs are restored correctly, the file handle must also have been created with the **WRITE_OWNER** access right. For more information, see [File security and access rights](#).

The handle must be synchronous (nonoverlapped). This means that the **FILE_FLAG_OVERLAPPED** flag must not be set when [CreateFile](#) is called. This function does not validate that the handle it receives is synchronous, so it does not return an error code for a synchronous handle, but calling it with an asynchronous (overlapped) handle can result in subtle errors that are very difficult to debug.

The **BackupWrite** function may fail if [CreateFile](#) was called with the flag **FILE_FLAG_NO_BUFFERING**. In this case, the [GetLastError](#) function returns the value **ERROR_INVALID_PARAMETER**.

[in] lpBuffer

Pointer to a buffer that the function writes data from.

[in] `nNumberOfBytesToWrite`

Size of the buffer, in bytes. The buffer size must be greater than the size of a [WIN32_STREAM_ID](#) structure.

[out] `lpNumberOfBytesWritten`

Pointer to a variable that receives the number of bytes written.

[in] `bAbort`

Indicates whether you have finished using **BackupWrite** on the handle. While you are restoring the file, specify this parameter as **FALSE**. After you are done using **BackupWrite**, you must call **BackupWrite** one more time specifying **TRUE** for this parameter and passing the appropriate *lpContext*. *lpContext* must be passed when *bAbort* is **TRUE**; all other parameters are ignored.

[in] `bProcessSecurity`

Specifies whether the function will restore the access-control list (ACL) data for the file or directory.

If *bProcessSecurity* is **TRUE**, you need to specify **WRITE_OWNER** and **WRITE_DAC** access when opening the file or directory handle. If the handle does not have those access rights, the operating system denies access to the ACL data, and ACL data restoration will not occur.

[out] `lpContext`

Pointer to a variable that receives a pointer to an internal data structure used by **BackupWrite** to maintain context information during a restore operation.

You must set the variable pointed to by *lpContext* to **NULL** before the first call to **BackupWrite** for the specified file or directory. The function allocates memory for the data structure, and then sets the variable to point to that structure. You must not change *lpContext* or the variable that it points to between calls to **BackupWrite**.

To release the memory used by the data structure, call **BackupWrite** with the *bAbort* parameter set to **TRUE** when the restore operation is complete.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero, indicating that an I/O error occurred. To get extended error information, call [GetLastError](#).

Remarks

This function is not intended for use in restoring files encrypted under the [Encrypted File System](#). Use [WriteEncryptedFileRaw](#) for that purpose.

The data read from the backup medium must be substreams separated by [WIN32_STREAM_ID](#) structures.

The **BACKUP_LINK** stream type lets you restore files with hard links.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BackupRead](#)

[BackupSeek](#)

[CreateFile](#)

[WIN32_STREAM_ID](#)

[WriteEncryptedFileRaw](#)

Feedback



Was this page helpful?  

Get help at Microsoft Q&A

BeginUpdateResourceA function (winbase.h)

Article02/09/2023

Retrieves a handle that can be used by the [UpdateResource](#) function to add, delete, or replace resources in a binary module.

Syntax

C++

```
HANDLE BeginUpdateResourceA(
    [in] LPCSTR pFileName,
    [in] BOOL    bDeleteExistingResources
);
```

Parameters

[in] *pFileName*

Type: **LPCTSTR**

The binary file in which to update resources. An application must be able to obtain write-access to this file; the file referenced by *pFileName* cannot be currently executing. If *pFileName* does not specify a full path, the system searches for the file in the current directory.

[in] *bDeleteExistingResources*

Type: **BOOL**

Indicates whether to delete the *pFileName* parameter's existing resources. If this parameter is **TRUE**, existing resources are deleted and the updated file includes only resources added with the [UpdateResource](#) function. If this parameter is **FALSE**, the updated file includes existing resources unless they are explicitly deleted or replaced by using [UpdateResource](#).

Return value

Type: **HANDLE**

If the function succeeds, the return value is a handle that can be used by the [UpdateResource](#) and [EndUpdateResource](#) functions. The return value is **NULL** if the specified file is not a PE, the file does not exist, or the file cannot be opened for writing. To get extended error information, call [GetLastError](#).

Remarks

It is recommended that the resource file is not loaded before this function is called. However, if that file is already loaded, it will not cause an error to be returned.

There are some restrictions on resource updates in files that contain Resource Configuration(RC Config) data: LN files and the associated .mui files. Details on which types of resources are allowed to be updated in these files are in the Remarks section for the [UpdateResource](#) function.

This function can update resources within modules that contain both code and resources. As noted above, there are restrictions on resource updates in LN files and .mui files, both of which contain RC Config data; details of the restrictions are in the reference for the [UpdateResource](#) function.

Examples

For an example see, [Updating Resources](#).

ⓘ Note

The winbase.h header defines BeginUpdateResource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[EndUpdateResource](#)

[Reference](#)

[Resources](#)

[UpdateResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BeginUpdateResourceW function (winbase.h)

Article02/09/2023

Retrieves a handle that can be used by the [UpdateResource](#) function to add, delete, or replace resources in a binary module.

Syntax

C++

```
HANDLE BeginUpdateResourceW(
    [in] LPCWSTR pFileName,
    [in] BOOL     bDeleteExistingResources
);
```

Parameters

[in] *pFileName*

Type: **LPCTSTR**

The binary file in which to update resources. An application must be able to obtain write-access to this file; the file referenced by *pFileName* cannot be currently executing. If *pFileName* does not specify a full path, the system searches for the file in the current directory.

[in] *bDeleteExistingResources*

Type: **BOOL**

Indicates whether to delete the *pFileName* parameter's existing resources. If this parameter is **TRUE**, existing resources are deleted and the updated file includes only resources added with the [UpdateResource](#) function. If this parameter is **FALSE**, the updated file includes existing resources unless they are explicitly deleted or replaced by using [UpdateResource](#).

Return value

Type: **HANDLE**

If the function succeeds, the return value is a handle that can be used by the [UpdateResource](#) and [EndUpdateResource](#) functions. The return value is **NULL** if the specified file is not a PE, the file does not exist, or the file cannot be opened for writing. To get extended error information, call [GetLastError](#).

Remarks

It is recommended that the resource file is not loaded before this function is called. However, if that file is already loaded, it will not cause an error to be returned.

There are some restrictions on resource updates in files that contain Resource Configuration(RC Config) data: LN files and the associated .mui files. Details on which types of resources are allowed to be updated in these files are in the Remarks section for the [UpdateResource](#) function.

This function can update resources within modules that contain both code and resources. As noted above, there are restrictions on resource updates in LN files and .mui files, both of which contain RC Config data; details of the restrictions are in the reference for the [UpdateResource](#) function.

Examples

For an example see, [Updating Resources](#).

ⓘ Note

The winbase.h header defines BeginUpdateResource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[EndUpdateResource](#)

[Reference](#)

[Resources](#)

[UpdateResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BindIoCompletionCallback function (winbase.h)

Article03/11/2023

Associates the I/O completion port owned by the [thread pool](#) with the specified file handle. On completion of an I/O request involving this file, a non-I/O worker thread will execute the specified callback function.

Syntax

C++

```
BOOL BindIoCompletionCallback(
    [in] HANDLE FileHandle,
    [in] LPOVERLAPPED_COMPLETION_ROUTINE Function,
    [in] ULONG Flags
);
```

Parameters

[in] FileHandle

A handle to the file opened for overlapped I/O completion. This handle is returned by the [CreateFile](#) function, with the **FILE_FLAG_OVERLAPPED** flag.

[in] Function

A pointer to the callback function to be executed in a non-I/O worker thread when the I/O operation is complete. This callback function must not call the [TerminateThread](#) function.

For more information about the completion routine, see [FileIOCompletionRoutine](#).

[in] Flags

This parameter must be zero.

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 the [GetLastError](#) function.

Remarks

The callback function might not be executed if the process issues an asynchronous request on the file specified by the *FileHandle* parameter but the request returns immediately with an error code other than ERROR_IO_PENDING.

Be sure that the thread that initiates the asynchronous I/O request does not terminate before the request is completed. Also, if a function in a DLL is queued to a worker thread, be sure that the function in the DLL has completed execution before the DLL is unloaded.

The thread pool maintains an I/O completion port. When you call [BindIoCompletionCallback](#), it associates the specified file with the thread pool's I/O completion port. Asynchronous requests on that file object will complete by posting to the completion port, where they will be picked up by thread pool worker threads. For callbacks that must issue an I/O request that completes as an asynchronous procedure call, the thread pool provides an I/O worker pool. The I/O worker threads do not wait on the completion port; they sleep in an alertable wait state so that I/O request packets that complete can wake them up. Both types of worker threads check whether there is I/O pending on them and if there is, they do not exit. For more information, see [Asynchronous Procedure Calls](#).

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see [Using the Windows Headers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FileIOCompletionRoutine](#)

[Process and Thread Functions](#)

[Thread Pooling](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BuildCommDCBA function (winbase.h)

Article 02/09/2023

Fills a specified [DCB](#) structure with values specified in a device-control string. The device-control string uses the syntax of the **mode** command.

Syntax

C++

```
BOOL BuildCommDCBA(
    [in]  LPCSTR lpDef,
    [out] LPDCB  lpDCB
);
```

Parameters

[in] *lpDef*

The device-control information. The function takes this string, parses it, and then sets appropriate values in the [DCB](#) structure pointed to by *lpDCB*.

The string must have the same form as the **mode** command's command-line arguments:

COMX[:][baud=*b*][parity=*p*][data=*d*][stop=*s*][to={on|off}][xon={on|off}][odsr={on|off}][octs={on|off}][dtr={on|off|hs}][rts={on|off|hs|tg}][idsr={on|off}]

The device name is optional, but it must specify a valid device if used.

For example, the following string specifies a baud rate of 1200, no parity, 8 data bits, and 1 stop bit:

baud=1200 parity=N data=8 stop=1

[out] *lpDCB*

A pointer to a [DCB](#) structure that receives the information.

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

The **BuildCommDCB** function adjusts only those members of the **DCB** structure that are specifically affected by the *lpDef* parameter, with the following exceptions:

- If the specified baud rate is 110, the function sets the stop bits to 2 to remain compatible with the system's **mode** command.
- By default, **BuildCommDCB** disables XON/XOFF and hardware flow control. To enable flow control, you must explicitly set the appropriate members of the **DCB** structure.

The **BuildCommDCB** function only fills in the members of the **DCB** structure. To apply these settings to a serial port, use the [SetCommState](#) function.

There are older and newer forms of the **mode** syntax. The **BuildCommDCB** function supports both forms. However, you cannot mix the two forms together.

The newer form of the **mode** syntax lets you explicitly set the values of the flow control members of the **DCB** structure. If you use an older form of the **mode** syntax, the **BuildCommDCB** function sets the flow control members of the **DCB** structure, as follows:

- For a string that does not end with an x or a p:
 - **fInX**, **fOutX**, **fOutXDsrFlow**, and **fOutXCtsFlow** are all set to **FALSE**
 - **fDtrControl** is set to **DTR_CONTROL_ENABLE**
 - **fRtsControl** is set to **RTS_CONTROL_ENABLE**
- For a string that ends with an x:
 - **fInX** and **fOutX** are both set to **TRUE**
 - **fOutXDsrFlow** and **fOutXCtsFlow** are both set to **FALSE**
 - **fDtrControl** is set to **DTR_CONTROL_ENABLE**
 - **fRtsControl** is set to **RTS_CONTROL_ENABLE**
- For a string that ends with a p:
 - **fInX** and **fOutX** are both set to **FALSE**
 - **fOutXDsrFlow** and **fOutXCtsFlow** are both set to **TRUE**
 - **fDtrControl** is set to **DTR_CONTROL_HANDSHAKE**
 - **fRtsControl** is set to **RTS_CONTROL_HANDSHAKE**

 **Note**

The winbase.h header defines BuildCommDCB as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[DCB](#)

[SetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BuildCommDCBAndTimeoutsA function (winbase.h)

Article02/09/2023

Translates a device-definition string into appropriate device-control block codes and places them into a device control block. The function can also set up time-out values, including the possibility of no time-outs, for a device; the function's behavior in this regard depends on the contents of the device-definition string.

Syntax

C++

```
BOOL BuildCommDCBAndTimeoutsA(
    [in]    LPCSTR      lpDef,
    [out]   LPDCB       lpDCB,
    [out]   LPCOMMTIMEOUTS lpCommTimeouts
);
```

Parameters

[in] *lpDef*

The device-control information. The function takes this string, parses it, and then sets appropriate values in the [DCB](#) structure pointed to by *lpDCB*.

The string must have the same form as the **mode** command's command-line arguments:

```
COMx[:][baud={11|110|15|150|30|300|60|600|12|1200|24|2400|48|4800|96|9600|19|19200}]
[parity={n|e|o|m|s}][data={5|6|7|8}][stop={1|1.5|2}][to={on|off}][xon={on|off}][odsr=
{on|off}][octs={on|off}][dtr={on|off|hs}][rts={on|off|hs|tg}][idsr={on|off}]
```

The "baud" substring can be any of the values listed, which are in pairs. The two-digit values are the first two digits of the associated values that they represent. For example, 11 represents 110 baud, 19 represents 19,200 baud.

The "parity" substring indicates how the parity bit is used to detect transmission errors. The values represent "none", "even", "odd", "mark", and "space".

For more information, see the [Mode](#) command reference in TechNet.

For example, the following string specifies a baud rate of 1200, no parity, 8 data bits, and 1 stop bit:

```
baud=1200 parity=N data=8 stop=1
```

```
[out] lpDCB
```

A pointer to a [DCB](#) structure that receives information from the device-control information string pointed to by *lpDef*. This **DCB** structure defines the control settings for a communications device.

```
[out] lpCommTimeouts
```

A pointer to a [COMMTIMEOUTS](#) structure that receives time-out information.

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

The [BuildCommDCBAndTimeouts](#) function modifies its time-out setting behavior based on the presence or absence of a "to={on|off}" substring in *lpDef*.

- If that string contains the substring "to=on", the function sets the [WriteTotalTimeoutConstant](#) member of the [COMMTIMEOUTS](#) structure to 60000 and all other members to 0.
- If that string contains the substring "to=off", the function sets the members of [COMMTIMEOUTS](#) to 0.
- If that string does not specify a "to={on|off}" substring, the function ignores the [COMMTIMEOUTS](#) structure in *lpCommTimeouts*.

For more information, see the Remarks for the [BuildCommDCB](#) function.

Note

The winbase.h header defines [BuildCommDCBAndTimeouts](#) as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BuildCommDCB](#)

[COMMTIMEOUTS](#)

[Communications Functions](#)

[Communications Resources](#)

[DCB](#)

[GetCommTimeouts](#)

[SetCommTimeouts](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BuildCommDCBAndTimeoutsW function (winbase.h)

Article02/09/2023

Translates a device-definition string into appropriate device-control block codes and places them into a device control block. The function can also set up time-out values, including the possibility of no time-outs, for a device; the function's behavior in this regard depends on the contents of the device-definition string.

Syntax

C++

```
BOOL BuildCommDCBAndTimeoutsW(
    [in]    LPCWSTR      lpDef,
    [out]   LPDCB        lpDCB,
    [out]   LPCOMMTIMEOUTS lpCommTimeouts
);
```

Parameters

[in] *lpDef*

The device-control information. The function takes this string, parses it, and then sets appropriate values in the **DCB** structure pointed to by *lpDCB*.

The string must have the same form as the **mode** command's command-line arguments:

```
COMx[:][baud={11|110|15|150|30|300|60|600|12|1200|24|2400|48|4800|96|9600|19|19200}]
[parity={n|e|o|m|s}][data={5|6|7|8}][stop={1|1.5|2}][to={on|off}][xon={on|off}][odsr=
{on|off}][octs={on|off}][dtr={on|off|hs}][rts={on|off|hs|tg}][idsr={on|off}]
```

The "baud" substring can be any of the values listed, which are in pairs. The two-digit values are the first two digits of the associated values that they represent. For example, 11 represents 110 baud, 19 represents 19,200 baud.

The "parity" substring indicates how the parity bit is used to detect transmission errors. The values represent "none", "even", "odd", "mark", and "space".

For more information, see the [Mode](#) command reference in TechNet.

For example, the following string specifies a baud rate of 1200, no parity, 8 data bits, and 1 stop bit:

```
baud=1200 parity=N data=8 stop=1
```

```
[out] lpDCB
```

A pointer to a [DCB](#) structure that receives information from the device-control information string pointed to by *lpDef*. This **DCB** structure defines the control settings for a communications device.

```
[out] lpCommTimeouts
```

A pointer to a [COMMTIMEOUTS](#) structure that receives time-out information.

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

The [BuildCommDCBAndTimeouts](#) function modifies its time-out setting behavior based on the presence or absence of a "to={on|off}" substring in *lpDef*.

- If that string contains the substring "to=on", the function sets the [WriteTotalTimeoutConstant](#) member of the [COMMTIMEOUTS](#) structure to 60000 and all other members to 0.
- If that string contains the substring "to=off", the function sets the members of [COMMTIMEOUTS](#) to 0.
- If that string does not specify a "to={on|off}" substring, the function ignores the [COMMTIMEOUTS](#) structure in *lpCommTimeouts*.

For more information, see the Remarks for the [BuildCommDCB](#) function.

Note

The winbase.h header defines [BuildCommDCBAndTimeouts](#) as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BuildCommDCB](#)

[COMMTIMEOUTS](#)

[Communications Functions](#)

[Communications Resources](#)

[DCB](#)

[GetCommTimeouts](#)

[SetCommTimeouts](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

BuildCommDCBW function (winbase.h)

Article 02/09/2023

Fills a specified [DCB](#) structure with values specified in a device-control string. The device-control string uses the syntax of the **mode** command.

Syntax

C++

```
BOOL BuildCommDCBW(
    [in]  LPCWSTR lpDef,
    [out] LPDCB   lpDCB
);
```

Parameters

[in] *lpDef*

The device-control information. The function takes this string, parses it, and then sets appropriate values in the [DCB](#) structure pointed to by *lpDCB*.

The string must have the same form as the **mode** command's command-line arguments:

COMX[:][baud=*b*][parity=*p*][data=*d*][stop=*s*][to={on|off}][xon={on|off}][odsr={on|off}][octs={on|off}][dtr={on|off|hs}][rts={on|off|hs|tg}][idsr={on|off}]

The device name is optional, but it must specify a valid device if used.

For example, the following string specifies a baud rate of 1200, no parity, 8 data bits, and 1 stop bit:

baud=1200 parity=N data=8 stop=1

[out] *lpDCB*

A pointer to a [DCB](#) structure that receives the information.

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

The **BuildCommDCB** function adjusts only those members of the **DCB** structure that are specifically affected by the *lpDef* parameter, with the following exceptions:

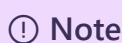
- If the specified baud rate is 110, the function sets the stop bits to 2 to remain compatible with the system's **mode** command.
- By default, **BuildCommDCB** disables XON/XOFF and hardware flow control. To enable flow control, you must explicitly set the appropriate members of the **DCB** structure.

The **BuildCommDCB** function only fills in the members of the **DCB** structure. To apply these settings to a serial port, use the [SetCommState](#) function.

There are older and newer forms of the **mode** syntax. The **BuildCommDCB** function supports both forms. However, you cannot mix the two forms together.

The newer form of the **mode** syntax lets you explicitly set the values of the flow control members of the **DCB** structure. If you use an older form of the **mode** syntax, the **BuildCommDCB** function sets the flow control members of the **DCB** structure, as follows:

- For a string that does not end with an x or a p:
 - **fInX**, **fOutX**, **fOutXDsrFlow**, and **fOutXCtsFlow** are all set to **FALSE**
 - **fDtrControl** is set to **DTR_CONTROL_ENABLE**
 - **fRtsControl** is set to **RTS_CONTROL_ENABLE**
- For a string that ends with an x:
 - **fInX** and **fOutX** are both set to **TRUE**
 - **fOutXDsrFlow** and **fOutXCtsFlow** are both set to **FALSE**
 - **fDtrControl** is set to **DTR_CONTROL_ENABLE**
 - **fRtsControl** is set to **RTS_CONTROL_ENABLE**
- For a string that ends with a p:
 - **fInX** and **fOutX** are both set to **FALSE**
 - **fOutXDsrFlow** and **fOutXCtsFlow** are both set to **TRUE**
 - **fDtrControl** is set to **DTR_CONTROL_HANDSHAKE**
 - **fRtsControl** is set to **RTS_CONTROL_HANDSHAKE**



Note

The winbase.h header defines BuildCommDCB as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[DCB](#)

[SetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CallNamedPipeA function (winbase.h)

Article 02/02/2023

Connects to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe.

Syntax

C++

```
BOOL CallNamedPipeA(
    [in]    LPCSTR    lpNamedPipeName,
    [in]    LPVOID    lpInBuffer,
    [in]    DWORD     nInBufferSize,
    [out]   LPVOID    lpOutBuffer,
    [in]    DWORD     nOutBufferSize,
    [out]   LPDWORD   lpBytesRead,
    [in]    DWORD     nTimeOut
);
```

Parameters

[in] lpNamedPipeName

The pipe name.

[in] lpInBuffer

The data to be written to the pipe.

[in] nInBufferSize

The size of the write buffer, in bytes.

[out] lpOutBuffer

A pointer to the buffer that receives the data read from the pipe.

[in] nOutBufferSize

The size of the read buffer, in bytes.

[out] lpBytesRead

A pointer to a variable that receives the number of bytes read from the pipe.

[in] nTimeOut

The number of milliseconds to wait for the named pipe to be available. In addition to numeric values, the following special values can be specified.

Value	Meaning
NMPWAIT_NOWAIT 0x00000001	Does not wait for the named pipe. If the named pipe is not available, the function returns an error.
NMPWAIT_WAIT_FOREVER 0xffffffff	Waits indefinitely.
NMPWAIT_USE_DEFAULT_WAIT 0x00000000	Uses the default time-out specified in a call to the CreateNamedPipe 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](#).

If the message written to the pipe by the server process is longer than *nOutBufferSize*, **CallNamedPipe** returns **FALSE**, and [GetLastError](#) returns **ERROR_MORE_DATA**. The remainder of the message is discarded, because **CallNamedPipe** closes the handle to the pipe before returning.

Remarks

Calling **CallNamedPipe** is equivalent to calling the [CreateFile](#) (or [WaitNamedPipe](#), if [CreateFile](#) cannot open the pipe immediately), [TransactNamedPipe](#), and [CloseHandle](#) functions. [CreateFile](#) is called with an access flag of **GENERIC_READ** | **GENERIC_WRITE**, and an inherit handle flag of **FALSE**.

CallNamedPipe fails if the pipe is a byte-type pipe.

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\pipe\LOCAL\` for the pipe name.

Examples

For an example, see [Transactions on Named Pipes](#).

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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFile](#)

[CreateNamedPipe](#)

[Pipe Functions](#)

[Pipes Overview](#)

[TransactNamedPipe](#)

[WaitNamedPipe](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CheckNameLegalDOS8Dot3A function (winbase.h)

Article 02/09/2023

Determines whether the specified name can be used to create a file on a FAT file system.

Syntax

C++

```
BOOL CheckNameLegalDOS8Dot3A(
    [in]          LPCSTR lpName,
    [out, optional] LPSTR  lpOemName,
    [in]          DWORD   OemNameSize,
    [out, optional] PBOOL  pbNameContainsSpaces,
    [out]          PBOOL  pbNameLegal
);
```

Parameters

[in] lpName

The file name, in 8.3 format.

[out, optional] lpOemName

A pointer to a buffer that receives the OEM string that corresponds to *Name*. This parameter can be **NULL**.

[in] OemNameSize

The size of the *lpOemName* buffer, in characters. If *lpOemName* is **NULL**, this parameter must be 0 (zero).

[out, optional] pbNameContainsSpaces

Indicates whether or not a name contains spaces. This parameter can be **NULL**. If the name is not a valid 8.3 FAT file system name, this parameter is undefined.

[out] pbNameLegal

If the function succeeds, this parameter indicates whether a file name is a valid 8.3 FAT file name when the current OEM code page is applied to the file name.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

This function can be used to determine whether or not a file name can be passed to a 16-bit Windows-based application or an MS-DOS-based application.

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

Note that SMB 3.0 does not support short names on shares with continuous availability capability, so function will always return zero (fail).

Note

The winbase.h header defines CheckNameLegalDOS8Dot3 as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetOEMCP](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CheckNameLegalDOS8Dot3W function (winbase.h)

Article02/09/2023

Determines whether the specified name can be used to create a file on a FAT file system.

Syntax

C++

```
BOOL CheckNameLegalDOS8Dot3W(
    [in]          LPCWSTR lpName,
    [out, optional] LPSTR   lpOemName,
    [in]          DWORD    OemNameSize,
    [out, optional] PBOOL   pbNameContainsSpaces,
    [out]          PBOOL   pbNameLegal
);
```

Parameters

[in] lpName

The file name, in 8.3 format.

[out, optional] lpOemName

A pointer to a buffer that receives the OEM string that corresponds to *Name*. This parameter can be **NULL**.

[in] OemNameSize

The size of the *lpOemName* buffer, in characters. If *lpOemName* is **NULL**, this parameter must be 0 (zero).

[out, optional] pbNameContainsSpaces

Indicates whether or not a name contains spaces. This parameter can be **NULL**. If the name is not a valid 8.3 FAT file system name, this parameter is undefined.

[out] pbNameLegal

If the function succeeds, this parameter indicates whether a file name is a valid 8.3 FAT file name when the current OEM code page is applied to the file name.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

This function can be used to determine whether or not a file name can be passed to a 16-bit Windows-based application or an MS-DOS-based application.

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

Note that SMB 3.0 does not support short names on shares with continuous availability capability, so function will always return zero (fail).

Note

The winbase.h header defines CheckNameLegalDOS8Dot3 as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetOEMCP](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ClearCommBreak function (winbase.h)

Article10/13/2021

Restores character transmission for a specified communications device and places the transmission line in a nonbreak state.

Syntax

C++

```
BOOL ClearCommBreak(
    [in] HANDLE hFile
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

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

A communications device is placed in a break state by the [SetCommBreak](#) or [EscapeCommFunction](#) function. Character transmission is then suspended until the break state is cleared by calling [ClearCommBreak](#).

Requirements

Minimum supported client

Windows XP [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ClearCommError](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[EscapeCommFunction](#)

[SetCommBreak](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

ClearCommError function (winbase.h)

Article10/13/2021

Retrieves information about a communications error and reports the current status of a communications device. The function is called when a communications error occurs, and it clears the device's error flag to enable additional input and output (I/O) operations.

Syntax

C++

```
BOOL ClearCommError(
    [in]          HANDLE     hFile,
    [out, optional] LPDWORD   lpErrors,
    [out, optional] LPCOMSTAT lpStat
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[out, optional] lpErrors

A pointer to a variable that receives a mask indicating the type of error. This parameter can be one or more of the following values.

Value	Meaning
CE_BREAK 0x0010	The hardware detected a break condition.
CE_FRAME 0x0008	The hardware detected a framing error.
CE_OVERRUN 0x0002	A character-buffer overrun has occurred. The next character is lost.
CE_RXOVER 0x0001	An input buffer overflow has occurred. There is either no room in the input buffer, or a character was received after the end-of-file (EOF) character.
CE_RXPARITY	The hardware detected a parity error.

The following values are not supported:

[out, optional] *lpStat*

A pointer to a [COMSTAT](#) structure in which the device's status information is returned. If this parameter is **NULL**, no status information is returned.

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

If a communications port has been set up with a **TRUE** value for the **fAbortOnError** member of the setup [DCB](#) structure, the communications software will terminate all read and write operations on the communications port when a communications error occurs. No new read or write operations will be accepted until the application acknowledges the communications error by calling the [ClearCommError](#) function.

The [ClearCommError](#) function fills the status buffer pointed to by the *lpStat* parameter with the current status of the communications device specified by the *hFile* parameter.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMSTAT](#)

[ClearCommBreak](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[DCB](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ClearEventLogA function (winbase.h)

Article02/09/2023

Clears the specified event log, and optionally saves the current copy of the log to a backup file.

Syntax

C++

```
BOOL ClearEventLogA(
    [in] HANDLE hEventLog,
    [in] LPCSTR lpBackupFileName
);
```

Parameters

[in] `hEventLog`

A handle to the event log to be cleared. The [OpenEventLog](#) function returns this handle.

[in] `lpBackupFileName`

The absolute or relative path of the backup file. If this file already exists, the function fails.

If the `lpBackupFileName` parameter is **NULL**, the event log is not backed up.

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](#). The **ClearEventLog** function can fail if the event log is empty or the backup file already exists.

Remarks

After this function returns, any handles that reference the cleared event log cannot be used to read the log.

ⓘ Note

The winbase.h header defines ClearEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[Event Logging Functions](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ClearEventLogW function (winbase.h)

Article02/09/2023

Clears the specified event log, and optionally saves the current copy of the log to a backup file.

Syntax

C++

```
BOOL ClearEventLogW(
    [in] HANDLE hEventLog,
    [in] LPCWSTR lpBackupFileName
);
```

Parameters

[in] `hEventLog`

A handle to the event log to be cleared. The [OpenEventLog](#) function returns this handle.

[in] `lpBackupFileName`

The absolute or relative path of the backup file. If this file already exists, the function fails.

If the `lpBackupFileName` parameter is **NULL**, the event log is not backed up.

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](#). The **ClearEventLog** function can fail if the event log is empty or the backup file already exists.

Remarks

After this function returns, any handles that reference the cleared event log cannot be used to read the log.

ⓘ Note

The winbase.h header defines ClearEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[Event Logging Functions](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CloseEncryptedFileRaw function (winbase.h)

Article10/13/2021

Closes an encrypted file after a backup or restore operation, and frees associated system resources. This is one of a group of Encrypted File System (EFS) functions that is intended to implement backup and restore functionality, while maintaining files in their encrypted state.

Syntax

C++

```
void CloseEncryptedFileRaw(
    [in] PVOID pvContext
);
```

Parameters

[in] `pvContext`

A pointer to a system-defined context block. The [OpenEncryptedFileRaw](#) function returns the context block.

Return value

None

Remarks

The `CloseEncryptedFileRaw` function frees allocated system resources such as the system-defined context block and closes the file.

The [BackupRead](#) and [BackupWrite](#) functions handle backup and restore of unencrypted files.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support EFS on shares with continuous availability capability.

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)

See also

[BackupRead](#)

[BackupWrite](#)

[File Encryption](#)

[File Management Functions](#)

[OpenEncryptedFileRaw](#)

[ReadEncryptedFileRaw](#)

[WriteEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CloseEventLog function (winbase.h)

Article07/27/2022

Closes the specified event log.

Syntax

C++

```
BOOL CloseEventLog(  
    [in, out] HANDLE hEventLog  
)
```

Parameters

[in, out] hEventLog

A handle to the event log to be closed. The [OpenEventLog](#) or [OpenBackupEventLog](#) function returns this handle.

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](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

API set

ext-ms-win-advapi32-eventlog-l1-1-0 (introduced in Windows 8)

See also

[Event Logging Functions](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

COMMCONFIG structure (winbase.h)

Article09/01/2022

Contains information about the configuration state of a communications device.

Syntax

C++

```
typedef struct _COMMCONFIG {
    DWORD dwSize;
    WORD wVersion;
    WORD wReserved;
    DCB dcb;
    DWORD dwProviderSubType;
    DWORD dwProviderOffset;
    DWORD dwProviderSize;
    WCHAR wcProviderData[1];
} COMMCONFIG, *LPCOMMCONFIG;
```

Members

`dwSize`

The size of the structure, in bytes. The caller must set this member to `sizeof(COMMCONFIG)`.

`wVersion`

The version number of the structure. This parameter can be 1. The version of the provider-specific structure should be included in the `wcProviderData` member.

`wReserved`

Reserved; do not use.

`dcb`

The device-control block ([DCB](#)) structure for RS-232 serial devices. A **DCB** structure is always present regardless of the port driver subtype specified in the device's [COMMPROP](#) structure.

`dwProviderSubType`

The type of communications provider, and thus the format of the provider-specific data. For a list of communications provider types, see the description of the [COMMPROP](#) structure.

`dwProviderOffset`

The offset of the provider-specific data relative to the beginning of the structure, in bytes. This member is zero if there is no provider-specific data.

`dwProviderSize`

The size of the provider-specific data, in bytes.

`wcProviderData[1]`

Optional provider-specific data. This member can be of any size or can be omitted. Because the [COMMCONFIG](#) structure may be expanded in the future, applications should use the `dwProviderOffset` member to determine the location of this member.

Remarks

If the provider subtype is [PST_RS232](#) or [PST_PARALLELPORT](#), the `wcProviderData` member is omitted. If the provider subtype is [PST_MODEM](#), the `wcProviderData` member contains a [MODEMSETTINGS](#) structure.

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Header	winbase.h (include Windows.h)

See also

[COMMPROP](#)

[DCB](#)

[GetCommProperties](#)

[MODEMSETTINGS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CommConfigDialogA function (winbase.h)

Article02/09/2023

Displays a driver-supplied configuration dialog box.

Syntax

C++

```
BOOL CommConfigDialogA(
    [in]      LPCSTR      lpszName,
    [in]      HWND        hWnd,
    [in, out] LPCOMMCONFIG lpCC
);
```

Parameters

[in] lpszName

The name of the device for which a dialog box should be displayed. For example, COM1 through COM9 are serial ports and LPT1 through LPT9 are parallel ports.

[in] hWnd

A handle to the window that owns the dialog box. This parameter can be any valid window handle, or it should be **NULL** if the dialog box is to have no owner.

[in, out] lpCC

A pointer to a [COMMCONFIG](#) structure. This structure contains initial settings for the dialog box before the call, and changed values after the call.

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

The **CommConfigDialog** function requires a dynamic-link library (DLL) provided by the communications hardware vendor.

ⓘ Note

The winbase.h header defines **CommConfigDialog** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

CommConfigDialogW function (winbase.h)

Article02/09/2023

Displays a driver-supplied configuration dialog box.

Syntax

C++

```
BOOL CommConfigDialogW(
    [in]     LPCWSTR     lpszName,
    [in]     HWND        hWnd,
    [in, out] LPCOMMCONFIG lpCC
);
```

Parameters

[in] lpszName

The name of the device for which a dialog box should be displayed. For example, COM1 through COM9 are serial ports and LPT1 through LPT9 are parallel ports.

[in] hWnd

A handle to the window that owns the dialog box. This parameter can be any valid window handle, or it should be **NULL** if the dialog box is to have no owner.

[in, out] lpCC

A pointer to a [COMMCONFIG](#) structure. This structure contains initial settings for the dialog box before the call, and changed values after the call.

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

The **CommConfigDialog** function requires a dynamic-link library (DLL) provided by the communications hardware vendor.

ⓘ Note

The winbase.h header defines **CommConfigDialog** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

COMMPROP structure (winbase.h)

Article09/01/2022

Contains information about a communications driver.

Syntax

C++

```
typedef struct _COMMPROP {
    WORD    wPacketLength;
    WORD    wPacketVersion;
    DWORD   dwServiceMask;
    DWORD   dwReserved1;
    DWORD   dwMaxTxQueue;
    DWORD   dwMaxRxQueue;
    DWORD   dwMaxBaud;
    DWORD   dwProvSubType;
    DWORD   dwProvCapabilities;
    DWORD   dwSettableParams;
    DWORD   dwSettableBaud;
    WORD    wSettableData;
    WORD    wSettableStopParity;
    DWORD   dwCurrentTxQueue;
    DWORD   dwCurrentRxQueue;
    DWORD   dwProvSpec1;
    DWORD   dwProvSpec2;
    WCHAR   wcProvChar[1];
} COMMPROP, *LPCOMMPROP;
```

Members

wPacketLength

The size of the entire data packet, regardless of the amount of data requested, in bytes.

wPacketVersion

The version of the structure.

dwServiceMask

A bitmask indicating which services are implemented by this provider. The **SP_SERIALCOMM** value is always specified for communications providers, including modem providers.

dwReserved1

Reserved; do not use.

dwMaxTxQueue

The maximum size of the driver's internal output buffer, in bytes. A value of zero indicates that no maximum value is imposed by the serial provider.

dwMaxRxQueue

The maximum size of the driver's internal input buffer, in bytes. A value of zero indicates that no maximum value is imposed by the serial provider.

dwMaxBaud

The maximum allowable baud rate, in bits per second (bps). This member can be one of the following values.

Value	Meaning
BAUD_075 0x00000001	75 bps
BAUD_110 0x00000002	110 bps
BAUD_134_5 0x00000004	134.5 bps
BAUD_150 0x00000008	150 bps
BAUD_300 0x00000010	300 bps
BAUD_600 0x00000020	600 bps
BAUD_1200 0x00000040	1200 bps
BAUD_1800 0x00000080	1800 bps
BAUD_2400 0x00000100	2400 bps
BAUD_4800 0x00000200	4800 bps

BAUD_7200	7200 bps
0x00000400	
BAUD_9600	9600 bps
0x00000800	
BAUD_14400	14400 bps
0x00001000	
BAUD_19200	19200 bps
0x00002000	
BAUD_38400	38400 bps
0x00004000	
BAUD_56K	56K bps
0x00008000	
BAUD_57600	57600 bps
0x00040000	
BAUD_115200	115200 bps
0x00020000	
BAUD_128K	128K bps
0x00010000	
BAUD_USER	Programmable baud rate.
0x10000000	

dwProvSubType

The communications-provider type.

Value	Meaning
PST_FAX	FAX device
0x00000021	
PST_LAT	LAT protocol
0x00000101	
PST_MODEM	Modem device
0x00000006	
PST_NETWORK_BRIDGE	Unspecified network bridge
0x00000100	
PST_PARALLELPORT	Parallel port
0x00000002	

PST_RS232	RS-232 serial port
0x00000001	
PST_RS422	RS-422 port
0x00000003	
PST_RS423	RS-423 port
0x00000004	
PST_RS449	RS-449 port
0x00000005	
PST_SCANNER	Scanner device
0x00000022	
PST_TCPIP_TELNET	TCP/IP Telnet protocol
0x00000102	
PST_UNSPECIFIED	Unspecified
0x00000000	
PST_X25	X.25 standards
0x00000103	

dwProvCapabilities

A bitmask indicating the capabilities offered by the provider. This member can be a combination of the following values.

Value	Meaning
PCF_16BITMODE 0x0200	Special 16-bit mode supported
PCF_DTRDSR 0x0001	DTR (data-terminal-ready)/DSR (data-set-ready) supported
PCF_INTTIMEOUTS 0x0080	Interval time-outs supported
PCF_PARITY_CHECK 0x0008	Parity checking supported
PCF_RLSD 0x0004	RLSD (receive-line-signal-detect) supported
PCF_RTSCTS 0x0002	RTS (request-to-send)/CTS (clear-to-send) supported
PCF_SETXCHAR 0x0020	Settable XON/XOFF supported

PCF_SPECIALCHARS 0x0100	Special character support provided
PCF_TOTALTIMEOUTS 0x0040	The total (elapsed) time-outs supported
PCF_XONXOFF 0x0010	XON/XOFF flow control supported

dwSettableParams

A bitmask indicating the communications parameters that can be changed. This member can be a combination of the following values.

Value	Meaning
SP_BAUD 0x0002	Baud rate
SP_DATABITS 0x0004	Data bits
SP_HANDSHAKING 0x0010	Handshaking (flow control)
SP_PARITY 0x0001	Parity
SP_PARITY_CHECK 0x0020	Parity checking
SP_RLSD 0x0040	RLSD (receive-line-signal-detect)
SP_STOPBITS 0x0008	Stop bits

dwSettableBaud

The baud rates that can be used. For values, see the **dwMaxBaud** member.

wSettableData

A bitmask indicating the number of data bits that can be set. This member can be a combination of the following values.

Value	Meaning
DATABITS_5	5 data bits

0x0001	
DATABITS_6	6 data bits
0x0002	
DATABITS_7	7 data bits
0x0004	
DATABITS_8	8 data bits
0x0008	
DATABITS_16	16 data bits
0x0010	
DATABITS_16X	Special wide path through serial hardware lines
0x0020	

wSettableStopParity

A bitmask indicating the stop bit and parity settings that can be selected. This member can be a combination of the following values.

Value	Meaning
STOPBITS_10	1 stop bit
0x0001	
STOPBITS_15	1.5 stop bits
0x0002	
STOPBITS_20	2 stop bits
0x0004	
PARITY_NONE	No parity
0x0100	
PARITY_ODD	Odd parity
0x0200	
PARITY_EVEN	Even parity
0x0400	
PARITY_MARK	Mark parity
0x0800	
PARITY_SPACE	Space parity
0x1000	

dwCurrentTxQueue

The size of the driver's internal output buffer, in bytes. A value of zero indicates that the value is unavailable.

dwCurrentRxQueue

The size of the driver's internal input buffer, in bytes. A value of zero indicates that the value is unavailable.

dwProvSpec1

Any provider-specific data. Applications should ignore this member unless they have detailed information about the format of the data required by the provider.

Set this member to **COMMPROP_INITIALIZED** before calling the [GetCommProperties](#) function to indicate that the **wPacketLength** member is already valid.

dwProvSpec2

Any provider-specific data. Applications should ignore this member unless they have detailed information about the format of the data required by the provider.

wcProvChar[1]

Any provider-specific data. Applications should ignore this member unless they have detailed information about the format of the data required by the provider.

Remarks

The contents of the **dwProvSpec1**, **dwProvSpec2**, and **wcProvChar** members depend on the provider subtype (specified by the **dwProvSubType** member).

If the provider subtype is **PST_MODEM**, these members are used as follows.

Value	Meaning
dwProvSpec1	Not used.
dwProvSpec2	Not used.
wcProvChar	Contains a MODEMDEVCAPS structure.

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Header	winbase.h (include Windows.h)

See also

[GetCommProperties](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

COMMTIMEOUTS structure (winbase.h)

Article04/02/2021

Contains the time-out parameters for a communications device. The parameters determine the behavior of [ReadFile](#), [WriteFile](#), [ReadFileEx](#), and [WriteFileEx](#) operations on the device.

Syntax

C++

```
typedef struct _COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;
    DWORD ReadTotalTimeoutMultiplier;
    DWORD ReadTotalTimeoutConstant;
    DWORD WriteTotalTimeoutMultiplier;
    DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS, *LPCOMMTIMEOUTS;
```

Members

ReadIntervalTimeout

The maximum time allowed to elapse before the arrival of the next byte on the communications line, in milliseconds. If the interval between the arrival of any two bytes exceeds this amount, the [ReadFile](#) operation is completed and any buffered data is returned. A value of zero indicates that interval time-outs are not used.

A value of **MAXDWORD**, combined with zero values for both the **ReadTotalTimeoutConstant** and **ReadTotalTimeoutMultiplier** members, specifies that the read operation is to return immediately with the bytes that have already been received, even if no bytes have been received.

ReadTotalTimeoutMultiplier

The multiplier used to calculate the total time-out period for read operations, in milliseconds. For each read operation, this value is multiplied by the requested number of bytes to be read.

ReadTotalTimeoutConstant

A constant used to calculate the total time-out period for read operations, in milliseconds. For each read operation, this value is added to the product of the **ReadTotalTimeoutMultiplier** member and the requested number of bytes.

A value of zero for both the **ReadTotalTimeoutMultiplier** and **ReadTotalTimeoutConstant** members indicates that total time-outs are not used for read operations.

WriteTotalTimeoutMultiplier

The multiplier used to calculate the total time-out period for write operations, in milliseconds. For each write operation, this value is multiplied by the number of bytes to be written.

WriteTotalTimeoutConstant

A constant used to calculate the total time-out period for write operations, in milliseconds. For each write operation, this value is added to the product of the **WriteTotalTimeoutMultiplier** member and the number of bytes to be written.

A value of zero for both the **WriteTotalTimeoutMultiplier** and **WriteTotalTimeoutConstant** members indicates that total time-outs are not used for write operations.

Remarks

If an application sets **ReadIntervalTimeout** and **ReadTotalTimeoutMultiplier** to **MAXDWORD** and sets **ReadTotalTimeoutConstant** to a value greater than zero and less than **MAXDWORD**, one of the following occurs when the [ReadFile](#) function is called:

- If there are any bytes in the input buffer, [ReadFile](#) returns immediately with the bytes in the buffer.
- If there are no bytes in the input buffer, [ReadFile](#) waits until a byte arrives and then returns immediately.
- If no bytes arrive within the time specified by **ReadTotalTimeoutConstant**, [ReadFile](#) times out.

Requirements

Minimum supported client

Windows XP

Minimum supported server	Windows Server 2003
Header	winbase.h (include Windows.h)

See also

[GetCommTimeouts](#)

[ReadFile](#)

[ReadFileEx](#)

[SetCommTimeouts](#)

[WriteFile](#)

[WriteFileEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

COMSTAT structure (winbase.h)

Article 04/02/2021

Contains information about a communications device. This structure is filled by the [ClearCommError](#) function.

Syntax

C++

```
typedef struct _COMSTAT {  
    DWORD fCtsHold : 1;  
    DWORD fDsrHold : 1;  
    DWORD fRlsdHold : 1;  
    DWORD fXoffHold : 1;  
    DWORD fXoffSent : 1;  
    DWORD fEof : 1;  
    DWORD fTxim : 1;  
    DWORD fReserved : 25;  
    DWORD cbInQue;  
    DWORD cbOutQue;  
} COMSTAT, *LPCOMSTAT;
```

Members

`fCtsHold`

If this member is **TRUE**, transmission is waiting for the CTS (clear-to-send) signal to be sent.

`fDsrHold`

If this member is **TRUE**, transmission is waiting for the DSR (data-set-ready) signal to be sent.

`fRlsdHold`

If this member is **TRUE**, transmission is waiting for the RLSD (receive-line-signal-detect) signal to be sent.

`fXoffHold`

If this member is **TRUE**, transmission is waiting because the XOFF character was received.

`fXoffSent`

If this member is **TRUE**, transmission is waiting because the XOFF character was transmitted. (Transmission halts when the XOFF character is transmitted to a system that takes the next character as XON, regardless of the actual character.)

`fEof`

If this member is **TRUE**, the end-of-file (EOF) character has been received.

`fTxim`

If this member is **TRUE**, there is a character queued for transmission that has come to the communications device by way of the [TransmitCommChar](#) function. The communications device transmits such a character ahead of other characters in the device's output buffer.

`fReserved`

Reserved; do not use.

`cbInQue`

The number of bytes received by the serial provider but not yet read by a [ReadFile](#) operation.

`cbOutQue`

The number of bytes of user data remaining to be transmitted for all write operations. This value will be zero for a nonoverlapped write.

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Header	winbase.h (include Windows.h)

See also

[ClearCommError](#)

[ReadFile](#)

[TransmitCommChar](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ConvertFiberToThread function (winbase.h)

Article 06/29/2021

Converts the current fiber into a thread.

Syntax

C++

```
BOOL ConvertFiberToThread();
```

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

The function releases the resources allocated by the [ConvertThreadToFiber](#) function. After calling this function, you cannot call any of the fiber functions from the thread.

To compile an application that uses this function, define _WIN32_WINNT as _WIN32_WINNT_WS03 (0x0502) or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConvertThreadToFiber](#)

[Fibers](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ConvertThreadToFiber function (winbase.h)

Article07/27/2022

Converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers.

Syntax

C++

```
LPVOID ConvertThreadToFiber(  
    [in, optional] LPVOID lpParameter  
);
```

Parameters

[in, optional] *lpParameter*

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the [GetFiberData](#) macro.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks

Only fibers can execute other fibers. If a thread needs to execute a fiber, it must call [ConvertThreadToFiber](#) or [ConvertThreadToFiberEx](#) to create an area in which to save fiber state information. The thread is now the current fiber. The state information for this fiber includes the fiber data specified by *lpParameter*.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Examples

For an example, see [Using Fibers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConvertFiberToThread](#)

[ConvertThreadToFiberEx](#)

[Fibers](#)

[GetFiberData](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ConvertThreadToFiberEx function (winbase.h)

Article07/27/2022

Converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers.

Syntax

C++

```
LPVOID ConvertThreadToFiberEx(
    [in, optional] LPVOID lpParameter,
    [in]           DWORD  dwFlags
);
```

Parameters

[in, optional] lpParameter

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the [GetFiberData](#) macro.

[in] dwFlags

If this parameter is zero, the floating-point state on x86 systems is not switched and data can be corrupted if a fiber uses floating-point arithmetic. If this parameter is FIBER_FLAG_FLOAT_SWITCH, the floating-point state is switched for the fiber.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks

Only fibers can execute other fibers. If a thread needs to execute a fiber, it must call [ConvertThreadToFiber](#) or [ConvertThreadToFiberEx](#) to create an area in which to save fiber state information. The thread is now the current fiber. The state information for this fiber includes the fiber data specified by *lpParameter*.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConvertFiberToThread](#)

[Fibers](#)

[GetFiberData](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CopyContext function (winbase.h)

Article 10/13/2021

Copies a source context structure (including any XState) onto an initialized destination context structure.

Syntax

C++

```
BOOL CopyContext(
    [in, out] PCONTEXT Destination,
    [in]      DWORD    ContextFlags,
    [in]      PCONTEXT Source
);
```

Parameters

[in, out] Destination

A pointer to a [CONTEXT](#) structure that receives the context copied from the *Source*. The [CONTEXT](#) structure should be initialized by calling [InitializeContext](#) before calling this function.

[in] ContextFlags

Flags specifying the pieces of the *Source* [CONTEXT](#) structure that will be copied into the destination. This must be a subset of the *ContextFlags* specified when calling [InitializeContext](#) on the *Destination* [CONTEXT](#).

[in] Source

A pointer to a [CONTEXT](#) structure from which to copy processor context data.

Return value

This function returns **TRUE** if the context was copied successfully, otherwise **FALSE**. To get extended error information, call [GetLastError](#).

Remarks

The function copies data from the *Source CONTEXT* over the corresponding data in the *Destination CONTEXT*, including extended context if any is present. The *Destination CONTEXT* must have been initialized with [InitializeContext](#) to ensure proper alignment and initialization. If any data is present in the *Destination CONTEXT* and the corresponding flag is not set in the *Source CONTEXT* or in the *ContextFlags* parameter, the data remains valid in the *Destination*.

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The [AVX API](#) is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1 . Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this documentation and get pointers to them using [GetModuleHandle](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CONTEXT](#)

[InitializeContext](#)

[Intel AVX](#)

[Working with XState Context](#)

Feedback



Was this page helpful? [Yes](#) | [No](#)

Get help at Microsoft Q&A

CopyFile function (winbase.h)

Article06/01/2023

Copies an existing file to a new file.

The [CopyFileEx](#) function provides two additional capabilities. [CopyFileEx](#) can call a specified callback function each time a portion of the copy operation is completed, and [CopyFileEx](#) can be canceled during the copy operation.

To perform this operation as a transacted operation, use the [CopyFileTransacted](#) function.

Syntax

C++

```
BOOL CopyFile(
    [in] LPCTSTR lpExistingFileName,
    [in] LPCTSTR lpNewFileName,
    [in] BOOL     bFailIfExists
);
```

Parameters

[in] `lpExistingFileName`

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

If `lpExistingFileName` does not exist, [CopyFile](#) fails, and [GetLastError](#) returns `ERROR_FILE_NOT_FOUND`.

[in] *lpNewFileName*

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] *bFailIfExists*

If this parameter is **TRUE** and the new file specified by *lpNewFileName* already exists, the function fails. If this parameter is **FALSE** and the new file already exists, the function overwrites the existing file and succeeds.

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

The security resource properties (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are copied to the new file.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource properties for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

File attributes for the existing file are copied to the new file. For example, if an existing file has the **FILE_ATTRIBUTE_READONLY** file attribute, a copy created through a call to **CopyFile** will also have the **FILE_ATTRIBUTE_READONLY** file attribute. For more information, see [Retrieving and Changing File Attributes](#).

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

When **CopyFile** is used to copy an encrypted file, it attempts to encrypt the destination file with the keys used in the encryption of the source file. If this cannot be done, this function attempts to encrypt the destination file with default keys. If neither of these methods can be done, **CopyFile** fails with an **ERROR_ENCRYPTION_FAILED** error code.

Symbolic link behavior—if the source file is a symbolic link, the actual file copied is the target of the symbolic link.

If the destination file already exists and is a symbolic link, the target of the symbolic link is overwritten by the source file.

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 an example, see [Retrieving and Changing File Attributes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileEx](#)

[CopyFileTransacted](#)

[CreateFile](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFile](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CopyFile2 function (winbase.h)

Article07/27/2022

Copies an existing file to a new file, notifying the application of its progress through a callback function.

Syntax

C++

```
HRESULT CopyFile2(
    [in]          PCWSTR pwszExistingFileName,
    [in]          PCWSTR pwszNewFileName,
    [in, optional] COPYFILE2_EXTENDED_PARAMETERS *pExtendedParameters
);
```

Parameters

[in] `pwszExistingFileName`

The name of an existing file.

To extend this limit to 32,767 wide characters, prepend "\?" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip Starting in Windows 10, version 1607, you can opt-in to remove the **MAX_PATH** character limitation without prepending "\?\\". See the "Maximum Path Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

If *lpExistingFileName* does not exist, the **CopyFile2** function fails returns `HRESULT_FROM_WIN32(ERROR_FILE_NOT_FOUND)`.

[in] `pwszNewFileName`

The name of the new file.

To extend this limit to 32,767 wide characters, prepend "\?" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip Starting in Windows 10, version 1607, you can opt-in to remove the **MAX_PATH** character limitation without prepending "\\?\\". See the "Maximum Path Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *pExtendedParameters*

Optional address of a [COPYFILE2_EXTENDED_PARAMETERS](#) structure.

Return value

If the function succeeds, the return value will return **TRUE** when passed to the [SUCCEEDED](#) macro.

Return code	Description
S_OK	The copy operation completed successfully.
HRESULT_FROM_WIN32(ERROR_REQUEST_PAUSED)	The copy operation was paused by a COPYFILE2_PROGRESS_PAUSE return from the CopyFile2ProgressRoutine callback function.
HRESULT_FROM_WIN32(ERROR_REQUEST_ABORTED)	The copy operation was paused by a COPYFILE2_PROGRESS_CANCEL or COPYFILE2_PROGRESS_STOP return from the CopyFile2ProgressRoutine callback function.
HRESULT_FROM_WIN32(ERROR_ALREADY_EXISTS)	The <i>dwCopyFlags</i> member of the COPYFILE2_EXTENDED_PARAMETERS structure passed through the <i>pExtendedParameters</i> parameter contains the COPY_FILE_FAIL_IF_EXISTS flag and a conflicting name existed.
HRESULT_FROM_WIN32(ERROR_FILE_EXISTS)	The <i>dwCopyFlags</i> member of the COPYFILE2_EXTENDED_PARAMETERS structure passed through the <i>pExtendedParameters</i> parameter contains the COPY_FILE_FAIL_IF_EXISTS flag and a conflicting name existed.

Remarks

This function preserves extended attributes, OLE structured storage, NTFS file system alternate data streams, and file attributes. Security attributes for the existing file are not copied to the new file. To copy security attributes, use the [SHFileOperation](#) function.

This function fails with `HRESULT_FROM_WIN32(ERROR_ACCESS_DENIED)` if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

To compile an application that uses this function, define the **_WIN32_WINNT** macro as **_WIN32_WINNT_WIN8** or later. For more information, see [Using the Windows Headers](#).

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

Starting with Windows 10, version 1903 and Windows Server 2022, a new value, **COPY_FILE_REQUEST_COMPRESSED_TRAFFIC**, is supported for the *dwCopyFlags* field of the [**COPYFILE2_EXTENDED_PARAMETERS**](#) structure passed in the *pExtendedParameters* argument to this function. This new value requests that the underlying transfer channel compress the data during the copy operation. The request may not be supported for all mediums, in which case it is ignored. The compression attributes and parameters (computational complexity, memory usage) are not configurable through this API, and are subject to change between different OS releases. On Windows 10, the flag is supported for files residing on SMB shares, where the negotiated SMB protocol version is SMB v3.1.1 or greater.

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
--------------------------	-------------------------------------

Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COPYFILE2_EXTENDED_PARAMETERS](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

COPYFILE2_COPY_PHASE enumeration (winbase.h)

Article01/31/2022

Indicates the phase of a copy at the time of an error. This is used in the **Error** structure embedded in the [COPYFILE2_MESSAGE](#) structure.

Syntax

C++

```
typedef enum _COPYFILE2_COPY_PHASE {
    COPYFILE2_PHASE_NONE = 0,
    COPYFILE2_PHASE_PREPARE_SOURCE,
    COPYFILE2_PHASE_PREPARE_DEST,
    COPYFILE2_PHASE_READ_SOURCE,
    COPYFILE2_PHASE_WRITE_DESTINATION,
    COPYFILE2_PHASE_SERVER_COPY,
    COPYFILE2_PHASE_NAMEGRAFT_COPY,
    COPYFILE2_PHASE_MAX
} COPYFILE2_COPY_PHASE;
```

Constants

`COPYFILE2_PHASE_NONE`

Value: 0

The copy had not yet started processing.

`COPYFILE2_PHASE_PREPARE_SOURCE`

The source was being prepared including opening a handle to the source. This phase happens once per stream copy operation.

`COPYFILE2_PHASE_PREPARE_DEST`

The destination was being prepared including opening a handle to the destination. This phase happens once per stream copy operation.

`COPYFILE2_PHASE_READ_SOURCE`

The source file was being read. This phase happens one or more times per stream copy operation.

COPYFILE2_PHASE_WRITE_DESTINATION

The destination file was being written. This phase happens one or more times per stream copy operation.

COPYFILE2_PHASE_SERVER_COPY

Both the source and destination were on the same remote server and the copy was being processed remotely.

This phase happens once per stream copy operation.

COPYFILE2_PHASE_NAMEGRAFT_COPY

The copy operation was processing symbolic links and/or reparse points. This phase happens once per file copy operation.

COPYFILE2_PHASE_MAX

One greater than the maximum value. Valid values for this enumeration will be less than this value.

Remarks

To compile an application that uses this enumeration, define the `_WIN32_WINNT` macro as 0x0601 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

See also

[COPYFILE2_MESSAGE](#)

[CopyFile2](#)

[File Management Enumerations](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

COPYFILE2_EXTENDED_PARAMETERS structure (winbase.h)

Article11/13/2023

Contains extended parameters for the [CopyFile2](#) function.

Syntax

C++

```
typedef struct COPYFILE2_EXTENDED_PARAMETERS {
    DWORD           dwSize;
    DWORD           dwCopyFlags;
    BOOL            *pfCancel;
    PCOPYFILE2_PROGRESS_ROUTINE pProgressRoutine;
    PVOID           pvCallbackContext;
} COPYFILE2_EXTENDED_PARAMETERS;
```

Members

`dwSize`

Contains the size of this structure, `sizeof(COPYFILE2_EXTENDED_PARAMETERS)`.

`dwCopyFlags`

Contains a combination of zero or more of these flag values.

Value	Meaning
<code>COPY_FILE_FAIL_IF_EXISTS</code> <code>0x00000001</code>	If the destination file exists the copy operation fails immediately. If a file or directory exists with the destination name then the CopyFile2 function call will fail with either <code>HRESULT_FROM_WIN32(ERROR_ALREADY_EXISTS)</code> or <code>HRESULT_FROM_WIN32(ERROR_FILE_EXISTS)</code> . If <code>COPY_FILE_RESUME_FROM_PAUSE</code> is also specified then a failure is only triggered if the destination file does not have a valid restart header.
<code>COPY_FILE_RESTARTABLE</code> <code>0x00000002</code>	The file is copied in a manner that can be restarted if the same source and destination

Value	Meaning
	filenames are used again. This is slower.
COPY_FILE_OPEN_SOURCE_FOR_WRITE 0x00000004	The file is copied and the source file is opened for write access.
COPY_FILE_ALLOW_DECRYPTED_DESTINATION 0x00000008	The copy will be attempted even if the destination file cannot be encrypted.
COPY_FILE_COPY_SYMLINK 0x00000800	If the source file is a symbolic link, the destination file is also a symbolic link pointing to the same file as the source symbolic link.
COPY_FILE_NO_BUFFERING 0x00001000	The copy is performed using unbuffered I/O, bypassing the system cache resources. This flag is recommended for very large file copies. It is not recommended to pause copies that are using this flag.
COPY_FILE_REQUEST_SECURITY_PRIVILEGES 0x00002000	The copy is attempted, specifying <code>ACCESS_SYSTEM_SECURITY</code> for the source file and <code>ACCESS_SYSTEM_SECURITY \ WRITE_DAC \ WRITE_OWNER</code> for the destination file. If these requests are denied the access request will be reduced to the highest privilege level for which access is granted. For more information see SACL Access Right . This can be used to allow the CopyFile2ProgressRoutine callback to perform operations requiring higher privileges, such as copying the security attributes for the file.
COPY_FILE_RESUME_FROM_PAUSE 0x00004000	The destination file is examined to see if it was copied using COPY_FILE_RESTARTABLE . If so the copy is resumed. If not the file will be fully copied.
COPY_FILE_NO_OFFLOAD 0x00040000	Do not attempt to use the Windows Copy Offload mechanism. This is not generally recommended.
COPY_FILE_IGNORE_EDP_BLOCK 0x00400000	Instead of blocking, the file should be copied and encrypted on the destination if supported by the destination file system. Supported on Windows 10 and later.
COPY_FILE_IGNORE_SOURCE_ENCRYPTION 0x00800000	Ignore the source file's encrypted state. Supported on Windows 10 and later.

Value	Meaning
<code>COPY_FILE_DONT_REQUEST_DEST_WRITE_DAC</code> 0x02000000	Don't request WRITE_DAC for the destination file access. Supported on Windows 10 and later.
<code>COPY_FILE_OPEN_AND_COPY_REPARSE_POINT</code> 0x00200000	Always copy the reparse point regardless of type. It's the caller's responsibility to understand the reparse point meaning. Supported on Windows 10, build 19041 and later.
<code>COPY_FILE_DIRECTORY</code> 0x00000080	Indicates the source file is a directory file. When provided, the source file is opened with <code>FILE_OPEN_FOR_BACKUP_INTENT</code> . The directory file will have its alternate data streams, reparse point info, and EAs copied like a normal file. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_SKIP_ALTERNATE_STREAMS</code> 0x00008000	Do not copy alternate data streams. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_DISABLE_PRE_ALLOCATION</code> 0x04000000	Do not preallocate the destination file size before performing the copy. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_ENABLE_LOW_FREE_SPACE_MODE</code> 0x08000000	Enable LowFreeSpace mode. No overlapped I/Os are used. ODX and SMB offload are not attempted. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_REQUEST_COMPRESSED_TRAFFIC</code> 0x10000000	Request the underlying transfer channel compress the data during the copy operation. The request may not be supported for all mediums, in which case it is ignored. The compression attributes and parameters (computational complexity, memory usage) are not configurable through this API, and are subject to change between different OS releases.
	This flag was introduced in Windows 10, version 1903 and Windows Server 2022. On Windows 10, the flag is supported for files residing on SMB shares, where the negotiated SMB protocol version is SMB v3.1.1 or greater.
<code>COPY_FILE_ENABLE_SPARSE_COPY</code> 0x20000000	Enable retaining the sparse state of the file during copy. Supported in Windows 11, build

Value	Meaning
	22H2 and later.

pfCancel

If this flag is set to **TRUE** during the copy operation then the copy operation is canceled.

pProgressRoutine

The optional address of a callback function of type **PCOPYFILE2_PROGRESS_ROUTINE** that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information on the progress callback function, see the [CopyFile2ProgressRoutine](#) callback function.

pvCallbackContext

A pointer to application-specific context information to be passed to the [CopyFile2ProgressRoutine](#).

Remarks

To compile an application that uses this structure, define the **_WIN32_WINNT** macro as **_WIN32_WINNT_WIN8** or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

See also

[CopyFile2](#)

[CopyFile2ProgressRoutine](#)

[File Management Structures](#)

[COPYFILE2_EXTENDED_PARAMETERS_V2](#)

Feedback

Was this page helpful?

 Yes

 No

COPYFILE2_EXTENDED_PARAMETERS_V2 structure (winbase.h)

Article11/13/2023

Contains updated, additional functionality beyond the [COPYFILE2_EXTENDED_PARAMETERS](#) structure for the [CopyFile2](#) function.

Syntax

C++

```
typedef struct COPYFILE2_EXTENDED_PARAMETERS_V2 {
    DWORD             dwSize;
    DWORD             dwCopyFlags;
    BOOL              *pfCancel;
    PCOPYFILE2_PROGRESS_ROUTINE pProgressRoutine;
    PVOID             pvCallbackContext;
    DWORD             dwCopyFlagsV2;
    ULONG             ioDesiredSize;
    ULONG             ioDesiredRate;
    PVOID             reserved[8];
} COPYFILE2_EXTENDED_PARAMETERS_V2;
```

Members

`dwSize`

Contains the size of this structure, `sizeof(COPYFILE2_EXTENDED_PARAMETERS_V2)`.

`dwCopyFlags`

Contains a combination of zero or more of these flag values.

Value	Meaning
<code>COPY_FILE_FAIL_IF_EXISTS</code> <code>0x00000001</code>	If the destination file exists the copy operation fails immediately. If a file or directory exists with the destination name then the CopyFile2 function call will fail with either <code>HRESULT_FROM_WIN32(ERROR_ALREADY_EXISTS)</code> or <code>HRESULT_FROM_WIN32(ERROR_FILE_EXISTS)</code> . If <code>COPY_FILE_RESUME_FROM_PAUSE</code> is also specified then a failure is only triggered if the

Value	Meaning
	destination file does not have a valid restart header.
COPY_FILE_RESTARTABLE 0x00000002	The file is copied in a manner that can be restarted if the same source and destination filenames are used again. This is slower.
COPY_FILE_OPEN_SOURCE_FOR_WRITE 0x00000004	The file is copied and the source file is opened for write access.
COPY_FILE_ALLOW_DECRYPTED_DESTINATION 0x00000008	The copy will be attempted even if the destination file cannot be encrypted.
COPY_FILE_COPY_SYMLINK 0x00000800	If the source file is a symbolic link, the destination file is also a symbolic link pointing to the same file as the source symbolic link.
COPY_FILE_NO_BUFFERING 0x00001000	The copy is performed using unbuffered I/O, bypassing the system cache resources. This flag is recommended for very large file copies. It is not recommended to pause copies that are using this flag.
COPY_FILE_REQUEST_SECURITY_PRIVILEGES 0x00002000	The copy is attempted, specifying <code>ACCESS_SYSTEM_SECURITY</code> for the source file and <code>ACCESS_SYSTEM_SECURITY \ WRITE_DAC \ WRITE_OWNER</code> for the destination file. If these requests are denied the access request will be reduced to the highest privilege level for which access is granted. For more information see SACL Access Right . This can be used to allow the CopyFile2ProgressRoutine callback to perform operations requiring higher privileges, such as copying the security attributes for the file.
COPY_FILE_RESUME_FROM_PAUSE 0x00004000	The destination file is examined to see if it was copied using COPY_FILE_RESTARTABLE . If so the copy is resumed. If not the file will be fully copied.
COPY_FILE_NO_OFFLOAD 0x00040000	Do not attempt to use the Windows Copy Offload mechanism. This is not generally recommended.
COPY_FILE_IGNORE_EDP_BLOCK 0x00400000	Instead of blocking, the file should be copied and encrypted on the destination if supported by the destination file system. Supported on Windows 10 and later.

Value	Meaning
<code>COPY_FILE_IGNORE_SOURCE_ENCRYPTION</code> 0x00800000	Ignore the source file's encrypted state. Supported on Windows 10 and later.
<code>COPY_FILE_DONT_REQUEST_DEST_WRITE_DAC</code> 0x02000000	Don't request WRITE_DAC for the destination file access. Supported on Windows 10 and later.
<code>COPY_FILE_OPEN_AND_COPY_REPARSE_POINT</code> 0x00200000	Always copy the reparse point regardless of type. It's the caller's responsibility to understand the reparse point meaning. Supported on Windows 10, build 19041 and later.
<code>COPY_FILE_DIRECTORY</code> 0x00000080	Indicates the source file is a directory file. When provided, the source file is opened with <code>FILE_OPEN_FOR_BACKUP_INTENT</code> . The directory file will have its alternate data streams, reparse point info, and EAs copied like a normal file. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_SKIP_ALTERNATE_STREAMS</code> 0x00008000	Do not copy alternate data streams. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_DISABLE_PRE_ALLOCATION</code> 0x04000000	Do not preallocate the destination file size before performing the copy. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_ENABLE_LOW_FREE_SPACE_MODE</code> 0x08000000	Enable LowFreeSpace mode. No overlapped I/Os are used. ODX and SMB offload are not attempted. Supported in Windows 10, build 19041 and later.
<code>COPY_FILE_REQUEST_COMPRESSED_TRAFFIC</code> 0x10000000	<p>Request the underlying transfer channel compress the data during the copy operation. The request may not be supported for all mediums, in which case it is ignored. The compression attributes and parameters (computational complexity, memory usage) are not configurable through this API, and are subject to change between different OS releases.</p> <p>This flag was introduced in Windows 10, version 1903 and Windows Server 2022. On Windows 10, the flag is supported for files residing on SMB shares, where the negotiated SMB protocol version is SMB v3.1.1 or greater.</p>

Value	Meaning
COPY_FILE_ENABLE_SPARSE_COPY 0x20000000	Enable retaining the sparse state of the file during copy. Supported in Windows 11, build 22H2 and later.

`pfCancel`

If this flag is set to **TRUE** during the copy operation then the copy operation is canceled.

`pProgressRoutine`

The optional address of a callback function of type **PCOPYFILE2_PROGRESS_ROUTINE** that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information on the progress callback function, see the [CopyFile2ProgressRoutine](#) callback function. If both *pProgressRoutineOld* and *pProgressRoutine* are provided, *pProgressRoutineOld* takes precedence.

`pvCallbackContext`

A pointer to application-specific context information to be passed to the [CopyFile2ProgressRoutine](#).

`dwCopyFlagsV2`

Contains a combination of zero or more of these flag values.

Value	Meaning
COPY_FILE2_V2_DONT_COPY_JUNCTIONS 0x00000001	Disable copying junctions.

`ioDesiredSize`

Optional. The requested size (in bytes) for each I/O operation (i.e. one read/write cycle while copying the file). This may be reduced if insufficient memory is available. If zero, the default size is used. This may be ignored if *ioDesiredRate* is also provided

`ioDesiredRate`

Optional. The requested average I/O rate, in kilobytes per second. If zero, copies are performed as fast as possible.

`reserved[8]`

pProgressRoutineOld. Optional. The address of an old-style callback function of type [LPPROGRESS_ROUTINE](#) that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information, on the progress callback function, see [LPPROGRESS_ROUTINE callback](#). If both *pProgressRoutineOld* and *pProgressRoutine* are provided, *pProgressRoutineOld* takes precedence.

Remarks

To compile an application that uses this structure, define the [_WIN32_WINNT](#) macro as [_WIN32_WINNT_WIN8](#) or later. For more information, see [Using the Windows Headers](#).

Requirements

Requirements		
Minimum supported client	Windows 11 [desktop apps]	UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps]	UWP apps]
Header	winbase.h (include Windows.h)	

See also

[CopyFile2](#)

[COPYFILE2_EXTENDED_PARAMETERS](#)

[CopyFile2ProgressRoutine](#)

[File Management Structures](#)

[LPPROGRESS_ROUTINE](#)

[Using the Windows Headers](#)

Feedback

Was this page helpful?

 Yes

 No

COPYFILE2_MESSAGE structure (winbase.h)

Article04/02/2021

Passed to the [CopyFile2ProgressRoutine](#) callback function with information about a pending copy operation.

Syntax

C++

```
typedef struct COPYFILE2_MESSAGE {
    COPYFILE2_MESSAGE_TYPE Type;
    DWORD                 dwPadding;
    union {
        struct {
            DWORD      dwStreamNumber;
            DWORD      dwReserved;
            HANDLE     hSourceFile;
            HANDLE     hDestinationFile;
            ULONG     uliChunkNumber;
            ULONG     uliChunkSize;
            ULONG     uliStreamSize;
            ULONG     uliTotalFileSize;
        } ChunkStarted;
        struct {
            DWORD      dwStreamNumber;
            DWORD      dwFlags;
            HANDLE     hSourceFile;
            HANDLE     hDestinationFile;
            ULONG     uliChunkNumber;
            ULONG     uliChunkSize;
            ULONG     uliStreamSize;
            ULONG     uliStreamBytesTransferred;
            ULONG     uliTotalFileSize;
            ULONG     uliTotalBytesTransferred;
        } ChunkFinished;
        struct {
            DWORD      dwStreamNumber;
            DWORD      dwReserved;
            HANDLE     hSourceFile;
            HANDLE     hDestinationFile;
            ULONG     uliStreamSize;
            ULONG     uliTotalFileSize;
        } StreamStarted;
        struct {
            DWORD      dwStreamNumber;
            DWORD      dwReserved;
        }
    }
}
```

```

HANDLE          hSourceFile;
HANDLE          hDestinationFile;
ULARGE_INTEGER  uliStreamSize;
ULARGE_INTEGER  uliStreamBytesTransferred;
ULARGE_INTEGER  uliTotalFileSize;
ULARGE_INTEGER  uliTotalBytesTransferred;
} StreamFinished;
struct {
    DWORD dwReserved;
} PollContinue;
struct {
    COPYFILE2_COPY_PHASE CopyPhase;
    DWORD               dwStreamNumber;
    HRESULT             hrFailure;
    DWORD               dwReserved;
    ULARGE_INTEGER      uliChunkNumber;
    ULARGE_INTEGER      uliStreamSize;
    ULARGE_INTEGER      uliStreamBytesTransferred;
    ULARGE_INTEGER      uliTotalFileSize;
    ULARGE_INTEGER      uliTotalBytesTransferred;
} Error;
} Info;
} COPYFILE2_MESSAGE;

```

Members

Type

Value from the [COPYFILE2_MESSAGE_TYPE](#) enumeration used as a discriminant for the **Info** union within this structure.

Value	Meaning
COPYFILE2_CALLBACK_CHUNK_STARTED 1	Indicates a single chunk of a stream has started to be copied. Information is in the ChunkStarted structure within the Info union.
COPYFILE2_CALLBACK_CHUNK_FINISHED 2	Indicates the copy of a single chunk of a stream has completed. Information is in the ChunkFinished structure within the Info union.
COPYFILE2_CALLBACK_STREAM_STARTED 3	Indicates both source and destination handles for a stream have been opened and the copy of the stream is about to be started. Information is in the StreamStarted structure within the Info union. This does not indicate that the copy has started for that stream.
COPYFILE2_CALLBACK_STREAM_FINISHED 4	Indicates the copy operation for a stream have started to be completed, either successfully or due

	to a COPYFILE2_PROGRESS_STOP return from CopyFile2ProgressRoutine . Information is in the StreamFinished structure within the Info union.
COPYFILE2_CALLBACK_POLL_CONTINUE 5	May be sent periodically. Information is in the PollContinue structure within the Info union.
COPYFILE2_CALLBACK_ERROR 6	An error was encountered during the copy operation. Information is in the Error structure within the Info union.

`dwPadding`

`Info`

`Info.ChunkStarted`

This structure is selected if the **Type** member is set to **COPYFILE2_CALLBACK_CHUNK_STARTED** (1).

`Info.ChunkStarted.dwStreamNumber`

Indicates which stream within the file is about to be copied. The value used for identifying a stream within a file will start at one (1) and will always be higher than any previous stream for that file.

`Info.ChunkStarted.dwReserved`

This member is reserved for internal use.

`Info.ChunkStarted.hSourceFile`

Handle to the source stream.

`Info.ChunkStarted.hDestinationFile`

Handle to the destination stream.

`Info.ChunkStarted.ulichunkNumber`

Indicates which chunk within the current stream is about to be copied. The value used for a chunk will start at zero (0) and will always be higher than that of any previous chunk for the current stream.

`Info.ChunkStarted.ulichunkSize`

Size of the copied chunk, in bytes.

Info.ChunkStarted.ulStreamSize

Size of the current stream, in bytes.

Info.ChunkStarted.ulTotalFileSize

Size of all streams for this file, in bytes.

Info.ChunkFinished

This structure is selected if the **Type** member is set to **COPYFILE2_CALLBACK_CHUNK_FINISHED** (2).

ChunkFinished.dwReserved

This member is reserved for internal use.

Info.ChunkFinished.dwStreamNumber

Indicates which stream within the file is about to be copied. The value used for identifying a stream within a file will start at one (1) and will always be higher than any previous stream for that file.

Info.ChunkFinished.dwFlags

Info.ChunkFinished.hSourceFile

Handle to the source stream.

Info.ChunkFinished.hDestinationFile

Handle to the destination stream.

Info.ChunkFinished.ulChunkNumber

Indicates which chunk within the current stream is in process. The value used for a chunk will start at zero (0) and will always be higher than that of any previous chunk for the current stream.

Info.ChunkFinished.ulChunkSize

Size of the copied chunk, in bytes.

Info.ChunkFinished.ulStreamSize

Size of the current stream, in bytes.

`Info.ChunkFinished.ulStreamBytesTransferred`

Total bytes copied for this stream so far.

`Info.ChunkFinished.ulTotalFileSize`

Size of all streams for this file, in bytes.

`Info.ChunkFinished.ulTotalBytesTransferred`

Total bytes copied for this file so far.

`Info.StreamStarted`

This structure is selected if the **Type** member is set to **COPYFILE2_CALLBACK_STREAM_STARTED** (3).

`Info.StreamStarted.dwStreamNumber`

Indicates which stream within the file is about to be copied. The value used for identifying a stream within a file will start at one (1) and will always be higher than any previous stream for that file.

`Info.StreamStarted.dwReserved`

This member is reserved for internal use.

`Info.StreamStarted.hSourceFile`

Handle to the source stream.

`Info.StreamStarted.hDestinationFile`

Handle to the destination stream.

`Info.StreamStarted.ulStreamSize`

Size of the current stream, in bytes.

`Info.StreamStarted.ulTotalFileSize`

Size of all streams for this file, in bytes.

`Info.StreamFinished`

This structure is selected if the **Type** member is set to **COPYFILE2_CALLBACK_STREAM_FINISHED** (4).

`Info.StreamFinished.dwStreamNumber`

Indicates which stream within the file is about to be copied. The value used for identifying a stream within a file will start at one (1) and will always be higher than any previous stream for that file.

`Info.StreamFinished.dwReserved`

This member is reserved for internal use.

`Info.StreamFinished.hSourceFile`

Handle to the source stream.

`Info.StreamFinished.hDestinationFile`

Handle to the destination stream.

`Info.StreamFinished.ulStreamSize`

Size of the current stream, in bytes.

`Info.StreamFinished.ulStreamBytesTransferred`

Total bytes copied for this stream so far.

`Info.StreamFinished.ulTotalFileSize`

Size of all streams for this file, in bytes.

`Info.StreamFinished.ulTotalBytesTransferred`

Total bytes copied for this file so far.

`Info.PollContinue`

This structure is selected if the `Type` member is set to `COPYFILE2_CALLBACK_POLL_CONTINUE` (5).

`Info.PollContinue.dwReserved`

This member is reserved for internal use.

`Info.Error`

This structure is selected if the `Type` member is set to `COPYFILE2_CALLBACK_ERROR` (6).

`Info.Error.CopyPhase`

Value from the [COPYFILE2_COPY_PHASE](#) enumeration indicating the current phase of the copy at the time of the error.

Info.Error.dwStreamNumber

The number of the stream that was being processed at the time of the error.

Info.Error.hrFailure

Value indicating the problem.

Info.Error.dwReserved

This member is reserved for internal use.

Info.Error.ulChunkNumber

Indicates which chunk within the current stream was being processed at the time of the error. The value used for a chunk will start at zero (0) and will always be higher than that of any previous chunk for the current stream.

Info.Error.ulStreamSize

Size, in bytes, of the stream being processed.

Info.Error.ulStreamBytesTransferred

Number of bytes that had been successfully transferred for the stream being processed.

Info.Error.ulTotalFileSize

Size, in bytes, of the total file being processed.

Info.Error.ulTotalBytesTransferred

Number of bytes that had been successfully transferred for the entire copy operation.

Remarks

To compile an application that uses the [COPYFILE2_MESSAGE](#) structure, define the [_WIN32_WINNT](#) macro as 0x0601 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

COPYFILE2_MESSAGE_ACTION enumeration (winbase.h)

Article01/31/2022

Returned by the [CopyFile2ProgressRoutine](#) callback function to indicate what action should be taken for the pending copy operation.

Syntax

C++

```
typedef enum _COPYFILE2_MESSAGE_ACTION {
    COPYFILE2_PROGRESS_CONTINUE = 0,
    COPYFILE2_PROGRESS_CANCEL,
    COPYFILE2_PROGRESS_STOP,
    COPYFILE2_PROGRESS QUIET,
    COPYFILE2_PROGRESS_PAUSE
} COPYFILE2_MESSAGE_ACTION;
```

Constants

`COPYFILE2_PROGRESS_CONTINUE`

Value: 0

Continue the copy operation.

`COPYFILE2_PROGRESS_CANCEL`

Cancel the copy operation. The [CopyFile2](#) call will fail and return `HRESULT_FROM_WIN32(ERROR_REQUEST_ABORTED)` and any partially copied fragments will be deleted.

`COPYFILE2_PROGRESS_STOP`

Stop the copy operation. The [CopyFile2](#) call will fail and return `HRESULT_FROM_WIN32(ERROR_REQUEST_ABORTED)` and any partially copied fragments will be left intact. The operation can be restarted using the `COPY_FILE_RESUME_FROM_PAUSE` flag only if the `COPY_FILE_RESTARTABLE` flag was set in the `dwCopyFlags` member of the `COPYFILE2_EXTENDED_PARAMETERS` structure passed to the [CopyFile2](#) function.

`COPYFILE2_PROGRESS_QUIET`

Continue the copy operation but do not call the [CopyFile2ProgressRoutine](#) callback function again for this operation.

`COPYFILE2_PROGRESS_PAUSE`

Pause the copy operation and write a restart header. This value is not compatible with the `COPY_FILE_RESTARTABLE` flag for the `dwCopyFlags` member of the [COPYFILE2_EXTENDED_PARAMETERS](#) structure. In most cases the [CopyFile2](#) call will fail and return `HRESULT_FROM_WIN32(ERROR_REQUEST_PAUSED)` and any partially copied fragments will be left intact (except for the header written that is used to resume the copy operation later.) In case the copy operation was complete at the time the pause request is processed the [CopyFile2](#) call will complete successfully and no resume header will be written. After this value is processed one more callback will be made to the [CopyFile2ProgressRoutine](#) with the message specifying a `COPYFILE2_CALLBACK_STREAM_FINISHED` (4) value in the `Type` member of the [COPYFILE2_MESSAGE](#) structure. After the callback has returned [CopyFile2](#) will fail with `HRESULT_FROM_WIN32(ERROR_REQUEST_PAUSED)`.

Remarks

To compile an application that uses this enumeration, define the `_WIN32_WINNT` macro as 0x0601 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

See also

[CopyFile2](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

COPYFILE2_MESSAGE_TYPE enumeration (winbase.h)

Article01/31/2022

Indicates the type of message passed in the [COPYFILE2_MESSAGE](#) structure to the [CopyFile2ProgressRoutine](#) callback function.

Syntax

C++

```
typedef enum _COPYFILE2_MESSAGE_TYPE {
    COPYFILE2_CALLBACK_NONE = 0,
    COPYFILE2_CALLBACK_CHUNK_STARTED,
    COPYFILE2_CALLBACK_CHUNK_FINISHED,
    COPYFILE2_CALLBACK_STREAM_STARTED,
    COPYFILE2_CALLBACK_STREAM_FINISHED,
    COPYFILE2_CALLBACK_POLL_CONTINUE,
    COPYFILE2_CALLBACK_ERROR,
    COPYFILE2_CALLBACK_MAX
} COPYFILE2_MESSAGE_TYPE;
```

Constants

`COPYFILE2_CALLBACK_NONE`

Value: 0

Not a valid value.

`COPYFILE2_CALLBACK_CHUNK_STARTED`

Indicates a single chunk of a stream has started to be copied.

`COPYFILE2_CALLBACK_CHUNK_FINISHED`

Indicates the copy of a single chunk of a stream has completed.

`COPYFILE2_CALLBACK_STREAM_STARTED`

Indicates both source and destination handles for a stream have been opened and the copy of the stream is about to be started.

`COPYFILE2_CALLBACK_STREAM_FINISHED`

Indicates the copy operation for a stream have started to be completed.

`COPYFILE2_CALLBACK_POLL_CONTINUE`

May be sent periodically.

`COPYFILE2_CALLBACK_ERROR`

`COPYFILE2_CALLBACK_MAX`

An error was encountered during the copy operation.

Remarks

To compile an application that uses this enumeration, define the `_WIN32_WINNT` macro as 0x0601 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

See also

[File Management Enumerations](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CopyFileA function (winbase.h)

Article06/01/2023

Copies an existing file to a new file.

The [CopyFileEx](#) function provides two additional capabilities. [CopyFileEx](#) can call a specified callback function each time a portion of the copy operation is completed, and [CopyFileEx](#) can be canceled during the copy operation.

To perform this operation as a transacted operation, use the [CopyFileTransacted](#) function.

Syntax

C++

```
BOOL CopyFileA(
    [in] LPCSTR lpExistingFileName,
    [in] LPCSTR lpNewFileName,
    [in] BOOL     bFailIfExists
);
```

Parameters

[in] *lpExistingFileName*

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

If *lpExistingFileName* does not exist, [CopyFile](#) fails, and [GetLastError](#) returns [ERROR_FILE_NOT_FOUND](#).

[in] *lpNewFileName*

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] *bFailIfExists*

If this parameter is **TRUE** and the new file specified by *lpNewFileName* already exists, the function fails. If this parameter is **FALSE** and the new file already exists, the function overwrites the existing file and succeeds.

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

The security resource properties (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are copied to the new file.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource properties for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

File attributes for the existing file are copied to the new file. For example, if an existing file has the **FILE_ATTRIBUTE_READONLY** file attribute, a copy created through a call to **CopyFile** will also have the **FILE_ATTRIBUTE_READONLY** file attribute. For more information, see [Retrieving and Changing File Attributes](#).

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

When **CopyFile** is used to copy an encrypted file, it attempts to encrypt the destination file with the keys used in the encryption of the source file. If this cannot be done, this function attempts to encrypt the destination file with default keys. If neither of these methods can be done, **CopyFile** fails with an **ERROR_ENCRYPTION_FAILED** error code.

Symbolic link behavior—if the source file is a symbolic link, the actual file copied is the target of the symbolic link.

If the destination file already exists and is a symbolic link, the target of the symbolic link is overwritten by the source file.

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 an example, see [Retrieving and Changing File Attributes](#).

Note

The winbase.h header defines **CopyFile** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileEx](#)

[CopyFileTransacted](#)

[CreateFile](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFile](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CopyFileExA function (winbase.h)

Article06/01/2023

Copies an existing file to a new file, notifying the application of its progress through a callback function.

To perform this operation as a transacted operation, use the [CopyFileTransacted](#) function.

Syntax

C++

```
BOOL CopyFileExA(
    [in]           LPCSTR      lpExistingFileName,
    [in]           LPCSTR      lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID       lpData,
    [in, optional] LPBOOL      pbCancel,
    [in]           DWORD       dwCopyFlags
);
```

Parameters

[in] `lpExistingFileName`

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

If `lpExistingFileName` does not exist, the `CopyFileEx` function fails, and the `GetLastError` function returns `ERROR_FILE_NOT_FOUND`.

[in] lpNewFileName

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 **Tip**

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] lpProgressRoutine

The address of a callback function of type **LPPROGRESS_ROUTINE** that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information on the progress callback function, see the [CopyProgressRoutine](#) function.

[in, optional] lpData

The argument to be passed to the callback function. This parameter can be **NULL**.

[in, optional] pbCancel

If this flag is set to **TRUE** during the copy operation, the operation is canceled. Otherwise, the copy operation will continue to completion.

[in] dwCopyFlags

Flags that specify how the file is to be copied. This parameter can be a combination of the following values.

Value	Meaning
COPY_FILE_ALLOW_DECRYPTED_DESTINATION 0x00000008	An attempt to copy an encrypted file will succeed even if the destination copy cannot be encrypted.
COPY_FILE_COPY_SYMLINK 0x00000800	If the source file is a symbolic link, the destination file is also a symbolic link pointing to the same file that the source symbolic link is pointing to. Windows Server 2003 and Windows XP: This value is not supported.

COPY_FILE_FAIL_IF_EXISTS 0x00000001	The copy operation fails immediately if the target file already exists.
COPY_FILE_NO_BUFFERING 0x00001000	The copy operation is performed using unbuffered I/O, bypassing system I/O cache resources. Recommended for very large file transfers. Windows Server 2003 and Windows XP: This value is not supported.
COPY_FILE_OPEN_SOURCE_FOR_WRITE 0x00000004	The file is copied and the original file is opened for write access.
COPY_FILE_RESTARTABLE 0x00000002	Progress of the copy is tracked in the target file in case the copy fails. The failed copy can be restarted at a later time by specifying the same values for <i>lpExistingFileName</i> and <i>lpNewFileName</i> as those used in the call that failed. This can significantly slow down the copy operation as the new file may be flushed multiple times during the copy operation.
COPY_FILE_REQUEST_COMPRESSED_TRAFFIC 0x10000000	Request the underlying transfer channel compress the data during the copy operation. The request may not be supported for all mediums, in which case it is ignored. The compression attributes and parameters (computational complexity, memory usage) are not configurable through this API, and are subject to change between different OS releases. This flag was introduced in Windows 10, version 1903 and Windows Server 2022. On Windows 10, the flag is supported for files residing on SMB shares, where the negotiated SMB protocol version is SMB v3.1.1 or greater.

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 *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **CopyFileEx** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is deleted.

If *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **CopyFileEx** will return zero and **GetLastError** will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is left intact.

Remarks

This function preserves extended attributes, OLE structured storage, NTFS file system alternate data streams, security resource attributes, and file attributes.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource attributes (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

The security resource properties (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are copied to the new file.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource properties for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

When encrypted files are copied using **CopyFileEx**, the function attempts to encrypt the destination file with the keys used in the encryption of the source file. If this cannot be done, this function attempts to encrypt the destination file with default keys. If both of these methods cannot be done, **CopyFileEx** fails with an **ERROR_ENCRYPTION_FAILED** error code. If you want **CopyFileEx** to complete the copy operation even if the destination file cannot be encrypted, include the **COPY_FILE_ALLOW_DECRYPTED_DESTINATION** as the value of the *dwCopyFlags* parameter in your call to **CopyFileEx**.

If **COPY_FILE_COPY_SYMLINK** is specified, the following rules apply:

- If the source file is a symbolic link, the symbolic link is copied, not the target file.
- If the source file is not a symbolic link, there is no change in behavior.
- If the destination file is an existing symbolic link, the symbolic link is overwritten, not the target file.
- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails in all cases.

If **COPY_FILE_COPY_SYMLINK** is not specified, the following rules apply:

- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails only if the target of the symbolic link exists.
- If **COPY_FILE_FAIL_IF_EXISTS** is not specified, there is no change in behavior.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: If you are writing an application that is optimizing file copy operations across a LAN, consider using the [TransmitFile](#) function from Windows Sockets (Winsock). **TransmitFile** supports high-performance network transfers and provides a simple interface to send the contents of a file to a remote computer. To use **TransmitFile**, you must write a Winsock client application that sends the file from the source computer as well as a Winsock server application that uses other Winsock functions to receive the file on the remote computer.

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

① Note

The `winbase.h` header defines `CopyFileEx` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client

Windows XP [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[CopyFileTransacted](#)

[CopyProgressRoutine](#)

[CreateFile](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFile](#)

[MoveFileWithProgress](#)

[Symbolic Links](#)

[TransmitFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CopyFileExW function (winbase.h)

Article06/01/2023

Copies an existing file to a new file, notifying the application of its progress through a callback function.

To perform this operation as a transacted operation, use the [CopyFileTransacted](#) function.

Syntax

C++

```
BOOL CopyFileExW(
    [in]           LPCWSTR      lpExistingFileName,
    [in]           LPCWSTR      lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID        lpData,
    [in, optional] LPBOOL       pbCancel,
    [in]           DWORD         dwCopyFlags
);
```

Parameters

[in] `lpExistingFileName`

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

If `lpExistingFileName` does not exist, the `CopyFileEx` function fails, and the `GetLastError` function returns `ERROR_FILE_NOT_FOUND`.

[in] lpNewFileName

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 **Tip**

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] lpProgressRoutine

The address of a callback function of type **LPPROGRESS_ROUTINE** that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information on the progress callback function, see the [CopyProgressRoutine](#) function.

[in, optional] lpData

The argument to be passed to the callback function. This parameter can be **NULL**.

[in, optional] pbCancel

If this flag is set to **TRUE** during the copy operation, the operation is canceled. Otherwise, the copy operation will continue to completion.

[in] dwCopyFlags

Flags that specify how the file is to be copied. This parameter can be a combination of the following values.

Value	Meaning
COPY_FILE_ALLOW_DECRYPTED_DESTINATION 0x00000008	An attempt to copy an encrypted file will succeed even if the destination copy cannot be encrypted.
COPY_FILE_COPY_SYMLINK 0x00000800	If the source file is a symbolic link, the destination file is also a symbolic link pointing to the same file that the source symbolic link is pointing to. Windows Server 2003 and Windows XP: This value is not supported.

COPY_FILE_FAIL_IF_EXISTS 0x00000001	The copy operation fails immediately if the target file already exists.
COPY_FILE_NO_BUFFERING 0x00001000	The copy operation is performed using unbuffered I/O, bypassing system I/O cache resources. Recommended for very large file transfers. Windows Server 2003 and Windows XP: This value is not supported.
COPY_FILE_OPEN_SOURCE_FOR_WRITE 0x00000004	The file is copied and the original file is opened for write access.
COPY_FILE_RESTARTABLE 0x00000002	Progress of the copy is tracked in the target file in case the copy fails. The failed copy can be restarted at a later time by specifying the same values for <i>lpExistingFileName</i> and <i>lpNewFileName</i> as those used in the call that failed. This can significantly slow down the copy operation as the new file may be flushed multiple times during the copy operation.
COPY_FILE_REQUEST_COMPRESSED_TRAFFIC 0x10000000	Request the underlying transfer channel compress the data during the copy operation. The request may not be supported for all mediums, in which case it is ignored. The compression attributes and parameters (computational complexity, memory usage) are not configurable through this API, and are subject to change between different OS releases. This flag was introduced in Windows 10, version 1903 and Windows Server 2022. On Windows 10, the flag is supported for files residing on SMB shares, where the negotiated SMB protocol version is SMB v3.1.1 or greater.

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 *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **CopyFileEx** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is deleted.

If *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **CopyFileEx** will return zero and **GetLastError** will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is left intact.

Remarks

This function preserves extended attributes, OLE structured storage, NTFS file system alternate data streams, security resource attributes, and file attributes.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource attributes (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

The security resource properties (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are copied to the new file.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource properties for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

When encrypted files are copied using **CopyFileEx**, the function attempts to encrypt the destination file with the keys used in the encryption of the source file. If this cannot be done, this function attempts to encrypt the destination file with default keys. If both of these methods cannot be done, **CopyFileEx** fails with an **ERROR_ENCRYPTION_FAILED** error code. If you want **CopyFileEx** to complete the copy operation even if the destination file cannot be encrypted, include the **COPY_FILE_ALLOW_DECRYPTED_DESTINATION** as the value of the *dwCopyFlags* parameter in your call to **CopyFileEx**.

If **COPY_FILE_COPY_SYMLINK** is specified, the following rules apply:

- If the source file is a symbolic link, the symbolic link is copied, not the target file.
- If the source file is not a symbolic link, there is no change in behavior.
- If the destination file is an existing symbolic link, the symbolic link is overwritten, not the target file.
- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails in all cases.

If **COPY_FILE_COPY_SYMLINK** is not specified, the following rules apply:

- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails only if the target of the symbolic link exists.
- If **COPY_FILE_FAIL_IF_EXISTS** is not specified, there is no change in behavior.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: If you are writing an application that is optimizing file copy operations across a LAN, consider using the [TransmitFile](#) function from Windows Sockets (Winsock). **TransmitFile** supports high-performance network transfers and provides a simple interface to send the contents of a file to a remote computer. To use **TransmitFile**, you must write a Winsock client application that sends the file from the source computer as well as a Winsock server application that uses other Winsock functions to receive the file on the remote computer.

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

① Note

The `winbase.h` header defines `CopyFileEx` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client

Windows XP [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[CopyFileTransacted](#)

[CopyProgressRoutine](#)

[CreateFile](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFile](#)

[MoveFileWithProgress](#)

[Symbolic Links](#)

[TransmitFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CopyFileTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Copies an existing file to a new file as a transacted operation, notifying the application of its progress through a callback function.

Syntax

C++

```
BOOL CopyFileTransactedA(
    [in]          LPCSTR      lpExistingFileName,
    [in]          LPCSTR      lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID       lpData,
    [in, optional] LPBOOL      pbCancel,
    [in]          DWORD        dwCopyFlags,
    [in]          HANDLE      hTransaction
);
```

Parameters

`[in] lpExistingFileName`

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation"

section of [Naming Files, Paths, and Namespaces](#) for details.>.

If *lpExistingFileName* does not exist, the **CopyFileTransacted** function fails, and the **GetLastError** function returns **ERROR_FILE_NOT_FOUND**.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] *lpNewFileName*

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpProgressRoutine*

The address of a callback function of type **LPPROGRESS_ROUTINE** that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information on the progress callback function, see the [CopyProgressRoutine](#) function.

[in, optional] *lpData*

The argument to be passed to the callback function. This parameter can be **NULL**.

[in, optional] *pbCancel*

If this flag is set to **TRUE** during the copy operation, the operation is canceled. Otherwise, the copy operation will continue to completion.

[in] *dwCopyFlags*

Flags that specify how the file is to be copied. This parameter can be a combination of the following values.

Value	Meaning

COPY_FILE_COPY_SYMLINK 0x00000800	If the source file is a symbolic link, the destination file is also a symbolic link pointing to the same file that the source symbolic link is pointing to.
COPY_FILE_FAIL_IF_EXISTS 0x00000001	The copy operation fails immediately if the target file already exists.
COPY_FILE_OPEN_SOURCE_FOR_WRITE 0x00000004	The file is copied and the original file is opened for write access.
COPY_FILE_RESTARTABLE 0x00000002	Progress of the copy is tracked in the target file in case the copy fails. The failed copy can be restarted at a later time by specifying the same values for <i>lpExistingFileName</i> and <i>lpNewFileName</i> as those used in the call that failed. This can significantly slow down the copy operation as the new file may be flushed multiple times during the copy operation.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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](#).

If *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **CopyFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is deleted.

If *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **CopyFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is left intact.

If you attempt to call this function with a handle to a transaction that has already been rolled back, **CopyFileTransacted** will return either **ERROR_TRANSACTION_NOT_ACTIVE** or **ERROR_INVALID_TRANSACTION**.

Remarks

This function preserves extended attributes, OLE structured storage, NTFS file system alternate data streams, security attributes, and file attributes.

Windows 7, Windows Server 2008 R2, Windows Server 2008 and Windows Vista: Security resource attributes (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

Encrypted files are not supported by TxF.

If **COPY_FILE_COPY_SYMLINK** is specified, the following rules apply:

- If the source file is a symbolic link, the symbolic link is copied, not the target file.
- If the source file is not a symbolic link, there is no change in behavior.
- If the destination file is an existing symbolic link, the symbolic link is overwritten, not the target file.
- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails in all cases.

If **COPY_FILE_COPY_SYMLINK** is not specified, the following rules apply:

- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails only if the target of the symbolic link exists.
- If **COPY_FILE_FAIL_IF_EXISTS** is not specified, there is no change in behavior.

Link tracking is not supported by TxF.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support TxF.

Note

The winbase.h header defines CopyFileTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyProgressRoutine](#)

[CreateFileTransacted](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFileTransacted](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback



Was this page helpful?  Yes  No

Get help at Microsoft Q&A

CopyFileTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Copies an existing file to a new file as a transacted operation, notifying the application of its progress through a callback function.

Syntax

C++

```
BOOL CopyFileTransactedW(
    [in]          LPCWSTR      lpExistingFileName,
    [in]          LPCWSTR      lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID        lpData,
    [in, optional] LPBOOL       pbCancel,
    [in]          DWORD         dwCopyFlags,
    [in]          HANDLE        hTransaction
);
```

Parameters

`[in] lpExistingFileName`

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation"

section of [Naming Files, Paths, and Namespaces](#) for details.

If *lpExistingFileName* does not exist, the **CopyFileTransacted** function fails, and the **GetLastError** function returns **ERROR_FILE_NOT_FOUND**.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] *lpNewFileName*

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpProgressRoutine*

The address of a callback function of type **LPPROGRESS_ROUTINE** that is called each time another portion of the file has been copied. This parameter can be **NULL**. For more information on the progress callback function, see the [CopyProgressRoutine](#) function.

[in, optional] *lpData*

The argument to be passed to the callback function. This parameter can be **NULL**.

[in, optional] *pbCancel*

If this flag is set to **TRUE** during the copy operation, the operation is canceled. Otherwise, the copy operation will continue to completion.

[in] *dwCopyFlags*

Flags that specify how the file is to be copied. This parameter can be a combination of the following values.

Value	Meaning

COPY_FILE_COPY_SYMLINK 0x00000800	If the source file is a symbolic link, the destination file is also a symbolic link pointing to the same file that the source symbolic link is pointing to.
COPY_FILE_FAIL_IF_EXISTS 0x00000001	The copy operation fails immediately if the target file already exists.
COPY_FILE_OPEN_SOURCE_FOR_WRITE 0x00000004	The file is copied and the original file is opened for write access.
COPY_FILE_RESTARTABLE 0x00000002	Progress of the copy is tracked in the target file in case the copy fails. The failed copy can be restarted at a later time by specifying the same values for <i>lpExistingFileName</i> and <i>lpNewFileName</i> as those used in the call that failed. This can significantly slow down the copy operation as the new file may be flushed multiple times during the copy operation.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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](#).

If *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **CopyFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is deleted.

If *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **CopyFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. In this case, the partially copied destination file is left intact.

If you attempt to call this function with a handle to a transaction that has already been rolled back, **CopyFileTransacted** will return either **ERROR_TRANSACTION_NOT_ACTIVE** or **ERROR_INVALID_TRANSACTION**.

Remarks

This function preserves extended attributes, OLE structured storage, NTFS file system alternate data streams, security attributes, and file attributes.

Windows 7, Windows Server 2008 R2, Windows Server 2008 and Windows Vista: Security resource attributes (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

Encrypted files are not supported by TxF.

If **COPY_FILE_COPY_SYMLINK** is specified, the following rules apply:

- If the source file is a symbolic link, the symbolic link is copied, not the target file.
- If the source file is not a symbolic link, there is no change in behavior.
- If the destination file is an existing symbolic link, the symbolic link is overwritten, not the target file.
- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails in all cases.

If **COPY_FILE_COPY_SYMLINK** is not specified, the following rules apply:

- If **COPY_FILE_FAIL_IF_EXISTS** is also specified, and the destination file is an existing symbolic link, the operation fails only if the target of the symbolic link exists.
- If **COPY_FILE_FAIL_IF_EXISTS** is not specified, there is no change in behavior.

Link tracking is not supported by TxF.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support TxF.

Note

The winbase.h header defines CopyFileTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyProgressRoutine](#)

[CreateFileTransacted](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFileTransacted](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback



Was this page helpful?  Yes  No

Get help at Microsoft Q&A

CopyFileW function (winbase.h)

Article06/01/2023

Copies an existing file to a new file.

The [CopyFileEx](#) function provides two additional capabilities. [CopyFileEx](#) can call a specified callback function each time a portion of the copy operation is completed, and [CopyFileEx](#) can be canceled during the copy operation.

To perform this operation as a transacted operation, use the [CopyFileTransacted](#) function.

Syntax

C++

```
BOOL CopyFileW(
    [in] LPCWSTR lpExistingFileName,
    [in] LPCWSTR lpNewFileName,
    [in] BOOL     bFailIfExists
);
```

Parameters

[in] *lpExistingFileName*

The name of an existing file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

If *lpExistingFileName* does not exist, [CopyFile](#) fails, and [GetLastError](#) returns [ERROR_FILE_NOT_FOUND](#).

[in] *lpNewFileName*

The name of the new file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] *bFailIfExists*

If this parameter is **TRUE** and the new file specified by *lpNewFileName* already exists, the function fails. If this parameter is **FALSE** and the new file already exists, the function overwrites the existing file and succeeds.

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

The security resource properties (**ATTRIBUTE_SECURITY_INFORMATION**) for the existing file are copied to the new file.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource properties for the existing file are not copied to the new file until Windows 8 and Windows Server 2012.

File attributes for the existing file are copied to the new file. For example, if an existing file has the **FILE_ATTRIBUTE_READONLY** file attribute, a copy created through a call to **CopyFile** will also have the **FILE_ATTRIBUTE_READONLY** file attribute. For more information, see [Retrieving and Changing File Attributes](#).

This function fails with **ERROR_ACCESS_DENIED** if the destination file already exists and has the **FILE_ATTRIBUTE_HIDDEN** or **FILE_ATTRIBUTE_READONLY** attribute set.

When **CopyFile** is used to copy an encrypted file, it attempts to encrypt the destination file with the keys used in the encryption of the source file. If this cannot be done, this function attempts to encrypt the destination file with default keys. If neither of these methods can be done, **CopyFile** fails with an **ERROR_ENCRYPTION_FAILED** error code.

Symbolic link behavior—if the source file is a symbolic link, the actual file copied is the target of the symbolic link.

If the destination file already exists and is a symbolic link, the target of the symbolic link is overwritten by the source file.

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 an example, see [Retrieving and Changing File Attributes](#).

Note

The winbase.h header defines **CopyFile** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileEx](#)

[CopyFileTransacted](#)

[CreateFile](#)

[File Attribute Constants](#)

[File Management Functions](#)

[MoveFile](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateActCtxA function (winbase.h)

Article 02/09/2023

The **CreateActCtx** function creates an activation context.

Syntax

C++

```
HANDLE CreateActCtxA(  
    [in, out] PCACTCTXA pActCtx  
);
```

Parameters

[in, out] pActCtx

Pointer to an [ACTCTX](#) structure that contains information about the activation context to be created.

Return value

If the function succeeds, it returns a handle to the returned activation context.

Otherwise, it returns INVALID_HANDLE_VALUE.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

Set any undefined bits in dwFlags of [ACTCTX](#) to 0. If any undefined bits are not set to 0, the call to **CreateActCtx** that creates the activation context fails and returns an invalid parameter error code. The handle returned from **CreateActCtx** is passed in a call to [ActivateActCtx](#) to activate the context for the current thread.

 Note

The winbase.h header defines CreateActCtx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateActCtxW function (winbase.h)

Article 02/09/2023

The **CreateActCtx** function creates an activation context.

Syntax

C++

```
HANDLE CreateActCtxW(
    [in, out] PCACTCTXW pActCtx
);
```

Parameters

[in, out] pActCtx

Pointer to an [ACTCTX](#) structure that contains information about the activation context to be created.

Return value

If the function succeeds, it returns a handle to the returned activation context.

Otherwise, it returns INVALID_HANDLE_VALUE.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

Set any undefined bits in **dwFlags** of [ACTCTX](#) to 0. If any undefined bits are not set to 0, the call to **CreateActCtx** that creates the activation context fails and returns an invalid parameter error code. The handle returned from **CreateActCtx** is passed in a call to [ActivateActCtx](#) to activate the context for the current thread.

 Note

The winbase.h header defines CreateActCtx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateBoundaryDescriptorA function (winbase.h)

Article 08/03/2022

Creates a boundary descriptor.

Syntax

C++

```
HANDLE CreateBoundaryDescriptorA(
    [in] LPCSTR Name,
    [in] ULONG Flags
);
```

Parameters

[in] Name

The name of the boundary descriptor.

[in] Flags

A combination of the following flags that are combined by using a bitwise **OR** operation.

Flag	Description
CREATE_BOUNDARY_DESCRIPTOR_ADD_APPCONTAINER_SID (0x01) Note: This value is not supported prior to Windows 8.	Required for creating a boundary descriptor in an appcontainer process, regardless of producer or consumer.

Return value

If the function succeeds, the return value is a handle to the boundary descriptor.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

A new boundary descriptor must have at least one security identifier (SID). To add a SID to a boundary descriptor, use the [AddSIDToBoundaryDescriptor](#) function.

To compile an application that uses this function, define `_WIN32_WINNT` as `0x0600` or later.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddSIDToBoundaryDescriptor](#)

[CreatePrivateNamespace](#)

[DeleteBoundaryDescriptor](#)

[Object Namespaces](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateDirectory function (winbase.h)

Article08/03/2022

Creates a new directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory.

To specify a template directory, use the [.CreateDirectoryEx](#) function.

To perform this operation as a transacted operation, use the [.CreateDirectoryTransacted](#) function.

Syntax

C++

```
BOOL CreateDirectory(
    [in]          LPCTSTR      lpPathName,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Parameters

[in] `lpPathName`

The path of the directory to be created.

For the ANSI version of this function, there is a default string size limit for paths of 248 characters (**MAX_PATH** - enough room for a 8.3 filename). To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\?" to the path.

For more information, see [Naming a File](#)

Tip Starting with Windows 10, version 1607, for the unicode version of this function (**CreateDirectoryW**), you can opt-in to remove the 248 character limitation without prepending "\?\\". The 255 character limit per path segment still applies. See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] `lpSecurityAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The *lpSecurityDescriptor* member of the structure specifies a security descriptor for the new directory. If *lpSecurityAttributes* is **NULL**, the directory gets a default security descriptor. The ACLs in the default security descriptor for a directory are inherited from its parent directory.

The target file system must support security on files and directories for this parameter to have an effect. (This is indicated when [GetVolumeInformation](#) returns **FS_PERSISTENT_ACLS**.)

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](#). Possible errors include the following.

Return code	Description
ERROR_ALREADY_EXISTS	The specified directory already exists.
ERROR_PATH_NOT_FOUND	One or more intermediate directories do not exist; this function will only create the final directory in the path.

Remarks

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

An application can obtain a handle to a directory by calling [CreateFile](#) with the **FILE_FLAG_BACKUP_SEMANTICS** flag set. For a code example, see [CreateFile](#).

To support inheritance functions that query the security descriptor of this object may heuristically determine and report that inheritance is in effect. See [Automatic Propagation of Inheritable ACEs](#) for more information.

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 an example, see [Retrieving and Changing File Attributes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[.CreateDirectoryEx](#)

[.CreateDirectoryTransacted](#)

[CreateFile](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[RemoveDirectory](#)

[SECURITY_ATTRIBUTES](#)

[SECURITY_INFORMATION](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateDirectoryExA function (winbase.h)

Article06/01/2023

Creates a new directory with the attributes of a specified template directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory. The new directory retains the other attributes of the specified template directory.

To perform this operation as a transacted operation, use the [CreateDirectoryTransacted](#) function.

Syntax

C++

```
BOOL CreateDirectoryExA(
    [in]          LPCSTR           lpTemplateDirectory,
    [in]          LPCSTR           lpNewDirectory,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Parameters

[in] lpTemplateDirectory

The path of the directory to use as a template when creating the new directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpNewDirectory

The path of the directory to be created.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The **IpSecurityDescriptor** member of the structure specifies a security descriptor for the new directory.

If *lpSecurityAttributes* is **NULL**, the directory gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a directory are inherited from its parent directory.

The target file system must support security on files and directories for this parameter to have an effect. This is indicated when [GetVolumeInformation](#) returns **FS_PERSISTENT_ACLS**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#). Possible errors include the following.

Return code	Description
ERROR_ALREADY_EXISTS	The specified directory already exists.
ERROR_PATH_NOT_FOUND	One or more intermediate directories do not exist. This function only creates the final directory in the path. To create all intermediate directories on the path, use the SHCreateDirectoryEx function.

Remarks

The [.CreateDirectoryEx](#) function allows you to create directories that inherit stream information from other directories. This function is useful, for example, when you are using Macintosh directories, which have a resource stream that is needed to properly identify directory contents as an attribute.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

You can obtain a handle to a directory by calling the [CreateFile](#) function with the **FILE_FLAG_BACKUP_SEMANTICS** flag set. For a code example, see [CreateFile](#).

To support inheritance functions that query the security descriptor of this object can heuristically determine and report that inheritance is in effect. For more information, see [Automatic Propagation of Inheritable ACEs](#).

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

Note

The `winbase.h` header defines `CreateDirectoryEx` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[.CreateDirectory](#)

[.CreateDirectoryTransacted](#)

[CreateFile](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[RemoveDirectory](#)

[SECURITY_ATTRIBUTES](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateDirectoryExW function (winbase.h)

Article06/01/2023

Creates a new directory with the attributes of a specified template directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory. The new directory retains the other attributes of the specified template directory.

To perform this operation as a transacted operation, use the [.CreateDirectoryTransacted](#) function.

Syntax

C++

```
BOOL CreateDirectoryExW(
    [in]          LPCWSTR      lpTemplateDirectory,
    [in]          LPCWSTR      lpNewDirectory,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Parameters

[in] lpTemplateDirectory

The path of the directory to use as a template when creating the new directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpNewDirectory

The path of the directory to be created.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The **IpSecurityDescriptor** member of the structure specifies a security descriptor for the new directory.

If *lpSecurityAttributes* is **NULL**, the directory gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a directory are inherited from its parent directory.

The target file system must support security on files and directories for this parameter to have an effect. This is indicated when [GetVolumeInformation](#) returns **FS_PERSISTENT_ACLS**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#). Possible errors include the following.

Return code	Description
ERROR_ALREADY_EXISTS	The specified directory already exists.
ERROR_PATH_NOT_FOUND	One or more intermediate directories do not exist. This function only creates the final directory in the path. To create all intermediate directories on the path, use the SHCreateDirectoryEx function.

Remarks

The [.CreateDirectoryEx](#) function allows you to create directories that inherit stream information from other directories. This function is useful, for example, when you are using Macintosh directories, which have a resource stream that is needed to properly identify directory contents as an attribute.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

You can obtain a handle to a directory by calling the [CreateFile](#) function with the **FILE_FLAG_BACKUP_SEMANTICS** flag set. For a code example, see [CreateFile](#).

To support inheritance functions that query the security descriptor of this object can heuristically determine and report that inheritance is in effect. For more information, see [Automatic Propagation of Inheritable ACEs](#).

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

Note

The `winbase.h` header defines `CreateDirectoryEx` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[.CreateDirectory](#)

[.CreateDirectoryTransacted](#)

[CreateFile](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[RemoveDirectory](#)

[SECURITY_ATTRIBUTES](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateDirectoryTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Creates a new directory as a transacted operation, with the attributes of a specified template directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory. The new directory retains the other attributes of the specified template directory.

Syntax

C++

```
BOOL CreateDirectoryTransactedA(
    [in, optional] LPCSTR             lpTemplateDirectory,
    [in]          LPCSTR             lpNewDirectory,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in]          HANDLE            hTransaction
);
```

Parameters

`[in, optional] lpTemplateDirectory`

The path of the directory to use as a template when creating the new directory. This parameter can be **NULL**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The directory must reside on the local computer; otherwise, the function fails and the last error code is set to [ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE](#).

[in] `lpNewDirectory`

The path of the directory to be created.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] `lpSecurityAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The `IpSecurityDescriptor` member of the structure specifies a security descriptor for the new directory.

If `lpSecurityAttributes` is `NULL`, the directory gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a directory are inherited from its parent directory.

The target file system must support security on files and directories for this parameter to have an effect. This is indicated when [GetVolumeInformation](#) returns `FS_PERSISTENT_ACLS`.

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#). Possible errors include the following.

Return code	Description
ERROR_ALREADY_EXISTS	The specified directory already exists.
ERROR_EFS_NOT_ALLOWED_IN_TRANSACTION	You cannot create a child directory with a parent directory that has encryption disabled.
ERROR_PATH_NOT_FOUND	One or more intermediate directories do not exist. This function only creates the final directory in the path.

Remarks

The [.CreateDirectoryTransacted](#) function allows you to create directories that inherit stream information from other directories. This function is useful, for example, when you are using Macintosh directories, which have a resource stream that is needed to properly identify directory contents as an attribute.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

This function fails with **ERROR_EFS_NOT_ALLOWED_IN_TRANSACTION** if you try to create a child directory with a parent directory that has encryption disabled.

You can obtain a handle to a directory by calling the [CreateFileTransacted](#) function with the **FILE_FLAG_BACKUP_SEMANTICS** flag set.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The winbase.h header defines CreateDirectoryTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFileTransacted](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[RemoveDirectoryTransacted](#)

[SECURITY_ATTRIBUTES](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

CreateDirectoryTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Creates a new directory as a transacted operation, with the attributes of a specified template directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory. The new directory retains the other attributes of the specified template directory.

Syntax

C++

```
BOOL CreateDirectoryTransacted(
    [in, optional] LPCWSTR             lpTemplateDirectory,
    [in]          LPCWSTR             lpNewDirectory,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in]          HANDLE              hTransaction
);
```

Parameters

`[in, optional] lpTemplateDirectory`

The path of the directory to use as a template when creating the new directory. This parameter can be **NULL**.

The directory must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] `lpNewDirectory`

The path of the directory to be created.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details..

[in, optional] `lpSecurityAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The `lpSecurityDescriptor` member of the structure specifies a security descriptor for the new directory.

If `lpSecurityAttributes` is **NULL**, the directory gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a directory are inherited from its parent directory.

The target file system must support security on files and directories for this parameter to have an effect. This is indicated when [GetVolumeInformation](#) returns **FS_PERSISTENT_ACLS**.

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#). Possible errors include the following.

Return code	Description
ERROR_ALREADY_EXISTS	The specified directory already exists.
ERROR_EFS_NOT_ALLOWED_IN_TRANSACTION	You cannot create a child directory with a parent directory that has encryption disabled.
ERROR_PATH_NOT_FOUND	One or more intermediate directories do not exist. This function only creates the final directory in the path.

Remarks

The [.CreateDirectoryTransacted](#) function allows you to create directories that inherit stream information from other directories. This function is useful, for example, when you are using Macintosh directories, which have a resource stream that is needed to properly identify directory contents as an attribute.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

This function fails with **ERROR_EFS_NOT_ALLOWED_IN_TRANSACTION** if you try to create a child directory with a parent directory that has encryption disabled.

You can obtain a handle to a directory by calling the [CreateFileTransacted](#) function with the **FILE_FLAG_BACKUP_SEMANTICS** flag set.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The winbase.h header defines CreateDirectoryTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFileTransacted](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[RemoveDirectoryTransacted](#)

[SECURITY_ATTRIBUTES](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

CreateFiber function (winbase.h)

Article07/27/2022

Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber.

To specify both a commit and reserve stack size, use the [CreateFiberEx](#) function.

Syntax

C++

```
LPVOID CreateFiber(
    [in]          SIZE_T           dwStackSize,
    [in]          LPFIBER_START_ROUTINE lpStartAddress,
    [in, optional] LPVOID          lpParameter
);
```

Parameters

[in] dwStackSize

The initial committed size of the stack, in bytes. If this parameter is zero, the new fiber uses the default commit stack size for the executable. For more information, see [Thread Stack Size](#).

[in] lpStartAddress

A pointer to the application-defined function to be executed by the fiber and represents the starting address of the fiber. Execution of the newly created fiber does not begin until another fiber calls the [SwitchToFiber](#) function with this address. For more information of the fiber callback function, see [FiberProc](#).

[in, optional] lpParameter

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the [GetFiberData](#) macro.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks

The number of fibers a process can create is limited by the available virtual memory. For example, if you create each fiber with 1 megabyte of reserved stack space, you can create at most 2028 fibers. If you reduce the default stack size by using the STACKSIZE statement in the module definition (.def) file or by using [CreateFiberEx](#), you can create more fibers. However, your application will have better performance if you use an alternate strategy for processing requests instead of creating such a large number of fibers.

Before a thread can schedule a fiber using the [SwitchToFiber](#) function, it must call the [ConvertThreadToFiber](#) function so there is a fiber associated with the thread.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Examples

For an example, see [Using Fibers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConvertThreadToFiber](#)

[CreateFiberEx](#)

[FiberProc](#)

[Fibers](#)

[GetFiberData](#)

[Process and Thread Functions](#)

[SwitchToFiber](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateFiberEx function (winbase.h)

Article07/27/2022

Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber.

Syntax

C++

```
LPVOID CreateFiberEx(
    [in]          SIZE_T      dwStackCommitSize,
    [in]          SIZE_T      dwStackReserveSize,
    [in]          DWORD       dwFlags,
    [in]          LPFIBER_START_ROUTINE lpStartAddress,
    [in, optional] LPVOID      lpParameter
);
```

Parameters

[in] dwStackCommitSize

The initial commit size of the stack, in bytes. If this parameter is zero, the new fiber uses the default commit stack size for the executable. For more information, see [Thread Stack Size](#).

[in] dwStackReserveSize

The initial reserve size of the stack, in bytes. If this parameter is zero, the new fiber uses the default reserved stack size for the executable. For more information, see [Thread Stack Size](#).

[in] dwFlags

If this parameter is zero, the floating-point state on x86 systems is not switched and data can be corrupted if a fiber uses floating-point arithmetic. If this parameter is **FIBER_FLAG_FLOAT_SWITCH**, the floating-point state is switched for the fiber.

Windows XP: The **FIBER_FLAG_FLOAT_SWITCH** flag is not supported.

[in] lpStartAddress

A pointer to the application-defined function to be executed by the fiber and represents the starting address of the fiber. Execution of the newly created fiber does not begin until another fiber calls the [SwitchToFiber](#) function with this address. For more information on the fiber callback function, see [FiberProc](#).

[in, optional] lpParameter

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the [GetFiberData](#) macro.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks

The number of fibers a process can create is limited by the available virtual memory. By default, every fiber has 1 megabyte of reserved stack space. Therefore, you can create at most 2028 fibers. If you reduce the default stack size, you can create more fibers.

However, your application will have better performance if you use an alternate strategy for processing requests.

Before a thread can schedule a fiber using the [SwitchToFiber](#) function, it must call the [ConvertThreadToFiber](#) function so there is a fiber associated with the thread.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConvertThreadToFiber](#)

[FiberProc](#)

[Fibers](#)

[GetFiberData](#)

[Process and Thread Functions](#)

[SwitchToFiber](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateFileMappingA function (winbase.h)

Article07/27/2022

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++

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

Parameters

[in] `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.

[in, optional] `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](#).

[in] *f1Protect*

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 0x20	<p>Allows views to be mapped for read-only, copy-on-write, or execute access.</p> <p>The file handle specified by the <i>hFile</i> parameter must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.</p> <p>Windows Server 2003 and Windows XP: This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1.</p>
PAGE_EXECUTE_READWRITE 0x40	<p>Allows views to be mapped for read-only, copy-on-write, read/write, or execute access.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ, GENERIC_WRITE, and GENERIC_EXECUTE access rights.</p> <p>Windows Server 2003 and Windows XP: This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1.</p>
PAGE_EXECUTE_WRITECOPY 0x80	<p>Allows views to be mapped for read-only, copy-on-write, or execute access. This value is equivalent to PAGE_EXECUTE_READ.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.</p> <p>Windows Vista: This value is not available until Windows Vista with SP1.</p> <p>Windows Server 2003 and Windows XP: This value is not supported.</p>

PAGE_READONLY 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. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.
PAGE_READWRITE 0x04	Allows views to be mapped for read-only, copy-on-write, or read/write access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_WRITE access rights.
PAGE_WRITECOPY 0x08	Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to PAGE_READONLY . The file handle that the <i>hFile</i> 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 0x8000000	If the file mapping object is backed by the operating system paging file (the <i>hfile</i> 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. This attribute has no effect for file mapping objects that are backed by executable image files or data files (the <i>hfile</i> parameter is a handle to a file). SEC_COMMIT cannot be combined with SEC_RESERVE . If no attribute is specified, SEC_COMMIT is assumed.
SEC_IMAGE 0x1000000	Specifies that the file that the <i>hFile</i> parameter specifies is an executable image file. 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.

No other attributes are valid with **SEC_IMAGE**.

SEC_IMAGE_NO_EXECUTE

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.

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**.

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

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).

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.

SEC_LARGE_PAGES requires the [SeLockMemoryPrivilege](#) privilege to be enabled in the caller's token.

If **SEC_LARGE_PAGES** is specified, **SEC_COMMIT** must also be specified.

Windows Server 2003: This value is not supported until Windows Server 2003 with SP1.

Windows XP: This value is not supported.

SEC_NOCACHE

0x10000000

Sets all pages to be non-cacheable.

Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with

	<p>SEC_NOCACHE can result in an EXCEPTION_ILLEGAL_INSTRUCTION exception.</p>
	<p>SEC_NOCACHE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p>
SEC_RESERVE 0x4000000	<p>If the file mapping object is backed by the operating system paging file (the <i>hfile</i> 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.</p> <p>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.</p> <p>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the <i>hfile</i> parameter is a handle to a file).</p> <p>SEC_RESERVE cannot be combined with SEC_COMMIT.</p>
SEC_WRITECOMBINE 0x40000000	<p>Sets all pages to be write-combined. 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.</p> <p>SEC_WRITECOMBINE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p> <p>Windows Server 2003 and Windows XP: This flag is not supported until Windows Vista.</p>

[in] dwMaximumSizeHigh

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

[in] 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.

[in, optional] *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 [GetLastError](#) returns **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](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateFileMappingNumaA function (winbase.h)

Article07/27/2022

Creates or opens a named or unnamed file mapping object for a specified file and specifies the NUMA node for the physical memory.

Syntax

C++

```
HANDLE CreateFileMappingNumaA(
    [in]          HANDLE           hFile,
    [in, optional] LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    [in]          DWORD            flProtect,
    [in]          DWORD            dwMaximumSizeHigh,
    [in]          DWORD            dwMaximumSizeLow,
    [in, optional] LPCSTR          lpName,
    [in]          DWORD            nndPreferred
);
```

Parameters

[in] `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, `CreateFileMappingNuma` 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.

[in, optional] `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](#).

[in] *f1Protect*

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 0x20	Allows views to be mapped for read-only, copy-on-write, or execute access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.
PAGE_EXECUTE_READWRITE 0x40	Allows views to be mapped for read-only, copy-on-write, read/write or execute access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ , GENERIC_WRITE , and GENERIC_EXECUTE access rights.
PAGE_EXECUTE_WRITECOPY 0x80	Allows views to be mapped for read-only, copy-on-write, or execute access. This value is equivalent to PAGE_EXECUTE_READ . The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights. Windows Vista: This value is not available until Windows Vista with SP1.
PAGE_READONLY 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. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.
PAGE_READWRITE 0x04	Allows views to be mapped for read-only, copy-on-write, or read/write access.

	The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_WRITE access rights.
PAGE_WRITECOPY 0x08	Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to PAGE_READONLY .
	The file handle that the <i>hFile</i> 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 0x8000000	Allocates physical storage in memory or the paging file for all pages. This is the default setting.
SEC_IMAGE 0x1000000	Sets the file that is specified to be an executable image file. 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. No other attributes are valid with SEC_IMAGE .
SEC_IMAGE_NO_EXECUTE 0x11000000	Specifies that the file that the <i>hFile</i> 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. 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 . Windows Server 2008 R2, Windows 7, Windows Server 2008 and Windows Vista: This value is not supported before Windows Server 2012 and Windows 8.
SEC_LARGE_PAGES 0x80000000	Enables large pages to be used when mapping images or backing from the pagefile, but not when mapping data

	<p>for regular files. Be sure to specify the maximum size of the file mapping object as the minimum size of a large page reported by the GetLargePageMinimum function and to enable the SeLockMemoryPrivilege privilege.</p>
SEC_NOCACHE 0x10000000	<p>Sets all pages to noncacheable. Applications should not use this flag except when explicitly required for a device. Using the interlocked functions with memory mapped with SEC_NOCACHE can result in an EXCEPTION_ILLEGAL_INSTRUCTION exception.</p> <p>SEC_NOCACHE requires either SEC_RESERVE or SEC_COMMIT to be set.</p>
SEC_RESERVE 0x4000000	<p>Reserves all pages without allocating physical storage. The reserved range of pages cannot be used by any other allocation operations until the range of pages is released. Reserved pages can be identified in subsequent calls to the VirtualAllocExNuma function. This attribute is valid only if the <i>hFile</i> parameter is INVALID_HANDLE_VALUE (that is, a file mapping object that is backed by the system paging file).</p>
SEC_WRITECOMBINE 0x40000000	<p>Sets all pages to be write-combined. 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.</p> <p>SEC_WRITECOMBINE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p>

[in] dwMaximumSizeHigh

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

[in] dwMaximumSizeLow

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

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

[in, optional] *lpName*

The name of the file mapping object.

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

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

If the *lpName* parameter 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 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 so that applications can support multiple users.

[in] *nndPreferred*

The NUMA node where the physical memory should reside.

Value	Meaning
NUMA_NO_PREFERRED_NODE 0xffffffff	No NUMA node is preferred. This is the same as calling the CreateFileMapping function.

Return value

If the function succeeds, the return value is a handle to the 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 the [GetLastError](#) function returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call the [GetLastError](#) function.

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.

The file mapping object can be shared by duplication, inheritance, or by name. The initial contents of the pages in a file mapping object backed by the page file are 0 (zero).

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 cannot be increased, the result is a failure to create the file mapping object and the [GetLastError](#) function returns **ERROR_DISK_FULL**.

The handle that the [CreateFileMappingNuma](#) function 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. A file mapping object can be shared through process creation, handle duplication, or by name. For more information, see the [DuplicateHandle](#) and [OpenFileMapping](#) functions.

Creating a file mapping object creates the potential for mapping a view of the file but does not map the view. The [MapViewOfFileExNuma](#) function maps a view of a file into a process address space.

With one important exception, file views derived from a single file mapping object are coherent or identical at a specific time. If multiple processes have handles of the same file mapping object, they see a coherent view of the data when they map a view of the file.

The exception is related to remote files. Although the [CreateFileMappingNuma](#) function 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 sees only its own writes to the page. When the data gets updated on the disk, the page 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.

To fully close a file mapping object, an application must unmap all mapped views of the file mapping object by calling the [UnMapViewOfFile](#) function and then close the file mapping object handle by calling the [CloseHandle](#) function.

These functions can be called in any order. The call to the [UnMapViewOfFile](#) function is necessary, because mapped views of a file mapping object maintain internal open handles to the object, and a file mapping object does not close until all open handles to it are closed.

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 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, any process running in any session can access that file mapping object provided that the caller has the required access rights.

Use structured exception handling to protect any code that writes to or reads from a memory mapped view. For more information, see [Reading and Writing From a File View](#).

To have a mapping with executable permissions, an application must call the [CreateFileMappingNuma](#) function with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ** and then call the [MapViewOfFileExNuma](#) function 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

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h, Memoryapi.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileMapping](#)

[DuplicateHandle](#)

[File Mapping Functions](#)

[MapViewOfFileExNuma](#)

[NUMA Support](#)

[OpenFileMapping](#)

[ReadFile](#)

[SECURITY_ATTRIBUTES](#)

[UnmapViewOfFile](#)

[VirtualAllocExNuma](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

CreateFileTransactedA function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Creates or opens a file, file stream, or directory as a transacted operation. The function returns a handle that can be used to access the object.

To perform this operation as a nontransacted operation or to access objects other than files (for example, named pipes, physical devices, mailslots), use the [CreateFile](#) function.

For more information about transactions, see the Remarks section of this topic.

Syntax

C++

```
HANDLE CreateFileTransactedA(
    [in]          LPCSTR             lpFileName,
    [in]          DWORD              dwDesiredAccess,
    [in]          DWORD              dwShareMode,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in]          DWORD              dwCreationDisposition,
    [in]          DWORD              dwFlagsAndAttributes,
    [in, optional] HANDLE             hTemplateFile,
    [in]          HANDLE             hTransaction,
    [in, optional] PUSHORT           pusMiniVersion,
                                PVOID              lpExtendedParameter
);
```

Parameters

[in] lpFileName

The name of an object to be created or opened.

The object must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

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](#).

[in] dwDesiredAccess

The access to the object, 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](#) and [File Security and Access Rights](#).

If this parameter is zero, the application can query file, directory, or device attributes without accessing that file or device. For more information, see the Remarks section of this topic.

You cannot request an access mode that conflicts with the sharing mode that is specified in an open request that has an open handle. For more information, see [Creating and Opening Files](#).

[in] dwShareMode

The sharing mode of an object, which can be read, write, both, delete, all of these, or none (refer to the following table).

If this parameter is zero and **CreateFileTransacted** succeeds, the object cannot be shared and cannot be opened again until the handle is closed. For more information, see the Remarks section of this topic.

You cannot request a sharing mode that conflicts with the access mode that is specified in an open request that has an open handle, because that would result in the following sharing violation: **ERROR_SHARING_VIOLATION**. For more information, see [Creating and Opening Files](#).

To enable a process to share an object while another process has the object open, use a combination of one or more of the following values to specify the access mode they can request to open the object.

Note The sharing options for each open handle remain in effect until that handle is closed, regardless of process context.

Value	Meaning
0 0x00000000	Disables subsequent open operations on an object to request any type of access to that object.
FILE_SHARE_DELETE 0x00000004	Enables subsequent open operations on an object to request delete access. Otherwise, other processes cannot open the object if they request delete access. If this flag is not specified, but the object has been opened for delete access, the function fails.
FILE_SHARE_READ 0x00000001	Enables subsequent open operations on an object to request read access. Otherwise, other processes cannot open the object if they request read access. If this flag is not specified, but the object has been opened for read access, the function fails.
FILE_SHARE_WRITE 0x00000002	Enables subsequent open operations on an object to request write access. Otherwise, other processes cannot open the object if they request write access. If this flag is not specified, but the object has been opened for write access or has a file mapping with write access, the function fails.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure that contains an optional [security descriptor](#) and also determines whether or not the returned handle can be inherited by child processes. The parameter can be **NULL**.

If the *lpSecurityAttributes* parameter is **NULL**, the handle returned by [CreateFileTransacted](#) cannot be inherited by any child processes your application may

create and the object associated with the returned handle gets a default security descriptor.

The **bInheritHandle** member of the structure specifies whether the returned handle can be inherited.

The **lpSecurityDescriptor** member of the structure specifies a [security descriptor](#) for an object, but may also be **NULL**.

If **lpSecurityDescriptor** member is **NULL**, the object associated with the returned handle is assigned a default security descriptor.

CreateFileTransacted ignores the **lpSecurityDescriptor** member when opening an existing file, but continues to use the **bInheritHandle** member.

For more information, see the Remarks section of this topic.

[in] dwCreationDisposition

An action to take on files that exist and do not exist.

For more information, see the Remarks section of this topic.

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 truncates 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 does not exist, the function fails and the last-error code is set to ERROR_FILE_NOT_FOUND (2). For more information, see the Remarks section of this topic.
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 <i>dwDesiredAccess</i> parameter.

[in] dwFlagsAndAttributes

The file attributes and flags, **FILE_ATTRIBUTE_NORMAL** being the most common default value.

This parameter can include any combination of the available file attributes (**FILE_ATTRIBUTE_***). All other file attributes override **FILE_ATTRIBUTE_NORMAL**.

This parameter can also contain combinations of flags (**FILE_FLAG_**) for control of buffering behavior, access modes, and other special-purpose flags. These combine with any **FILE_ATTRIBUTE_** values.

This parameter can also contain Security Quality of Service (SQOS) information by specifying the **SECURITY_SQOS_PRESENT** flag. Additional SQOS-related flags information is presented in the table following the attributes and flags tables.

Note

When **CreateFileTransacted** opens an existing file, it generally combines the file flags with the file attributes of the existing file, and ignores any file attributes supplied as part of *dwFlagsAndAttributes*. Special cases are detailed in **Creating and Opening Files**.

The following file attributes and flags are used only for file objects, not other types of objects that **CreateFileTransacted** opens (additional information can be found in the

Remarks section of this topic). For more advanced access to file attributes, see [SetFileAttributes](#). For a complete list of all file attributes with their values and descriptions, see [File Attribute Constants](#).

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE 32 (0x20)	The file should be archived. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_ENCRYPTED 16384 (0x4000)	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories. For more information, see File Encryption .
	This flag has no effect if FILE_ATTRIBUTE_SYSTEM is also specified.
FILE_ATTRIBUTE_HIDDEN 2 (0x2)	The file is hidden. Do not include it in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL 128 (0x80)	The file does not have other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE 4096 (0x1000)	The data of a file is not immediately available. This attribute indicates that file data is physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY 1 (0x1)	The file is read only. Applications can read the file, but cannot write to or delete it.
FILE_ATTRIBUTE_SYSTEM 4 (0x4)	The file is part of or used exclusively by an operating system.
FILE_ATTRIBUTE_TEMPORARY 256 (0x100)	The file is being used for temporary storage. File systems avoid writing data back to mass storage if sufficient cache memory is available, because an application deletes a temporary file after a handle is closed. In that case, the system can entirely avoid writing the data. Otherwise, the data is written after the handle is closed.

Flag	Meaning
FILE_FLAG_BACKUP_SEMANTICS 0x02000000	The file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks when the process has SE_BACKUP_NAME and SE_RESTORE_NAME

privileges. For more information, see [Changing Privileges in a Token](#).

You must set this flag to obtain a handle to a directory. A directory handle can be passed to some functions instead of a file handle. For more information, see [Directory Handles](#).

FILE_FLAG_DELETE_ON_CLOSE 0x04000000	<p>The file is to be deleted immediately after the last transacted writer handle to the file is closed, provided that the transaction is still active. If a file has been marked for deletion and a transacted writer handle is still open after the transaction completes, the file will not be deleted.</p> <p>If there are existing open handles to a file, the call fails unless they were all opened with the FILE_SHARE_DELETE share mode.</p> <p>Subsequent open requests for the file fail, unless the FILE_SHARE_DELETE share mode is specified.</p>
--	---

FILE_FLAG_NO_BUFFERING 0x20000000	<p>The file is being opened with no system caching. This flag does not affect hard disk caching or memory mapped files. When combined with FILE_FLAG_OVERLAPPED, the flag gives maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations take more time, because data is not being held in the cache. Also, the file metadata may still be cached. To flush the metadata to disk, use the FlushFileBuffers function.</p> <p>An application must meet certain requirements when working with files that are opened with FILE_FLAG_NO_BUFFERING:</p>
---	---

- File access must begin at byte offsets within a file that are integer multiples of the volume sector size.
- File access must be for numbers of bytes that are integer multiples of the volume sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, 1536, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations should be sector aligned, which means aligned on addresses in memory that are integer multiples of the volume sector size. Depending on the disk, this requirement may not be enforced.

One way to align buffers on integer multiples of the volume sector size is to use [VirtualAlloc](#) to allocate the

buffers. It allocates memory that is aligned on addresses that are integer multiples of the operating system's memory page size. Because both memory page and volume sector sizes are powers of 2, this memory is also aligned on addresses that are integer multiples of a volume sector size. Memory pages are 4 or 8 KB in size; sectors are 512 bytes (hard disks), 2048 bytes (CD), or 4096 bytes (hard disks), and therefore, volume sectors can never be larger than memory pages.

An application can determine a volume sector size by calling the [GetDiskFreeSpace](#) function.

FILE_FLAG_OPEN_NO_RECALL 0x00100000	The file data is requested, but it should continue to be located in remote storage. It should not be transported back to local storage. This flag is for use by remote storage systems.
FILE_FLAG_OPEN_REPARSE_POINT 0x00200000	Normal reparse point processing will not occur; CreateFileTransacted will attempt to open the reparse point. When a file is opened, a file handle is returned, whether or not the filter that controls the reparse point is operational. This flag cannot be used with the CREATE_ALWAYS flag. If the file is not a reparse point, then this flag is ignored.
FILE_FLAG_OVERLAPPED 0x40000000	<p>The file is being opened or created for asynchronous I/O. When the operation is complete, the event specified in the OVERLAPPED structure is set to the signaled state. Operations that take a significant amount of time to process return ERROR_IO_PENDING.</p> <p>If this flag is specified, the file can be used for simultaneous read and write operations. The system does not maintain the file pointer, therefore you must pass the file position to the read and write functions in the OVERLAPPED structure or update the file pointer.</p> <p>If this flag is not specified, then I/O operations are serialized, even if the calls to the read and write functions specify an OVERLAPPED structure.</p>
FILE_FLAG_POSIX_SEMANTICS 0x01000000	The file is to be accessed according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support that naming. Use care when using this option, because files created with this flag may not be accessible by applications that are written for MS-DOS or 16-bit Windows.
FILE_FLAG_RANDOM_ACCESS 0x10000000	The file is to be accessed randomly. The system can use this as a hint to optimize file caching.

FILE_FLAG_SESSION_AWARE 0x00800000	<p>The file or device is being opened with session awareness. If this flag is not specified, then per-session devices (such as a device using RemoteFX USB Redirection) cannot be opened by processes running in session 0. This flag has no effect for callers not in session 0. This flag is supported only on server editions of Windows.</p> <p>Windows Server 2008 R2 and Windows Server 2008: This flag is not supported before Windows Server 2012.</p>
FILE_FLAG_SEQUENTIAL_SCAN 0x08000000	<p>The file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur. However, correct operation is still guaranteed. Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.</p> <p>This flag has no effect if the file system does not support cached I/O and FILE_FLAG_NO_BUFFERING.</p>
FILE_FLAG_WRITE_THROUGH 0x80000000	<p>Write operations will not go through any intermediate cache, they will go directly to disk. If FILE_FLAG_NO_BUFFERING is not also specified, so that system caching is in effect, then the data is written to the system cache, but is flushed to disk without delay.</p> <p>If FILE_FLAG_NO_BUFFERING is also specified, so that system caching is not in effect, then the data is immediately flushed to disk without going through the system cache. The operating system also requests a write-through the hard disk cache to persistent media. However, not all hardware supports this write-through capability.</p>

The *dwFlagsAndAttributes* parameter can also specify Security Quality of Service information. For more information, see [Impersonation Levels](#). When the calling application specifies the **SECURITY_SQOS_PRESENT** flag as part of *dwFlagsAndAttributes*, it can also contain one or more of the following values.

Security flag	Meaning
SECURITY_ANONYMOUS	Impersonates a client at the Anonymous impersonation level.

SECURITY_CONTEXT_TRACKING	The security tracking mode is dynamic. If this flag is not specified, the security tracking mode is static.
SECURITY_DELEGATION	Impersonates a client at the Delegation impersonation level.
SECURITY_EFFECTIVE_ONLY	Only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available. This allows the client to limit the groups and privileges that a server can use while impersonating the client.
SECURITY_IDENTIFICATION	Impersonates a client at the Identification impersonation level.
SECURITY_IMPERSONATION	Impersonate a client at the impersonation level. This is the default behavior if no other flags are specified along with the SECURITY_SQOS_PRESENT flag.

[in, optional] `hTemplateFile`

A valid handle to a template file with the **GENERIC_READ** access right. The template file supplies file attributes and extended attributes for the file that is being created. This parameter can be **NULL**.

When opening an existing file, [CreateFileTransacted](#) ignores the template file.

When opening a new EFS-encrypted file, the file inherits the DACL from its parent directory.

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

[in, optional] `pusMiniVersion`

The miniversion to be opened. If the transaction specified in *hTransaction* is not the transaction that is modifying the file, this parameter should be **NULL**. Otherwise, this parameter can be a miniversion identifier returned by the [FSCTL_TXFS_CREATE_MINIVERSION](#) control code, or one of the following values.

Value	Meaning
TXFS_MINIVERSION_COMMITTED_VIEW 0x0000	The view of the file as of its last commit.
TXFS_MINIVERSION_DIRTY_VIEW 0xFFFF	The view of the file as it is being modified by the transaction.

TXFS_MINIVERSION_DEFAULT_VIEW	Either the committed or dirty view of the file, depending on the context. A transaction that is modifying the file gets the dirty view, while a transaction that is not modifying the file gets the committed view.
0xFFFFE	

`lpExtendedParameter`

This parameter is reserved and must be **NULL**.

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

When using the handle returned by [CreateFileTransacted](#), use the transacted version of file I/O functions instead of the standard file I/O functions where appropriate. For more information, see [Programming Considerations for Transactional NTFS](#).

When opening a transacted handle to a directory, that handle must have **FILE_WRITE_DATA** (**FILE_ADD_FILE**) and **FILE_APPEND_DATA** (**FILE_ADD_SUBDIRECTORY**) permissions. These are included in **FILE_GENERIC_WRITE** permissions. You should open directories with fewer permissions if you are just using the handle to create files or subdirectories; otherwise, sharing violations can occur.

You cannot open a file with **FILE_EXECUTE** access level when that file is a part of another transaction (that is, another application opened it by calling [CreateFileTransacted](#)). This means that [CreateFileTransacted](#) fails if the access level **FILE_EXECUTE** or **FILE_ALL_ACCESS** is specified.

When a non-transacted application calls [CreateFileTransacted](#) with **MAXIMUM_ALLOWED** specified for *lpSecurityAttributes*, a handle is opened with the same access level every time. When a transacted application calls [CreateFileTransacted](#) with **MAXIMUM_ALLOWED** specified for *lpSecurityAttributes*, a handle is opened with a differing amount of access based on whether the file is locked by a transaction. For example, if the calling application has **FILE_EXECUTE** access level for a file, the application only obtains this access if the file that is being opened is either not locked

by a transaction, or is locked by a transaction and the application is already a transacted reader for that file.

See [Transactional NTFS](#) for a complete description of transacted operations.

Use the [CloseHandle](#) function to close an object handle returned by [CreateFileTransacted](#) when the handle is no longer needed, and prior to committing or rolling back the transaction.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes that are formatted for that kind of file system, a new file inherits the compression and encryption attributes of its directory.

You cannot use [CreateFileTransacted](#) to control compression on a file or directory. For more information, see [File Compression and Decompression](#), and [File Encryption](#).

Symbolic link behavior—if the call to this function creates a new file, there is no change in behavior.

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.

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). A single-sector write is atomic. Multi-sector writes that are cached may not always be written to the disk; therefore, specify **FILE_FLAG_WRITE_THROUGH** to ensure that an entire multi-sector write is written to the disk without caching.

As stated previously, if the *lpSecurityAttributes* parameter is **NULL**, the handle returned by [CreateFileTransacted](#) cannot be inherited by any child processes your application may create. The following information regarding this parameter also applies:

- If **bInheritHandle** 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 **IpSecurityDescriptor** 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	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support TxF.

Files

If you try to create a file on a floppy drive that does not have a floppy disk or a CD-ROM drive that does not have a CD, the system displays a message for the user to insert a disk or a CD. To prevent the system from displaying this message, call the [SetErrorMode](#) function with **SEM_FAILCRITICALERRORS**.

For more information, see [Creating and Opening 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 [CreateFileTransacted](#) 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** 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**.

File Streams

On NTFS file systems, you can use [CreateFileTransacted](#) to create separate streams within a file.

For more information, see [File Streams](#).

Directories

An application cannot create a directory by using [CreateFileTransacted](#), therefore only the **OPEN_EXISTING** value is valid for *dwCreationDisposition* for this use case. To create a directory, the application must call [.CreateDirectoryTransacted](#), [.CreateDirectory](#) or [.CreateDirectoryEx](#).

To open a directory using [CreateFileTransacted](#), specify the **FILE_FLAG_BACKUP_SEMANTICS** flag as part of *dwFlagsAndAttributes*. Appropriate security checks still apply when this flag is used without **SE_BACKUP_NAME** and **SE_RESTORE_NAME** privileges.

When using [CreateFileTransacted](#) 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](#).

Note

The winbase.h header defines [CreateFileTransacted](#) as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CopyFileTransacted](#)

[CreateDirectoryTransacted](#)

[DeleteFileTransacted](#)

[File Compression and Decompression](#)

[File Encryption](#)

[File Management Functions](#)

[File Security and Access Rights](#)

[File Streams](#)

[FindFirstFileTransacted](#)

Functions

[GetFileAttributesTransacted](#)

[MoveFileTransacted](#)

Overview Topics

[Programming Considerations for Transactional NTFS](#)

[ReadFile](#)

[Transactional NTFS \(TxF\)](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

CreateFileTransactedW function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Creates or opens a file, file stream, or directory as a transacted operation. The function returns a handle that can be used to access the object.

To perform this operation as a nontransacted operation or to access objects other than files (for example, named pipes, physical devices, mailslots), use the [CreateFile](#) function.

For more information about transactions, see the Remarks section of this topic.

Syntax

C++

```
HANDLE CreateFileTransactedW(
    [in]          LPCWSTR             lpFileName,
    [in]          DWORD               dwDesiredAccess,
    [in]          DWORD               dwShareMode,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in]          DWORD               dwCreationDisposition,
    [in]          DWORD               dwFlagsAndAttributes,
    [in, optional] HANDLE              hTemplateFile,
    [in]          HANDLE              hTransaction,
    [in, optional] PUSHORT            pusMiniVersion,
    [in]          PVOID               lpExtendedParameter
);
```

Parameters

`[in] lpFileName`

The name of an object to be created or opened.

The object must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

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](#).

[in] dwDesiredAccess

The access to the object, 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](#) and [File Security and Access Rights](#).

If this parameter is zero, the application can query file, directory, or device attributes without accessing that file or device. For more information, see the Remarks section of this topic.

You cannot request an access mode that conflicts with the sharing mode that is specified in an open request that has an open handle. For more information, see [Creating and Opening Files](#).

[in] dwShareMode

The sharing mode of an object, which can be read, write, both, delete, all of these, or none (refer to the following table).

If this parameter is zero and **CreateFileTransacted** succeeds, the object cannot be shared and cannot be opened again until the handle is closed. For more information, see the Remarks section of this topic.

You cannot request a sharing mode that conflicts with the access mode that is specified in an open request that has an open handle, because that would result in the following sharing violation: **ERROR_SHARING_VIOLATION**. For more information, see [Creating and Opening Files](#).

To enable a process to share an object while another process has the object open, use a combination of one or more of the following values to specify the access mode they can request to open the object.

Note The sharing options for each open handle remain in effect until that handle is closed, regardless of process context.

Value	Meaning
0 0x00000000	Disables subsequent open operations on an object to request any type of access to that object.
FILE_SHARE_DELETE 0x00000004	Enables subsequent open operations on an object to request delete access. Otherwise, other processes cannot open the object if they request delete access. If this flag is not specified, but the object has been opened for delete access, the function fails.
FILE_SHARE_READ 0x00000001	Enables subsequent open operations on an object to request read access. Otherwise, other processes cannot open the object if they request read access. If this flag is not specified, but the object has been opened for read access, the function fails.
FILE_SHARE_WRITE 0x00000002	Enables subsequent open operations on an object to request write access. Otherwise, other processes cannot open the object if they request write access. If this flag is not specified, but the object has been opened for write access or has a file mapping with write access, the function fails.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure that contains an optional [security descriptor](#) and also determines whether or not the returned handle can be inherited by child processes. The parameter can be **NULL**.

If the *lpSecurityAttributes* parameter is **NULL**, the handle returned by [CreateFileTransacted](#) cannot be inherited by any child processes your application may

create and the object associated with the returned handle gets a default security descriptor.

The **bInheritHandle** member of the structure specifies whether the returned handle can be inherited.

The **lpSecurityDescriptor** member of the structure specifies a [security descriptor](#) for an object, but may also be **NULL**.

If **lpSecurityDescriptor** member is **NULL**, the object associated with the returned handle is assigned a default security descriptor.

CreateFileTransacted ignores the **lpSecurityDescriptor** member when opening an existing file, but continues to use the **bInheritHandle** member.

For more information, see the Remarks section of this topic.

[in] dwCreationDisposition

An action to take on files that exist and do not exist.

For more information, see the Remarks section of this topic.

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 does not exist, the function fails and the last-error code is set to ERROR_FILE_NOT_FOUND (2). For more information, see the Remarks section of this topic.
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 <i>dwDesiredAccess</i> parameter.

[in] `dwFlagsAndAttributes`

The file attributes and flags, **FILE_ATTRIBUTE_NORMAL** being the most common default value.

This parameter can include any combination of the available file attributes (**FILE_ATTRIBUTE_***). All other file attributes override **FILE_ATTRIBUTE_NORMAL**.

This parameter can also contain combinations of flags (**FILE_FLAG_**) for control of buffering behavior, access modes, and other special-purpose flags. These combine with any **FILE_ATTRIBUTE_** values.

This parameter can also contain Security Quality of Service (SQOS) information by specifying the **SECURITY_SQOS_PRESENT** flag. Additional SQOS-related flags information is presented in the table following the attributes and flags tables.

Note

When **CreateFileTransacted** opens an existing file, it generally combines the file flags with the file attributes of the existing file, and ignores any file attributes supplied as part of *dwFlagsAndAttributes*. Special cases are detailed in **Creating and Opening Files**.

The following file attributes and flags are used only for file objects, not other types of objects that **CreateFileTransacted** opens (additional information can be found in the

Remarks section of this topic). For more advanced access to file attributes, see [SetFileAttributes](#). For a complete list of all file attributes with their values and descriptions, see [File Attribute Constants](#).

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE 32 (0x20)	The file should be archived. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_ENCRYPTED 16384 (0x4000)	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories. For more information, see File Encryption .
	This flag has no effect if FILE_ATTRIBUTE_SYSTEM is also specified.
FILE_ATTRIBUTE_HIDDEN 2 (0x2)	The file is hidden. Do not include it in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL 128 (0x80)	The file does not have other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE 4096 (0x1000)	The data of a file is not immediately available. This attribute indicates that file data is physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY 1 (0x1)	The file is read only. Applications can read the file, but cannot write to or delete it.
FILE_ATTRIBUTE_SYSTEM 4 (0x4)	The file is part of or used exclusively by an operating system.
FILE_ATTRIBUTE_TEMPORARY 256 (0x100)	The file is being used for temporary storage. File systems avoid writing data back to mass storage if sufficient cache memory is available, because an application deletes a temporary file after a handle is closed. In that case, the system can entirely avoid writing the data. Otherwise, the data is written after the handle is closed.

Flag	Meaning
FILE_FLAG_BACKUP_SEMANTICS 0x02000000	The file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks when the process has SE_BACKUP_NAME and SE_RESTORE_NAME

privileges. For more information, see [Changing Privileges in a Token](#).

You must set this flag to obtain a handle to a directory. A directory handle can be passed to some functions instead of a file handle. For more information, see [Directory Handles](#).

FILE_FLAG_DELETE_ON_CLOSE 0x04000000	<p>The file is to be deleted immediately after the last transacted writer handle to the file is closed, provided that the transaction is still active. If a file has been marked for deletion and a transacted writer handle is still open after the transaction completes, the file will not be deleted.</p> <p>If there are existing open handles to a file, the call fails unless they were all opened with the FILE_SHARE_DELETE share mode.</p> <p>Subsequent open requests for the file fail, unless the FILE_SHARE_DELETE share mode is specified.</p>
FILE_FLAG_NO_BUFFERING 0x20000000	<p>The file is being opened with no system caching. This flag does not affect hard disk caching or memory mapped files. When combined with FILE_FLAG_OVERLAPPED, the flag gives maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations take more time, because data is not being held in the cache. Also, the file metadata may still be cached. To flush the metadata to disk, use the FlushFileBuffers function.</p> <p>An application must meet certain requirements when working with files that are opened with FILE_FLAG_NO_BUFFERING:</p> <ul style="list-style-type: none">• File access must begin at byte offsets within a file that are integer multiples of the volume sector size.• File access must be for numbers of bytes that are integer multiples of the volume sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, 1536, or 2048 bytes, but not of 335, 981, or 7171 bytes.• Buffer addresses for read and write operations should be sector aligned, which means aligned on addresses in memory that are integer multiples of the volume sector size. Depending on the disk, this requirement may not be enforced.

One way to align buffers on integer multiples of the volume sector size is to use [VirtualAlloc](#) to allocate the

buffers. It allocates memory that is aligned on addresses that are integer multiples of the operating system's memory page size. Because both memory page and volume sector sizes are powers of 2, this memory is also aligned on addresses that are integer multiples of a volume sector size. Memory pages are 4 or 8 KB in size; sectors are 512 bytes (hard disks), 2048 bytes (CD), or 4096 bytes (hard disks), and therefore, volume sectors can never be larger than memory pages.

An application can determine a volume sector size by calling the [GetDiskFreeSpace](#) function.

FILE_FLAG_OPEN_NO_RECALL 0x00100000	The file data is requested, but it should continue to be located in remote storage. It should not be transported back to local storage. This flag is for use by remote storage systems.
FILE_FLAG_OPEN_REPARSE_POINT 0x00200000	Normal reparse point processing will not occur; CreateFileTransacted will attempt to open the reparse point. When a file is opened, a file handle is returned, whether or not the filter that controls the reparse point is operational. This flag cannot be used with the CREATE_ALWAYS flag. If the file is not a reparse point, then this flag is ignored.
FILE_FLAG_OVERLAPPED 0x40000000	<p>The file is being opened or created for asynchronous I/O. When the operation is complete, the event specified in the OVERLAPPED structure is set to the signaled state. Operations that take a significant amount of time to process return ERROR_IO_PENDING.</p> <p>If this flag is specified, the file can be used for simultaneous read and write operations. The system does not maintain the file pointer, therefore you must pass the file position to the read and write functions in the OVERLAPPED structure or update the file pointer.</p> <p>If this flag is not specified, then I/O operations are serialized, even if the calls to the read and write functions specify an OVERLAPPED structure.</p>
FILE_FLAG_POSIX_SEMANTICS 0x01000000	The file is to be accessed according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support that naming. Use care when using this option, because files created with this flag may not be accessible by applications that are written for MS-DOS or 16-bit Windows.
FILE_FLAG_RANDOM_ACCESS 0x10000000	The file is to be accessed randomly. The system can use this as a hint to optimize file caching.

FILE_FLAG_SESSION_AWARE 0x00800000	The file or device is being opened with session awareness. If this flag is not specified, then per-session devices (such as a device using RemoteFX USB Redirection) cannot be opened by processes running in session 0. This flag has no effect for callers not in session 0. This flag is supported only on server editions of Windows. Windows Server 2008 R2 and Windows Server 2008: This flag is not supported before Windows Server 2012.
FILE_FLAG_SEQUENTIAL_SCAN 0x08000000	The file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur. However, correct operation is still guaranteed. Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes. This flag has no effect if the file system does not support cached I/O and FILE_FLAG_NO_BUFFERING .
FILE_FLAG_WRITE_THROUGH 0x80000000	Write operations will not go through any intermediate cache, they will go directly to disk. If FILE_FLAG_NO_BUFFERING is not also specified, so that system caching is in effect, then the data is written to the system cache, but is flushed to disk without delay. If FILE_FLAG_NO_BUFFERING is also specified, so that system caching is not in effect, then the data is immediately flushed to disk without going through the system cache. The operating system also requests a write-through the hard disk cache to persistent media. However, not all hardware supports this write-through capability.

The *dwFlagsAndAttributes* parameter can also specify Security Quality of Service information. For more information, see [Impersonation Levels](#). When the calling application specifies the **SECURITY_SQOS_PRESENT** flag as part of *dwFlagsAndAttributes*, it can also contain one or more of the following values.

Security flag	Meaning
SECURITY_ANONYMOUS	Impersonates a client at the Anonymous impersonation level.

SECURITY_CONTEXT_TRACKING	The security tracking mode is dynamic. If this flag is not specified, the security tracking mode is static.
SECURITY_DELEGATION	Impersonates a client at the Delegation impersonation level.
SECURITY_EFFECTIVE_ONLY	Only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available. This allows the client to limit the groups and privileges that a server can use while impersonating the client.
SECURITY_IDENTIFICATION	Impersonates a client at the Identification impersonation level.
SECURITY_IMPERSONATION	Impersonate a client at the impersonation level. This is the default behavior if no other flags are specified along with the SECURITY_SQOS_PRESENT flag.

[in, optional] `hTemplateFile`

A valid handle to a template file with the **GENERIC_READ** access right. The template file supplies file attributes and extended attributes for the file that is being created. This parameter can be **NULL**.

When opening an existing file, [CreateFileTransacted](#) ignores the template file.

When opening a new EFS-encrypted file, the file inherits the DACL from its parent directory.

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

[in, optional] `pusMiniVersion`

The miniversion to be opened. If the transaction specified in *hTransaction* is not the transaction that is modifying the file, this parameter should be **NULL**. Otherwise, this parameter can be a miniversion identifier returned by the [FSCTL_TXFS_CREATE_MINIVERSION](#) control code, or one of the following values.

Value	Meaning
TXFS_MINIVERSION_COMMITTED_VIEW 0x0000	The view of the file as of its last commit.
TXFS_MINIVERSION_DIRTY_VIEW 0xFFFF	The view of the file as it is being modified by the transaction.

TXFS_MINIVERSION_DEFAULT_VIEW 0xFFFFE	Either the committed or dirty view of the file, depending on the context. A transaction that is modifying the file gets the dirty view, while a transaction that is not modifying the file gets the committed view.
---	---

`lpExtendedParameter`

This parameter is reserved and must be **NULL**.

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

When using the handle returned by [CreateFileTransacted](#), use the transacted version of file I/O functions instead of the standard file I/O functions where appropriate. For more information, see [Programming Considerations for Transactional NTFS](#).

When opening a transacted handle to a directory, that handle must have **FILE_WRITE_DATA** (**FILE_ADD_FILE**) and **FILE_APPEND_DATA** (**FILE_ADD_SUBDIRECTORY**) permissions. These are included in **FILE_GENERIC_WRITE** permissions. You should open directories with fewer permissions if you are just using the handle to create files or subdirectories; otherwise, sharing violations can occur.

You cannot open a file with **FILE_EXECUTE** access level when that file is a part of another transaction (that is, another application opened it by calling [CreateFileTransacted](#)). This means that [CreateFileTransacted](#) fails if the access level **FILE_EXECUTE** or **FILE_ALL_ACCESS** is specified.

When a non-transacted application calls [CreateFileTransacted](#) with **MAXIMUM_ALLOWED** specified for *lpSecurityAttributes*, a handle is opened with the same access level every time. When a transacted application calls [CreateFileTransacted](#) with **MAXIMUM_ALLOWED** specified for *lpSecurityAttributes*, a handle is opened with a differing amount of access based on whether the file is locked by a transaction. For example, if the calling application has **FILE_EXECUTE** access level for a file, the application only obtains this access if the file that is being opened is either not locked

by a transaction, or is locked by a transaction and the application is already a transacted reader for that file.

See [Transactional NTFS](#) for a complete description of transacted operations.

Use the [CloseHandle](#) function to close an object handle returned by [CreateFileTransacted](#) when the handle is no longer needed, and prior to committing or rolling back the transaction.

Some file systems, such as the NTFS file system, support compression or encryption for individual files and directories. On volumes that are formatted for that kind of file system, a new file inherits the compression and encryption attributes of its directory.

You cannot use [CreateFileTransacted](#) to control compression on a file or directory. For more information, see [File Compression and Decompression](#), and [File Encryption](#).

Symbolic link behavior—if the call to this function creates a new file, there is no change in behavior.

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.

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). A single-sector write is atomic. Multi-sector writes that are cached may not always be written to the disk; therefore, specify **FILE_FLAG_WRITE_THROUGH** to ensure that an entire multi-sector write is written to the disk without caching.

As stated previously, if the *lpSecurityAttributes* parameter is **NULL**, the handle returned by [CreateFileTransacted](#) cannot be inherited by any child processes your application may create. The following information regarding this parameter also applies:

- If **bInheritHandle** 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 **IpSecurityDescriptor** 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	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support TxF.

Files

If you try to create a file on a floppy drive that does not have a floppy disk or a CD-ROM drive that does not have a CD, the system displays a message for the user to insert a disk or a CD. To prevent the system from displaying this message, call the [SetErrorMode](#) function with **SEM_FAILCRITICALERRORS**.

For more information, see [Creating and Opening 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 [CreateFileTransacted](#) 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** 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**.

File Streams

On NTFS file systems, you can use [CreateFileTransacted](#) to create separate streams within a file.

For more information, see [File Streams](#).

Directories

An application cannot create a directory by using [CreateFileTransacted](#), therefore only the **OPEN_EXISTING** value is valid for *dwCreationDisposition* for this use case. To create a directory, the application must call [.CreateDirectoryTransacted](#), [.CreateDirectory](#) or [.CreateDirectoryEx](#).

To open a directory using [CreateFileTransacted](#), specify the **FILE_FLAG_BACKUP_SEMANTICS** flag as part of *dwFlagsAndAttributes*. Appropriate security checks still apply when this flag is used without **SE_BACKUP_NAME** and **SE_RESTORE_NAME** privileges.

When using [CreateFileTransacted](#) 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](#).

Note

The winbase.h header defines [CreateFileTransacted](#) as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CopyFileTransacted](#)

[CreateDirectoryTransacted](#)

[DeleteFileTransacted](#)

[File Compression and Decompression](#)

[File Encryption](#)

[File Management Functions](#)

[File Security and Access Rights](#)

[File Streams](#)

[FindFirstFileTransacted](#)

Functions

[GetFileAttributesTransacted](#)

[MoveFileTransacted](#)

Overview Topics

[Programming Considerations for Transactional NTFS](#)

[ReadFile](#)

[Transactional NTFS \(TxF\)](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateHardLinkA function (winbase.h)

Article06/01/2023

Establishes a hard link between an existing file and a new file. This function is only supported on the NTFS file system, and only for files, not directories.

To perform this operation as a transacted operation, use the [CreateHardLinkTransacted](#) function.

Syntax

C++

```
BOOL CreateHardLinkA(
    [in] LPCSTR             lpFileName,
    [in] LPCSTR             lpExistingFileName,
    LPSECURITY_ATTRIBUTES  lpSecurityAttributes
);
```

Parameters

[in] lpFileName

The name of the new file.

This parameter may include the path but cannot specify the name of a directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpExistingFileName

The name of the existing file.

This parameter may include the path cannot specify the name of a directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

`lpSecurityAttributes`

Reserved; must be NULL.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

The maximum number of hard links that can be created with this function is 1023 per file. If more than 1023 links are created for a file, an error results.

If you pass a name longer than MAX_PATH characters to the *lpFileName* or *lpExistingFileName* parameter of the ANSI version of this function or to the Unicode version of this function without prepending "\\?\" to the path, the function returns ERROR_PATH_NOT_FOUND.

Remarks

Any directory entry for a file that is created with [CreateFile](#) or [CreateHardLink](#) is a hard link to an associated file. An additional hard link that is created with the [CreateHardLink](#) function allows you to have multiple directory entries for a file, that is, multiple hard links to the same file, which can be different names in the same directory, or the same or different names in different directories. However, all hard links to a file must be on the same volume.

Because hard links are only directory entries for a file, many changes to that file are instantly visible to applications that access it through the hard links that reference it.

However, the directory entry size and attribute information is updated only for the link through which the change was made.

The security descriptor belongs to the file to which a hard link points. The link itself is only a directory entry, and does not have a security descriptor. Therefore, when you change the security descriptor of a hard link, you change the security descriptor of the underlying file, and all hard links that point to the file allow the newly specified access. You cannot give a file different security descriptors on a per-hard-link basis.

This function does not modify the security descriptor of the file to be linked to, even if security descriptor information is passed in the *lpSecurityAttributes* parameter.

Use [DeleteFile](#) to delete hard links. You can delete them in any order regardless of the order in which they are created.

Flags, attributes, access, and sharing that are specified in [CreateFile](#) operate on a per-file basis. That is, if you open a file that does not allow sharing, another application cannot share the file by creating a new hard link to the file.

When you create a hard link on the NTFS file system, the file attribute information in the directory entry is refreshed only when the file is opened, or when [GetFileInformationByHandle](#) is called with the handle of a specific file.

Symbolic link behavior—if the path points to a symbolic link, the function creates a hard link to the symbolic link.

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)	No

Note that SMB 3.0 does not support creation of hard links on shares with continuous availability capability.

Examples

The following code snippet shows you how to call **CreateHardLink** so that it does not modify the security descriptor of a file. The *pszExistingFileName* parameter can be the original file name, or any existing link to a file. After this code is executed, *pszNewLinkName* refers to the file.

C++

```
BOOL fCreatedLink = CreateHardLink( pszNewLinkName,
                                    pszExistingFileName,
                                    NULL ); // reserved, must be NULL

if ( fCreatedLink == FALSE )
{
    // handle error condition
}
```

ⓘ Note

The winbase.h header defines **CreateHardLink** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[CreateHardLinkTransacted](#)

[DeleteFile](#)

[File Management Functions](#)

[Hard Links and Junctions](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateHardLinkTransactedA function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Establishes a hard link between an existing file and a new file as a transacted operation. This function is only supported on the NTFS file system, and only for files, not directories.

Syntax

C++

```
BOOL CreateHardLinkTransactedA(
    [in] LPCSTR             lpFileName,
    [in] LPCSTR             lpExistingFileName,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in] HANDLE              hTransaction
);
```

Parameters

[in] lpFileName

The name of the new file.

This parameter cannot specify the name of a directory.

[in] lpExistingFileName

The name of the existing file.

This parameter cannot specify the name of a directory.

lpSecurityAttributes

Reserved; must be **NULL**.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

The maximum number of hard links that can be created with this function is 1023 per file. If more than 1023 links are created for a file, an error results.

The files must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

Remarks

Any directory entry for a file that is created with [CreateFileTransacted](#) or [CreateHardLinkTransacted](#) is a hard link to an associated file. An additional hard link that is created with the [CreateHardLinkTransacted](#) function allows you to have multiple directory entries for a file, that is, multiple hard links to the same file, which can be different names in the same directory, or the same or different names in different directories. However, all hard links to a file must be on the same volume.

Because hard links are only directory entries for a file, when an application modifies a file through any hard link, all applications that use any other hard link to the file see the changes. Also, all of the directory entries are updated if the file changes. For example, if a file size changes, all of the hard links to the file show the new file size.

The security descriptor belongs to the file to which a hard link points. The link itself is only a directory entry, and does not have a security descriptor. Therefore, when you change the security descriptor of a hard link, you change the security descriptor of the underlying file, and all hard links that point to the file allow the newly specified access. You cannot give a file different security descriptors on a per-hard-link basis.

This function does not modify the security descriptor of the file to be linked to, even if security descriptor information is passed in the *lpSecurityAttributes* parameter.

Use [DeleteFileTransacted](#) to delete hard links. You can delete them in any order regardless of the order in which they are created.

Flags, attributes, access, and sharing that are specified in [CreateFileTransacted](#) operate on a per-file basis. That is, if you open a file that does not allow sharing, another application cannot share the file by creating a new hard link to the file.

When you create a hard link on the NTFS file system, the file attribute information in the directory entry is refreshed only when the file is opened, or when [GetFileInformationByHandle](#) is called with the handle of a specific file.

Symbolic links: If the path points to a symbolic link, the function creates a hard link to the target.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support TxF.

 **Note**

The winbase.h header defines `CreateHardLinkTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
--------------------------	-----------------------------------

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFileTransacted](#)

[DeleteFileTransacted](#)

[File Management Functions](#)

[Hard Links and Junctions](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateHardLinkTransactedW function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Establishes a hard link between an existing file and a new file as a transacted operation. This function is only supported on the NTFS file system, and only for files, not directories.

Syntax

C++

```
BOOL CreateHardLinkTransactedW(
    [in] LPCWSTR             lpFileName,
    [in] LPCWSTR             lpExistingFileName,
    LPSECURITY_ATTRIBUTES   lpSecurityAttributes,
    [in] HANDLE              hTransaction
);
```

Parameters

[in] lpFileName

The name of the new file.

This parameter cannot specify the name of a directory.

[in] lpExistingFileName

The name of the existing file.

This parameter cannot specify the name of a directory.

lpSecurityAttributes

Reserved; must be **NULL**.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

The maximum number of hard links that can be created with this function is 1023 per file. If more than 1023 links are created for a file, an error results.

The files must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

Remarks

Any directory entry for a file that is created with [CreateFileTransacted](#) or [CreateHardLinkTransacted](#) is a hard link to an associated file. An additional hard link that is created with the [CreateHardLinkTransacted](#) function allows you to have multiple directory entries for a file, that is, multiple hard links to the same file, which can be different names in the same directory, or the same or different names in different directories. However, all hard links to a file must be on the same volume.

Because hard links are only directory entries for a file, when an application modifies a file through any hard link, all applications that use any other hard link to the file see the changes. Also, all of the directory entries are updated if the file changes. For example, if a file size changes, all of the hard links to the file show the new file size.

The security descriptor belongs to the file to which a hard link points. The link itself is only a directory entry, and does not have a security descriptor. Therefore, when you change the security descriptor of a hard link, you change the security descriptor of the underlying file, and all hard links that point to the file allow the newly specified access. You cannot give a file different security descriptors on a per-hard-link basis.

This function does not modify the security descriptor of the file to be linked to, even if security descriptor information is passed in the *lpSecurityAttributes* parameter.

Use [DeleteFileTransacted](#) to delete hard links. You can delete them in any order regardless of the order in which they are created.

Flags, attributes, access, and sharing that are specified in [CreateFileTransacted](#) operate on a per-file basis. That is, if you open a file that does not allow sharing, another application cannot share the file by creating a new hard link to the file.

When you create a hard link on the NTFS file system, the file attribute information in the directory entry is refreshed only when the file is opened, or when [GetFileInformationByHandle](#) is called with the handle of a specific file.

Symbolic links: If the path points to a symbolic link, the function creates a hard link to the target.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

Note that SMB 3.0 does not support TxF.

 **Note**

The winbase.h header defines `CreateHardLinkTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
--------------------------	-----------------------------------

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFileTransacted](#)

[DeleteFileTransacted](#)

[File Management Functions](#)

[Hard Links and Junctions](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateHardLinkW function (winbase.h)

Article06/01/2023

Establishes a hard link between an existing file and a new file. This function is only supported on the NTFS file system, and only for files, not directories.

To perform this operation as a transacted operation, use the [CreateHardLinkTransacted](#) function.

Syntax

C++

```
BOOL CreateHardLinkW(
    [in] LPCWSTR             lpFileName,
    [in] LPCWSTR             lpExistingFileName,
    LPSECURITY_ATTRIBUTES   lpSecurityAttributes
);
```

Parameters

[in] lpFileName

The name of the new file.

This parameter may include the path but cannot specify the name of a directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpExistingFileName

The name of the existing file.

This parameter may include the path cannot specify the name of a directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

`lpSecurityAttributes`

Reserved; must be NULL.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

The maximum number of hard links that can be created with this function is 1023 per file. If more than 1023 links are created for a file, an error results.

If you pass a name longer than MAX_PATH characters to the *lpFileName* or *lpExistingFileName* parameter of the ANSI version of this function or to the Unicode version of this function without prepending "\\?\" to the path, the function returns ERROR_PATH_NOT_FOUND.

Remarks

Any directory entry for a file that is created with [CreateFile](#) or [CreateHardLink](#) is a hard link to an associated file. An additional hard link that is created with the [CreateHardLink](#) function allows you to have multiple directory entries for a file, that is, multiple hard links to the same file, which can be different names in the same directory, or the same or different names in different directories. However, all hard links to a file must be on the same volume.

Because hard links are only directory entries for a file, many changes to that file are instantly visible to applications that access it through the hard links that reference it.

However, the directory entry size and attribute information is updated only for the link through which the change was made.

The security descriptor belongs to the file to which a hard link points. The link itself is only a directory entry, and does not have a security descriptor. Therefore, when you change the security descriptor of a hard link, you change the security descriptor of the underlying file, and all hard links that point to the file allow the newly specified access. You cannot give a file different security descriptors on a per-hard-link basis.

This function does not modify the security descriptor of the file to be linked to, even if security descriptor information is passed in the *lpSecurityAttributes* parameter.

Use [DeleteFile](#) to delete hard links. You can delete them in any order regardless of the order in which they are created.

Flags, attributes, access, and sharing that are specified in [CreateFile](#) operate on a per-file basis. That is, if you open a file that does not allow sharing, another application cannot share the file by creating a new hard link to the file.

When you create a hard link on the NTFS file system, the file attribute information in the directory entry is refreshed only when the file is opened, or when [GetFileInformationByHandle](#) is called with the handle of a specific file.

Symbolic link behavior—if the path points to a symbolic link, the function creates a hard link to the symbolic link.

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)	No

Note that SMB 3.0 does not support creation of hard links on shares with continuous availability capability.

Examples

The following code snippet shows you how to call **CreateHardLink** so that it does not modify the security descriptor of a file. The *pszExistingFileName* parameter can be the original file name, or any existing link to a file. After this code is executed, *pszNewLinkName* refers to the file.

C++

```
BOOL fCreatedLink = CreateHardLink( pszNewLinkName,
                                    pszExistingFileName,
                                    NULL ); // reserved, must be NULL

if ( fCreatedLink == FALSE )
{
    // handle error condition
}
```

ⓘ Note

The winbase.h header defines **CreateHardLink** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[CreateHardLinkTransacted](#)

[DeleteFile](#)

[File Management Functions](#)

[Hard Links and Junctions](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateJobObjectA function (winbase.h)

Article07/27/2022

Creates or opens a job object.

Syntax

C++

```
HANDLE CreateJobObjectA(
    [in, optional] LPSECURITY_ATTRIBUTES lpJobAttributes,
    [in, optional] LPCSTR                 lpName
);
```

Parameters

[in, optional] lpJobAttributes

A pointer to a [SECURITY_ATTRIBUTES](#) structure that specifies the security descriptor for the job object and determines whether child processes can inherit the returned handle. If *lpJobAttributes* is **NULL**, the job object gets a default security descriptor and the handle cannot be inherited. The ACLs in the default security descriptor for a job object come from the primary or impersonation token of the creator.

[in, optional] lpName

The name of the job. The name is limited to **MAX_PATH** characters. Name comparison is case-sensitive.

If *lpName* is **NULL**, the job is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or file-mapping object, the function fails and the [GetLastError](#) function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The object can be created in a private namespace. For more information, see [Object Namespaces](#).

Terminal Services: 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 (\). For more information, see [Kernel Object Namespaces](#).

Return value

If the function succeeds, the return value is a handle to the job object. The handle has the **JOB_OBJECT_ALL_ACCESS** access right. If the object existed before the function call, the function returns a handle to the existing job object and [GetLastError](#) returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks

When a job is created, its accounting information is initialized to zero, all limits are inactive, and there are no associated processes. To assign a process to a job object, use the [AssignProcessToJobObject](#) function. To set limits for a job, use the [SetInformationJobObject](#) function. To query accounting information, use the [QueryInformationJobObject](#) function.

All processes associated with a job must run in the same session. A job is associated with the session of the first process to be assigned to the job.

Windows Server 2003 and Windows XP: A job is associated with the session of the process that created it.

To close a job object handle, use the [CloseHandle](#) function. The job is destroyed when its last handle has been closed and all associated processes have exited. However, if the job has the **JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE** flag specified, closing the last job object handle terminates all associated processes and then destroys the job object itself.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see [Using the Windows Headers](#).

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, Jobapi2.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AssignProcessToJobObject](#)

[CloseHandle](#)

[Job Objects](#)

[Process and Thread Functions](#)

[QueryInformationJobObject](#)

[SECURITY_ATTRIBUTES](#)

[SetInformationJobObject](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateMailslotA function (winbase.h)

Article02/09/2023

Creates a mailslot with the specified name and returns a handle that a mailslot server can use to perform operations on the mailslot. The mailslot is local to the computer that creates it. An error occurs if a mailslot with the specified name already exists.

Syntax

C++

```
HANDLE CreateMailslotA(
    [in]          LPCSTR      lpName,
    [in]          DWORD       nMaxMessageSize,
    [in]          DWORD       lReadTimeout,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Parameters

[in] lpName

The name of the mailslot. This name must have the following form:

\.\mailslot\[path]name

The name field must be unique. The name may include multiple levels of pseudo directories separated by backslashes. For example, both \\.\mailslot\example_mailslot_name and \\.\mailslot\abc\def\ghi are valid names.

[in] nMaxMessageSize

The maximum size of a single message that can be written to the mailslot, in bytes. To specify that the message can be of any size, set this value to zero.

[in] lReadTimeout

The time a read operation can wait for a message to be written to the mailslot before a time-out occurs, in milliseconds. The following values have special meanings.

Value	Meaning
0	Returns immediately if no message is present. (The

system does not treat an immediate return as an error.)

MAILSLOT_WAIT_FOREVER ((DWORD)-1)	Waits forever for a message.
---	------------------------------

This time-out value applies to all subsequent read operations and all inherited mailslot handles.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The **bInheritHandle** member of the structure determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is **NULL**, the handle cannot be inherited.

Return value

If the function succeeds, the return value is a handle to the mailslot, for use in server mailslot operations. The handle returned by this function is asynchronous, or overlapped.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call [GetLastError](#).

Remarks

The mailslot exists until one of the following conditions is true:

- The last (possibly inherited or duplicated) handle to it is closed using the [CloseHandle](#) function.
- The process owning the last (possibly inherited or duplicated) handle exits.

The system uses the second method to destroy mailslots.

To write a message to a mailslot, a process uses the [CreateFile](#) function, specifying the mailslot name by using one of the following formats.

Format	Usage
<code>\.\mailslot\name</code>	Retrieves a client handle to a local mailslot.
<code>\computername\mailslot\name</code>	Retrieves a client handle to a remote mailslot.
<code>\domainname\mailslot\name</code>	Retrieves a client handle to all mailslots with the specified name in the specified domain.

<code>*\mailslot\name</code>	Retrieves a client handle to all mailslots with the specified name in the system's primary domain.
-------------------------------	--

If [CreateFile](#) specifies a domain or uses the asterisk format to specify the system's primary domain, the application cannot write more than 424 bytes at a time to the mailslot. If the application attempts to do so, the [WriteFile](#) function fails and [GetLastError](#) returns [ERROR_BAD_NETPATH](#).

An application must specify the [FILE_SHARE_READ](#) flag when using [CreateFile](#) to retrieve a client handle to a mailslot.

If [CreateFile](#) is called to access a non-existent mailslot, the [ERROR_FILE_NOT_FOUND](#) error code will be set.

Examples

For an example, see [Creating a Mailslot](#).

 Note

The winbase.h header defines CreateMailslot as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFile](#)

[GetMailslotInfo](#)

[Mailslot Functions](#)

[Mailslots Overview](#)

[SECURITY_ATTRIBUTES](#)

[SetMailslotInfo](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateMailslotW function (winbase.h)

Article 02/09/2023

Creates a mailslot with the specified name and returns a handle that a mailslot server can use to perform operations on the mailslot. The mailslot is local to the computer that creates it. An error occurs if a mailslot with the specified name already exists.

Syntax

C++

```
HANDLE CreateMailslotW(
    [in]          LPCWSTR      lpName,
    [in]          DWORD        nMaxMessageSize,
    [in]          DWORD        lReadTimeout,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Parameters

[in] lpName

The name of the mailslot. This name must have the following form:

\.\mailslot\[path]name

The name field must be unique. The name may include multiple levels of pseudo directories separated by backslashes. For example, both \\.\mailslot\example_mailslot_name and \\.\mailslot\abc\def\ghi are valid names.

[in] nMaxMessageSize

The maximum size of a single message that can be written to the mailslot, in bytes. To specify that the message can be of any size, set this value to zero.

[in] lReadTimeout

The time a read operation can wait for a message to be written to the mailslot before a time-out occurs, in milliseconds. The following values have special meanings.

Value	Meaning
0	Returns immediately if no message is present. (The

system does not treat an immediate return as an error.)

MAILSLOT_WAIT_FOREVER ((DWORD)-1)	Waits forever for a message.
---	------------------------------

This time-out value applies to all subsequent read operations and all inherited mailslot handles.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure. The **bInheritHandle** member of the structure determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is **NULL**, the handle cannot be inherited.

Return value

If the function succeeds, the return value is a handle to the mailslot, for use in server mailslot operations. The handle returned by this function is asynchronous, or overlapped.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call [GetLastError](#).

Remarks

The mailslot exists until one of the following conditions is true:

- The last (possibly inherited or duplicated) handle to it is closed using the [CloseHandle](#) function.
- The process owning the last (possibly inherited or duplicated) handle exits.

The system uses the second method to destroy mailslots.

To write a message to a mailslot, a process uses the [CreateFile](#) function, specifying the mailslot name by using one of the following formats.

Format	Usage
<code>\.\mailslot\name</code>	Retrieves a client handle to a local mailslot.
<code>\computername\mailslot\name</code>	Retrieves a client handle to a remote mailslot.
<code>\domainname\mailslot\name</code>	Retrieves a client handle to all mailslots with the specified name in the specified domain.

<code>*\mailslot\name</code>	Retrieves a client handle to all mailslots with the specified name in the system's primary domain.
-------------------------------	--

If [CreateFile](#) specifies a domain or uses the asterisk format to specify the system's primary domain, the application cannot write more than 424 bytes at a time to the mailslot. If the application attempts to do so, the [WriteFile](#) function fails and [GetLastError](#) returns **ERROR_BAD_NETPATH**.

An application must specify the **FILE_SHARE_READ** flag when using [CreateFile](#) to retrieve a client handle to a mailslot.

If [CreateFile](#) is called to access a non-existent mailslot, the **ERROR_FILE_NOT_FOUND** error code will be set.

Examples

For an example, see [Creating a Mailslot](#).

ⓘ Note

The winbase.h header defines CreateMailslot as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFile](#)

[GetMailslotInfo](#)

[Mailslot Functions](#)

[Mailslots Overview](#)

[SECURITY_ATTRIBUTES](#)

[SetMailslotInfo](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateNamedPipeA function (winbase.h)

Article 02/02/2023

Creates an instance of a named pipe and returns a handle for subsequent pipe operations. A named pipe server process uses this function either to create the first instance of a specific named pipe and establish its basic attributes or to create a new instance of an existing named pipe.

Syntax

C++

```
HANDLE CreateNamedPipeA(
    [in]          LPCSTR             lpName,
    [in]          DWORD              dwOpenMode,
    [in]          DWORD              dwPipeMode,
    [in]          DWORD              nMaxInstances,
    [in]          DWORD              nOutBufferSize,
    [in]          DWORD              nInBufferSize,
    [in]          DWORD              nDefaultTimeOut,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Parameters

[in] lpName

The unique pipe name. This string must have the following form:

\.\pipe\pipename

The pipename part of the name can include any character other than a backslash, including numbers and special characters. The entire pipe name string can be up to 256 characters long. Pipe names are not case sensitive.

[in] dwOpenMode

The open mode.

The function fails if *dwOpenMode* specifies anything other than 0 or the flags listed in the following tables.

This parameter must specify one of the following pipe access modes. The same mode must be specified for each instance of the pipe.

Mode	Meaning
PIPE_ACCESS_DUPLEX 0x00000003	The pipe is bi-directional; both server and client processes can read from and write to the pipe. This mode gives the server the equivalent of GENERIC_READ and GENERIC_WRITE access to the pipe. The client can specify GENERIC_READ or GENERIC_WRITE , or both, when it connects to the pipe using the CreateFile function.
PIPE_ACCESS_INBOUND 0x00000001	The flow of data in the pipe goes from client to server only. This mode gives the server the equivalent of GENERIC_READ access to the pipe. The client must specify GENERIC_WRITE access when connecting to the pipe. If the client must read pipe settings by calling the GetNamedPipeInfo or GetNamedPipeHandleState functions, the client must specify GENERIC_WRITE and FILE_READ_ATTRIBUTES access when connecting to the pipe.
PIPE_ACCESS_OUTBOUND 0x00000002	The flow of data in the pipe goes from server to client only. This mode gives the server the equivalent of GENERIC_WRITE access to the pipe. The client must specify GENERIC_READ access when connecting to the pipe. If the client must change pipe settings by calling the SetNamedPipeHandleState function, the client must specify GENERIC_READ and FILE_WRITE_ATTRIBUTES access when connecting to the pipe.

This parameter can also include one or more of the following flags, which enable the write-through and overlapped modes. These modes can be different for different instances of the same pipe.

Mode	Meaning
FILE_FLAG_FIRST_PIPE_INSTANCE 0x00080000	If you attempt to create multiple instances of a pipe with this flag, creation of the first instance succeeds, but creation of the next instance fails with ERROR_ACCESS_DENIED .
FILE_FLAG_WRITE_THROUGH 0x80000000	Write-through mode is enabled. This mode affects only write operations on byte-type pipes and, then, only when the client and server processes are on different computers. If this mode is enabled, functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer.

on the remote computer. If this mode is not enabled, the system enhances the efficiency of network operations by buffering data until a minimum number of bytes accumulate or until a maximum time elapses.

FILE_FLAG_OVERLAPPED

0x40000000

Overlapped mode is enabled. If this mode is enabled, functions performing read, write, and connect operations that may take a significant time to be completed can return immediately. This mode enables the thread that started the operation to perform other operations while the time-consuming operation executes in the background. For example, in overlapped mode, a thread can handle simultaneous input and output (I/O) operations on multiple instances of a pipe or perform simultaneous read and write operations on the same pipe handle. If overlapped mode is not enabled, functions performing read, write, and connect operations on the pipe handle do not return until the operation is finished. The [ReadFileEx](#) and [WriteFileEx](#) functions can only be used with a pipe handle in overlapped mode. The [ReadFile](#), [WriteFile](#), [ConnectNamedPipe](#), and [TransactNamedPipe](#) functions can execute either synchronously or as overlapped operations.

This parameter can include any combination of the following security access modes. These modes can be different for different instances of the same pipe.

Mode	Meaning
WRITE_DAC 0x00040000L	The caller will have write access to the named pipe's discretionary access control list (ACL).
WRITE_OWNER 0x00080000L	The caller will have write access to the named pipe's owner.
ACCESS_SYSTEM_SECURITY 0x01000000L	The caller will have write access to the named pipe's SACL. For more information, see Access-Control Lists (ACLs) and SACL Access Right .

[in] dwPipeMode

The pipe mode.

The function fails if *dwPipeMode* specifies anything other than 0 or the flags listed in the following tables.

One of the following type modes can be specified. The same type mode must be specified for each instance of the pipe.

Mode	Meaning
PIPE_TYPE_BYTE 0x00000000	Data is written to the pipe as a stream of bytes. This mode cannot be used with PIPE_READMODE_MESSAGE . The pipe does not distinguish bytes written during different write operations.
PIPE_TYPE_MESSAGE 0x00000004	Data is written to the pipe as a stream of messages. The pipe treats the bytes written during each write operation as a message unit. The GetLastError function returns ERROR_MORE_DATA when a message is not read completely. This mode can be used with either PIPE_READMODE_MESSAGE or PIPE_READMODE_BYTE .

One of the following read modes can be specified. Different instances of the same pipe can specify different read modes.

Mode	Meaning
PIPE_READMODE_BYTE 0x00000000	Data is read from the pipe as a stream of bytes. This mode can be used with either PIPE_TYPE_MESSAGE or PIPE_TYPE_BYTE .
PIPE_READMODE_MESSAGE 0x00000002	Data is read from the pipe as a stream of messages. This mode can be only used if PIPE_TYPE_MESSAGE is also specified.

One of the following wait modes can be specified. Different instances of the same pipe can specify different wait modes.

Mode	Meaning
PIPE_WAIT 0x00000000	Blocking mode is enabled. When the pipe handle is specified in the ReadFile , WriteFile , or ConnectNamedPipe function, the operations are not completed until there is data to read, all data is written, or a client is connected. Use of this mode can mean waiting indefinitely in some situations for a client process to perform an action.
PIPE_NOWAIT 0x00000001	Nonblocking mode is enabled. In this mode, ReadFile , WriteFile , and ConnectNamedPipe always return immediately.

Note that nonblocking mode is supported for compatibility with Microsoft LAN Manager version 2.0 and should not be used to achieve asynchronous I/O with named pipes. For more information on asynchronous pipe I/O, see [Synchronous and Overlapped Input and Output](#).

One of the following remote-client modes can be specified. Different instances of the same pipe can specify different remote-client modes.

Mode	Meaning
PIPE_ACCEPT_REMOTE_CLIENTS 0x00000000	Connections from remote clients can be accepted and checked against the security descriptor for the pipe.
PIPE_REJECT_REMOTE_CLIENTS 0x00000008	Connections from remote clients are automatically rejected.

[in] nMaxInstances

The maximum number of instances that can be created for this pipe. The first instance of the pipe can specify this value; the same number must be specified for other instances of the pipe. Acceptable values are in the range 1 through **PIPE_UNLIMITED_INSTANCES** (255).

If this parameter is **PIPE_UNLIMITED_INSTANCES**, the number of pipe instances that can be created is limited only by the availability of system resources. If *nMaxInstances* is greater than **PIPE_UNLIMITED_INSTANCES**, the return value is **INVALID_HANDLE_VALUE** and [GetLastError](#) returns **ERROR_INVALID_PARAMETER**.

[in] nOutBufferSize

The number of bytes to reserve for the output buffer. For a discussion on sizing named pipe buffers, see the following Remarks section.

[in] nInBufferSize

The number of bytes to reserve for the input buffer. For a discussion on sizing named pipe buffers, see the following Remarks section.

[in] nDefaultTimeOut

The default time-out value, in milliseconds, if the [WaitNamedPipe](#) function specifies **NMPWAIT_USE_DEFAULT_WAIT**. Each instance of a named pipe must specify the same value.

A value of zero will result in a default time-out of 50 milliseconds.

[in, optional] *lpSecurityAttributes*

A pointer to a [SECURITY_ATTRIBUTES](#) structure that specifies a security descriptor for the new named pipe and determines whether child processes can inherit the returned handle. If *lpSecurityAttributes* is **NULL**, the named pipe gets a default security descriptor and the handle cannot be inherited. The ACLs in the default security descriptor for a named pipe grant full control to the LocalSystem account, administrators, and the creator owner. They also grant read access to members of the Everyone group and the anonymous account.

Return value

If the function succeeds, the return value is a handle to the server end of a named pipe instance.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call [GetLastError](#).

Remarks

To create an instance of a named pipe by using [CreateNamedPipe](#), the user must have **FILE_CREATE_PIPE_INSTANCE** access to the named pipe object. If a new named pipe is being created, the access control list (ACL) from the security attributes parameter defines the discretionary access control for the named pipe.

All instances of a named pipe must specify the same pipe type (byte-type or message-type), pipe access (duplex, inbound, or outbound), instance count, and time-out value. If different values are used, this function fails and [GetLastError](#) returns **ERROR_ACCESS_DENIED**.

A client process connects to a named pipe by using the [CreateFile](#) or [CallNamedPipe](#) function. The client side of a named pipe starts out in byte mode, even if the server side is in message mode. To avoid problems receiving data, set the client side to message mode as well. To change the mode of the pipe, the pipe client must open a read-only pipe with **GENERIC_READ** and **FILE_WRITE_ATTRIBUTES** access.

The pipe server should not perform a blocking read operation until the pipe client has started. Otherwise, a race condition can occur. This typically occurs when initialization code, such as the C run-time, needs to lock and examine inherited handles.

Every time a named pipe is created, the system creates the inbound and/or outbound buffers using nonpaged pool, which is the physical memory used by the kernel. The number of pipe instances (as well as objects such as threads and processes) that you can create is limited by the available nonpaged pool. Each read or write request requires space in the buffer for the read or write data, plus additional space for the internal data structures.

The input and output buffer sizes are advisory. The actual buffer size reserved for each end of the named pipe is either the system default, the system minimum or maximum, or the specified size rounded up to the next allocation boundary. The buffer size specified should be small enough that your process will not run out of nonpaged pool, but large enough to accommodate typical requests.

Whenever a pipe write operation occurs, the system first tries to charge the memory against the pipe write quota. If the remaining pipe write quota is enough to fulfill the request, the write operation completes immediately. If the remaining pipe write quota is too small to fulfill the request, the system will try to expand the buffers to accommodate the data using nonpaged pool reserved for the process. The write operation will block until the data is read from the pipe so that the additional buffer quota can be released. Therefore, if your specified buffer size is too small, the system will grow the buffer as needed, but the downside is that the operation will block. If the operation is overlapped, a system thread is blocked; otherwise, the application thread is blocked.

To free resources used by a named 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 named pipe may have more than one handle associated with it. An instance of a named pipe is always deleted when the last handle to the instance of the named pipe is closed.

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\pipe\LOCAL\` for the pipe name.

Examples

For an example, see [Multithreaded Pipe Server](#).

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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConnectNamedPipe](#)

[CreateFile](#)

[Pipe Functions](#)

[Pipes Overview](#)

[ReadFile](#)

[ReadFileEx](#)

[SECURITY_ATTRIBUTES](#)

[TransactNamedPipe](#)

[WaitNamedPipe](#)

[WriteFile](#)

[WriteFileEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreatePrivateNamespaceA function (winbase.h)

Article 08/03/2022

Creates a private namespace.

Syntax

C++

```
HANDLE CreatePrivateNamespaceA(
    [in, optional] LPSECURITY_ATTRIBUTES lpPrivateNamespaceAttributes,
    [in]           LPVOID             lpBoundaryDescriptor,
    [in]           LPCSTR            lpAliasPrefix
);
```

Parameters

[in, optional] lpPrivateNamespaceAttributes

A pointer to a [SECURITY_ATTRIBUTES](#) structure that specifies the security attributes of the namespace object.

[in] lpBoundaryDescriptor

A descriptor that defines how the namespace is to be isolated. The caller must be within this boundary. The [CreateBoundaryDescriptor](#) function creates a boundary descriptor.

[in] lpAliasPrefix

The prefix for the namespace. To create an object in this namespace, specify the object name as *prefix\objectname*.

The system supports multiple private namespaces with the same name, as long as they define different boundaries.

Return value

If the function succeeds, it returns a handle to the new namespace.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Other applications can access the namespace using the [OpenPrivateNamespace](#) function.

The application that created the namespace can use the [ClosePrivateNamespace](#) function to close the handle to the namespace. The handle is also closed when the creating process terminates. After the namespace handle is closed, subsequent calls to [OpenPrivateNamespace](#) fail, but all operations on objects in the namespace succeed.

To compile an application that uses this function, define `_WIN32_WINNT` as `0x0600` or later.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ClosePrivateNamespace](#)

[Object Namespaces](#)

[OpenPrivateNamespace](#)

Feedback



Was this page helpful? [!\[\]\(f6ab692719ed5eee547c211195503f41_img.jpg\) Yes](#) [!\[\]\(8637541a1f2765892d6e307fe9a9e585_img.jpg\) No](#)

Get help at Microsoft Q&A

CreateProcessWithLogonW function (winbase.h)

Article02/02/2023

Creates a new process and its primary thread. Then the new process runs the specified executable file in the security context of the specified credentials (user, domain, and password). It can optionally load the user profile for a specified user.

This function is similar to the [CreateProcessAsUser](#) and [CreateProcessWithTokenW](#) functions, except that the caller does not need to call the [LogonUser](#) function to authenticate the user and get a token.

Syntax

C++

```
BOOL CreateProcessWithLogonW(
    [in]                 LPCWSTR             lpUsername,
    [in, optional]       LPCWSTR             lpDomain,
    [in]                 LPCWSTR             lpPassword,
    [in]                 DWORD               dwLogonFlags,
    [in, optional]       LPCWSTR             lpApplicationName,
    [in, out, optional] LPVOID              lpCommandLine,
    [in]                 DWORD               dwCreationFlags,
    [in, optional]       LPVOID              lpEnvironment,
    [in, optional]       LPCWSTR             lpCurrentDirectory,
    [in]                 LPSTARTUPINFO     lpStartupInfo,
    [out]                LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

[in] lpUsername

The name of the user. This is the name of the user account to log on to. If you use the UPN format, *user@DNS_domain_name*, the *lpDomain* parameter must be NULL.

The user account must have the Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

[in, optional] lpDomain

The name of the domain or server whose account database contains the *lpUsername* account. If this parameter is NULL, the user name must be specified in UPN format.

[in] lpPassword

The clear-text password for the *lpUsername* account.

[in] dwLogonFlags

The logon option. This parameter can be 0 (zero) or one of the following values.

Value	Meaning
LOGON_WITH_PROFILE 0x00000001	Log on, then load the user profile in the HKEY_USERS registry key. The function returns after the profile is loaded. Loading the profile can be time-consuming, so it is best to use this value only if you must access the information in the HKEY_CURRENT_USER registry key. Windows Server 2003: The profile is unloaded after the new process is terminated, whether or not it has created child processes. Windows XP: The profile is unloaded after the new process and all child processes it has created are terminated.
LOGON_NETCREDENTIALS_ONLY 0x00000002	Log on, but use the specified credentials on the network only. The new process uses the same token as the caller, but the system creates a new logon session within LSA, and the process uses the specified credentials as the default credentials. This value can be used to create a process that uses a different set of credentials locally than it does remotely. This is useful in inter-domain scenarios where there is no trust relationship. The system does not validate the specified credentials. Therefore, the process can start, but it may not have access to network resources.

[in, optional] lpApplicationName

The name of the module to be executed. This module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. If it is a partial name, the function uses the current drive and

current directory to complete the specification. The function does not use the search path. This parameter must include the file name extension; no default extension is assumed.

The *lpApplicationName* parameter can be NULL, and the module name must be the first white space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous.

For example, the following string can be interpreted in different ways:

"c:\program files\sub dir\program name"

The system tries to interpret the possibilities in the following order:

1. c:\program.exe files\sub dir\program name
2. c:\program files\sub.exe dir\program name
3. c:\program files\sub dir\program.exe name
4. c:\program files\sub dir\program name.exe

If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module and its arguments.

[in, out, optional] lpCommandLine

The command line to be executed. The maximum length of this string is 1024 characters. If *lpApplicationName* is **NULL**, the module name portion of *lpCommandLine* is limited to **MAX_PATH** characters.

The function can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a **const** variable or a literal string). If this parameter is a constant string, the function may cause an access violation.

The *lpCommandLine* parameter can be **NULL**, and the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-**NULL**, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use [GetCommandLine](#) to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because *argv[0]* is the module name, C programmers typically repeat the module name as the first token in the command line.

If *lpApplicationName* is **NULL**, the first white space-delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the [GetSystemDirectory](#) function to get the path of this directory.
4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this function does not search the per-application path specified by the **App Paths** registry key. To include this per-application path in the search sequence, use the [ShellExecute](#) function.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

[in] dwCreationFlags

The flags that control how the process is created. The **CREATE_DEFAULT_ERROR_MODE**, **CREATE_NEW_CONSOLE**, and **CREATE_NEW_PROCESS_GROUP** flags are enabled by default. For a list of values, see [Process Creation Flags](#).

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see [GetPriorityClass](#). If none of the priority class flags is specified, the priority class defaults to **NORMAL_PRIORITY_CLASS** unless the priority class of the creating process is **IDLE_PRIORITY_CLASS** or **BELOW_NORMAL_PRIORITY_CLASS**. In this case, the child process receives the default priority class of the calling process.

If the dwCreationFlags parameter has a value of 0:

- The process gets the default error mode, creates a new console and creates a new process group.
- The environment block for the new process is assumed to contain ANSI characters (see *lpEnvironment* parameter for additional information).
- A 16-bit Windows-based application runs in a shared Virtual DOS machine (VDM).

[in, optional] *lpEnvironment*

A pointer to an environment block for the new process. If this parameter is **NULL**, the new process uses an environment created from the profile of the user specified by *lpUsername*.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the following form:

name=*value*

Because the equal sign (=) is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, ensure that *dwCreationFlags* includes **CREATE_UNICODE_ENVIRONMENT**.

An ANSI environment block is terminated by two 0 (zero) bytes: one for the last string and one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string and two more to terminate the block.

To retrieve a copy of the environment block for a specific user, use the [CreateEnvironmentBlock](#) function.

[in, optional] *lpCurrentDirectory*

The full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is **NULL**, the new process has the same current drive and directory as the calling process. This feature is provided primarily for shells that need to start an application, and specify its initial drive and working directory.

[in] *lpStartupInfo*

A pointer to a [STARTUPINFO](#) structure.

The application must add permission for the specified user account to the specified window station and desktop, even for WinSta0\Default.

If the **lpDesktop** member is **NULL** or an empty string, the new process inherits the desktop and window station of its parent process. The application must add permission for the specified user account to the inherited window station and desktop.

Windows XP: **CreateProcessWithLogonW** adds permission for the specified user account to the inherited window station and desktop.

Handles in **STARTUPINFO** must be closed with **CloseHandle** when they are no longer needed.

Important If the **dwFlags** member of the **STARTUPINFO** structure specifies **STARTF_USESTDHANDLES**, the standard handle fields are copied unchanged to the child process without validation. The caller is responsible for ensuring that these fields contain valid handle values. Incorrect values can cause the child process to misbehave or crash. Use the **Application Verifier** runtime verification tool to detect invalid handles.

[out] **lpProcessInformation**

A pointer to a **PROCESS_INFORMATION** structure that receives identification information for the new process, including a handle to the process.

Handles in **PROCESS_INFORMATION** must be closed with the **CloseHandle** function when they are not needed.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call **GetLastError**.

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call **GetExitCodeProcess**.

Remarks

By default, **CreateProcessWithLogonW** does not load the specified user profile into the **HKEY_USERS** registry key. This means that access to information in the

`HKEY_CURRENT_USER` registry key may not produce results that are consistent with a normal interactive logon. It is your responsibility to load the user registry hive into `HKEY_USERS` before calling `CreateProcessWithLogonW`, by using `LOGON_WITH_PROFILE`, or by calling the [LoadUserProfile](#) function.

If the *lpEnvironment* parameter is NULL, the new process uses an environment block created from the profile of the user specified by *lpUserName*. If the HOMEDRIVE and HOMEPATH variables are not set, `CreateProcessWithLogonW` modifies the environment block to use the drive and path of the user's working directory.

When created, the new process and thread handles receive full access rights (`PROCESS_ALL_ACCESS` and `THREAD_ALL_ACCESS`). For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the process or thread.

To retrieve a security token, pass the process handle in the `PROCESS_INFORMATION` structure to the [OpenProcessToken](#) function.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or it can be specified in the [OpenProcess](#) function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the [OpenThread](#) function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in `PROCESS_INFORMATION`.

The calling thread can use the [WaitForInputIdle](#) function to wait until the new process has completed its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because `CreateProcessWithLogonW` returns without waiting for the new process to finish its initialization. For example, the creating process would use [WaitForInputIdle](#) before trying to find a window that is associated with the new process.

The preferred way to shut down a process is by using the [ExitProcess](#) function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls [ExitProcess](#), other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For more information, see [Terminating a Process](#).

CreateProcessWithLogonW accesses the specified directory and executable image in the security context of the target user. If the executable image is on a network and a network drive letter is specified in the path, the network drive letter is not available to the target user, as network drive letters can be assigned for each logon. If a network drive letter is specified, this function fails. If the executable image is on a network, use the UNC path.

There is a limit to the number of child processes that can be created by this function and run simultaneously. For example, on Windows XP, this limit is **MAXIMUM_WAIT_OBJECTS***4. However, you may not be able to create this many processes due to system-wide quota limits.

Windows XP with SP2, Windows Server 2003, or later: You cannot call **CreateProcessWithLogonW** from a process that is running under the "LocalSystem" account, because the function uses the logon SID in the caller token, and the token for the "LocalSystem" account does not contain this SID. As an alternative, use the [CreateProcessAsUser](#) and [LogonUser](#) functions.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see [Using the Windows Headers](#).

Security Remarks

The *lpApplicationName* parameter can be **NULL**, and the executable name must be the first white space-delimited string in *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. Avoid the following example, because the function attempts to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
LPTSTR szCmdline[]=_tcstrup(TEXT("C:\\\\Program Files\\\\MyApp"));
CreateProcessWithLogonW(..., szCmdline, ...)
```

If a malicious user creates an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcessWithLogonW** using the Program Files directory runs the malicious user application instead of the intended application.

To avoid this issue, do not pass **NULL** for *lpApplicationName*. If you pass **NULL** for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the following example:

syntax

```
LPTSTR szCmdline[]=_tcstrup(TEXT("C:\\Program Files\\MyApp\\"));
CreateProcessWithLogonW(..., szCmdline, ...)
```

Examples

The following example demonstrates how to call this function.

C++

```
#include <windows.h>
#include <stdio.h>
#include <userenv.h>

void DisplayError(LPWSTR pszAPI)
{
    LPVOID lpvMessageBuffer;

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM,
                  NULL, GetLastError(),
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPWSTR)&lpvMessageBuffer, 0, NULL);

    //
    //... now display this string
    //
    wprintf(L"ERROR: API      = %s.\n", pszAPI);
    wprintf(L"        error code = %d.\n", GetLastError());
    wprintf(L"        message   = %s.\n", (LPSTR)lpvMessageBuffer);

    //
    // Free the buffer allocated by the system
    //
    LocalFree(lpvMessageBuffer);

    ExitProcess(GetLastError());
}

void wmain(int argc, WCHAR *argv[])
{
    DWORD      dwSize;
    HANDLE     hToken;
    LPVOID     lpvEnv;
    PROCESS_INFORMATION pi = {0};
    STARTUPINFO      si = {0};
    WCHAR          szUserProfile[256] = L"";

    si.cb = sizeof(STARTUPINFO);

    if (argc != 4)
```

```

{
    wprintf(L"Usage: %s [user@domain] [password] [cmd]", argv[0]);
    wprintf(L"\n\n");
    return;
}

// TO DO: change NULL to '.' to use local account database
//
if (!LogonUser(argv[1], NULL, argv[2], LOGON32_LOGON_INTERACTIVE,
    LOGON32_PROVIDER_DEFAULT, &hToken))
    DisplayError(L"LogonUser");

if (!CreateEnvironmentBlock(&lpvEnv, hToken, TRUE))
    DisplayError(L>CreateEnvironmentBlock");

dwSize = sizeof(szUserProfile)/sizeof(WCHAR);

if (!GetUserProfileDirectory(hToken, szUserProfile, &dwSize))
    DisplayError(L"GetUserProfileDirectory");

//
// TO DO: change NULL to '.' to use local account database
//
if (!CreateProcessWithLogonW(argv[1], NULL, argv[2],
    LOGON_WITH_PROFILE, NULL, argv[3],
    CREATE_UNICODE_ENVIRONMENT, lpvEnv, szUserProfile,
    &si, &pi))
    DisplayError(L>CreateProcessWithLogonW");

if (!DestroyEnvironmentBlock(lpvEnv))
    DisplayError(L"DestroyEnvironmentBlock");

CloseHandle(hToken);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}

```

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)

Library	Advapi32.lib
DLL	Advapi32.dll

See also

[CloseHandle](#)

[CreateEnvironmentBlock](#)

[CreateProcessAsUser](#)

[ExitProcess](#)

[GetEnvironmentStrings](#)

[GetExitCodeProcess](#)

[OpenProcess](#)

[PROCESS_INFORMATION](#)

[Process and Thread Functions](#)

[Processes](#)

[STARTUPINFO](#)

[SetErrorMode](#)

[WaitForInputIdle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateProcessWithTokenW function (winbase.h)

Article02/02/2023

Creates a new process and its primary thread. The new process runs in the security context of the specified token. It can optionally load the user profile for the specified user.

The process that calls **CreateProcessWithTokenW** must have the SE_IMPERSONATE_NAME privilege. If this function fails with ERROR_PRIVILEGE_NOT_HELD (1314), use the [CreateProcessAsUser](#) or [CreateProcessWithLogonW](#) function instead. Typically, the process that calls **CreateProcessAsUser** must have the SE_INCREASE_QUOTA_NAME privilege and may require the SE_ASSIGNPRIMARYTOKEN_NAME privilege if the token is not assignable. **CreateProcessWithLogonW** requires no special privileges, but the specified user account must be allowed to log on interactively. Generally, it is best to use **CreateProcessWithLogonW** to create a process with alternate credentials.

Syntax

C++

```
BOOL CreateProcessWithTokenW(
    [in]                 HANDLE          hToken,
    [in]                 DWORD           dwLogonFlags,
    [in, optional]       LPCWSTR        lpApplicationName,
    [in, out, optional] LPWSTR         lpCommandLine,
    [in]                 DWORD           dwCreationFlags,
    [in, optional]       LPVOID          lpEnvironment,
    [in, optional]       LPCWSTR        lpCurrentDirectory,
    [in]                 LPSTARTUPINFO   lpStartupInfo,
    [out]                LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

[in] hToken

A handle to the primary token that represents a user. The handle must have the TOKEN_QUERY, TOKEN_DUPLICATE, and TOKEN_ASSIGN_PRIMARY access rights. For more information, see [Access Rights for Access-Token Objects](#). The user represented by

the token must have read and execute access to the application specified by the *lpApplicationName* or the *lpCommandLine* parameter.

To get a primary token that represents the specified user, call the [LogonUser](#) function. Alternatively, you can call the [DuplicateTokenEx](#) function to convert an impersonation token into a primary token. This allows a server application that is impersonating a client to create a process that has the security context of the client.

Terminal Services: The caller's process always runs in the caller's session, not in the session specified in the token. To run a process in the session specified in the token, use the [CreateProcessAsUser](#) function.

[in] dwLogonFlags

The logon option. This parameter can be zero or one of the following values.

Value	Meaning
LOGON_WITH_PROFILE 0x00000001	Log on, then load the user's profile in the HKEY_USERS registry key. The function returns after the profile has been loaded. Loading the profile can be time-consuming, so it is best to use this value only if you must access the information in the HKEY_CURRENT_USER registry key. Windows Server 2003: The profile is unloaded after the new process has been terminated, regardless of whether it has created child processes.
LOGON_NETCREDENTIALS_ONLY 0x00000002	Log on, but use the specified credentials on the network only. The new process uses the same token as the caller, but the system creates a new logon session within LSA, and the process uses the specified credentials as the default credentials. This value can be used to create a process that uses a different set of credentials locally than it does remotely. This is useful in inter-domain scenarios where there is no trust relationship. The system does not validate the specified credentials. Therefore, the process can start, but it may not have access to network resources.

[in, optional] lpApplicationName

The name of the module to be executed. This module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path. This parameter must include the file name extension; no default extension is assumed.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe c:\program files\sub.exe c:\program files\sub dir\program.exe
c:\program files\sub dir\program name.exe If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments.

[in, out, optional] *lpCommandLine*

The command line to be executed.

The maximum length of this string is 1024 characters. If *lpApplicationName* is NULL, the module name portion of *lpCommandLine* is limited to MAX_PATH characters.

The function can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a **const** variable or a literal string). If this parameter is a constant string, the function may cause an access violation.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use [GetCommandLine](#) to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because *argv*[0] is the module name, C programmers generally repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white space-delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an

extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the [GetSystemDirectory](#) function to get the path of this directory.
4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this function does not search the per-application path specified by the [App Paths](#) registry key. To include this per-application path in the search sequence, use the [ShellExecute](#) function.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

[in] dwCreationFlags

The flags that control how the process is created. The CREATE_DEFAULT_ERROR_MODE, CREATE_NEW_CONSOLE, and CREATE_NEW_PROCESS_GROUP flags are enabled by default. For a list of values, see [Process Creation Flags](#).

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see [GetPriorityClass](#). If none of the priority class flags is specified, the priority class defaults to NORMAL_PRIORITY_CLASS unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process.

If the dwCreationFlags parameter has a value of 0:

- The process gets the default error mode, creates a new console and creates a new process group.
- The environment block for the new process is assumed to contain ANSI characters (see *lpEnvironment* parameter for additional information).
- A 16-bit Windows-based application runs in a shared Virtual DOS machine (VDM).

[in, optional] *lpEnvironment*

A pointer to an environment block for the new process. If this parameter is NULL, the new process uses an environment created from the profile of the user specified by *lpUsername*.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the following form:

name=*value*

Because the equal sign (=) is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, be sure that *dwCreationFlags* includes CREATE_UNICODE_ENVIRONMENT.

An ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string and two more to terminate the block.

To retrieve a copy of the environment block for a specific user, use the [CreateEnvironmentBlock](#) function.

[in, optional] *lpCurrentDirectory*

The full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is NULL, the new process will have the same current drive and directory as the calling process. (This feature is provided primarily for shells that need to start an application and specify its initial drive and working directory.)

[in] *lpStartupInfo*

A pointer to a [STARTUPINFO](#) or [STARTUPINFOEX](#) structure.

If the **lpDesktop** member is NULL or an empty string, the new process inherits the desktop and window station of its parent process. The function adds permission for the specified user account to the inherited window station and desktop. Otherwise, if this member specifies a desktop, it is the responsibility of the application to add permission for the specified user account to the specified window station and desktop, even for WinSta0\Default.

Handles in [STARTUPINFO](#) or [STARTUPINFOEX](#) must be closed with [CloseHandle](#) when they are no longer needed.

Important If the `dwFlags` member of the [STARTUPINFO](#) structure specifies `STARTF_USESTDHANDLES`, the standard handle fields are copied unchanged to the child process without validation. The caller is responsible for ensuring that these fields contain valid handle values. Incorrect values can cause the child process to misbehave or crash. Use the [Application Verifier](#) runtime verification tool to detect invalid handles.

[out] `lpProcessInformation`

A pointer to a [PROCESS_INFORMATION](#) structure that receives identification information for the new process, including a handle to the process.

Handles in [PROCESS_INFORMATION](#) must be closed with the [CloseHandle](#) function when they are no longer needed.

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](#).

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call [GetExitCodeProcess](#).

Remarks

By default, [CreateProcessWithTokenW](#) does not load the specified user's profile into the [HKEY_USERS](#) registry key. This means that access to information in the [HKEY_CURRENT_USER](#) registry key may not produce results consistent with a normal interactive logon. It is your responsibility to load the user's registry hive into [HKEY_USERS](#) by either using `LOGON_WITH_PROFILE`, or by calling the [LoadUserProfile](#) function before calling this function.

If the `lpEnvironment` parameter is `NULL`, the new process uses an environment block created from the profile of the user specified by `lpUserName`. If the `HOMEDRIVE` and

HOMEPATH variables are not set, **CreateProcessWithTokenW** modifies the environment block to use the drive and path of the user's working directory.

When created, the new process and thread handles receive full access rights (PROCESS_ALL_ACCESS and THREAD_ALL_ACCESS). For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the process or thread.

To retrieve a security token, pass the process handle in the **PROCESS_INFORMATION** structure to the [OpenProcessToken](#) function.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the [OpenProcess](#) function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the [OpenThread](#) function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in **PROCESS_INFORMATION**.

The calling thread can use the [WaitForInputIdle](#) function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcessWithTokenW** returns without waiting for the new process to finish its initialization. For example, the creating process would use [WaitForInputIdle](#) before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the [ExitProcess](#) function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls [ExitProcess](#), other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For more information, see [Terminating a Process](#).

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see [Using the Windows Headers](#).

Security Remarks

The *lpApplicationName* parameter can be NULL, in which case the executable name must be the first white space-delimited string in *lpCommandLine*. If the executable or

path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
LPTSTR szCmdline = L"C:\\Program Files\\MyApp";
CreateProcessWithTokenW(/*...*/, szCmdline, /*...*/);
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcessWithTokenW** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, do not pass NULL for *lpApplicationName*. If you do pass NULL for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the example below.

syntax

```
LPTSTR szCmdline = L"C:\\Program Files\\MyApp\";
CreateProcessWithTokenW(/*...*/, szCmdline, /*...*/);
```

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[CloseHandle](#)

[CreateEnvironmentBlock](#)

[ExitProcess](#)

[GetEnvironmentStrings](#)

[GetExitCodeProcess](#)

[OpenProcess](#)

[PROCESS_INFORMATION](#)

[Process and Thread Functions](#)

[Processes](#)

[STARTUPINFO](#)

[SetErrorMode](#)

[WaitForInputIdle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateSemaphoreA function (winbase.h)

Article07/27/2022

Creates or opens a named or unnamed semaphore object.

To specify an access mask for the object, use the [CreateSemaphoreEx](#) function.

Syntax

C++

```
HANDLE CreateSemaphoreA(
    [in, optional] LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    [in]           LONG             lInitialCount,
    [in]           LONG             lMaximumCount,
    [in, optional] LPCSTR          lpName
);
```

Parameters

[in, optional] `lpSemaphoreAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure. If this parameter is **NULL**, the handle cannot be inherited by child processes.

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

[in] `lInitialCount`

The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to `lMaximumCount`. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the [ReleaseSemaphore](#) function.

[in] `lMaximumCount`

The maximum count for the semaphore object. This value must be greater than zero.

[in, optional] *lpName*

The name of the semaphore object. The name is limited to **MAX_PATH** characters. Name comparison is case sensitive.

If *lpName* matches the name of an existing named semaphore object, this function requests the **SEMAPHORE_ALL_ACCESS** access right. In this case, the *lInitialCount* and *lMaximumCount* parameters are ignored because they have already been set by the creating process. If the *lpSemaphoreAttributes* parameter is not **NULL**, it determines whether the handle can be inherited, but its security-descriptor member is ignored.

If *lpName* is **NULL**, the semaphore object is created without a name.

If *lpName* matches the name of an existing event, mutex, waitable timer, job, or file-mapping 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 (\). For more information, see [Kernel Object Namespaces](#). Fast user switching is implemented using Terminal Services sessions. Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users.

The object can be created in a private namespace. For more information, see [Object Namespaces](#).

Return value

If the function succeeds, the return value is a handle to the semaphore object. If the named semaphore object existed before the function call, the function returns a handle to the existing object and [GetLastError](#) returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The handle returned by CreateSemaphore has the **SEMAPHORE_ALL_ACCESS** access right; it can be used in any function that requires a handle to a semaphore object, provided that the caller has been granted access. If a semaphore is created from a

service or a thread that is impersonating a different user, you can either apply a security descriptor to the semaphore when you create it, or change the default security descriptor for the creating process by changing its default DACL. For more information, see [Synchronization Object Security and Access Rights](#).

The state of a semaphore object is signaled when its count is greater than zero, and nonsignaled when its count is equal to zero. The *lInitialCount* parameter specifies the initial count. The count can never be less than zero or greater than the value specified in the *lMaximumCount* parameter.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the [wait functions](#). The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution. Each time a thread completes a wait for a semaphore object, the count of the semaphore object is decremented by one. When the thread has finished, it calls the [ReleaseSemaphore](#) function, which increments the count of the semaphore object.

Multiple processes can have handles of the same semaphore object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

- A child process created by the [CreateProcess](#) function can inherit a handle to a semaphore object if the *lpSemaphoreAttributes* parameter of [CreateSemaphore](#) enabled inheritance.
- A process can specify the semaphore-object handle in a call to the [DuplicateHandle](#) function to create a duplicate handle that can be used by another process.
- A process can specify the name of a semaphore object in a call to the [\[OpenSemaphore\]\(/windows/win32/api/synchapi/nf-synchapi-opensemaphorew\)](#) or [CreateSemaphore](#) function.

Use the [CloseHandle](#) function to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

Examples

For an example that uses [CreateSemaphore](#), see [Using Semaphore Objects](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateProcess](#)

[CreateSemaphoreEx](#)

[DuplicateHandle](#)

[Object Names](#)

[OpenSemaphore](#)

[ReleaseSemaphore](#)

[SECURITY_ATTRIBUTES](#)

[Semaphore Objects](#)

[Synchronization Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateSemaphoreExA function (winbase.h)

Article07/27/2022

Creates or opens a named or unnamed semaphore object and returns a handle to the object.

Syntax

C++

```
HANDLE CreateSemaphoreExA(
    [in, optional] LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    [in]           LONG             lInitialCount,
    [in]           LONG             lMaximumCount,
    [in, optional] LPCSTR          lpName,
                           DWORD            dwFlags,
    [in]           DWORD            dwDesiredAccess
);
```

Parameters

[in, optional] `lpSemaphoreAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure. If this parameter is **NULL**, the semaphore handle cannot be inherited by child processes.

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

[in] `lInitialCount`

The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to `lMaximumCount`. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the [ReleaseSemaphore](#) function.

[in] lMaximumCount

The maximum count for the semaphore object. This value must be greater than zero.

[in, optional] lpName

A pointer to a null-terminated string specifying the name of the semaphore object. The name is limited to MAX_PATH characters. Name comparison is case sensitive.

If *lpName* matches the name of an existing named semaphore object, the *lInitialCount* and *lMaximumCount* parameters are ignored because they have already been set by the creating process. If the *lpSemaphoreAttributes* parameter is not **NULL**, it determines whether the handle can be inherited.

If *lpName* is **NULL**, the semaphore object is created without a name.

If *lpName* matches the name of an existing event, mutex, waitable timer, job, or file-mapping 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 (\). For more information, see [Kernel Object Namespaces](#). Fast user switching is implemented using Terminal Services sessions. Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users.

The object can be created in a private namespace. For more information, see [Object Namespaces](#).

dwFlags

This parameter is reserved and must be 0.

[in] dwDesiredAccess

The access mask for the semaphore object. For a list of access rights, see [Synchronization Object Security and Access Rights](#).

Return value

If the function succeeds, the return value is a handle to the semaphore object. If the named semaphore object existed before the function call, the function returns a handle

to the existing object and [GetLastError](#) returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The state of a semaphore object is signaled when its count is greater than zero, and nonsignaled when its count is equal to zero. The *lInitialCount* parameter specifies the initial count. The count can never be less than zero or greater than the value specified in the *lMaximumCount* parameter.

Any thread of the calling process can specify the semaphore-object handle in a call to one of the [wait functions](#). The single-object wait functions return when the state of the specified object is signaled. The multiple-object wait functions can be instructed to return either when any one or when all of the specified objects are signaled. When a wait function returns, the waiting thread is released to continue its execution. Each time a thread completes a wait for a semaphore object, the count of the semaphore object is decremented by one. When the thread has finished, it calls the [ReleaseSemaphore](#) function, which increments the count of the semaphore object.

Multiple processes can have handles of the same semaphore object, enabling use of the object for interprocess synchronization. The following object-sharing mechanisms are available:

- A child process created by the [CreateProcess](#) function can inherit a handle to a semaphore object if the *lpSemaphoreAttributes* parameter of [CreateSemaphoreEx](#) enabled inheritance.
- A process can specify the semaphore-object handle in a call to the [DuplicateHandle](#) function to create a duplicate handle that can be used by another process.
- A process can specify the name of a semaphore object in a call to the [\[OpenSemaphore\]\(../synchapi/nf-synchapi-signalobjectandwait.md\)](#) or [CreateSemaphoreEx](#) function.

Use the [CloseHandle](#) function to close the handle. The system closes the handle automatically when the process terminates. The semaphore object is destroyed when its last handle has been closed.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[Semaphore Objects](#)

[Synchronization Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateSymbolicLinkA function (winbase.h)

Article06/01/2023

Creates a symbolic link.

To perform this operation as a transacted operation, use the [CreateSymbolicLinkTransacted](#) function.

Syntax

C++

```
BOOLEAN CreateSymbolicLinkA(
    [in] LPCSTR lpSymlinkFileName,
    [in] LPCSTR lpTargetFileName,
    [in] DWORD   dwFlags
);
```

Parameters

[in] lpSymlinkFileName

The symbolic link to be created.

This parameter may include the path.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpTargetFileName

The name of the target for the symbolic link to be created.

If *lpTargetFileName* has a device name associated with it, the link is treated as an absolute link; otherwise, the link is treated as a relative link.

This parameter may include the path.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] dwFlags

Indicates whether the link target, *lpTargetFileName*, is a directory.

Value	Meaning
0x0	The link target is a file.
SYMBOLIC_LINK_FLAG_DIRECTORY 0x1	The link target is a directory.
SYMBOLIC_LINK_FLAG_ALLOW_UNPRIVILEGED_CREATE 0x2	Specify this flag to allow creation of symbolic links when the process is not elevated. In UWP, Developer Mode must first be enabled on the machine before this option will function. Under MSIX, developer mode is not required to be enabled for this flag.

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

Symbolic links can either be absolute or relative links. Absolute links are links that specify each portion of the path name; relative links are determined relative to where relative-link specifiers are in a specified path. Relative links are specified using the following conventions:

- Dot (. and ..) conventions—for example, "..\" resolves the path relative to the parent directory.
- Names with no slashes (\)—for example, "tmp" resolves the path relative to the current directory.
- Root relative—for example, "\Windows\System32" resolves to "*current drive*:\\Windows\\System32".
- Current working directory-relative—for example, if the current working directory is C:\\Windows\\System32, "C:\\File.txt" resolves to "C:\\Windows\\System32\\File.txt".

Note If you specify a current working directory-relative link, it is created as an absolute link, due to the way the current working directory is processed based on the user and the thread.

To remove a symbolic link, delete the file (using [DeleteFile](#) or similar APIs) or remove the directory (using [RemoveDirectory](#) or similar APIs) depending on what type of symbolic link is used.

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)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	Yes

CsvFs does not support soft link or any other reparse points.

 **Note**

The winbase.h header defines CreateSymbolicLink as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateSymbolicLinkTransacted](#)

[File Management Functions](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateSymbolicLinkTransactedA function (winbase.h)

Article 02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Creates a symbolic link as a transacted operation.

Syntax

C++

```
BOOLEAN CreateSymbolicLinkTransactedA(
    [in] LPCSTR lpSymlinkFileName,
    [in] LPCSTR lpTargetFileName,
    [in] DWORD dwFlags,
    [in] HANDLE hTransaction
);
```

Parameters

`[in] lpSymlinkFileName`

The symbolic link to be created.

`[in] lpTargetFileName`

The name of the target for the symbolic link to be created.

If *lpTargetFileName* has a device name associated with it, the link is treated as an absolute link; otherwise, the link is treated as a relative link.

`[in] dwFlags`

Indicates whether the link target, *lpTargetFileName*, is a directory.

Value	Meaning

0x0	The link target is a file.
SYMBOLIC_LINK_FLAG_DIRECTORY 0x1	The link target is a directory.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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

Symbolic links can either be absolute or relative links. Absolute links are links that specify each portion of the path name; relative links are determined relative to where relative-link specifiers are in a specified path. Relative links are specified using the following conventions:

- Dot (.) and (..) conventions—for example, "..\" resolves the path relative to the parent directory.
- Names with no slashes (\)—for example, "tmp" resolves the path relative to the current directory.
- Root relative—for example, "\\Windows\System32" resolves to "*current drive*:\\Windows\\System32".
- Current working directory-relative—for example, if the current working directory is C:\\Windows\\System32, "C:File.txt" resolves to "C:\\Windows\\System32\\File.txt".

Note If you specify a current working directory-relative link, it is created as an absolute link, due to the way the current working directory is processed based on the user and the thread.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported

Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The `winbase.h` header defines `CreateSymbolicLinkTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[File Management Functions](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateSymbolicLinkTransactedW function (winbase.h)

Article 02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Creates a symbolic link as a transacted operation.

Syntax

C++

```
BOOLEAN CreateSymbolicLinkTransactedW(
    [in] LPCWSTR lpSymlinkFileName,
    [in] LPCWSTR lpTargetFileName,
    [in] DWORD    dwFlags,
    [in] HANDLE   hTransaction
);
```

Parameters

`[in] lpSymlinkFileName`

The symbolic link to be created.

`[in] lpTargetFileName`

The name of the target for the symbolic link to be created.

If *lpTargetFileName* has a device name associated with it, the link is treated as an absolute link; otherwise, the link is treated as a relative link.

`[in] dwFlags`

Indicates whether the link target, *lpTargetFileName*, is a directory.

Value	Meaning

0x0	The link target is a file.
SYMBOLIC_LINK_FLAG_DIRECTORY 0x1	The link target is a directory.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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

Symbolic links can either be absolute or relative links. Absolute links are links that specify each portion of the path name; relative links are determined relative to where relative-link specifiers are in a specified path. Relative links are specified using the following conventions:

- Dot (. and ..) conventions—for example, "..\" resolves the path relative to the parent directory.
- Names with no slashes (\)—for example, "tmp" resolves the path relative to the current directory.
- Root relative—for example, "\\Windows\System32" resolves to "*current drive*:\\Windows\\System32".
- Current working directory-relative—for example, if the current working directory is C:\\Windows\\System32, "C:File.txt" resolves to "C:\\Windows\\System32\\File.txt".

Note If you specify a current working directory-relative link, it is created as an absolute link, due to the way the current working directory is processed based on the user and the thread.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported

Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The `winbase.h` header defines `CreateSymbolicLinkTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[File Management Functions](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateSymbolicLinkW function (winbase.h)

Article06/01/2023

Creates a symbolic link.

To perform this operation as a transacted operation, use the [CreateSymbolicLinkTransacted](#) function.

Syntax

C++

```
BOOLEAN CreateSymbolicLinkW(
    [in] LPCWSTR lpSymlinkFileName,
    [in] LPCWSTR lpTargetFileName,
    [in] DWORD    dwFlags
);
```

Parameters

[in] lpSymlinkFileName

The symbolic link to be created.

This parameter may include the path.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpTargetFileName

The name of the target for the symbolic link to be created.

If *lpTargetFileName* has a device name associated with it, the link is treated as an absolute link; otherwise, the link is treated as a relative link.

This parameter may include the path.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] dwFlags

Indicates whether the link target, *lpTargetFileName*, is a directory.

Value	Meaning
0x0	The link target is a file.
SYMBOLIC_LINK_FLAG_DIRECTORY 0x1	The link target is a directory.
SYMBOLIC_LINK_FLAG_ALLOW_UNPRIVILEGED_CREATE 0x2	Specify this flag to allow creation of symbolic links when the process is not elevated. Developer Mode must first be enabled on the machine before this option will 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

Symbolic links can either be absolute or relative links. Absolute links are links that specify each portion of the path name; relative links are determined relative to where

relative-link specifiers are in a specified path. Relative links are specified using the following conventions:

- Dot (. and ..) conventions—for example, "..\" resolves the path relative to the parent directory.
- Names with no slashes (\)—for example, "tmp" resolves the path relative to the current directory.
- Root relative—for example, "\Windows\System32" resolves to "*current drive*:\\Windows\\System32".
- Current working directory-relative—for example, if the current working directory is C:\\Windows\\System32, "C:\\File.txt" resolves to "C:\\Windows\\System32\\File.txt".

Note If you specify a current working directory-relative link, it is created as an absolute link, due to the way the current working directory is processed based on the user and the thread.

To remove a symbolic link, delete the file (using [DeleteFile](#) or similar APIs) or remove the directory (using [RemoveDirectory](#) or similar APIs) depending on what type of symbolic link is used.

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)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	Yes

CsvFs does not support soft link or any other reparse points.

ⓘ Note

The winbase.h header defines CreateSymbolicLink as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with

code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateSymbolicLinkTransacted](#)

[File Management Functions](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateTapePartition function (winbase.h)

Article 10/13/2021

The **CreateTapePartition** function reformats a tape.

Syntax

C++

```
DWORD CreateTapePartition(
    [in] HANDLE hDevice,
    [in] DWORD dwPartitionMethod,
    [in] DWORD dwCount,
    [in] DWORD dwSize
);
```

Parameters

[in] `hDevice`

Handle to the device where the new partition is to be created. This handle is created by using the [CreateFile](#) function.

[in] `dwPartitionMethod`

Type of partition to create. To determine what type of partitions your device supports, see the documentation for your hardware. This parameter can have one of the following values.

Value	Meaning
<code>TAPE_FIXED_PARTITIONS</code> 0L	Partitions the tape based on the device's default definition of partitions. The <code>dwCount</code> and <code>dwSize</code> parameters are ignored.
<code>TAPE_INITIATOR_PARTITIONS</code> 2L	Partitions the tape into the number and size of partitions specified by <code>dwCount</code> and <code>dwSize</code> , respectively, except for the last partition. The size of the last partition is the remainder of the tape.
<code>TAPE_SELECT_PARTITIONS</code> 1L	Partitions the tape into the number of partitions specified by <code>dwCount</code> . The <code>dwSize</code> parameter is ignored. The size of

the partitions is determined by the device's default partition size. For more specific information, see the documentation for your tape device.

[in] dwCount

Number of partitions to create. The [GetTapeParameters](#) function provides the maximum number of partitions a tape can support.

[in] dwSize

Size of each partition, in megabytes. This value is ignored if the *dwPartitionMethod* parameter is **TAPE_SELECT_PARTITIONS**.

Return value

If the function succeeds, the return value is NO_ERROR.

If the function fails, it can return one of the following error codes.

Error	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.

ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT 19L	The media is write protected.

Remarks

Creating partitions reformats the tape. All previous information recorded on the tape is destroyed.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[GetTapeParameters](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateUmsCompletionList function (winbase.h)

Article03/18/2022

Creates a user-mode scheduling (UMS) completion list.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL CreateUmsCompletionList(
    [out] PUMS_COMPLETION_LIST *UmsCompletionList
);
```

Parameters

[out] `UmsCompletionList`

A `PUMS_COMPLETION_LIST` variable. On output, this parameter receives a pointer to an empty UMS completion list.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include the following.

Return code	Description
<code>ERROR_NOT_ENOUGH_MEMORY</code>	Not enough memory is available to create the completion list.
<code>ERROR_NOT_SUPPORTED</code>	UMS is not supported.

Remarks

A completion list is associated with a UMS scheduler thread when the [EnterUmsSchedulingMode](#) function is called to create the scheduler thread. The system queues newly created UMS worker threads to the completion list. It also queues previously blocked UMS worker threads to the completion list when the threads are no longer blocked.

When an application's [UmsSchedulerProc](#) entry point function is called, the application's scheduler should retrieve items from the completion list by calling [DequeueUmsCompletionListItems](#).

Each completion list has an associated completion list event which is signaled whenever the system queues items to an empty list. Use the [GetUmsCompletionListEvent](#) to obtain a handle to the event for a specified completion list.

When a completion list is no longer needed, use the [DeleteUmsCompletionList](#) to release the list. The list must be empty before it can be released.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[DequeueUmsCompletionListItems](#)

[EnterUmsSchedulingMode](#)

[GetUmsCompletionListEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateUmsThreadContext function (winbase.h)

Article03/18/2022

Creates a user-mode scheduling (UMS) thread context to represent a UMS worker thread.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL CreateUmsThreadContext(
    [out] PUMS_CONTEXT *lpUmsThread
);
```

Parameters

`[out] lpUmsThread`

A PUMS_CONTEXT variable. On output, this parameter receives a pointer to a UMS thread context.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include the following.

Return code	Description
<code>ERROR_NOT_ENOUGH_MEMORY</code>	Not enough memory is available to create the UMS thread context.

Remarks

A UMS thread context represents the state of a UMS worker thread. Thread contexts are used to specify UMS worker threads in function calls.

A UMS worker thread is created by calling the [CreateRemoteThreadEx](#) function after using [InitializeProcThreadAttributeList](#) and [UpdateProcThreadAttribute](#) to prepare a list of UMS attributes for the thread.

The underlying structures for a UMS thread context are managed by the system and should not be modified directly. To get and set information about a UMS worker thread, use the [QueryUmsThreadInformation](#) and [SetUmsThreadInformation](#) functions.

After a UMS worker thread terminates, its thread context should be released by calling [DeleteUmsThreadContext](#).

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[CreateRemoteThreadEx](#)

[DeleteUmsThreadContext](#)

[InitializeProcThreadAttributeList](#)

[QueryUmsThreadInformation](#)

[SetUmsThreadInformation](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DCB structure (winbase.h)

Article04/02/2021

Defines the control setting for a serial communications device.

Syntax

C++

```
typedef struct _DCB {
    DWORD DCBlength;
    DWORD BaudRate;
    DWORD fBinary : 1;
    DWORD fParity : 1;
    DWORD fOutxCtsFlow : 1;
    DWORD fOutxDsrFlow : 1;
    DWORD fDtrControl : 2;
    DWORD fDsrSensitivity : 1;
    DWORD fTXContinueOnXoff : 1;
    DWORD fOutX : 1;
    DWORD fInX : 1;
    DWORD fErrorChar : 1;
    DWORD fNull : 1;
    DWORD fRtsControl : 2;
    DWORD fAbortOnError : 1;
    DWORD fDummy2 : 17;
    WORD wReserved;
    WORD XonLim;
    WORD XoffLim;
    BYTE ByteSize;
    BYTE Parity;
    BYTE StopBits;
    char XonChar;
    char XoffChar;
    char ErrorChar;
    char EofChar;
    char EvtChar;
    WORD wReserved1;
} DCB, *LPDCB;
```

Members

DCBlength

The length of the structure, in bytes. The caller must set this member to `sizeof(DCB)`.

BaudRate

The baud rate at which the communications device operates. This member can be an actual baud rate value, or one of the following indexes.

Value	Meaning
CBR_110 110	110 bps
CBR_300 300	300 bps
CBR_600 600	600 bps
CBR_1200 1200	1200 bps
CBR_2400 2400	2400 bps
CBR_4800 4800	4800 bps
CBR_9600 9600	9600 bps
CBR_14400 14400	14400 bps
CBR_19200 19200	19200 bps
CBR_38400 38400	38400 bps
CBR_57600 57600	57600 bps
CBR_115200 115200	115200 bps
CBR_128000 128000	128000 bps
CBR_256000 256000	256000 bps

If this member is **TRUE**, binary mode is enabled. Windows does not support nonbinary mode transfers, so this member must be **TRUE**.

fParity

If this member is **TRUE**, parity checking is performed and errors are reported.

fOutxCtsFlow

If this member is **TRUE**, the CTS (clear-to-send) signal is monitored for output flow control. If this member is **TRUE** and CTS is turned off, output is suspended until CTS is sent again.

fOutxDsrFlow

If this member is **TRUE**, the DSR (data-set-ready) signal is monitored for output flow control. If this member is **TRUE** and DSR is turned off, output is suspended until DSR is sent again.

fDtrControl

The DTR (data-terminal-ready) flow control. This member can be one of the following values.

Value	Meaning
DTR_CONTROL_DISABLE 0x00	Disables the DTR line when the device is opened and leaves it disabled.
DTR_CONTROL_ENABLE 0x01	Enables the DTR line when the device is opened and leaves it on.
DTR_CONTROL_HANDSHAKE 0x02	Enables DTR handshaking. If handshaking is enabled, it is an error for the application to adjust the line by using the EscapeCommFunction function.

fDsrSensitivity

If this member is **TRUE**, the communications driver is sensitive to the state of the DSR signal. The driver ignores any bytes received, unless the DSR modem input line is high.

fTXContinueOnXoff

If this member is **TRUE**, transmission continues after the input buffer has come within **XoffLim** bytes of being full and the driver has transmitted the **XoffChar** character to stop receiving bytes. If this member is **FALSE**, transmission does not continue until the

input buffer is within **XonLim** bytes of being empty and the driver has transmitted the **XonChar** character to resume reception.

fOutX

Indicates whether XON/XOFF flow control is used during transmission. If this member is **TRUE**, transmission stops when the **XoffChar** character is received and starts again when the **XonChar** character is received.

fInX

Indicates whether XON/XOFF flow control is used during reception. If this member is **TRUE**, the **XoffChar** character is sent when the input buffer comes within **XoffLim** bytes of being full, and the **XonChar** character is sent when the input buffer comes within **XonLim** bytes of being empty.

fErrorChar

Indicates whether bytes received with parity errors are replaced with the character specified by the **ErrorChar** member. If this member is **TRUE** and the **fParity** member is **TRUE**, replacement occurs.

fNull

If this member is **TRUE**, null bytes are discarded when received.

fRtsControl

The RTS (request-to-send) flow control. This member can be one of the following values.

Value	Meaning
RTS_CONTROL_DISABLE 0x00	Disables the RTS line when the device is opened and leaves it disabled.
RTS_CONTROL_ENABLE 0x01	Enables the RTS line when the device is opened and leaves it on.
RTS_CONTROL_HANDSHAKE 0x02	Enables RTS handshaking. The driver raises the RTS line when the "type-ahead" (input) buffer is less than one-half full and lowers the RTS line when the buffer is more than three-quarters full. If handshaking is enabled, it is an error for the application to adjust the line by using the EscapeCommFunction function.
RTS_CONTROL_TOGGLE 0x03	Specifies that the RTS line will be high if bytes are available for transmission. After all buffered bytes have been sent, the RTS line will be low.

fAbortOnError

If this member is **TRUE**, the driver terminates all read and write operations with an error status if an error occurs. The driver will not accept any further communications operations until the application has acknowledged the error by calling the [ClearCommError](#) function.

fDummy2

Reserved; do not use.

wReserved

Reserved; must be zero.

XonLim

The minimum number of bytes in use allowed in the input buffer before flow control is activated to allow transmission by the sender. This assumes that either XON/XOFF, RTS, or DTR input flow control is specified in the **fInX**, **fRtsControl**, or **fDtrControl** members.

XoffLim

The minimum number of free bytes allowed in the input buffer before flow control is activated to inhibit the sender. Note that the sender may transmit characters after the flow control signal has been activated, so this value should never be zero. This assumes that either XON/XOFF, RTS, or DTR input flow control is specified in the **fInX**, **fRtsControl**, or **fDtrControl** members. The maximum number of bytes in use allowed is calculated by subtracting this value from the size, in bytes, of the input buffer.

ByteSize

The number of bits in the bytes transmitted and received.

Parity

The parity scheme to be used. This member can be one of the following values.

Value	Meaning
EVENPARITY 2	Even parity.
MARKPARITY 3	Mark parity.

NOPARITY	No parity.
0	
ODDPARITY	Odd parity.
1	
SPACEPARITY	Space parity.
4	

StopBits

The number of stop bits to be used. This member can be one of the following values.

Value	Meaning
ONESTOPBIT	1 stop bit.
0	
ONE5STOPBITS	1.5 stop bits.
1	
TWOSTOPBITS	2 stop bits.
2	

XonChar

The value of the XON character for both transmission and reception.

XoffChar

The value of the XOFF character for both transmission and reception.

ErrorChar

The value of the character used to replace bytes received with a parity error.

EofChar

The value of the character used to signal the end of data.

EvtChar

The value of the character used to signal an event.

wReserved1

Reserved; do not use.

Remarks

When a **DCB** structure is used to configure the 8250, the following restrictions apply to the values specified for the **ByteSize** and **StopBits** members:

- The number of data bits must be 5 to 8 bits.
- The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Header	winbase.h (include Windows.h)

See also

[BuildCommDCB](#)

[ClearCommError](#)

[EscapeCommFunction](#)

[GetCommState](#)

[SetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeactivateActCtx function (winbase.h)

Article10/13/2021

The **DeactivateActCtx** function deactivates the activation context corresponding to the specified cookie.

Syntax

C++

```
BOOL DeactivateActCtx(
    [in] DWORD      dwFlags,
    [in] ULONG_PTR ulCookie
);
```

Parameters

[in] *dwFlags*

Flags that indicate how the deactivation is to occur.

Value	Meaning
0	If this value is set and the cookie specified in the <i>ulCookie</i> parameter is in the top frame of the activation stack, the activation context is popped from the stack and thereby deactivated.
	If this value is set and the cookie specified in the <i>ulCookie</i> parameter is not in the top frame of the activation stack, this function searches down the stack for the cookie.
	If the cookie is found, a STATUS_SXS_EARLY_DEACTIVATION exception is thrown.
	If the cookie is not found, a STATUS_SXS_INVALID_DEACTIVATION exception is thrown.

	This value should be specified in most cases.
DEACTIVATE_ACTCTX_FLAG_FORCE_EARLY_DEACTIVATION	If this value is set and the cookie specified in the <i>ulCookie</i> parameter is in the top frame of the activation stack, the function returns an ERROR_INVALID_PARAMETER error code. Call GetLastError to obtain this code.
	If this value is set and the cookie is not on the activation stack, a STATUS_SXS_INVALID_DEACTIVATION exception will be thrown.
	If this value is set and the cookie is in a lower frame of the activation stack, all of the frames down to and including the frame the cookie is in is popped from the stack.

[in] *ulCookie*

The **ULONG_PTR** that was passed into the call to [ActivateActCtx](#). This value is used as a cookie to identify a specific activated activation context.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

The deactivation of activation contexts must occur in the reverse order of activation. It can be understood as popping an activation context from a stack.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ActivateActCtx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DebugBreakProcess function (winbase.h)

Article 10/13/2021

Causes a breakpoint exception to occur in the specified process. This allows the calling thread to signal the debugger to handle the exception.

Syntax

C++

```
BOOL DebugBreakProcess(  
    [in] HANDLE Process  
);
```

Parameters

[in] Process

A handle to the process.

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

If the process is not being debugged, the function uses the search logic of a standard exception handler. In most cases, this causes the process to terminate because of an unhandled breakpoint exception.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communicating with the Debugger](#)

[DebugBreak](#)

[Debugging Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DebugSetProcessKillOnExit function (winbase.h)

Article 10/13/2021

Sets the action to be performed when the calling thread exits.

Syntax

C++

```
BOOL DebugSetProcessKillOnExit(
    [in] BOOL KillOnExit
);
```

Parameters

[in] KillOnExit

If this parameter is **TRUE**, the thread terminates all attached processes on exit (note that this is the default). Otherwise, the thread detaches from all processes being debugged on exit.

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

The calling thread must have established at least one debugging connection using the [CreateProcess](#) or [DebugActiveProcess](#) function before calling this function.

`DebugSetProcessKillOnExit` affects all current and future debuggees connected to the calling thread. A thread can call this function multiple times to change the action as needed.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DebugActiveProcessStop](#)

[Debugging Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DecryptFileA function (winbase.h)

Article02/09/2023

Decrypts an encrypted file or directory.

Syntax

C++

```
BOOL DecryptFileA(
    [in] LPCSTR lpFileName,
    DWORD    dwReserved
);
```

Parameters

[in] lpFileName

The name of the file or directory to be decrypted.

The caller must have the **FILE_READ_DATA**, **FILE_WRITE_DATA**, **FILE_READ_ATTRIBUTES**, **FILE_WRITE_ATTRIBUTES**, and **SYNCHRONIZE** access rights. For more information, see [File Security and Access Rights](#).

dwReserved

Reserved; must be zero.

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

The **DecryptFile** function requires exclusive access to the file being decrypted, and will fail if another process is using the file. If the file is not encrypted, **DecryptFile** simply returns a nonzero value, which indicates success.

If *lpFileName* specifies a read-only file, the function fails and [GetLastError](#) returns **ERROR_FILE_READ_ONLY**. If *lpFileName* specifies a directory that contains a read-only file, the functions succeeds but the directory is not decrypted.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

Note

The winbase.h header defines DecryptFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Advapi32.lib
DLL	Advapi32.dll

API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)
---------	--

See also

[CreateFile](#)

[EncryptFile](#)

[File Encryption](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DecryptFileW function (winbase.h)

Article02/09/2023

Decrypts an encrypted file or directory.

Syntax

C++

```
BOOL DecryptFileW(
    [in] LPCWSTR lpFileName,
    DWORD     dwReserved
);
```

Parameters

[in] lpFileName

The name of the file or directory to be decrypted.

The caller must have the **FILE_READ_DATA**, **FILE_WRITE_DATA**, **FILE_READ_ATTRIBUTES**, **FILE_WRITE_ATTRIBUTES**, and **SYNCHRONIZE** access rights. For more information, see [File Security and Access Rights](#).

dwReserved

Reserved; must be zero.

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

The **DecryptFile** function requires exclusive access to the file being decrypted, and will fail if another process is using the file. If the file is not encrypted, **DecryptFile** simply returns a nonzero value, which indicates success.

If *lpFileName* specifies a read-only file, the function fails and [GetLastError](#) returns **ERROR_FILE_READ_ONLY**. If *lpFileName* specifies a directory that contains a read-only file, the functions succeeds but the directory is not decrypted.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

Note

The winbase.h header defines DecryptFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Advapi32.lib
DLL	Advapi32.dll

API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)
---------	--

See also

[CreateFile](#)

[EncryptFile](#)

[File Encryption](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DefineDosDeviceA function (winbase.h)

Article07/27/2022

Defines, redefines, or deletes MS-DOS device names.

Syntax

C++

```
BOOL DefineDosDeviceA(
    [in]          DWORD  dwFlags,
    [in]          LPCSTR lpDeviceName,
    [in, optional] LPCSTR lpTargetPath
);
```

Parameters

[in] dwFlags

The controllable aspects of the **DefineDosDevice** function. This parameter can be one or more of the following values.

Value	Meaning
DDD_EXACT_MATCH_ON_REMOVE 0x00000004	If this value is specified along with DDD_REMOVE_DEFINITION , the function will use an exact match to determine which mapping to remove. Use this value to ensure that you do not delete something that you did not define.
DDD_NO_BROADCAST_SYSTEM 0x00000008	Do not broadcast the WM_SETTINGCHANGE message. By default, this message is broadcast to notify the shell and applications of the change.
DDD_RAW_TARGET_PATH 0x00000001	Uses the <i>lpTargetPath</i> string as is. Otherwise, it is converted from an MS-DOS path to a path.
DDD_REMOVE_DEFINITION 0x00000002	Removes the specified definition for the specified device. To determine which definition to remove, the function walks the list of mappings for the device, looking for a match of <i>lpTargetPath</i> against a prefix of each mapping associated with this device. The first mapping that matches is the one removed, and then the function returns.

If *lpTargetPath* is **NULL** or a pointer to a **NULL** string, the function will remove the first mapping associated with the device and pop the most recent one pushed. If there is nothing left to pop, the device name will be removed.

If this value is not specified, the string pointed to by the *lpTargetPath* parameter will become the new mapping for this device.

[in] *lpDeviceName*

A pointer to an MS-DOS device name string specifying the device the function is defining, redefining, or deleting. The device name string must not have a colon as the last character, unless a drive letter is being defined, redefined, or deleted. For example, drive C would be the string "C:". In no case is a trailing backslash ("") allowed.

[in, optional] *lpTargetPath*

A pointer to a path string that will implement this device. The string is an MS-DOS path string unless the **DDD_RAW_TARGET_PATH** flag is specified, in which case this string is a path string.

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

MS-DOS device names are stored as junctions in the object namespace. The code that converts an MS-DOS path into a corresponding path uses these junctions to map MS-DOS devices and drive letters. The **DefineDosDevice** function enables an application to modify the junctions used to implement the MS-DOS device namespace.

To retrieve the current mapping for a particular MS-DOS device name or to obtain a list of all MS-DOS devices known to the system, use the [QueryDosDevice](#) function.

To define a drive letter assignment that is persistent across boots and not a network share, use the [SetVolumeMountPoint](#) function. If the volume to be mounted already has a drive letter assigned to it, use the [DeleteVolumeMountPoint](#) function to remove the assignment.

Drive letters and device names defined at system boot time are protected from redefinition and deletion unless the user is an administrator.

Starting with Windows XP, this function creates a device name for a caller that is not running in the "LocalSystem" context in its own Local MS-DOS device namespace. If the caller is running in the "LocalSystem" context, the function creates the device name in the Global MS-DOS device namespace. For more information, see [Defining an MS DOS Device Name](#) and [File Names, Paths, and Namespaces](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. For CsvFs, a new name will not be replicated to the other nodes on the cluster.

Examples

For an example, see [Editing Drive Letter Assignments](#).

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)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[DeleteVolumeMountPoint](#)

[QueryDosDevice](#)

[SetVolumeMountPoint](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteAtom function (winbase.h)

Article 10/13/2021

Decrements the reference count of a local string atom. If the atom's reference count is reduced to zero, **DeleteAtom** removes the string associated with the atom from the local atom table.

Syntax

C++

```
ATOM DeleteAtom(  
    [in] ATOM nAtom  
);
```

Parameters

[in] nAtom

Type: **ATOM**

The atom to be deleted.

Return value

Type: **ATOM**

If the function succeeds, the return value is zero.

If the function fails, the return value is the *nAtom* parameter. To get extended error information, call [GetLastError](#).

Remarks

A string atom's reference count specifies the number of times the atom has been added to the atom table. The [AddAtom](#) function increments the count on each call. The **DeleteAtom** function decrements the count on each call but removes the string only if the atom's reference count is zero.

Each call to [AddAtom](#) should have a corresponding call to [DeleteAtom](#). Do not call [DeleteAtom](#) more times than you call [AddAtom](#), or you may delete the atom while other clients are using it.

The [DeleteAtom](#) function has no effect on an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF). The function always returns zero for an integer atom.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[FindAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[MAKEINTATOM](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteFiber function (winbase.h)

Article10/13/2021

Deletes an existing fiber.

Syntax

C++

```
void DeleteFiber(  
    [in] LPVOID lpFiber  
);
```

Parameters

[in] lpFiber

The address of the fiber to be deleted.

Return value

None

Remarks

The **DeleteFiber** function deletes all data associated with the fiber. This data includes the stack, a subset of the registers, and the fiber data.

If the currently running fiber calls **DeleteFiber**, its thread calls **ExitThread** and terminates. However, if a currently running fiber is deleted by another fiber, the thread running the deleted fiber is likely to terminate abnormally because the fiber stack has been freed.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Examples

For an example, see [Using Fibers](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ExitThread](#)

[Fibers](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteFile function (winbase.h)

Article06/01/2023

Deletes an existing file.

To perform this operation as a transacted operation, use the [DeleteFileTransacted](#) function.

Syntax

C++

```
BOOL DeleteFile(
    [in] LPCTSTR lpFileName
);
```

Parameters

[in] lpFileName

The name of the file to be deleted.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

Remarks

If an application attempts to delete a file that does not exist, the **DeleteFile** function fails with **ERROR_FILE_NOT_FOUND**. If the file is a read-only file, the function fails with **ERROR_ACCESS_DENIED**.

The following list identifies some tips for deleting, removing, or closing files:

- To delete a read-only file, first you must remove the read-only attribute.
- To delete or rename a file, you must have either delete permission on the file, or delete child permission in the parent directory.
- To recursively delete the files in a directory, use the [SHFileOperation](#) function.
- To remove an empty directory, use the [RemoveDirectory](#) function.
- To close an open file, use the [CloseHandle](#) function.

If you set up a directory with all access except delete and delete child, and the access control lists (ACL) of new files are inherited, then you can create a file without being able to delete it. However, then you can create a file, and then get all the access you request on the handle that is returned to you at the time you create the file.

If you request delete permission at the time you create a file, you can delete or rename the file with that handle, but not with any other handle. For more information, see [File Security and Access Rights](#).

The **DeleteFile** function fails if an application attempts to delete a file that has other handles open for normal I/O or as a memory-mapped file (**FILE_SHARE_DELETE** must have been specified when other handles were opened).

The **DeleteFile** function marks a file for deletion on close. Therefore, the file deletion does not occur until the last handle to the file is closed. Subsequent calls to [CreateFile](#) to open the file fail with **ERROR_ACCESS_DENIED**.

Symbolic link behavior—

If the path points to a symbolic link, the symbolic link is deleted, not the target. To delete a target, you must call [CreateFile](#) and specify **FILE_FLAG_DELETE_ON_CLOSE**.

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 an example, see [Locking and Unlocking Byte Ranges in Files](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFile](#)

[DeleteFileTransacted](#)

[File Management Functions](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DeleteFileTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Deletes an existing file as a transacted operation.

Syntax

C++

```
BOOL DeleteFileTransactedA(
    [in] LPCSTR lpFileName,
    [in] HANDLE hTransaction
);
```

Parameters

[in] lpFileName

The name of the file to be deleted.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

If an application attempts to delete a file that does not exist, the [DeleteFileTransacted](#) function fails with [ERROR_FILE_NOT_FOUND](#). If the file is a read-only file, the function fails with [ERROR_ACCESS_DENIED](#).

The following list identifies some tips for deleting, removing, or closing files:

- To delete a read-only file, first you must remove the read-only attribute.
- To delete or rename a file, you must have either delete permission on the file, or delete child permission in the parent directory.
- To recursively delete the files in a directory, use the [SHFileOperation](#) function.
- To remove an empty directory, use the [RemoveDirectoryTransacted](#) function.
- To close an open file, use the [CloseHandle](#) function.

If you set up a directory with all access except delete and delete child, and the access control lists (ACL) of new files are inherited, then you can create a file without being able to delete it. However, then you can create a file, and then get all the access you request on the handle that is returned to you at the time you create the file.

If you request delete permission at the time you create a file, you can delete or rename the file with that handle, but not with any other handle. For more information, see [File Security and Access Rights](#).

The [DeleteFileTransacted](#) function fails if an application attempts to delete a file that has other handles open for normal I/O or as a memory-mapped file ([FILE_SHARE_DELETE](#) must have been specified when other handles were opened).

The [DeleteFileTransacted](#) function marks a file for deletion on close. The file is deleted after the last transacted writer handle to the file is closed, provided that the transaction is still active. If a file has been marked for deletion and a transacted writer handle is still open after the transaction completes, the file will not be deleted.

Symbolic links: If the path points to a symbolic link, the symbolic link is deleted, not the target. To delete a target, you must call [CreateFile](#) and specify **FILE_FLAG_DELETE_ON_CLOSE**.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

 **Note**

The winbase.h header defines DeleteFileTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileTransacted](#)

[File Management Functions](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteFileTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Deletes an existing file as a transacted operation.

Syntax

C++

```
BOOL DeleteFileTransactedW(
    [in] LPCWSTR lpFileName,
    [in] HANDLE hTransaction
);
```

Parameters

`[in] lpFileName`

The name of the file to be deleted.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

If an application attempts to delete a file that does not exist, the **DeleteFileTransacted** function fails with **ERROR_FILE_NOT_FOUND**. If the file is a read-only file, the function fails with **ERROR_ACCESS_DENIED**.

The following list identifies some tips for deleting, removing, or closing files:

- To delete a read-only file, first you must remove the read-only attribute.
- To delete or rename a file, you must have either delete permission on the file, or delete child permission in the parent directory.
- To recursively delete the files in a directory, use the [SHFileOperation](#) function.
- To remove an empty directory, use the [RemoveDirectoryTransacted](#) function.
- To close an open file, use the [CloseHandle](#) function.

If you set up a directory with all access except delete and delete child, and the access control lists (ACL) of new files are inherited, then you can create a file without being able to delete it. However, then you can create a file, and then get all the access you request on the handle that is returned to you at the time you create the file.

If you request delete permission at the time you create a file, you can delete or rename the file with that handle, but not with any other handle. For more information, see [File Security and Access Rights](#).

The **DeleteFileTransacted** function fails if an application attempts to delete a file that has other handles open for normal I/O or as a memory-mapped file (**FILE_SHARE_DELETE** must have been specified when other handles were opened).

The **DeleteFileTransacted** function marks a file for deletion on close. The file is deleted after the last transacted writer handle to the file is closed, provided that the transaction is still active. If a file has been marked for deletion and a transacted writer handle is still open after the transaction completes, the file will not be deleted.

Symbolic links: If the path points to a symbolic link, the symbolic link is deleted, not the target. To delete a target, you must call [CreateFile](#) and specify **FILE_FLAG_DELETE_ON_CLOSE**.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

 **Note**

The winbase.h header defines DeleteFileTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileTransacted](#)

[File Management Functions](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteUmsCompletionList function (winbase.h)

Article03/18/2022

Deletes the specified user-mode scheduling (UMS) completion list. The list must be empty.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL DeleteUmsCompletionList(
    [in] PUMS_COMPLETION_LIST UmsCompletionList
);
```

Parameters

`[in] UmsCompletionList`

A pointer to the UMS completion list to be deleted. The [CreateUmsCompletionList](#) function provides this pointer.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If the completion list is shared, the caller is responsible for ensuring that no active UMS thread holds a reference to the list before deleting it.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteUmsThreadContext function (winbase.h)

Article03/18/2022

Deletes the specified user-mode scheduling (UMS) thread context. The thread must be terminated.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL DeleteUmsThreadContext(  
    [in] PUMS_CONTEXT UmsThread  
);
```

Parameters

[in] `UmsThread`

A pointer to the UMS thread context to be deleted. The [CreateUmsThreadContext](#) function provides this pointer.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A UMS thread context cannot be deleted until the associated thread has terminated.

When a UMS worker thread finishes running (for example, by returning from its thread entry point function), the system terminates the thread, sets the termination status in the thread's UMS thread context, and queues the UMS thread context to the associated completion list.

Any attempt to execute the UMS thread will fail because the thread is already terminated.

To check the termination status of a thread, the application's scheduler should call [QueryUmsThreadInformation](#) with the **UmsIsThreadTerminated** information class.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[CreateUmsThreadContext](#)

[QueryUmsThreadInformation](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeleteVolumeMountPointA function (winbase.h)

Article07/27/2022

Deletes a drive letter or mounted folder.

Syntax

C++

```
BOOL DeleteVolumeMountPointA(
    [in] LPCSTR lpszVolumeMountPoint
);
```

Parameters

[in] *lpszVolumeMountPoint*

The drive letter or mounted folder to be deleted. A trailing backslash is required, for example, "X:" or "Y:\MountX".

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

Deleting a mounted folder does not cause the underlying directory to be deleted.

If the *lpszVolumeMountPoint* parameter is a directory that is not a mounted folder, the function does nothing. The directory is not deleted.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported

Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. For CsvFs, a new mount point will not be replicated to the other nodes on the cluster.

Examples

For an example, see [Unmounting a Volume at a Mount Point](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetVolumeNameForVolumeMountPoint](#)

[GetVolumePathName](#)

[Mounted Folders](#)

[SetVolumeMountPoint](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DequeueUmsCompletionListItems function (winbase.h)

Article 03/18/2022

Retrieves user-mode scheduling (UMS) worker threads from the specified UMS completion list.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL DequeueUmsCompletionListItems(
    [in] PUMS_COMPLETION_LIST UmsCompletionList,
    [in] DWORD              WaitTimeOut,
    [out] PUMS_CONTEXT       *UmsThreadList
);
```

Parameters

[in] `UmsCompletionList`

A pointer to the completion list from which to retrieve worker threads.

[in] `WaitTimeOut`

The time-out interval for the retrieval operation, in milliseconds. The function returns if the interval elapses, even if no worker threads are queued to the completion list.

If the `WaitTimeOut` parameter is zero, the completion list is checked for available worker threads without waiting for worker threads to become available. If the `WaitTimeOut` parameter is `INFINITE`, the function's time-out interval never elapses. This is not recommended, however, because it causes the function to block until one or more worker threads become available.

[out] `UmsThreadList`

A pointer to a UMS_CONTEXT variable. On output, this parameter receives a pointer to the first UMS thread context in a list of UMS thread contexts.

If no worker threads are available before the time-out specified by the *WaitTimeOut* parameter, this parameter is set to NULL.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include the following.

Return code	Description
ERROR_TIMEOUT	No threads became available before the specified time-out interval elapsed.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

The system queues a UMS worker thread to a completion list when the worker thread is created or when a previously blocked worker thread becomes unblocked. The [DequeueUmsCompletionListItems](#) function retrieves a pointer to a list of all thread contexts in the specified completion list. The [GetNextUmsListItem](#) function can be used to pop UMS thread contexts off the list into the scheduler's own ready thread queue. The scheduler is responsible for selecting threads to run based on priorities chosen by the application.

Do not run UMS threads directly from the list provided by [DequeueUmsCompletionListItems](#), or run a thread transferred from the list to the ready thread queue before the list is completely empty. This can cause unpredictable behavior in the application.

If more than one caller attempts to retrieve threads from a shared completion list, only the first caller retrieves the threads. For subsequent callers, the [DequeueUmsCompletionListItems](#) function returns success but the *UmsThreadList* parameter is set to NULL.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[GetNextUmsListItem](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DeregisterEventSource function (winbase.h)

Article07/27/2022

Closes the specified event log.

Syntax

C++

```
BOOL DeregisterEventSource(
    [in, out] HANDLE hEventLog
);
```

Parameters

[in, out] hEventLog

A handle to the event log. The [RegisterEventSource](#) function returns this handle.

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](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib

DLL

Advapi32.dll

See also

[Event Logging Functions](#)

[Event Sources](#)

[RegisterEventSource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DestroyThreadpoolEnvironment function (winbase.h)

Article07/27/2022

Deletes the specified callback environment. Call this function when the callback environment is no longer needed for creating new thread pool objects.

Syntax

C++

```
void DestroyThreadpoolEnvironment(
    [in, out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

[in, out] pcbe

A TP_CALLBACK_ENVIRON structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

Return value

None

Remarks

This function is implemented as an inline function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or higher.

Requirements

Minimum supported client

Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[InitializeThreadpoolEnvironment](#)

[SetThreadpoolCallbackCleanupGroup](#)

[SetThreadpoolCallbackLibrary](#)

[SetThreadpoolCallbackPool](#)

[SetThreadpoolCallbackPriority](#)

[SetThreadpoolCallbackRunsLong](#)

[Thread Pools](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DisableThreadProfiling function (winbase.h)

Article 10/13/2021

Disables thread profiling.

Syntax

C++

```
DWORD DisableThreadProfiling(
    [in] HANDLE PerformanceDataHandle
);
```

Parameters

[in] PerformanceDataHandle

The handle that the [EnableThreadProfiling](#) function returned.

Return value

Returns ERROR_SUCCESS if the call is successful; otherwise, a system error code (see Winerror.h).

Remarks

You must call this function from the same thread that enabled profiling for the specified handle. You must call this function before exiting the thread; otherwise, you will leak resources (the resources are not reclaimed until the process exits).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[EnableThreadProfiling](#)

[QueryThreadProfiling](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DnsHostnameToComputerNameA function (winbase.h)

Article02/09/2023

Converts a DNS-style host name to a NetBIOS-style computer name.

Syntax

C++

```
BOOL DnsHostnameToComputerNameA(
    [in]      LPCSTR Hostname,
    [out]     LPSTR ComputerName,
    [in, out] LPDWORD nSize
);
```

Parameters

[in] Hostname

The DNS name. If the DNS name is not a valid, translatable name, the function fails. For more information, see [Computer Names](#).

[out] ComputerName

A pointer to a buffer that receives the computer name. The buffer size should be large enough to contain MAX_COMPUTERNAME_LENGTH + 1 characters.

[in, out] nSize

On input, specifies the size of the buffer, in TCHARs. On output, receives the number of TCHARs copied to the destination buffer, not including the terminating null character.

If the buffer is too small, the function fails, [GetLastError](#) returns ERROR_MORE_DATA, and *nSize* receives the required buffer size, not including the terminating null character.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible values include the following.

Return code	Description
ERROR_MORE_DATA	The <i>ComputerName</i> buffer is too small. The <i>nSize</i> parameter contains the number of bytes required to receive the name.

Remarks

This function performs a textual mapping of the name. This convention limits the names of computers to be the common subset of the names. (Specifically, the leftmost label of the DNS name is truncated to 15-bytes of OEM characters.) Therefore, do not use this function to convert a DNS domain name to a NetBIOS domain name. There is no textual mapping for domain names.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see [Using the Windows Headers](#).

ⓘ Note

The winbase.h header defines DnsHostnameToComputerName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[GetComputerNameEx](#)

[SetComputerNameEx](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

DnsHostnameToComputerNameW function (winbase.h)

Article02/09/2023

Converts a DNS-style host name to a NetBIOS-style computer name.

Syntax

C++

```
BOOL DnsHostnameToComputerNameW(
    [in]      LPCWSTR Hostname,
    [out]     LPWSTR  ComputerName,
    [in, out] LPDWORD nSize
);
```

Parameters

[in] Hostname

The DNS name. If the DNS name is not a valid, translatable name, the function fails. For more information, see [Computer Names](#).

[out] ComputerName

A pointer to a buffer that receives the computer name. The buffer size should be large enough to contain MAX_COMPUTERNAME_LENGTH + 1 characters.

[in, out] nSize

On input, specifies the size of the buffer, in TCHARs. On output, receives the number of TCHARs copied to the destination buffer, not including the terminating null character.

If the buffer is too small, the function fails, [GetLastError](#) returns ERROR_MORE_DATA, and *nSize* receives the required buffer size, not including the terminating null character.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible values include the following.

Return code	Description
ERROR_MORE_DATA	The <i>ComputerName</i> buffer is too small. The <i>nSize</i> parameter contains the number of bytes required to receive the name.

Remarks

This function performs a textual mapping of the name. This convention limits the names of computers to be the common subset of the names. (Specifically, the leftmost label of the DNS name is truncated to 15-bytes of OEM characters.) Therefore, do not use this function to convert a DNS domain name to a NetBIOS domain name. There is no textual mapping for domain names.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see [Using the Windows Headers](#).

ⓘ Note

The winbase.h header defines DnsHostnameToComputerName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[GetComputerNameEx](#)

[SetComputerNameEx](#)

[System Information Functions](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

DosDateTimeToFileTime function (winbase.h)

Article 10/13/2021

Converts MS-DOS date and time values to a file time.

Syntax

C++

```
BOOL DosDateTimeToFileTime(
    [in] WORD      wFatDate,
    [in] WORD      wFatTime,
    [out] LPFILETIME lpFileTime
);
```

Parameters

[in] wFatDate

The MS-DOS date. The date is a packed value with the following format.

Bits	Description
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (add 1980 to get actual year)

[in] wFatTime

The MS-DOS time. The time is a packed value with the following format.

Bits	Description
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

[out] lpFileTime

A pointer to a [FILETIME](#) structure that receives the converted file time.

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](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FILETIME](#)

[FileTimeToDosDateTime](#)

[FileTimeToSystemTime](#)

[SystemTimeToFileTime](#)

[Time Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnableProcessOptionalXStateFeatures function (winbase.h)

Article11/04/2022

This function enables a set of optional XState features for the current process.

Syntax

C++

```
BOOL EnableProcessOptionalXStateFeatures(
    DWORD64 Features
);
```

Parameters

Features

A bitmask in which each bit represents an optional XState feature to enable for the current process.

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

In general, optional XState features are disabled by default for newly created threads and enabled on demand later. When this function returns, the specified optional XState features will be enabled for all existing threads in the current process, and all future threads created in the process will have the specified optional XState features enabled at thread creation time.

Only XState feature bits supported by the system are allowed to be supplied to this function, otherwise an error is returned. The XState feature bits supported by the system can be obtained via the [GetEnabledXStateFeatures](#) routine. If non-optional XState

feature bits supported by the system are supplied (for example AVX, AVX2, etc. are non-optional XState features), those are ignored and will not cause this function to return an error. Note that all non-optional XState features supported by the system are always enabled for every thread by default.

Requirements

Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
Header	winbase.h

See also

[GetEnabledXStateFeatures](#)

[GetThreadEnabledXStateFeatures](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnableThreadProfiling function (winbase.h)

Article 10/13/2021

Enables thread profiling on the specified thread.

Syntax

C++

```
DWORD EnableThreadProfiling(
    [in]  HANDLE ThreadHandle,
    [in]  DWORD   Flags,
    [in]  DWORD64 HardwareCounters,
    [out] HANDLE *PerformanceDataHandle
);
```

Parameters

[in] ThreadHandle

The handle to the thread on which you want to enable profiling. This must be the current thread.

[in] Flags

To receive thread profiling data such as context switch count, set this parameter to THREAD_PROFILING_FLAG_DISPATCH; otherwise, set to 0.

[in] HardwareCounters

To receive hardware performance counter data, set this parameter to a bitmask that identifies the hardware counters to collect. You can specify up to 16 performance counters. Each bit relates directly to the zero-based hardware counter index for the hardware performance counters that you configured. Set to zero if you are not collecting hardware counter data. If you set a bit for a hardware counter that has not been configured, the counter value that is read for that counter is zero.

[out] PerformanceDataHandle

An opaque handle that you use when calling the [ReadThreadProfilingData](#) and [DisableThreadProfiling](#) functions.

Return value

Returns ERROR_SUCCESS if the call is successful; otherwise, a system error code (see Winerror.h).

Remarks

You must call the [DisableThreadProfiling](#) function before exiting the thread.

To profile hardware performance counters, you need a driver to configure the counters. The performance counters are configured globally for the system, so every thread has access to the same hardware counter data. The counters must be configured before you enable profiling. For information on configuring hardware performance counters, see the [KeSetHardwareCounterConfiguration](#) function in the Windows Driver Kit (WDK).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DisableThreadProfiling](#)

[QueryThreadProfiling](#)

[ReadThreadProfilingData](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EncryptFileA function (winbase.h)

Article02/09/2023

Encrypts a file or directory. All data streams in a file are encrypted. All new files created in an encrypted directory are encrypted.

Syntax

C++

```
BOOL EncryptFileA(  
    [in] LPCSTR lpFileName  
);
```

Parameters

[in] lpFileName

The name of the file or directory to be encrypted.

The caller must have the **FILE_READ_DATA**, **FILE_WRITE_DATA**, **FILE_READ_ATTRIBUTES**, **FILE_WRITE_ATTRIBUTES**, and **SYNCHRONIZE** access rights. For more information, see [File Security and Access Rights](#).

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

The **EncryptFile** function requires exclusive access to the file being encrypted, and will fail if another process is using the file.

If the file is already encrypted, **EncryptFile** simply returns a nonzero value, which indicates success. If the file is compressed, **EncryptFile** will decompress the file before encrypting it.

If *lpFileName* specifies a read-only file, the function fails and [GetLastError](#) returns **ERROR_FILE_READ_ONLY**. If *lpFileName* specifies a directory that contains a read-only file, the functions succeeds but the directory is not encrypted.

To decrypt an encrypted file, use the [DecryptFile](#) function.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

 **Note**

The winbase.h header defines EncryptFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)

See also

[DecryptFile](#)

[File Encryption](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EncryptFileW function (winbase.h)

Article02/09/2023

Encrypts a file or directory. All data streams in a file are encrypted. All new files created in an encrypted directory are encrypted.

Syntax

C++

```
BOOL EncryptFileW(
    [in] LPCWSTR lpFileName
);
```

Parameters

[in] lpFileName

The name of the file or directory to be encrypted.

The caller must have the **FILE_READ_DATA**, **FILE_WRITE_DATA**, **FILE_READ_ATTRIBUTES**, **FILE_WRITE_ATTRIBUTES**, and **SYNCHRONIZE** access rights. For more information, see [File Security and Access Rights](#).

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

The **EncryptFile** function requires exclusive access to the file being encrypted, and will fail if another process is using the file.

If the file is already encrypted, **EncryptFile** simply returns a nonzero value, which indicates success. If the file is compressed, **EncryptFile** will decompress the file before encrypting it.

If *lpFileName* specifies a read-only file, the function fails and [GetLastError](#) returns **ERROR_FILE_READ_ONLY**. If *lpFileName* specifies a directory that contains a read-only file, the functions succeeds but the directory is not encrypted.

To decrypt an encrypted file, use the [DecryptFile](#) function.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

 **Note**

The winbase.h header defines EncryptFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)

See also

[DecryptFile](#)

[File Encryption](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EndUpdateResourceA function (winbase.h)

Article 02/09/2023

Commits or discards changes made prior to a call to [UpdateResource](#).

Syntax

C++

```
BOOL EndUpdateResourceA(
    [in] HANDLE hUpdate,
    [in] BOOL    fDiscard
);
```

Parameters

[in] hUpdate

Type: **HANDLE**

A module handle returned by the [BeginUpdateResource](#) function, and used by [UpdateResource](#), referencing the file to be updated.

[in] fDiscard

Type: **BOOL**

Indicates whether to write the resource updates to the file. If this parameter is **TRUE**, no changes are made. If it is **FALSE**, the changes are made: the resource updates will take effect.

Return value

Type: **BOOL**

Returns **TRUE** if the function succeeds; **FALSE** otherwise. If the function succeeds and *fDiscard* is **TRUE**, then no resource updates are made to the file; otherwise all successful resource updates are made to the file. To get extended error information, call [GetLastError](#).

Remarks

Before you call this function, make sure all file handles other than the one returned by [BeginUpdateResource](#) are closed.

This function can update resources within modules that contain both code and resources. There are restrictions on resource updates in LN files and .mui files, both of which contain Resource Configuration data; details of the restrictions are in the reference for the [UpdateResource](#) function.

Examples

For an example, see [Updating Resources](#).

 **Note**

The winbase.h header defines EndUpdateResource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BeginUpdateResource](#)

[Conceptual](#)

[Reference](#)

[Resources](#)

[UpdateResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EndUpdateResourceW function (winbase.h)

Article 02/09/2023

Commits or discards changes made prior to a call to [UpdateResource](#).

Syntax

C++

```
BOOL EndUpdateResourceW(
    [in] HANDLE hUpdate,
    [in] BOOL    fDiscard
);
```

Parameters

[in] hUpdate

Type: **HANDLE**

A module handle returned by the [BeginUpdateResource](#) function, and used by [UpdateResource](#), referencing the file to be updated.

[in] fDiscard

Type: **BOOL**

Indicates whether to write the resource updates to the file. If this parameter is **TRUE**, no changes are made. If it is **FALSE**, the changes are made: the resource updates will take effect.

Return value

Type: **BOOL**

Returns **TRUE** if the function succeeds; **FALSE** otherwise. If the function succeeds and *fDiscard* is **TRUE**, then no resource updates are made to the file; otherwise all successful resource updates are made to the file. To get extended error information, call [GetLastError](#).

Remarks

Before you call this function, make sure all file handles other than the one returned by [BeginUpdateResource](#) are closed.

This function can update resources within modules that contain both code and resources. There are restrictions on resource updates in LN files and .mui files, both of which contain Resource Configuration data; details of the restrictions are in the reference for the [UpdateResource](#) function.

Examples

For an example, see [Updating Resources](#).

 **Note**

The winbase.h header defines EndUpdateResource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BeginUpdateResource](#)

[Conceptual](#)

[Reference](#)

[Resources](#)

[UpdateResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnterUmsSchedulingMode function (winbase.h)

Article03/18/2022

Converts the calling thread into a user-mode scheduling (UMS) scheduler thread.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL EnterUmsSchedulingMode(
    [in] PUMS_SCHEDULER_STARTUP_INFO SchedulerStartupInfo
);
```

Parameters

[in] `SchedulerStartupInfo`

A pointer to a [UMS_SCHEDULER_STARTUP_INFO](#) structure that specifies UMS attributes for the thread, including a completion list and a [UmsSchedulerProc](#) entry point function.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

An application's UMS scheduler creates one UMS scheduler thread for each processor that will be used to run UMS threads. The scheduler typically sets the affinity of the scheduler thread for a single processor, effectively reserving the processor for the use of

that scheduler thread. For more information about thread affinity, see [Multiple Processors](#).

When a UMS scheduler thread is created, the system calls the [UmsSchedulerProc](#) entry point function specified with the [EnterUmsSchedulingMode](#) function call. The application's scheduler is responsible for finishing any application-specific initialization of the scheduler thread and selecting a UMS worker thread to run.

The application's scheduler selects a UMS worker thread to run by calling [ExecuteUmsThread](#) with the worker thread's UMS thread context. The worker thread runs until it yields control by calling [UmsThreadYield](#), blocks, or terminates. The scheduler thread is then available to run another worker thread.

A scheduler thread should continue to run until all of its worker threads reach a natural stopping point: that is, all worker threads have yielded, blocked, or terminated.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[ExecuteUmsThread](#)

[Multiple Processors](#)

[UMS_SCHEDULER_STARTUP_INFO](#)

[UmsSchedulerProc](#)

[User-Mode Scheduling](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnumResourceLanguagesA function (winbase.h)

Article 02/09/2023

Enumerates language-specific resources, of the specified type and name, associated with a binary module.

Syntax

C++

```
BOOL EnumResourceLanguagesA(
    [in] HMODULE           hModule,
    [in] LPCSTR            lpType,
    [in] LPCSTR            lpName,
    [in] ENUMRESLANGPROCA lpEnumFunc,
    [in] LONG_PTR          lParam
);
```

Parameters

[in] hModule

Type: **HMODULE**

The handle to a module to be searched. Starting with Windows Vista, if this is a [language-neutral Portable Executable](#) (LN file), then appropriate .mui files (if any exist) are included in the search. If this is a specific .mui file, only that file is searched for resources.

If this parameter is **NULL**, that is equivalent to passing in a handle to the module used to create the current process.

[in] lpType

Type: **LPCTSTR**

The type of resource for which the language is being enumerated. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is an integer value representing a predefined resource type. For a list of predefined resource types, see [Resource Types](#). For more information, see the Remarks section below.

[in] *lpName*

Type: **LPCTSTR**

The name of the resource for which the language is being enumerated. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(*ID*), where *ID* is the integer identifier of the resource. For more information, see the Remarks section below.

[in] *lpEnumFunc*

Type: **ENUMRESLANGPROC**

A pointer to the callback function to be called for each enumerated resource language. For more information, see [EnumResLangProcA](#).

[in] *lParam*

Type: **LONG_PTR**

An application-defined value passed to the callback function. This parameter can be used in error checking.

Return value

Type: **BOOL**

Returns **TRUE** if successful or **FALSE** otherwise. To get extended error information, call [GetLastError](#).

Remarks

If [IS_INTRESOURCE](#)(*lpType*) is **TRUE**, then *lpType* specifies the integer identifier of the given resource type. Otherwise, it is a pointer to a null-terminated string. If the first character of the string is a pound sign (#), then the remaining characters represent a decimal number that specifies the integer identifier of the resource type. For example, the string "#258" represents the identifier 258.

Similarly, if [IS_INTRESOURCE](#)(*lpName*) is **TRUE**, then *lpName* specifies the integer identifier of the given resource. Otherwise, it is a pointer to a null-terminated string. If the first character of the string is a pound sign (#), then the remaining characters represent a decimal number that specifies the integer identifier of the resource.

Starting with Windows Vista, the binary module is typically a [language-neutral Portable Executable](#) (LN file), and the enumeration will also include resources from the

corresponding language-specific resource files (.mui files) that contain localizable language resources.

For each resource found, **EnumResourceLanguages** calls an application-defined callback function *lpEnumFunc*, passing the language identifier (see [Language Identifiers](#)) of the language for which a resource was found, as well as the various other parameters that were passed to **EnumResourceLanguages**.

Alternately, applications can call [**EnumResourceLanguagesEx**](#), which provides more precise control of what resources are enumerated.

The **EnumResourceLanguages** function continues to enumerate resource languages until the callback function returns **FALSE** or all resource languages have been enumerated.

In Windows Vista and later, if *hModule* specifies an LN file, then the resources enumerated can reside either in the LN file or in an .mui file associated with it. If no .mui files are found, only resources from the LN file are returned. Unlike [**EnumResourceNames**](#) and [**EnumResourceTypes**](#), this search will look at multiple .mui files. The enumeration begins with .mui files in the folders associated with [**EnumUILanguages**](#). These are followed by any other .mui files whose paths conform to the scheme described at [MUI Resource Management](#). Finally, the file designated by *hModule* is also searched.

The enumeration never includes duplicates: if a resource with the same name, type, and language is contained in both the LN file and in an .mui file, the resource will only be enumerated once.

Examples

For an example, see [Creating a Resource List](#).

ⓘ Note

The winbase.h header defines **EnumResourceLanguages** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[EnumResLangProc](#)

[EnumResourceLanguagesEx](#)

[EnumResourceNames](#)

[EnumResourceTypes](#)

[Reference](#)

[Resources](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnumResourceLanguagesW function (winbase.h)

Article 02/09/2023

Enumerates language-specific resources, of the specified type and name, associated with a binary module.

Syntax

C++

```
BOOL EnumResourceLanguagesW(
    [in] HMODULE           hModule,
    [in] LPCWSTR           lpType,
    [in] LPCWSTR           lpName,
    [in] ENUMRESLANGPROCW lpEnumFunc,
    [in] LONG_PTR          lParam
);
```

Parameters

[in] hModule

Type: **HMODULE**

The handle to a module to be searched. Starting with Windows Vista, if this is a [language-neutral Portable Executable](#) (LN file), then appropriate .mui files (if any exist) are included in the search. If this is a specific .mui file, only that file is searched for resources.

If this parameter is **NULL**, that is equivalent to passing in a handle to the module used to create the current process.

[in] lpType

Type: **LPCTSTR**

The type of resource for which the language is being enumerated. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is an integer value representing a predefined resource type. For a list of predefined resource types, see [Resource Types](#). For more information, see the Remarks section below.

[in] *lpName*

Type: **LPCTSTR**

The name of the resource for which the language is being enumerated. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(*ID*), where *ID* is the integer identifier of the resource. For more information, see the Remarks section below.

[in] *lpEnumFunc*

Type: **ENUMRESLANGPROC**

A pointer to the callback function to be called for each enumerated resource language. For more information, see [EnumResLangProc](#).

[in] *lParam*

Type: **LONG_PTR**

An application-defined value passed to the callback function. This parameter can be used in error checking.

Return value

Type: **BOOL**

Returns **TRUE** if successful or **FALSE** otherwise. To get extended error information, call [GetLastError](#).

Remarks

If [IS_INTRESOURCE](#)(*lpType*) is **TRUE**, then *lpType* specifies the integer identifier of the given resource type. Otherwise, it is a pointer to a null-terminated string. If the first character of the string is a pound sign (#), then the remaining characters represent a decimal number that specifies the integer identifier of the resource type. For example, the string "#258" represents the identifier 258.

Similarly, if [IS_INTRESOURCE](#)(*lpName*) is **TRUE**, then *lpName* specifies the integer identifier of the given resource. Otherwise, it is a pointer to a null-terminated string. If the first character of the string is a pound sign (#), then the remaining characters represent a decimal number that specifies the integer identifier of the resource.

Starting with Windows Vista, the binary module is typically a [language-neutral Portable Executable](#) (LN file), and the enumeration will also include resources from the

corresponding language-specific resource files (.mui files) that contain localizable language resources.

For each resource found, **EnumResourceLanguages** calls an application-defined callback function *lpEnumFunc*, passing the language identifier (see [Language Identifiers](#)) of the language for which a resource was found, as well as the various other parameters that were passed to **EnumResourceLanguages**.

Alternately, applications can call [**EnumResourceLanguagesEx**](#), which provides more precise control of what resources are enumerated.

The **EnumResourceLanguages** function continues to enumerate resource languages until the callback function returns **FALSE** or all resource languages have been enumerated.

In Windows Vista and later, if *hModule* specifies an LN file, then the resources enumerated can reside either in the LN file or in an .mui file associated with it. If no .mui files are found, only resources from the LN file are returned. Unlike [**EnumResourceNames**](#) and [**EnumResourceTypes**](#), this search will look at multiple .mui files. The enumeration begins with .mui files in the folders associated with [**EnumUILanguages**](#). These are followed by any other .mui files whose paths conform to the scheme described at [MUI Resource Management](#). Finally, the file designated by *hModule* is also searched.

The enumeration never includes duplicates: if a resource with the same name, type, and language is contained in both the LN file and in an .mui file, the resource will only be enumerated once.

Examples

For an example, see [Creating a Resource List](#).

ⓘ Note

The winbase.h header defines **EnumResourceLanguages** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[EnumResLangProc](#)

[EnumResourceLanguagesEx](#)

[EnumResourceNames](#)

[EnumResourceTypes](#)

[Reference](#)

[Resources](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnumResourceTypesA function (winbase.h)

Article02/09/2023

Enumerates resource types within a binary module. Starting with Windows Vista, this is typically a [language-neutral Portable Executable](#) (LN file), and the enumeration also includes resources from one of the corresponding language-specific resource files (.mui files)—if one exists—that contain localizable language resources. It is also possible to use *hModule* to specify a .mui file, in which case only that file is searched for resource types.

Alternately, applications can call [EnumResourceTypesEx](#), which provides more precise control over which resource files to enumerate.

Syntax

C++

```
BOOL EnumResourceTypesA(
    [in, optional] HMODULE           hModule,
    [in]          ENUMRESTYPEPROCA lpEnumFunc,
    [in]          LONG_PTR          lParam
);
```

Parameters

[in, optional] *hModule*

Type: **HMODULE**

A handle to a module to be searched. This handle must be obtained through [LoadLibrary](#) or [LoadLibraryEx](#).

See Remarks for more information.

If this parameter is **NULL**, that is equivalent to passing in a handle to the module used to create the current process.

[in] *lpEnumFunc*

Type: **ENUMRESTYPEPROC**

A pointer to the callback function to be called for each enumerated resource type. For more information, see the [EnumResTypeProc](#) function.

[in] lParam

Type: **LONG_PTR**

An application-defined value passed to the callback function.

Return value

Type: **BOOL**

Returns **TRUE** if successful; otherwise, **FALSE**. To get extended error information, call [GetLastError](#).

Remarks

For each resource type found, **EnumResourceTypes** calls an application-defined callback function *lpEnumFunc*, passing each resource type it finds, as well as the various other parameters that were passed to **EnumResourceTypes**.

EnumResourceTypes continues to enumerate resource types until the callback function returns **FALSE** or all resource types have been enumerated.

Starting with Windows Vista, if *hModule* specifies an LN file, then the types enumerated correspond to resources that reside in the LN file and in the .mui file associated with it. If no .mui files are found, only types from the LN file are returned. The order in which .mui files are searched is the usual Resource Loader search order; see [User Interface Language Management](#) for details. After one appropriate .mui file is found, the search does not continue further to other .mui files associated with the LN file, because all .mui files that correspond to a single LN file have the same set of resource types.

The enumeration never includes duplicates: if a given resource type is contained in both the LN file and in an .mui file, the type is enumerated only once.

Examples

For an example, see [Creating a Resource List](#).

ⓘ Note

The winbase.h header defines EnumResourceTypes as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Conceptual

[EnumResTypeProc](#)

[EnumResourceLanguages](#)

[EnumResourceNames](#)

[EnumResourceTypesEx](#)

Reference

[Resources](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EnumResourceTypesW function (winbase.h)

Article02/09/2023

Enumerates resource types within a binary module. Starting with Windows Vista, this is typically a [language-neutral Portable Executable](#) (LN file), and the enumeration also includes resources from one of the corresponding language-specific resource files (.mui files)—if one exists—that contain localizable language resources. It is also possible to use *hModule* to specify a .mui file, in which case only that file is searched for resource types.

Alternately, applications can call [EnumResourceTypesEx](#), which provides more precise control over which resource files to enumerate.

Syntax

C++

```
BOOL EnumResourceTypesW(
    [in, optional] HMODULE           hModule,
    [in]          ENUMRESTYPEPROCW lpEnumFunc,
    [in]          LONG_PTR          lParam
);
```

Parameters

[in, optional] *hModule*

Type: **HMODULE**

A handle to a module to be searched. This handle must be obtained through [LoadLibrary](#) or [LoadLibraryEx](#).

See Remarks for more information.

If this parameter is **NULL**, that is equivalent to passing in a handle to the module used to create the current process.

[in] *lpEnumFunc*

Type: **ENUMRESTYPEPROC**

A pointer to the callback function to be called for each enumerated resource type. For more information, see the [EnumResTypeProc](#) function.

[in] lParam

Type: **LONG_PTR**

An application-defined value passed to the callback function.

Return value

Type: **BOOL**

Returns **TRUE** if successful; otherwise, **FALSE**. To get extended error information, call [GetLastError](#).

Remarks

For each resource type found, **EnumResourceTypes** calls an application-defined callback function *lpEnumFunc*, passing each resource type it finds, as well as the various other parameters that were passed to **EnumResourceTypes**.

EnumResourceTypes continues to enumerate resource types until the callback function returns **FALSE** or all resource types have been enumerated.

Starting with Windows Vista, if *hModule* specifies an LN file, then the types enumerated correspond to resources that reside in the LN file and in the .mui file associated with it. If no .mui files are found, only types from the LN file are returned. The order in which .mui files are searched is the usual Resource Loader search order; see [User Interface Language Management](#) for details. After one appropriate .mui file is found, the search does not continue further to other .mui files associated with the LN file, because all .mui files that correspond to a single LN file have the same set of resource types.

The enumeration never includes duplicates: if a given resource type is contained in both the LN file and in an .mui file, the type is enumerated only once.

Examples

For an example, see [Creating a Resource List](#).

ⓘ Note

The winbase.h header defines EnumResourceTypes as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Conceptual

[EnumResTypeProc](#)

[EnumResourceLanguages](#)

[EnumResourceNames](#)

[EnumResourceTypesEx](#)

Reference

[Resources](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EraseTape function (winbase.h)

Article 10/13/2021

The **EraseTape** function erases all or part of a tape.

Syntax

C++

```
DWORD EraseTape(
    [in] HANDLE hDevice,
    [in] DWORD dwEraseType,
    [in] BOOL bImmediate
);
```

Parameters

[in] `hDevice`

Handle to the device where the tape is to be erased. This handle is created by using the [CreateFile](#) function.

[in] `dwEraseType`

Erasing technique. This parameter can be one of the following values.

Value	Meaning
<code>TAPE_ERASE_LONG</code> 1L	Erases the tape from the current position to the end of the current partition.
<code>TAPE_ERASE_SHORT</code> 0L	Writes an erase gap or end-of-data marker at the current position.

[in] `bImmediate`

If this parameter is **TRUE**, the function returns immediately; if it is **FALSE**, the function does not return until the erase operation has been completed.

Return value

If the function succeeds, the return value is `NO_ERROR`.

If the function fails, it can return one of the following error codes.

Error code	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.
ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT 19L	The media is write protected.

Remarks

Some tape devices do not support certain tape operations. To determine your tape device's capabilities, see your tape device documentation and use the [GetTapeParameters](#) function.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[GetTapeParameters](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EscapeCommFunction function (winbase.h)

Article 10/13/2021

Directs the specified communications device to perform an extended function.

Syntax

C++

```
BOOL EscapeCommFunction(
    [in] HANDLE hFile,
    [in] DWORD dwFunc
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in] dwFunc

The extended function to be performed. This parameter can be one of the following values.

Value	Meaning
CLRBREAK 9	Restores character transmission and places the transmission line in a nonbreak state. The CLRBREAK extended function code is identical to the ClearCommBreak function.
CLRDTR 6	Clears the DTR (data-terminal-ready) signal.
CLRRTS 4	Clears the RTS (request-to-send) signal.
SETBREAK 8	Suspends character transmission and places the transmission line in a break state until the ClearCommBreak function is called (or EscapeCommFunction is called with the CLRBREAK

extended function code). The SETBREAK extended function code is identical to the [SetCommBreak](#) function. Note that this extended function does not flush data that has not been transmitted.

SETDTR	Sends the DTR (data-terminal-ready) signal.
5	
SETRTS	Sends the RTS (request-to-send) signal.
3	
SETXOFF	Causes transmission to act as if an XOFF character has been received.
1	
SETXON	Causes transmission to act as if an XON character has been received.
2	

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](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ClearCommBreak](#)

[Communications Functions](#)

[CreateFile](#)

[SetCommBreak](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

EVENTLOG_FULL_INFORMATION structure (winbase.h)

Article04/02/2021

Indicates whether the event log is full.

Syntax

C++

```
typedef struct _EVENTLOG_FULL_INFORMATION {
    DWORD dwFull;
} EVENTLOG_FULL_INFORMATION, *LPEVENTLOG_FULL_INFORMATION;
```

Members

`dwFull`

Indicates whether the event log is full. If the log is full, this member is **TRUE**. Otherwise, it is **FALSE**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[GetEventLogInformation](#)

Feedback



Was this page helpful? [Yes](#) | [No](#)

Get help at Microsoft Q&A

ExecuteUmsThread function (winbase.h)

Article 03/18/2022

Runs the specified UMS worker thread.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL ExecuteUmsThread(  
    [in, out] PUMS_CONTEXT UmsThread  
);
```

Parameters

`[in, out] UmsThread`

A pointer to the UMS thread context of the worker thread to run.

Return value

If the function succeeds, it does not return a value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include the following.

Return code	Description
<code>ERROR_RETRY</code>	The specified UMS worker thread is temporarily locked by the system. The caller can retry the operation.
<code>ERROR_NOT_SUPPORTED</code>	UMS is not supported.

Remarks

The **ExecuteUmsThread** function loads the state of the specified UMS worker thread over the state of the calling UMS scheduler thread so that the worker thread can run. The worker thread runs until it yields by calling the [UmsThreadYield](#) function, blocks, or terminates.

When a worker thread yields or blocks, the system calls the scheduler thread's [UmsSchedulerProc](#) entry point function. When a previously blocked worker thread becomes unblocked, the system queues the worker thread to the completion list specified with the [UpdateProcThreadAttribute](#) function when the worker thread was created.

The **ExecuteUmsThread** function does not return unless an error occurs. If the function returns **ERROR_RETRY**, the error is transitory and the operation can be retried.

If the function returns an error other than **ERROR_RETRY**, the application's scheduler should check whether the thread is suspended or terminated by calling [QueryUmsThreadInformation](#) with **UmsThreadIsSuspended** or **UmsThreadIsTerminated**, respectively. Other possible errors include calling the function on a thread that is not a UMS scheduler thread, passing an invalid UMS worker thread context, or specifying a worker thread that is already executing on another scheduler thread.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[UmsSchedulerProc](#)

[UmsThreadYield](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

FatalExit function (winbase.h)

Article10/13/2021

Transfers execution control to the debugger. The behavior of the debugger thereafter is specific to the type of debugger used.

Syntax

C++

```
__analysis_noreturn VOID FatalExit(
    [in] int ExitCode
);
```

Parameters

[in] ExitCode

The error code associated with the exit.

Return value

This function does not return a value.

Remarks

An application should only use **FatalExit** for debugging purposes. It should not call the function in a retail version of the application because doing so will terminate the application.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communicating with the Debugger](#)

[Debugging Functions](#)

[FatalAppExit](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_ALIGNMENT_INFO structure (winbase.h)

Article04/02/2021

Contains alignment information for a file. This structure is returned from the [GetFileInformationByHandleEx](#) function when **FileAlignmentInfo** is passed in the *FileInformationClass* parameter.

Syntax

C++

```
typedef struct _FILE_ALIGNMENT_INFO {
    ULONG AlignmentRequirement;
} FILE_ALIGNMENT_INFO, *PFILE_ALIGNMENT_INFO;
```

Members

`AlignmentRequirement`

Minimum alignment requirement, in bytes.

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[File Management Structures](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_ALLOCATION_INFO structure (winbase.h)

Article04/02/2021

Contains the total number of bytes that should be allocated for a file. This structure is used when calling the [SetFileInformationByHandle](#) function.

Syntax

C++

```
typedef struct _FILE_ALLOCATION_INFO {
    LARGE_INTEGER AllocationSize;
} FILE_ALLOCATION_INFO, *PFILE_ALLOCATION_INFO;
```

Members

`AllocationSize`

The new file allocation size, in bytes. This value is typically a multiple of the sector or cluster size for the underlying physical device.

Remarks

The end-of-file (EOF) position for a file must always be less than or equal to the file allocation size. If the allocation size is set to a value that is less than EOF, the EOF position is automatically adjusted to match the file allocation size.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

FILE_ATTRIBUTE_TAG_INFO structure (winbase.h)

Article04/02/2021

Receives the requested file attribute information. Used for any handles. Use only when calling [GetFileInformationByHandleEx](#).

Syntax

C++

```
typedef struct _FILE_ATTRIBUTE_TAG_INFO {
    DWORD FileAttributes;
    DWORD ReparseTag;
} FILE_ATTRIBUTE_TAG_INFO, *PFILE_ATTRIBUTE_TAG_INFO;
```

Members

FileAttributes

The file attribute information.

ReparseTag

The reparse tag.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

FILE_INFO_BY_HANDLE_CLASS

File Attribute Constants

GetFileInformationByHandleEx

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_BASIC_INFO structure (winbase.h)

Article 04/02/2021

Contains the basic information for a file. Used for file handles.

Syntax

C++

```
typedef struct _FILE_BASIC_INFO {
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    DWORD       FileAttributes;
} FILE_BASIC_INFO, *PFILE_BASIC_INFO;
```

Members

`CreationTime`

The time the file was created in [FILETIME](#) format, which is a 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601 (UTC).

`LastAccessTime`

The time the file was last accessed in [FILETIME](#) format.

`LastWriteTime`

The time the file was last written to in [FILETIME](#) format.

`ChangeTime`

The time the file was changed in [FILETIME](#) format.

`FileAttributes`

The file attributes. For a list of attributes, see [File Attribute Constants](#). If this is set to 0 in a `FILE_BASIC_INFO` structure passed to [SetFileInformationByHandle](#) then none of the attributes are changed.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[GetFileAttributes](#)

[GetFileInformationByHandleEx](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_COMPRESSION_INFO structure (winbase.h)

Article09/01/2022

Receives file compression information. Used for any handles. Use only when calling [GetFileInformationByHandleEx](#).

Syntax

C++

```
typedef struct _FILE_COMPRESSION_INFO {
    LARGE_INTEGER CompressedFileSize;
    WORD          CompressionFormat;
    UCHAR         CompressionUnitShift;
    UCHAR         ChunkShift;
    UCHAR         ClusterShift;
    UCHAR         Reserved[3];
} FILE_COMPRESSION_INFO, *PFILE_COMPRESSION_INFO;
```

Members

CompressedFileSize

The file size of the compressed file.

CompressionFormat

The compression format that is used to compress the file.

CompressionUnitShift

The factor that the compression uses.

ChunkShift

The number of chunks that are shifted by compression.

ClusterShift

The number of clusters that are shifted by compression.

Reserved[3]

Reserved.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

FILE_DISPOSITION_INFO structure (winbase.h)

Article04/02/2021

Indicates whether a file should be deleted. Used for any handles. Use only when calling [SetFileInformationByHandle](#).

Syntax

C++

```
typedef struct _FILE_DISPOSITION_INFO {
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFO, *PFILE_DISPOSITION_INFO;
```

Members

`DeleteFile`

Indicates whether the file should be deleted. Set to TRUE to delete the file. This member has no effect if the handle was opened with `FILE_FLAG_DELETE_ON_CLOSE`.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_END_OF_FILE_INFO structure (winbase.h)

Article04/02/2021

Contains the specified value to which the end of the file should be set. Used for file handles. Use only when calling [SetFileInformationByHandle](#).

Syntax

C++

```
typedef struct _FILE_END_OF_FILE_INFO {
    LARGE_INTEGER EndOfFile;
} FILE_END_OF_FILE_INFO, *PFILE_END_OF_FILE_INFO;
```

Members

EndOfFile

The specified value for the new end of the file.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_FULL_DIR_INFO structure (winbase.h)

Article09/01/2022

Contains directory information for a file. This structure is returned from the [GetFileInformationByHandleEx](#) function when **FileFullDirectoryInfo** or **FileFullDirectoryRestartInfo** is passed in the *FileInformationClass* parameter.

Syntax

C++

```
typedef struct _FILE_FULL_DIR_INFO {
    ULONG          NextEntryOffset;
    ULONG          FileIndex;
    LARGE_INTEGER  CreationTime;
    LARGE_INTEGER  LastAccessTime;
    LARGE_INTEGER  LastWriteTime;
    LARGE_INTEGER  ChangeTime;
    LARGE_INTEGER  EndOfFile;
    LARGE_INTEGER  AllocationSize;
    ULONG          FileAttributes;
    ULONG          FileNameLength;
    ULONG          EaSize;
    WCHAR          FileName[1];
} FILE_FULL_DIR_INFO, *PFILE_FULL_DIR_INFO;
```

Members

NextEntryOffset

The offset for the next **FILE_FULL_DIR_INFO** structure that is returned. Contains zero (0) if no other entries follow this one.

FileIndex

The byte offset of the file within the parent directory. This member is undefined for file systems, such as NTFS, in which the position of a file within the parent directory is not fixed and can be changed at any time to maintain sort order.

CreationTime

The time that the file was created.

`LastAccessTime`

The time that the file was last accessed.

`LastWriteTime`

The time that the file was last written to.

`ChangeTime`

The time that the file was last changed.

`EndOfFile`

The absolute new end-of-file position as a byte offset from the start of the file to the end of the default data stream of the file. Because this value is zero-based, it actually refers to the first free byte in the file. In other words, `EndOfFile` is the offset to the byte that immediately follows the last valid byte in the file.

`AllocationSize`

The number of bytes that are allocated for the file. This value is usually a multiple of the sector or cluster size of the underlying physical device.

`FileAttributes`

The file attributes. This member can be any valid combination of the following attributes:

FILE_ATTRIBUTE_ARCHIVE (0x00000020)

FILE_ATTRIBUTE_COMPRESSED (0x00000800)

FILE_ATTRIBUTE_DIRECTORY (0x00000010)

FILE_ATTRIBUTE_HIDDEN (0x00000002)

FILE_ATTRIBUTE_NORMAL (0x00000080)

FILE_ATTRIBUTE_READONLY (0x00000001)

FILE_ATTRIBUTE_SYSTEM (0x00000004)

FILE_ATTRIBUTE_TEMPORARY (0x00000100)

FileNameLength

The length of the file name.

EaSize

The size of the extended attributes for the file.

FileName[1]

The first character of the file name string. This is followed in memory by the remainder of the string.

Remarks

The **FILE_FULL_DIR_INFO** structure is a subset of the information in the **FILE_ID_BOTH_DIR_INFO** structure. If the additional information is not needed then the operation will be faster as it comes from the directory entry; **FILE_ID_BOTH_DIR_INFO** contains information from both the directory entry and the Master File Table (MFT).

No specific access rights are required to query this information.

All dates and times are in absolute system-time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601.

This **FILE_FULL_DIR_INFO** structure must be aligned on a **LONGLONG** (8-byte) boundary. If a buffer contains two or more of these structures, the **NextEntryOffset** value in each entry, except the last, falls on an 8-byte boundary.

To compile an application that uses this structure, define the **_WIN32_WINNT** macro as 0x0600 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]

Header

winbase.h (include Windows.h)

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[File Management Structures](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_ID_BOTH_DIR_INFO structure (winbase.h)

Article09/01/2022

Contains information about files in the specified directory. Used for directory handles. Use only when calling [GetFileInformationByHandleEx](#). The number of files that are returned for each call to [GetFileInformationByHandleEx](#) depends on the size of the buffer that is passed to the function. Any subsequent calls to [GetFileInformationByHandleEx](#) on the same handle will resume the enumeration operation after the last file is returned.

Syntax

C++

```
typedef struct _FILE_ID_BOTH_DIR_INFO {
    DWORD          NextEntryOffset;
    DWORD          FileIndex;
    LARGE_INTEGER  CreationTime;
    LARGE_INTEGER  LastAccessTime;
    LARGE_INTEGER  LastWriteTime;
    LARGE_INTEGER  ChangeTime;
    LARGE_INTEGER  EndOfFile;
    LARGE_INTEGER  AllocationSize;
    DWORD          FileAttributes;
    DWORD          FileNameLength;
    DWORD          EaSize;
    CCHAR          ShortNameLength;
    WCHAR          ShortName[12];
    LARGE_INTEGER  FileId;
    WCHAR          FileName[1];
} FILE_ID_BOTH_DIR_INFO, *PFILE_ID_BOTH_DIR_INFO;
```

Members

NextEntryOffset

The offset for the next **FILE_ID_BOTH_DIR_INFO** structure that is returned. Contains zero (0) if no other entries follow this one.

FileIndex

The byte offset of the file within the parent directory. This member is undefined for file systems, such as NTFS, in which the position of a file within the parent directory is not fixed and can be changed at any time to maintain sort order.

`CreationTime`

The time that the file was created.

`LastAccessTime`

The time that the file was last accessed.

`LastWriteTime`

The time that the file was last written to.

`ChangeTime`

The time that the file was last changed.

`EndOfFile`

The absolute new end-of-file position as a byte offset from the start of the file to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file. In other words, `EndOfFile` is the offset to the byte that immediately follows the last valid byte in the file.

`AllocationSize`

The number of bytes that are allocated for the file. This value is usually a multiple of the sector or cluster size of the underlying physical device.

`FileAttributes`

The file attributes. This member can be any valid combination of the following attributes:

FILE_ATTRIBUTE_ARCHIVE (0x00000020)

FILE_ATTRIBUTE_COMPRESSED (0x00000800)

FILE_ATTRIBUTE_DIRECTORY (0x00000010)

FILE_ATTRIBUTE_HIDDEN (0x00000002)

FILE_ATTRIBUTE_NORMAL (0x00000080)

FILE_ATTRIBUTE_READONLY (0x00000001)

FILE_ATTRIBUTE_SYSTEM (0x00000004)

FILE_ATTRIBUTE_TEMPORARY (0x00000100)

`FileNameLength`

The length of the file name.

`EaSize`

The size of the extended attributes for the file.

`ShortNameLength`

The length of **ShortName**.

`ShortName[12]`

The short 8.3 file naming convention (for example, "FILENAME.TXT") name of the file.

`FileId`

The file ID.

`FileName[1]`

The first character of the file name string. This is followed in memory by the remainder of the string.

Remarks

No specific access rights are required to query this information.

File reference numbers, also called file IDs, are guaranteed to be unique only within a static file system. They are not guaranteed to be unique over time, because file systems are free to reuse them. Nor are they guaranteed to remain constant. For example, the FAT file system generates the file reference number for a file from the byte offset of the

file's directory entry record (DIRENT) on the disk. Defragmentation can change this byte offset. Thus a FAT file reference number can change over time.

All dates and times are in absolute system-time format. Absolute system time is the number of 100-nanosecond intervals since the start of the year 1601.

This **FILE_ID_BOTH_DIR_INFO** structure must be aligned on a **DWORDLONG** (8-byte) boundary. If a buffer contains two or more of these structures, the **NextEntryOffset** value in each entry, except the last, falls on an 8-byte boundary.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[File Attribute Constants](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_ID_DESCRIPTOR structure (winbase.h)

Article04/02/2021

Specifies the type of ID that is being used.

Syntax

C++

```
typedef struct FILE_ID_DESCRIPTOR {
    DWORD          dwSize;
    FILE_ID_TYPE   Type;
    union {
        LARGE_INTEGER FileId;
        GUID          ObjectId;
        FILE_ID_128   ExtendedFileId;
    } DUMMYUNIONNAME;
} FILE_ID_DESCRIPTOR, *LPFILE_ID_DESCRIPTOR;
```

Members

dwSize

The size of this FILE_ID_DESCRIPTOR structure.

Type

The discriminator for the union indicating the type of identifier that is being passed.

Value	Meaning
FileType 0	Use the FileId member of the union.
ObjectIdType 1	Use the ObjectId member of the union.
ExtendedFileType 2	Use the ExtendedFileId member of the union. Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7 and Windows Server 2008 R2: This value is not supported before Windows 8 and Windows Server 2012.

DUMMYUNIONNAME

DUMMYUNIONNAME.FileId

The ID of the file to open.

DUMMYUNIONNAME.ObjectId

The ID of the object to open.

DUMMYUNIONNAME.ExtendedFileId

A [FILE_ID_128](#) structure containing the 128-bit file ID of the file. This is used on ReFS file systems.

Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7 and Windows Server 2008 R2: This member is not supported before Windows 8 and Windows Server 2012.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_ID_128](#)

[FILE_ID_TYPE](#)

[File Management Structures](#)

[OpenFileById](#)

Feedback



Was this page helpful? [!\[\]\(6cce504e6ca40d5867d027bcf3033ad2_img.jpg\) Yes](#) [!\[\]\(f8d7444c939582f34c08a51b3c16b574_img.jpg\) No](#)

Get help at Microsoft Q&A

FILE_ID_EXTD_DIR_INFO structure (winbase.h)

Article09/01/2022

Contains identification information for a file. This structure is returned from the [GetFileInformationByHandleEx](#) function when **FileIdExtdDirectoryInfo** (0x13) or **FileIdExtdDirectoryRestartInfo** (0x14) is passed in the *FileInformationClass* parameter.

Syntax

C++

```
typedef struct _FILE_ID_EXTD_DIR_INFO {
    ULONG          NextEntryOffset;
    ULONG          FileIndex;
    LARGE_INTEGER  CreationTime;
    LARGE_INTEGER  LastAccessTime;
    LARGE_INTEGER  LastWriteTime;
    LARGE_INTEGER  ChangeTime;
    LARGE_INTEGER  EndOfFile;
    LARGE_INTEGER  AllocationSize;
    ULONG          FileAttributes;
    ULONG          FileNameLength;
    ULONG          EaSize;
    ULONG          ReparsePointTag;
    FILE_ID_128    FileId;
    WCHAR          FileName[1];
} FILE_ID_EXTD_DIR_INFO, *PFILE_ID_EXTD_DIR_INFO;
```

Members

NextEntryOffset

The offset for the next **FILE_ID_EXTD_DIR_INFO** structure that is returned. Contains zero (0) if no other entries follow this one.

FileIndex

The byte offset of the file within the parent directory. This member is undefined for file systems, such as NTFS, in which the position of a file within the parent directory is not fixed and can be changed at any time to maintain sort order.

CreationTime

The time that the file was created.

LastAccessTime

The time that the file was last accessed.

LastWriteTime

The time that the file was last written to.

ChangeTime

The time that the file was last changed.

EndOfFile

The absolute new end-of-file position as a byte offset from the start of the file to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file. In other words, **EndOfFile** is the offset to the byte that immediately follows the last valid byte in the file.

AllocationSize

The number of bytes that are allocated for the file. This value is usually a multiple of the sector or cluster size of the underlying physical device.

FileAttributes

The file attributes. This member can be any valid combination of the following attributes:

Value	Meaning
FILE_ATTRIBUTE_ARCHIVE 32 (0x20)	A file or directory that is an archive file or directory. Applications typically use this attribute to mark files for backup or removal .
FILE_ATTRIBUTE_COMPRESSED 2048 (0x800)	A file or directory that is compressed. For a file, all of the data in the file is compressed. For a directory, compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DEVICE 64 (0x40)	This value is reserved for system use.
FILE_ATTRIBUTE_DIRECTORY 16 (0x10)	The handle that identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	A file or directory that is encrypted. For a file, all

16384 (0x4000)	data streams in the file are encrypted. For a directory, encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN 2 (0x2)	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL 128 (0x80)	A file that does not have other attributes set. This attribute is valid only when used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED 8192 (0x2000)	The file or directory is not to be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE 4096 (0x1000)	The data of a file is not available immediately. This attribute indicates that the file data is physically moved to offline storage. This attribute is used by Remote Storage, which is the hierarchical storage management software. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY 1 (0x1)	A file that is read-only. Applications can read the file, but cannot write to it or delete it. This attribute is not honored on directories. For more information, see You cannot view or change the Read-only or the System attributes of folders in Windows Server 2003, in Windows XP, in Windows Vista or in Windows 7 .
FILE_ATTRIBUTE_REPARSE_POINT 1024 (0x400)	A file or directory that has an associated reparse point, or a file that is a symbolic link.
FILE_ATTRIBUTE_SPARSE_FILE 512 (0x200)	A file that is a sparse file.
FILE_ATTRIBUTE_SYSTEM 4 (0x4)	A file or directory that the operating system uses a part of, or uses exclusively.
FILE_ATTRIBUTE_TEMPORARY 256 (0x100)	A file that is being used for temporary storage. File systems avoid writing data back to mass storage if sufficient cache memory is available, because typically, an application deletes a temporary file after the handle is closed. In that scenario, the system can entirely avoid writing the data. Otherwise, the data is written after the handle is closed.
FILE_ATTRIBUTE_VIRTUAL 65536 (0x10000)	This value is reserved for system use.

FileNameLength

The length of the file name.

EaSize

The size of the extended attributes for the file.

ReparsePointTag

If the **FileAttributes** member includes the **FILE_ATTRIBUTE_REPARSE_POINT** attribute, this member specifies the reparse point tag.

Otherwise, this value is undefined and should not be used.

For more information see [Reparse Point Tags](#).

IO_REPARSE_TAG_CSV (0x80000009)

IO_REPARSE_TAG_DEDUP (0x80000013)

IO_REPARSE_TAG_DFS (0x8000000A)

IO_REPARSE_TAG_DFSR (0x80000012)

IO_REPARSE_TAG_HSM (0xC0000004)

IO_REPARSE_TAG_HSM2 (0x80000006)

IO_REPARSE_TAG_MOUNT_POINT (0xA0000003)

IO_REPARSE_TAG_NFS (0x80000014)

IO_REPARSE_TAG_SIS (0x80000007)

IO_REPARSE_TAG_SYMLINK (0xA000000C)

IO_REPARSE_TAG_WIM (0x80000008)

FileId

The file ID.

FileName[1]

The first character of the file name string. This is followed in memory by the remainder of the string.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[FILE_ID_128](#)

[FILE_INFO_BY_HANDLE_CLASS](#)

[File Management Structures](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_ID_INFO structure (winbase.h)

Article07/27/2022

Contains identification information for a file. This structure is returned from the [GetFileInformationByHandleEx](#) function when **FileInfo** is passed in the *FileInformationClass* parameter.

Syntax

C++

```
typedef struct _FILE_ID_INFO {
    ULONGLONG    VolumeSerialNumber;
    FILE_ID_128  FileId;
} FILE_ID_INFO, *PFILE_ID_INFO;
```

Members

`VolumeSerialNumber`

The serial number of the volume that contains a file.

`FileId`

The 128-bit file identifier for the file. The file identifier and the volume serial number uniquely identify a file on a single computer. To determine whether two open handles represent the same file, combine the identifier and the volume serial number for each file and compare them.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

FILE_ID_128

FILE_INFO_BY_HANDLE_CLASS

File Management Structures

GetFileInformationByHandleEx

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

FILE_ID_TYPE enumeration (winbase.h)

Article06/02/2021

Discriminator for the union in the [FILE_ID_DESCRIPTOR](#) structure.

Syntax

C++

```
typedef enum _FILE_ID_TYPE {
    FileIdType,
    ObjectIdType,
    ExtendedFileDialogType,
    MaximumFileDialogType
} FILE_ID_TYPE, *PFILE_ID_TYPE;
```

Constants

`FileIdType`

Use the **FileId** member of the union.

`ObjectIdType`

Use the **ObjectId** member of the union.

`ExtendedFileDialogType`

Use the **ExtendedFileDialog** member of the union.

Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7 and Windows Server 2008 R2: This value is not supported before Windows 8 and Windows Server 2012.

`MaximumFileDialogType`

This value is used for comparison only. All valid values are less than this value.

Requirements

Minimum supported client

Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_ID_DESCRIPTOR](#)

[File Management Enumerations](#)

[OpenFileByIid](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_IO_PRIORITY_HINT_INFO structure (winbase.h)

Article04/02/2021

Specifies the priority hint for a file I/O operation.

Syntax

C++

```
typedef struct _FILE_IO_PRIORITY_HINT_INFO {
    PRIORITY_HINT PriorityHint;
} FILE_IO_PRIORITY_HINT_INFO, *PFILE_IO_PRIORITY_HINT_INFO;
```

Members

PriorityHint

The priority hint. This member is a value from the [PRIORITY_HINT](#) enumeration.

Remarks

The [SetFileInformationByHandle](#) function can be used with this structure to associate a priority hint with I/O operations on a file-handle basis. In addition to the idle priority (very low), this function allows normal priority and low priority. Whether these priorities are supported and honored by the underlying drivers depends on their implementation (which is why they are called hints). For more information, see the [I/O Prioritization in Windows Vista](#) white paper on the Windows Hardware Developer Central (WHDC) website.

This structure must be aligned on a **LONGLONG** (8-byte) boundary.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Header

winbase.h (include Windows.h)

See also

[PRIORITY_HINT](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_NAME_INFO structure (winbase.h)

Article 06/09/2023

Receives the file name. Used for any handles. Use only when calling [GetFileInformationByHandleEx](#).

Syntax

C++

```
typedef struct _FILE_NAME_INFO {
    DWORD FileNameLength;
    WCHAR FileName[1];
} FILE_NAME_INFO, *PFILE_NAME_INFO;
```

Members

`FileNameLength`

The size of the `FileName` string, in bytes.

`FileName[1]`

The file name that is returned.

Remarks

If the call to `GetFileInformationByHandleEx` fails with `ERROR_MORE_DATA` because there was not enough buffer space for the full length of the `FileName` then the `FileNameLength` field will contain the required length of the `FileName` in bytes.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)

Redistributable

Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_REMOTE_PROTOCOL_INFO structure (winbase.h)

Article09/01/2022

Contains file remote protocol information. This structure is returned from the [GetFileInformationByHandleEx](#) function when **FileRemoteProtocolInfo** is passed in the *FileInformationClass* parameter.

Syntax

C++

```
typedef struct _FILE_REMOTE_PROTOCOL_INFO {
    USHORT StructureVersion;
    USHORT StructureSize;
    ULONG Protocol;
    USHORT ProtocolMajorVersion;
    USHORT ProtocolMinorVersion;
    USHORT ProtocolRevision;
    USHORT Reserved;
    ULONG Flags;
    struct {
        ULONG Reserved[8];
    } GenericReserved;
    struct {
        ULONG Reserved[16];
    } ProtocolSpecificReserved;
    union {
        struct {
            struct {
                ULONG Capabilities;
            } Server;
            struct {
                ULONG Capabilities;
                ULONG ShareFlags;
                ULONG CachingFlags;
            } Share;
        } Smb2;
        ULONG Reserved[16];
    } ProtocolSpecific;
} FILE_REMOTE_PROTOCOL_INFO, *PFILE_REMOTE_PROTOCOL_INFO;
```

Members

StructureVersion

Version of this structure. This member should be set to 2 if the communication is between computers running Windows 8, Windows Server 2012, or later and 1 otherwise.

StructureSize

Size of this structure. This member should be set to `sizeof(FILE_REMOTE_PROTOCOL_INFO)`.

Protocol

Remote protocol (`WNNC_NET_*`) defined in Wnnc.h or Ntifs.h.

WNNC_NET_MSNET (0x00010000)

WNNC_NET_SMB (0x00020000)

WNNC_NET_LANMAN (0x00020000)

WNNC_NET_NETWARE (0x00030000)

WNNC_NET_VINES (0x00040000)

WNNC_NET_10NET (0x00050000)

WNNC_NET_LOCUS (0x00060000)

WNNC_NET_SUN_PC_NFS (0x00070000)

WNNC_NET_LANSTEP (0x00080000)

WNNC_NET_9TILES (0x00090000)

WNNC_NET_LANTASTIC (0x000A0000)

WNNC_NET_AS400 (0x000B0000)

WNNC_NET_FTP_NFS (0x000C0000)

WNNC_NET_PATHWORKS (0x000D0000)

WNNC_NET_LIFENET (0x000E0000)

WNNC_NET_POWERLAN (0x000F0000)

WNNC_NET_BWNFS (0x00100000)

WNNC_NET_COGENT (0x00110000)

WNNC_NET_FARALLON (0x00120000)

WNNC_NET_APPLETALK (0x00130000)

WNNC_NET_INTERGRAPH (0x00140000)

WNNC_NET_SYMFONET (0x00150000)

WNNC_NET_CLEARCASE (0x00160000)

WNNC_NET_FRONTIER (0x00170000)

WNNC_NET_BMC (0x00180000)

WNNC_NET_DCE (0x00190000)

WNNC_NET_AVID (0x001A0000)

WNNC_NET_DOCUSPACE (0x001B0000)

WNNC_NET_MANGOSOFT (0x001C0000)

WNNC_NET_SERNET (0x001D0000)

WNNC_NET_RIVERFRONT1 (0x001E0000)

WNNC_NET_RIVERFRONT2 (0x001F0000)

WNNC_NET_DECORB (0x00200000)

WNNC_NET_PROTSTOR (0x00210000)

WNNC_NET_FJ_REDIR (0x00220000)

WNNC_NET_DISTINCT (0x00230000)

WNNC_NET_TWINS (0x00240000)

WNNC_NET_RDR2SAMPLE (0x00250000)

WNNC_NET_CSC (0x00260000)

WNNC_NET_3IN1 (0x00270000)

WNNC_NET_EXTENDNET (0x00290000)

WNNC_NET_STAC (0x002A0000)

WNNC_NET_FOXBAT (0x002B0000)

WNNC_NET_YAHOO (0x002C0000)

WNNC_NET_EXIFS (0x002D0000)

WNNC_NET_DAV (0x002E0000)

WNNC_NET_KNOWARE (0x002F0000)

WNNC_NET_OBJECT_DIRE (0x00300000)

WNNC_NET_MASFAX (0x00310000)

WNNC_NET_HOB_NFS (0x00320000)

WNNC_NET_SHIVA (0x00330000)

WNNC_NET_IBMAL (0x00340000)

WNNC_NET_LOCK (0x00350000)

WNNC_NET_TERMSRV (0x00360000)

WNNC_NET_SRT (0x00370000)

WNNC_NET_QUINCY (0x00380000)

WNNC_NET_OPENAFS (0x00390000)

WNNC_NET_AVID1 (0x003A0000)

WNNC_NET_DFS (0x003B0000)

WNNC_NET_KWNP (0x003C0000)

WNNC_NET_ZENWORKS (0x003D0000)

WNNC_NET_DRIVEONWEB (0x003E0000)

WNNC_NET_VMWARE (0x003F0000)

WNNC_NET_RSFX (0x00400000)

WNNC_NET_MFILES (0x00410000)

WNNC_NET_MS_NFS (0x00420000)

WNNC_NET_GOOGLE (0x00430000)

ProtocolMajorVersion

Major version of the remote protocol.

ProtocolMinorVersion

Minor version of the remote protocol.

ProtocolRevision

Revision of the remote protocol.

Reserved

Should be set to zero. Do not use this member.

Flags

Remote protocol information. This member can be set to zero or more of the following flags.

Value	Meaning
REMOTE_PROTOCOL_FLAG_LOOPBACK 0x1	The remote protocol is using a loopback.
REMOTE_PROTOCOL_FLAG_OFFLINE 0x2	The remote protocol is using an offline cache.
REMOTE_PROTOCOL_INFO_FLAG_PERSISTENT_HANDLE 0x4	The remote protocol is using a persistent handle. Windows 7 and Windows Server 2008 R2: This flag is not supported before Windows 8 and Windows Server 2012.
REMOTE_PROTOCOL_INFO_FLAG_PRIVACY 0x8	The remote protocol is using privacy. This is only supported if the

	StructureVersion member is 2 or higher.
	Windows 7 and Windows Server 2008 R2: This flag is not supported before Windows 8 and Windows Server 2012.
REMOTE_PROTOCOL_INFO_FLAG_INTEGRITY 0x10	The remote protocol is using integrity so the data is signed. This is only supported if the StructureVersion member is 2 or higher.
	Windows 7 and Windows Server 2008 R2: This flag is not supported before Windows 8 and Windows Server 2012.
REMOTE_PROTOCOL_INFO_FLAG_MUTUAL_AUTH 0x20	The remote protocol is using mutual authentication using Kerberos. This is only supported if the StructureVersion member is 2 or higher.
	Windows 7 and Windows Server 2008 R2: This flag is not supported before Windows 8 and Windows Server 2012.

`GenericReserved`

Protocol-generic information structure.

`GenericReserved.Reserved[8]`

Should be set to zero. Do not use this member.

`ProtocolSpecificReserved`

Protocol-specific information structure.

`ProtocolSpecificReserved.Reserved[16]`

Should be set to zero. Do not use this member.

`ProtocolSpecific`

`ProtocolSpecific.Smb2`

`ProtocolSpecific.Smb2.Server`

`ProtocolSpecific.Smb2.Server.Capabilities`

`ProtocolSpecific.Smb2.Share`

`ProtocolSpecific.Smb2.Share.Capabilities`

`ProtocolSpecific.Smb2.Share.ShareFlags`

`ProtocolSpecific.Smb2.Share.CachingFlags`

`ProtocolSpecific.Reserved[16]`

Remarks

The `FILE_REMOTE_PROTOCOL_INFO` structure is valid only for use with the [GetFileInformationByHandleEx](#) function.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_RENAME_INFO structure (winbase.h)

Article09/01/2022

Contains the target name to which the source file should be renamed. Use only when calling [SetFileInformationByHandle](#).

Syntax

C++

```
typedef struct _FILE_RENAME_INFO {
    union {
        BOOLEAN ReplaceIfExists;
        DWORD   Flags;
    } DUMMYUNIONNAME;
    BOOLEAN ReplaceIfExists;
    HANDLE  RootDirectory;
    DWORD   FileNameLength;
    WCHAR   FileName[1];
} FILE_RENAME_INFO, *PFILE_RENAME_INFO;
```

Members

DUMMYUNIONNAME

DUMMYUNIONNAME.ReplaceIfExists

This field is used when [SetFileInformationByHandle](#)'s *FileInformationClass* parameter is set to **FileRenameInfo**. If this field is **TRUE** and the target file exists then the target file will be replaced by the source file. If this field is **FALSE** and the target file exists then operation will return an error.

DUMMYUNIONNAME.Flags

This field is used when [SetFileInformationByHandle](#)'s *FileInformationClass* parameter is set to **FileRenameInfoEx**.

ReplaceIfExists

RootDirectory

This field should be set to NULL.

FileNameLength

The size of **FileName** in bytes, not including the NUL-termination.

FileName[1]

A NUL-terminated wide-character string containing the new path to the file. The value can be one of the following:

- An absolute path (drive, directory, and filename).
- A path relative to the process's current directory.
- The new name of an NTFS file stream, starting with `\:`.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_STANDARD_INFO structure (winbase.h)

Article04/02/2021

Receives extended information for the file. Used for file handles. Use only when calling [GetFileInformationByHandleEx](#).

Syntax

C++

```
typedef struct _FILE_STANDARD_INFO {
    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER EndOfFile;
    DWORD       NumberOfLinks;
    BOOLEAN     DeletePending;
    BOOLEAN     Directory;
} FILE_STANDARD_INFO, *PFILE_STANDARD_INFO;
```

Members

`AllocationSize`

The amount of space that is allocated for the file.

`EndOfFile`

The end of the file.

`NumberOfLinks`

The number of links to the file.

`DeletePending`

TRUE if the file in the delete queue; otherwise, **false**.

`Directory`

TRUE if the file is a directory; otherwise, **false**.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_STORAGE_INFO structure (winbase.h)

Article07/27/2022

Contains directory information for a file. This structure is returned from the [GetFileInformationByHandleEx](#) function when **FileStorageInfo** is passed in the *FileInformationClass* parameter.

Syntax

C++

```
typedef struct _FILE_STORAGE_INFO {
    ULONG LogicalBytesPerSector;
    ULONG PhysicalBytesPerSectorForAtomicity;
    ULONG PhysicalBytesPerSectorForPerformance;
    ULONG FileSystemEffectivePhysicalBytesPerSectorForAtomicity;
    ULONG Flags;
    ULONG ByteOffsetForSectorAlignment;
    ULONG ByteOffsetForPartitionAlignment;
} FILE_STORAGE_INFO, *PFILE_STORAGE_INFO;
```

Members

`LogicalBytesPerSector`

Logical bytes per sector reported by physical storage. This is the smallest size for which uncached I/O is supported.

`PhysicalBytesPerSectorForAtomicity`

Bytes per sector for atomic writes. Writes smaller than this may require a read before the entire block can be written atomically.

`PhysicalBytesPerSectorForPerformance`

Bytes per sector for optimal performance for writes.

`FileSystemEffectivePhysicalBytesPerSectorForAtomicity`

This is the size of the block used for atomicity by the file system. This may be a trade-off between the optimal size of the physical media and one that is easier to adapt existing

code and structures.

Flags

This member can contain combinations of flags specifying information about the alignment of the storage.

Value	Meaning
STORAGE_INFO_FLAGS_ALIGNED_DEVICE 0x00000001	When set, this flag indicates that the logical sectors of the storage device are aligned to physical sector boundaries.
STORAGE_INFO_FLAGS_PARTITION_ALIGNED_ON_DEVICE 0x00000002	When set, this flag indicates that the partition is aligned to physical sector boundaries on the storage device.

ByteOffsetForSectorAlignment

Logical sector offset within the first physical sector where the first logical sector is placed, in bytes. If this value is set to **STORAGE_INFO_OFFSET_UNKNOWN** (0xffffffff), there was insufficient information to compute this field.

ByteOffsetForPartitionAlignment

Offset used to align the partition to a physical sector boundary on the storage device, in bytes. If this value is set to **STORAGE_INFO_OFFSET_UNKNOWN** (0xffffffff), there was insufficient information to compute this field.

Remarks

If a volume is built on top of storage devices with different properties (for example a mirrored, spanned, striped, or RAID configuration) the sizes returned are those of the largest size of any of the underlying storage devices.

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]

Header

winbase.h (include Windows.h)

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[File Management Structures](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FILE_STREAM_INFO structure (winbase.h)

Article09/01/2022

Receives file stream information for the specified file. Used for any handles. Use only when calling [GetFileInformationByHandleEx](#).

Syntax

C++

```
typedef struct _FILE_STREAM_INFO {
    DWORD      NextEntryOffset;
    DWORD      StreamNameLength;
    LARGE_INTEGER StreamSize;
    LARGE_INTEGER StreamAllocationSize;
    WCHAR      StreamName[1];
} FILE_STREAM_INFO, *PFILE_STREAM_INFO;
```

Members

`NextEntryOffset`

The offset for the next **FILE_STREAM_INFO** entry that is returned. This member is zero if no other entries follow this one.

`StreamNameLength`

The length, in bytes, of **StreamName**.

`StreamSize`

The size, in bytes, of the data stream.

`StreamAllocationSize`

The amount of space that is allocated for the stream, in bytes. This value is usually a multiple of the sector or cluster size of the underlying physical device.

`StreamName[1]`

The stream name.

Remarks

The **FILE_STREAM_INFO** structure is used to enumerate the streams for a file.

Support for named data streams is file-system-specific.

The **FILE_STREAM_INFO** structure must be aligned on a **LONGLONG** (8-byte) boundary. If a buffer contains two or more of these structures, the **NextEntryOffset** value in each entry, except the last, falls on an 8-byte boundary.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Header	winbase.h (include Windows.h)
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[GetFileInformationByHandleEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FileEncryptionStatusA function (winbase.h)

Article02/09/2023

Retrieves the encryption status of the specified file.

Syntax

C++

```
BOOL FileEncryptionStatusA(
    [in]  LPCSTR  lpFileName,
    [out] LPDWORD lpStatus
);
```

Parameters

[in] lpFileName

The name of the file.

[out] lpStatus

A pointer to a variable that receives the encryption status of the file. This parameter can be one of the following values.

Value	Meaning
FILE_ENCRYPTABLE 0	The file can be encrypted. Home, Home Premium, Starter, and ARM Editions of Windows: FILE_ENCRYPTABLE may be returned but EFS does not support encrypting files on these editions of Windows.
FILE_IS_ENCRYPTED 1	The file is encrypted.
FILE_READ_ONLY 8	The file is a read-only file.
FILE_ROOT_DIR 3	The file is a root directory. Root directories cannot be encrypted.
FILE_SYSTEM_ATTR	The file is a system file. System files cannot be encrypted.

FILE_SYSTEM_DIR 4	The file is a system directory. System directories cannot be encrypted.
FILE_SYSTEM_NOT_SUPPORT 6	The file system does not support file encryption.
FILE_UNKNOWN 5	The encryption status is unknown. The file may be encrypted.
FILE_USER_DISALLOWED 7	Reserved for future use.

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

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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

① Note

The winbase.h header defines FileEncryptionStatus as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with

code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[EncryptFile](#)

[File Encryption](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FileEncryptionStatusW function (winbase.h)

Article02/09/2023

Retrieves the encryption status of the specified file.

Syntax

C++

```
BOOL FileEncryptionStatusW(
    [in]  LPCWSTR lpFileName,
    [out] LPDWORD lpStatus
);
```

Parameters

[in] lpFileName

The name of the file.

[out] lpStatus

A pointer to a variable that receives the encryption status of the file. This parameter can be one of the following values.

Value	Meaning
FILE_ENCRYPTABLE 0	The file can be encrypted. Home, Home Premium, Starter, and ARM Editions of Windows: FILE_ENCRYPTABLE may be returned but EFS does not support encrypting files on these editions of Windows.
FILE_IS_ENCRYPTED 1	The file is encrypted.
FILE_READ_ONLY 8	The file is a read-only file.
FILE_ROOT_DIR 3	The file is a root directory. Root directories cannot be encrypted.
FILE_SYSTEM_ATTR	The file is a system file. System files cannot be encrypted.

FILE_SYSTEM_DIR 4	The file is a system directory. System directories cannot be encrypted.
FILE_SYSTEM_NOT_SUPPORT 6	The file system does not support file encryption.
FILE_UNKNOWN 5	The encryption status is unknown. The file may be encrypted.
FILE_USER_DISALLOWED 7	Reserved for future use.

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

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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

① Note

The winbase.h header defines FileEncryptionStatus as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with

code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[EncryptFile](#)

[File Encryption](#)

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FileTimeToDosDateTime function (winbase.h)

Article 10/13/2021

Converts a file time to MS-DOS date and time values.

Syntax

C++

```
BOOL FileTimeToDosDateTime(
    [in] const FILETIME *lpFileTime,
    [out] LPWORD     lpFatDate,
    [out] LPWORD     lpFatTime
);
```

Parameters

[in] lpFileTime

A pointer to a [FILETIME](#) structure containing the file time to convert to MS-DOS date and time format.

[out] lpFatDate

A pointer to a variable to receive the MS-DOS date. The date is a packed value with the following format.

Bits	Description
0–4	Day of the month (1–31)
5–8	Month (1 = January, 2 = February, etc.)
9–15	Year offset from 1980 (add 1980 to get actual year)

[out] lpFatTime

A pointer to a variable to receive the MS-DOS time. The time is a packed value with the following format.

Bits	Description
------	-------------

0–4	Second divided by 2
5–10	Minute (0–59)
11–15	Hour (0–23 on a 24-hour clock)

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

The MS-DOS date format can represent only dates between 1/1/1980 and 12/31/2107; this conversion fails if the input file time is outside this range.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DosDateTimeToFileTime](#)

[FileTimeToSystemTime](#)

[SystemTimeToFileTime](#)

[Time Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindActCtxSectionGuid function (winbase.h)

Article 10/13/2021

The **FindActCtxSectionGuid** function retrieves information on a specific GUID in the current activation context and returns a [ACTCTX_SECTION_KEYED_DATA](#) structure.

Syntax

C++

```
BOOL FindActCtxSectionGuid(
    [in]  DWORD          dwFlags,
    [in]  const GUID     *lpExtensionGuid,
    [in]  ULONG           ulSectionId,
    [in]  const GUID     *lpGuidToFind,
    [out] PACTCTX_SECTION_KEYED_DATA ReturnedData
);
```

Parameters

[in] dwFlags

Flags that determine how this function is to operate. Only the following flag is currently defined.

Value	Meaning
FIND_ACTCTX_SECTION_KEY_RETURN_HACTCTX	This function returns the activation context handle where the redirection data was found in the hActCtx member of the ACTCTX_SECTION_KEYED_DATA structure. The caller must use ReleaseActCtx to release this activation context.

[in] lpExtensionGuid

Reserved; must be null.

[in] ulSectionId

Identifier of the section of the activation context in which to search for the specified GUID.

The following are valid GUID section identifiers:

- ACTIVATION_CONTEXT_SECTION_COM_SERVER_REDIRECTION
- ACTIVATION_CONTEXT_SECTION_COM_INTERFACE_REDIRECTION
- ACTIVATION_CONTEXT_SECTION_COM_TYPE_LIBRARY_REDIRECTION

The following is a valid GUID section identifier beginning with Windows Server 2003 and Windows XP with SP1:

- ACTIVATION_CONTEXT_SECTION_CLR_SURROGATES

[in] lpGuidToFind

Pointer to a GUID to be used as the search criteria.

[out] ReturnedData

Pointer to an [ACTCTX_SECTION_KEYED_DATA](#) structure to be filled out with the requested GUID information.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

This function should only be called by the Side-by-side API functions or COM methods. Applications should not directly call this function.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX_SECTION_KEYED_DATA](#)

[FindActCtxSectionString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindActCtxSectionStringA function (winbase.h)

Article02/09/2023

The **FindActCtxSectionString** function retrieves information on a specific string in the current activation context and returns a [ACTCTX_SECTION_KEYED_DATA](#) structure.

Syntax

C++

```
BOOL FindActCtxSectionStringA(
    [in]  DWORD          dwFlags,
    [in]  const GUID     *lpExtensionGuid,
    [in]  ULONG           ulSectionId,
    [in]  LPCSTR          lpStringToFind,
    [out] PACTCTX_SECTION_KEYED_DATA ReturnedData
);
```

Parameters

[in] dwFlags

Flags that determine how this function is to operate. Only the following flag is currently defined.

Value	Meaning
FIND_ACTCTX_SECTION_KEY_RETURN_HACTCTX	This function returns the activation context handle where the redirection data was found in the hActCtx member of the ACTCTX_SECTION_KEYED_DATA structure. The caller must use ReleaseActCtx to release this activation context.

[in] lpExtensionGuid

Reserved; must be null.

[in] ulSectionId

Identifier of the string section of the activation context in which to search for the specific string.

The following are valid string section identifiers:

- ACTIVATION_CONTEXT_SECTION_ASSEMBLY_INFORMATION
- ACTIVATION_CONTEXT_SECTION_DLL_REDIRECTION
- ACTIVATION_CONTEXT_SECTION_WINDOW_CLASS_REDIRECTION
- ACTIVATION_CONTEXT_SECTION_COM_PROGID_REDIRECTION

[in] lpStringToFind

Pointer to a null-terminated string to be used as the search criteria.

[out] ReturnedData

Pointer to an [ACTCTX_SECTION_KEYED_DATA](#) structure to be filled out with the requested string information.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

This function should only be called by the Side-by-side API functions or COM methods. Applications should not directly call this function.

Note

The winbase.h header defines FindActCtxSectionString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX_SECTION_KEYED_DATA](#)

[FindActCtxSectionGuid](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindActCtxSectionStringW function (winbase.h)

Article02/09/2023

The **FindActCtxSectionString** function retrieves information on a specific string in the current activation context and returns a [ACTCTX_SECTION_KEYED_DATA](#) structure.

Syntax

C++

```
BOOL FindActCtxSectionStringW(
    [in] DWORD dwFlags,
    [in] const GUID *lpExtensionGuid,
    [in] ULONG ulSectionId,
    [in] LPCWSTR lpStringToFind,
    [out] PACTCTX_SECTION_KEYED_DATA ReturnedData
);
```

Parameters

[in] dwFlags

Flags that determine how this function is to operate. Only the following flag is currently defined.

Value	Meaning
FIND_ACTCTX_SECTION_KEY_RETURN_HACTCTX	This function returns the activation context handle where the redirection data was found in the hActCtx member of the ACTCTX_SECTION_KEYED_DATA structure. The caller must use ReleaseActCtx to release this activation context.

[in] lpExtensionGuid

Reserved; must be null.

[in] ulSectionId

Identifier of the string section of the activation context in which to search for the specific string.

The following are valid string section identifiers:

- ACTIVATION_CONTEXT_SECTION_ASSEMBLY_INFORMATION
- ACTIVATION_CONTEXT_SECTION_DLL_REDIRECTION
- ACTIVATION_CONTEXT_SECTION_WINDOW_CLASS_REDIRECTION
- ACTIVATION_CONTEXT_SECTION_COM_PROGID_REDIRECTION

[in] lpStringToFind

Pointer to a null-terminated string to be used as the search criteria.

[out] ReturnedData

Pointer to an [ACTCTX_SECTION_KEYED_DATA](#) structure to be filled out with the requested string information.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

This function should only be called by the Side-by-side API functions or COM methods. Applications should not directly call this function.

Note

The winbase.h header defines FindActCtxSectionString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX_SECTION_KEYED_DATA](#)

[FindActCtxSectionGuid](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindAtomA function (winbase.h)

Article02/09/2023

Searches the local atom table for the specified character string and retrieves the atom associated with that string.

Syntax

C++

```
ATOM FindAtomA(  
    [in] LPCSTR lpString  
);
```

Parameters

[in] lpString

Type: **LPCTSTR**

The character string for which to search.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See Remarks for more information.

Return value

Type: **ATOM**

If the function succeeds, the return value is the atom associated with the given string.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Even though the system preserves the case of a string in an atom table, the search performed by the **FindAtom** function is not case sensitive.

If *lpString* was created by the [MAKEINTATOM](#) macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Note

The winbase.h header defines FindAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindAtomW function (winbase.h)

Article02/09/2023

Searches the local atom table for the specified character string and retrieves the atom associated with that string.

Syntax

C++

```
ATOM FindAtomW(
    [in] LPCWSTR lpString
);
```

Parameters

[in] lpString

Type: **LPCTSTR**

The character string for which to search.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See Remarks for more information.

Return value

Type: **ATOM**

If the function succeeds, the return value is the atom associated with the given string.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Even though the system preserves the case of a string in an atom table, the search performed by the **FindAtom** function is not case sensitive.

If *lpString* was created by the [MAKEINTATOM](#) macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Note

The winbase.h header defines FindAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstFileNameTransactedW function (winbase.h)

Article 10/13/2021

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Creates an enumeration of all the hard links to the specified file as a transacted operation. The function returns a handle to the enumeration that can be used on subsequent calls to the [FindNextFileNameW](#) function.

Syntax

C++

```
HANDLE FindFirstFileNameTransactedW(
    [in]          LPCWSTR lpFileName,
    [in]          DWORD   dwFlags,
    [in, out]     LPDWORD StringLength,
    [in, out]     PWSTR   LinkName,
    [in, optional] HANDLE  hTransaction
);
```

Parameters

`[in] lpFileName`

The name of the file.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE` (6805).

`[in] dwFlags`

Reserved; specify zero (0).

`[in, out] StringLength`

The size of the buffer pointed to by the *LinkName* parameter, in characters. If this call fails and the error is **ERROR_MORE_DATA** (234), the value that is returned by this parameter is the size that the buffer pointed to by *LinkName* must be to contain all the data.

[in, out] *LinkName*

A pointer to a buffer to store the first link name found for *lpFileName*.

[in, optional] *hTransaction*

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is a search handle that can be used with the [FindNextFileNameW](#) function or closed with the [FindClose](#) function.

If the function fails, the return value is **INVALID_HANDLE_VALUE** (0xffffffff). To get extended error information, call the [GetLastError](#) function.

Remarks

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[FindClose](#)

[FindNextFileNameW](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstFileTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Searches a directory for a file or subdirectory with a name that matches a specific name as a transacted operation.

This function is the transacted form of the [FindFirstFileEx](#) function.

For the most basic version of this function, see [FindFirstFile](#).

Syntax

C++

```
HANDLE FindFirstFileTransactedA(
    [in]  LPCSTR          lpFileName,
    [in]  FINDEX_INFO_LEVELS fInfoLevelId,
    [out] LPVOID           lpFindFileData,
    [in]  FINDEX_SEARCH_OPS fSearchOp,
           LPVOID           lpSearchFilter,
    [in]  DWORD            dwAdditionalFlags,
    [in]  HANDLE           hTransaction
);
```

Parameters

`[in] lpFileName`

The directory or path, and the file name. The file name can include wildcard characters, for example, an asterisk (*) or a question mark (?).

This parameter should not be **NULL**, an invalid string (for example, an empty string or a string that is missing the terminating null character), or end in a trailing backslash (\).

If the string ends with a wildcard, period (.), or directory name, the user must have access to the root and all subdirectories on the path.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] fInfoLevelId

The information level of the returned data.

This parameter is one of the [FINDEX_INFO_LEVELS](#) enumeration values.

[out] lpFindFileData

A pointer to the [WIN32_FIND_DATA](#) structure that receives information about a found file or subdirectory.

[in] fSearchOp

The type of filtering to perform that is different from wildcard matching.

This parameter is one of the [FINDEX_SEARCH_OPS](#) enumeration values.

lpSearchFilter

A pointer to the search criteria if the specified *fSearchOp* needs structured search information.

At this time, none of the supported *fSearchOp* values require extended search information. Therefore, this pointer must be **NULL**.

[in] dwAdditionalFlags

Specifies additional flags that control the search.

Value	Meaning
FIND_FIRST_EX_CASE_SENSITIVE 1	Searches are case-sensitive.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is a search handle used in a subsequent call to [FindNextFile](#) or [FindClose](#), and the *lpFindFileData* parameter contains information about the first file or directory found.

If the function fails or fails to locate files from the search string in the *lpFileName* parameter, the return value is **INVALID_HANDLE_VALUE** and the contents of *lpFindFileData* are indeterminate. To get extended error information, call the [GetLastError](#) function.

Remarks

The **FindFirstFileTransacted** function opens a search handle and returns information about the first file that the file system finds with a name that matches the specified pattern. This may or may not be the first file or directory that appears in a directory-listing application (such as the dir command) when given the same file name string pattern. This is because **FindFirstFileTransacted** does no sorting of the search results. For additional information, see [FindNextFile](#).

The following list identifies some other search characteristics:

- The search is performed strictly on the name of the file, not on any attributes such as a date or a file type.
- The search includes the long and short file names.
- An attempt to open a search with a trailing backslash always fails.
- Passing an invalid string, **NULL**, or empty string for the *lpFileName* parameter is not a valid use of this function. Results in this case are undefined.

Note In rare cases, file information on NTFS file systems may not be current at the time you call this function. To be assured of getting the current file information, call the [GetFileInformationByHandle](#) function.

If the underlying file system does not support the specified type of filtering, other than directory filtering, **FindFirstFileTransacted** fails with the error **ERROR_NOT_SUPPORTED**. The application must use [FINDEX_SEARCH_OPS](#) type **FileExSearchNameMatch** and perform its own filtering.

After the search handle is established, use it in the [FindNextFile](#) function to search for other files that match the same pattern with the same filtering that is being performed. When the search handle is not needed, it should be closed by using the [FindClose](#) function.

As stated previously, you cannot use a trailing backslash (\) in the *lpFileName* input string for **FindFirstFileTransacted**, therefore it may not be obvious how to search root directories. If you want to see files or get the attributes of a root directory, the following options would apply:

- To examine files in a root directory, you can use "C:*" and step through the directory by using [FindNextFile](#).
- To get the attributes of a root directory, use the [GetFileAttributes](#) function.

Note Prepending the string "\\\\" does not allow access to the root directory.

On network shares, you can use an *lpFileName* in the form of the following: "\\server\service*". However, you cannot use an *lpFileName* that points to the share itself; for example, "\\server\service" is not valid.

To examine a directory that is not a root directory, use the path to that directory, without a trailing backslash. For example, an argument of "C:\Windows" returns information about the directory "C:\Windows", not about a directory or file in "C:\Windows". To examine the files and directories in "C:\Windows", use an *lpFileName* of "C:\Windows*".

Be aware that some other thread or process could create or delete a file with this name between the time you query for the result and the time you act on the information. If this is a potential concern for your application, one possible solution is to use the [CreateFile](#) function with **CREATE_NEW** (which fails if the file exists) or **OPEN_EXISTING** (which fails if the file does not exist).

If you are writing a 32-bit application to list all the files in a directory and the application may be run on a 64-bit computer, you should call [Wow64DisableWow64FsRedirection](#)

before calling **FindFirstFileTransacted** and call **Wow64RevertWow64FsRedirection** after the last call to **FindNextFile**. For more information, see [File System Redirector](#).

If the path points to a symbolic link, the **WIN32_FIND_DATA** buffer contains information about the symbolic link, not the target.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

 **Note**

The `winbase.h` header defines `FindFirstFileTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[FindClose](#)

[FindNextFile](#)

[GetFileAttributes](#)

[SetFileAttributes](#)

[Symbolic Links](#)

[Transactional NTFS](#)

[WIN32_FIND_DATA](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstFileTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Searches a directory for a file or subdirectory with a name that matches a specific name as a transacted operation.

This function is the transacted form of the [FindFirstFileEx](#) function.

For the most basic version of this function, see [FindFirstFile](#).

Syntax

C++

```
HANDLE FindFirstFileTransactedW(
    [in]  LPCWSTR          lpFileName,
    [in]  FINDEX_INFO_LEVELS fInfoLevelId,
    [out] LPVOID            lpFindFileData,
    [in]  FINDEX_SEARCH_OPS fSearchOp,
           LPVOID            lpSearchFilter,
    [in]  DWORD             dwAdditionalFlags,
    [in]  HANDLE            hTransaction
);
```

Parameters

`[in] lpFileName`

The directory or path, and the file name. The file name can include wildcard characters, for example, an asterisk (*) or a question mark (?).

This parameter should not be **NULL**, an invalid string (for example, an empty string or a string that is missing the terminating null character), or end in a trailing backslash (\).

If the string ends with a wildcard, period (.), or directory name, the user must have access to the root and all subdirectories on the path.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

`[in] fInfoLevelId`

The information level of the returned data.

This parameter is one of the [FINDEX_INFO_LEVELS](#) enumeration values.

`[out] lpFindFileData`

A pointer to the [WIN32_FIND_DATA](#) structure that receives information about a found file or subdirectory.

`[in] fSearchOp`

The type of filtering to perform that is different from wildcard matching.

This parameter is one of the [FINDEX_SEARCH_OPS](#) enumeration values.

`lpSearchFilter`

A pointer to the search criteria if the specified *fSearchOp* needs structured search information.

At this time, none of the supported *fSearchOp* values require extended search information. Therefore, this pointer must be **NULL**.

`[in] dwAdditionalFlags`

Specifies additional flags that control the search.

Value	Meaning
FIND_FIRST_EX_CASE_SENSITIVE 1	Searches are case-sensitive.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is a search handle used in a subsequent call to [FindNextFile](#) or [FindClose](#), and the *lpFindFileData* parameter contains information about the first file or directory found.

If the function fails or fails to locate files from the search string in the *lpFileName* parameter, the return value is **INVALID_HANDLE_VALUE** and the contents of *lpFindFileData* are indeterminate. To get extended error information, call the [GetLastError](#) function.

Remarks

The **FindFirstFileTransacted** function opens a search handle and returns information about the first file that the file system finds with a name that matches the specified pattern. This may or may not be the first file or directory that appears in a directory-listing application (such as the dir command) when given the same file name string pattern. This is because **FindFirstFileTransacted** does no sorting of the search results. For additional information, see [FindNextFile](#).

The following list identifies some other search characteristics:

- The search is performed strictly on the name of the file, not on any attributes such as a date or a file type.
- The search includes the long and short file names.
- An attempt to open a search with a trailing backslash always fails.
- Passing an invalid string, **NULL**, or empty string for the *lpFileName* parameter is not a valid use of this function. Results in this case are undefined.

Note In rare cases, file information on NTFS file systems may not be current at the time you call this function. To be assured of getting the current file information, call the [GetFileInformationByHandle](#) function.

If the underlying file system does not support the specified type of filtering, other than directory filtering, **FindFirstFileTransacted** fails with the error **ERROR_NOT_SUPPORTED**. The application must use [FINDEX_SEARCH_OPS](#) type **FileExSearchNameMatch** and perform its own filtering.

After the search handle is established, use it in the [FindNextFile](#) function to search for other files that match the same pattern with the same filtering that is being performed. When the search handle is not needed, it should be closed by using the [FindClose](#) function.

As stated previously, you cannot use a trailing backslash (\) in the *lpFileName* input string for **FindFirstFileTransacted**, therefore it may not be obvious how to search root directories. If you want to see files or get the attributes of a root directory, the following options would apply:

- To examine files in a root directory, you can use "C:*" and step through the directory by using [FindNextFile](#).
- To get the attributes of a root directory, use the [GetFileAttributes](#) function.

Note Prepending the string "\\\\" does not allow access to the root directory.

On network shares, you can use an *lpFileName* in the form of the following: "\\server\service*". However, you cannot use an *lpFileName* that points to the share itself; for example, "\\server\service" is not valid.

To examine a directory that is not a root directory, use the path to that directory, without a trailing backslash. For example, an argument of "C:\Windows" returns information about the directory "C:\Windows", not about a directory or file in "C:\Windows". To examine the files and directories in "C:\Windows", use an *lpFileName* of "C:\Windows*".

Be aware that some other thread or process could create or delete a file with this name between the time you query for the result and the time you act on the information. If this is a potential concern for your application, one possible solution is to use the [CreateFile](#) function with **CREATE_NEW** (which fails if the file exists) or **OPEN_EXISTING** (which fails if the file does not exist).

If you are writing a 32-bit application to list all the files in a directory and the application may be run on a 64-bit computer, you should call [Wow64DisableWow64FsRedirection](#)

before calling **FindFirstFileTransacted** and call **Wow64RevertWow64FsRedirection** after the last call to **FindNextFile**. For more information, see [File System Redirector](#).

If the path points to a symbolic link, the **WIN32_FIND_DATA** buffer contains information about the symbolic link, not the target.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

 **Note**

The `winbase.h` header defines `FindFirstFileTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[FindClose](#)

[FindNextFile](#)

[GetFileAttributes](#)

[SetFileAttributes](#)

[Symbolic Links](#)

[Transactional NTFS](#)

[WIN32_FIND_DATA](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstStreamTransactedW function (winbase.h)

Article 10/13/2021

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Enumerates the first stream in the specified file or directory as a transacted operation.

Syntax

C++

```
HANDLE FindFirstStreamTransactedW(
    [in]  LPCWSTR           lpFileName,
    [in]  STREAM_INFO_LEVELS InfoLevel,
    [out] LPVOID            lpFindStreamData,
    [in]  DWORD             dwFlags,
    [in]  HANDLE            hTransaction
);
```

Parameters

[in] lpFileName

The fully qualified file name.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE** (6805).

[in] InfoLevel

The information level of the returned data. This parameter is one of the values in the [STREAM_INFO_LEVELS](#) enumeration type.

Value	Meaning
FindStreamInfoStandard	The data is returned in a WIN32_FIND_STREAM_DATA structure.
0	

[out] lpFindStreamData

A pointer to a buffer that receives the file data. The format of this data depends on the value of the *InfoLevel* parameter.

dwFlags

Reserved for future use. This parameter must be zero.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is a search handle that can be used in subsequent calls to the [FindNextStreamW](#) function.

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call [GetLastError](#).

Remarks

All files contain a default data stream. On NTFS, files can also contain one or more named data streams. On FAT file systems, files cannot have more than the default data stream, and therefore, this function will not return valid results when used on FAT filesystem files. This function works on all file systems that supports hard links; otherwise, the function returns **ERROR_STATUS_NOT_IMPLEMENTED** (6805).

The [FindFirstStreamTransactedW](#) function opens a search handle and returns information about the first stream in the specified file or directory. For files, this is always the default data stream, ::\$DATA. After the search handle has been established, use it in the [FindNextStreamW](#) function to search for other streams in the specified file or directory. When the search handle is no longer needed, it should be closed using the [FindClose](#) function.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No

SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[FindClose](#)

[FindNextStreamW](#)

[STREAM_INFO_LEVELS](#)

[Transactional NTFS](#)

[WIN32_FIND_STREAM_DATA](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstVolumeA function (winbase.h)

Article07/27/2022

Retrieves the name of a volume on a computer. **FindFirstVolume** is used to begin scanning the volumes of a computer.

Syntax

C++

```
HANDLE FindFirstVolumeA(
    [out] LPSTR lpszVolumeName,
    [in]  DWORD cchBufferLength
);
```

Parameters

[out] `lpszVolumeName`

A pointer to a buffer that receives a null-terminated string that specifies a volume **GUID** path for the first volume that is found.

[in] `cchBufferLength`

The length of the buffer to receive the volume **GUID** path, in **TCHARs**.

Return value

If the function succeeds, the return value is a search handle used in a subsequent call to the [FindNextVolume](#) and [FindVolumeClose](#) functions.

If the function fails to find any volumes, the return value is the **INVALID_HANDLE_VALUE** error code. To get extended error information, call [GetLastError](#).

Remarks

The **FindFirstVolume** function opens a volume search handle and returns information about the first volume found on a computer. After the search handle is established, you

can use the [FindNextVolume](#) function to search for other volumes. When the search handle is no longer needed, close it by using the [FindVolumeClose](#) function.

You should not assume any correlation between the order of the volumes that are returned by these functions and the order of the volumes that are on the computer. In particular, do not assume any correlation between volume order and drive letters as assigned by the BIOS (if any) or the Disk Administrator.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

Examples

For an example, see [Displaying Volume Paths](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FindNextVolume](#)

[FindVolumeClose](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstVolumeMountPointA function (winbase.h)

Article02/09/2023

Retrieves the name of a mounted folder on the specified volume.

FindFirstVolumeMountPoint is used to begin scanning the mounted folders on a volume.

Syntax

C++

```
HANDLE FindFirstVolumeMountPointA(
    [in]  LPCSTR lpszRootPathName,
    [out] LPSTR  lpszVolumeMountPoint,
    [in]  DWORD   cchBufferLength
);
```

Parameters

[in] lpszRootPathName

A volume GUID path for the volume to scan for mounted folders. A trailing backslash is required.

[out] lpszVolumeMountPoint

A pointer to a buffer that receives the name of the first mounted folder that is found.

[in] cchBufferLength

The length of the buffer that receives the path to the mounted folder, in TCHARs.

Return value

If the function succeeds, the return value is a search handle used in a subsequent call to the [FindNextVolumeMountPoint](#) and [FindVolumeMountPointClose](#) functions.

If the function fails to find a mounted folder on the volume, the return value is the **INVALID_HANDLE_VALUE** error code. To get extended error information, call

[GetLastError](#).

Remarks

The [FindFirstVolumeMountPoint](#) function opens a mounted folder search handle and returns information about the first mounted folder that is found on the specified volume. After the search handle is established, you can use the [FindNextVolumeMountPoint](#) function to search for other mounted folders. When the search handle is no longer needed, close it with the [FindVolumeMountPointClose](#) function.

The [FindFirstVolumeMountPoint](#), [FindNextVolumeMountPoint](#), and [FindVolumeMountPointClose](#) functions return paths to mounted folders for a specified volume. They do not return drive letters or volume **GUID** paths. For information about enumerating the volume **GUID** paths for a volume, see [Enumerating Volume GUID Paths](#).

You should not assume any correlation between the order of the mounted folders that are returned by these functions and the order of the mounted folders that are returned by other functions or tools.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. CsvFS does not support adding mount point on a CSV volume. ReFS does not index mount points.

Note

The winbase.h header defines [FindFirstVolumeMountPoint](#) as an alias which automatically selects the ANSI or Unicode version of this function based on the

definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[FindNextVolumeMountPoint](#)

[FindVolumeMountPointClose](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindFirstVolumeMountPointW function (winbase.h)

Article02/09/2023

Retrieves the name of a mounted folder on the specified volume.

FindFirstVolumeMountPoint is used to begin scanning the mounted folders on a volume.

Syntax

C++

```
HANDLE FindFirstVolumeMountPoint(
    [in]  LPCWSTR lpszRootPathName,
    [out] LPWSTR  lpszVolumeMountPoint,
    [in]  DWORD   cchBufferLength
);
```

Parameters

[in] lpszRootPathName

A volume GUID path for the volume to scan for mounted folders. A trailing backslash is required.

[out] lpszVolumeMountPoint

A pointer to a buffer that receives the name of the first mounted folder that is found.

[in] cchBufferLength

The length of the buffer that receives the path to the mounted folder, in TCHARs.

Return value

If the function succeeds, the return value is a search handle used in a subsequent call to the [FindNextVolumeMountPoint](#) and [FindVolumeMountPointClose](#) functions.

If the function fails to find a mounted folder on the volume, the return value is the **INVALID_HANDLE_VALUE** error code. To get extended error information, call

[GetLastError](#).

Remarks

The [FindFirstVolumeMountPoint](#) function opens a mounted folder search handle and returns information about the first mounted folder that is found on the specified volume. After the search handle is established, you can use the [FindNextVolumeMountPoint](#) function to search for other mounted folders. When the search handle is no longer needed, close it with the [FindVolumeMountPointClose](#) function.

The [FindFirstVolumeMountPoint](#), [FindNextVolumeMountPoint](#), and [FindVolumeMountPointClose](#) functions return paths to mounted folders for a specified volume. They do not return drive letters or volume **GUID** paths. For information about enumerating the volume **GUID** paths for a volume, see [Enumerating Volume GUID Paths](#).

You should not assume any correlation between the order of the mounted folders that are returned by these functions and the order of the mounted folders that are returned by other functions or tools.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. CsvFS does not support adding mount point on a CSV volume. ReFS does not index mount points.

Note

The winbase.h header defines [FindFirstVolumeMountPoint](#) as an alias which automatically selects the ANSI or Unicode version of this function based on the

definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[FindNextVolumeMountPoint](#)

[FindVolumeMountPointClose](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindNextVolumeA function (winbase.h)

Article 07/27/2022

Continues a volume search started by a call to the [FindFirstVolume](#) function. **FindNextVolume** finds one volume per call.

Syntax

C++

```
BOOL FindNextVolumeA(
    [in] HANDLE hFindVolume,
    [out] LPSTR lpszVolumeName,
    [in] DWORD cchBufferLength
);
```

Parameters

[in] `hFindVolume`

The volume search handle returned by a previous call to the [FindFirstVolume](#) function.

[out] `lpszVolumeName`

A pointer to a string that receives the volume **GUID** path that is found.

[in] `cchBufferLength`

The length of the buffer that receives the volume **GUID** path, in **TCHARs**.

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 no matching files can be found, the [GetLastError](#) function returns the **ERROR_NO_MORE_FILES** error code. In that case, close the search with the [FindVolumeClose](#) function.

Remarks

After the search handle is established by calling [FindFirstVolume](#), you can use the [FindNextVolume](#) function to search for other volumes.

You should not assume any correlation between the order of the volumes that are returned by these functions and the order of the volumes that are on the computer. In particular, do not assume any correlation between volume order and drive letters as assigned by the BIOS (if any) or the Disk Administrator.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

Examples

For an example, see [Displaying Volume Paths](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FindFirstVolume](#)

[FindVolumeClose](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindNextVolumeMountPointA function (winbase.h)

Article02/09/2023

Continues a mounted folder search started by a call to the [FindFirstVolumeMountPoint](#) function. **FindNextVolumeMountPoint** finds one mounted folder per call.

Syntax

C++

```
BOOL FindNextVolumeMountPointA(
    [in] HANDLE hFindVolumeMountPoint,
    [out] LPSTR lpszVolumeMountPoint,
    [in] DWORD cchBufferLength
);
```

Parameters

[in] `hFindVolumeMountPoint`

A mounted folder search handle returned by a previous call to the [FindFirstVolumeMountPoint](#) function.

[out] `lpszVolumeMountPoint`

A pointer to a buffer that receives the name of the mounted folder that is found.

[in] `cchBufferLength`

The length of the buffer that receives the mounted folder name, in TCHARs.

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 no more mounted folders can be found, the [GetLastError](#) function returns the **ERROR_NO_MORE_FILES** error code. In that case, close the search with the [FindVolumeMountPointClose](#) function.

Remarks

After the search handle is established by calling [FindFirstVolumeMountPoint](#), you can use the [FindNextVolumeMountPoint](#) function to search for other mounted folders.

The [FindFirstVolumeMountPoint](#), [FindNextVolumeMountPoint](#), and [FindVolumeMountPointClose](#) functions return paths to mounted folders for a specified volume. They do not return drive letters or volume GUID paths. For information about enumerating the volume **GUID** paths for a volume, see [Enumerating Volume GUID Paths](#).

You should not assume any correlation between the order of the mounted folders that are returned with these functions and the order of the mounted folders that are returned by other functions or tools.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. CsvFS does not support adding mount point on a CSV volume. ReFS does not index mount points.

ⓘ Note

The winbase.h header defines `FindNextVolumeMountPoint` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FindFirstVolumeMountPoint](#)

[FindVolumeMountPointClose](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

FindNextVolumeMountPointW function (winbase.h)

Article02/09/2023

Continues a mounted folder search started by a call to the [FindFirstVolumeMountPoint](#) function. **FindNextVolumeMountPoint** finds one mounted folder per call.

Syntax

C++

```
BOOL FindNextVolumeMountPointW(
    [in]  HANDLE hFindVolumeMountPoint,
    [out] LPWSTR lpszVolumeMountPoint,
    [in]  DWORD  cchBufferLength
);
```

Parameters

[in] `hFindVolumeMountPoint`

A mounted folder search handle returned by a previous call to the [FindFirstVolumeMountPoint](#) function.

[out] `lpszVolumeMountPoint`

A pointer to a buffer that receives the name of the mounted folder that is found.

[in] `cchBufferLength`

The length of the buffer that receives the mounted folder name, in TCHARs.

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 no more mounted folders can be found, the [GetLastError](#) function returns the **ERROR_NO_MORE_FILES** error code. In that case, close the search with the [FindVolumeMountPointClose](#) function.

Remarks

After the search handle is established by calling [FindFirstVolumeMountPoint](#), you can use the [FindNextVolumeMountPoint](#) function to search for other mounted folders.

The [FindFirstVolumeMountPoint](#), [FindNextVolumeMountPoint](#), and [FindVolumeMountPointClose](#) functions return paths to mounted folders for a specified volume. They do not return drive letters or volume GUID paths. For information about enumerating the volume **GUID** paths for a volume, see [Enumerating Volume GUID Paths](#).

You should not assume any correlation between the order of the mounted folders that are returned with these functions and the order of the mounted folders that are returned by other functions or tools.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. CsvFS does not support adding mount point on a CSV volume. ReFS does not index mount points.

ⓘ Note

The winbase.h header defines `FindNextVolumeMountPoint` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FindFirstVolumeMountPoint](#)

[FindVolumeMountPointClose](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindResourceA function (winbase.h)

Article07/27/2022

Determines the location of a resource with the specified type and name in the specified module.

To specify a language, use the [FindResourceEx](#) function.

Syntax

C++

```
HRSRC FindResourceA(
    [in, optional] HMODULE hModule,
    [in]           LPCSTR  lpName,
    [in]           LPCSTR  lpType
);
```

Parameters

[in, optional] hModule

Type: **HMODULE**

A handle to the module whose portable executable file or an accompanying MUI file contains the resource. If this parameter is **NULL**, the function searches the module used to create the current process.

[in] lpName

Type: **LPCTSTR**

The name of the resource. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is the integer identifier of the resource. For more information, see the Remarks section below.

[in] lpType

Type: **LPCTSTR**

The resource type. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is the integer identifier of the given resource type. For

standard resource types, see [Resource Types](#). For more information, see the Remarks section below.

Return value

Type: HRSRC

If the function succeeds, the return value is a handle to the specified resource's information block. To obtain a handle to the resource, pass this handle to the [LoadResource](#) function.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

If [IS_INTRESOURCE](#) is **TRUE** for $x = lpName$ or $lpType$, x specifies the integer identifier of the name or type of the given resource. Otherwise, those parameters are long pointers to null-terminated strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer identifier of the resource's name or type. For example, the string "#258" represents the integer identifier 258.

To reduce the amount of memory required for a resource, an application should refer to it by integer identifier instead of by name.

An application can use [FindResource](#) to find any type of resource, but this function should be used only if the application must access the binary resource data by making subsequent calls to [LoadResource](#) and then to [LockResource](#).

To use a resource immediately, an application should use one of the following resource-specific functions to find the resource and convert the data into a more usable form.

Function	Action
FormatMessage	Loads and formats a message-table entry.
LoadAccelerators	Loads an accelerator table.
LoadBitmap	Loads a bitmap resource.
LoadCursor	Loads a cursor resource.
LoadIcon	Loads an icon resource.

LoadMenu	Loads a menu resource.
LoadString	Loads a string-table entry.

For example, an application can use the [LoadIcon](#) function to load an icon for display on the screen. However, the application should use [FindResource](#) and [LoadResource](#) if it is loading the icon to copy its data to another application.

String resources are stored in sections of up to 16 strings per section. The strings in each section are stored as a sequence of counted (not necessarily null-terminated) Unicode strings. The [LoadString](#) function will extract the string resource from its corresponding section.

Examples

For an example, see [Updating Resources](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[FindResourceEx](#)

[FormatMessage](#)

[IS_INTRESOURCE](#)

[LoadAccelerators](#)

[LoadBitmap](#)

[LoadCursor](#)

[LoadIcon](#)

[LoadMenu](#)

[LoadResource](#)

[LoadString](#)

[LockResource](#)

Other Resources

Reference

[Resources](#)

[SizeofResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindResourceExA function (winbase.h)

Article 07/27/2022

Determines the location of the resource with the specified type, name, and language in the specified module.

Syntax

C++

```
HRSRC FindResourceExA(
    [in, optional] HMODULE hModule,
    [in]           LPCSTR   lpType,
    [in]           LPCSTR   lpName,
    [in]           WORD     wLanguage
);
```

Parameters

[in, optional] hModule

Type: **HMODULE**

A handle to the module whose portable executable file or an accompanying MUI file contains the resource. If this parameter is **NULL**, the function searches the module used to create the current process.

[in] lpType

Type: **LPCTSTR**

The resource type. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is the integer identifier of the given

resource type. For standard resource types, see [Resource Types](#). For more information, see the Remarks section below.

[in] lpName

Type: **LPCTSTR**

The name of the resource. Alternately, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is the integer identifier of the resource. For more

information, see the Remarks section below.

[in] wLanguage

Type: WORD

The language of the resource. If this parameter is `MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL)`, the current language associated with the calling thread is used.

To specify a language other than the current language, use the [MAKELANGID](#) macro to create this parameter. For more information, see [MAKELANGID](#).

Return value

Type: HRSRC

If the function succeeds, the return value is a handle to the specified resource's information block. To obtain a handle to the resource, pass this handle to the [LoadResource](#) function.

If the function fails, the return value is `NULL`. To get extended error information, call [GetLastError](#).

Remarks

If [IS_INTRESOURCE](#) is `TRUE` for `x = lpType` or `lpName`, `x` specifies the integer identifier of the type or name of the given resource. Otherwise, those parameters are long pointers to null-terminated strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer identifier of the resource's name or type. For example, the string "#258" represents the integer identifier 258.

To reduce the amount of memory required for a resource, an application should refer to it by integer identifier instead of by name.

An application can use [FindResourceEx](#) to find any type of resource, but this function should be used only if the application must access the binary resource data by making subsequent calls to [LoadResource](#) and then to [LockResource](#).

To use a resource immediately, an application should use one of the following resource-specific functions to find the resource and convert the data into a more usable form.

Function	Action

FormatMessage	Loads and formats a message-table entry.
LoadAccelerators	Loads an accelerator table.
LoadBitmap	Loads a bitmap resource.
LoadCursor	Loads a cursor resource.
LoadIcon	Loads an icon resource.
LoadMenu	Loads a menu resource.
LoadString	Loads a string-table entry.

For example, an application can use the [LoadIcon](#) function to load an icon for display on the screen. However, the application should use [FindResourceEx](#) and [LoadResource](#) if it is loading the icon to copy its data to another application.

String resources are stored in sections of up to 16 strings per section. The strings in each section are stored as a sequence of counted (not necessarily null-terminated) Unicode strings. The [LoadString](#) function will extract the string resource from its corresponding section.

Examples

For an example, see [Creating a Resource List](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[FindResource](#)

[FormatMessage](#)

[IS_INTRESOURCE](#)

[LoadAccelerators](#)

[LoadBitmap](#)

[LoadCursor](#)

[LoadIcon](#)

[LoadMenu](#)

[LoadResource](#)

[LoadString](#)

[MAKELANGID](#)

Other Resources

[Reference](#)

[Resources](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FindVolumeMountPointClose function (winbase.h)

Article 10/13/2021

Closes the specified mounted folder search handle. The [FindFirstVolumeMountPoint](#) and [FindNextVolumeMountPoint](#) functions use this search handle to locate mounted folders on a specified volume.

Syntax

C++

```
BOOL FindVolumeMountPointClose(
    [in] HANDLE hFindVolumeMountPoint
);
```

Parameters

[in] *hFindVolumeMountPoint*

The mounted folder search handle to be closed. This handle must have been previously opened by the [FindFirstVolumeMountPoint](#) 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

After the **FindVolumeMountPointClose** function is called, the handle *hFindVolumeMountPoint* cannot be used in subsequent calls to either [FindNextVolumeMountPoint](#) or [FindVolumeMountPointClose](#).

The [FindFirstVolumeMountPoint](#), [FindNextVolumeMountPoint](#), and [FindVolumeMountPointClose](#) functions return paths to mounted folders for a specified

volume. They do not return drive letters or volume GUID paths. For information about enumerating the volume GUID paths for a volume, see [Enumerating Volume GUID Paths](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	Yes

SMB does not support volume management functions. CsvFS does not support adding mount point on a CSV volume.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FindFirstVolumeMountPoint](#)

[FindNextVolumeMountPoint](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FormatMessage function (winbase.h)

Article09/22/2022

Formats a message string. The function requires a message definition as input. The message definition can come from a buffer passed into the function. It can come from a message table resource in an already-loaded module. Or the caller can ask the function to search the system's message table resource(s) for the message definition. The function finds the message definition in a message table resource based on a message identifier and a language identifier. The function copies the formatted message text to an output buffer, processing any embedded insert sequences if requested.

Syntax

C++

```
DWORD FormatMessage(
    [in]           DWORD   dwFlags,
    [in, optional] LPCVOID lpSource,
    [in]           DWORD   dwMessageId,
    [in]           DWORD   dwLanguageId,
    [out]          LPTSTR  lpBuffer,
    [in]           DWORD   nSize,
    [in, optional] va_list *Arguments
);
```

Parameters

[in] dwFlags

The formatting options, and how to interpret the *lpSource* parameter. The low-order byte of *dwFlags* specifies how the function handles line breaks in the output buffer. The low-order byte can also specify the maximum width of a formatted output line.

This parameter can be one or more of the following values.

Value	Meaning
FORMAT_MESSAGE_ALLOCATE_BUFFER 0x00000100	The function allocates a buffer large enough to hold the formatted message, and places a pointer to the allocated buffer at the address specified by <i>lpBuffer</i> . The <i>lpBuffer</i> parameter is a pointer to an LPTSTR ; you must cast the pointer to an LPTSTR (for example, <code>(LPTSTR)&lpBuffer</code>). The <i>nSize</i> parameter specifies the

minimum number of TCHARs to allocate for an output message buffer. The caller should use the [LocalFree](#) function to free the buffer when it is no longer needed.

If the length of the formatted message exceeds 128K bytes, then **FormatMessage** will fail and a subsequent call to [GetLastError](#) will return **ERROR_MORE_DATA**.

In previous versions of Windows, this value was not available for use when compiling Windows Store apps. As of Windows 10 this value can be used.

Windows Server 2003 and Windows XP:

If the length of the formatted message exceeds 128K bytes, then **FormatMessage** will not automatically fail with an error of **ERROR_MORE_DATA**.

FORMAT_MESSAGE_ARGUMENT_ARRAY 0x00002000	The <i>Arguments</i> parameter is not a va_list structure, but is a pointer to an array of values that represent the arguments.
--	--

This flag cannot be used with 64-bit integer values. If you are using a 64-bit integer, you must use the **va_list** structure.

FORMAT_MESSAGE_FROM_HMODULE 0x00000800	The <i>lpSource</i> parameter is a module handle containing the message-table resource(s) to search. If this <i>lpSource</i> handle is NULL , the current process's application image file will be searched. This flag cannot be used with FORMAT_MESSAGE_FROM_STRING . If the module has no message table resource, the function fails with ERROR_RESOURCE_TYPE_NOT_FOUND .
--	--

FORMAT_MESSAGE_FROM_STRING 0x00000400	The <i>lpSource</i> parameter is a pointer to a null-terminated string that contains a message definition. The message definition may contain insert sequences, just as the message text in a message table resource may. This flag cannot be used with FORMAT_MESSAGE_FROM_HMODULE or FORMAT_MESSAGE_FROM_SYSTEM .
---	---

FORMAT_MESSAGE_FROM_SYSTEM 0x00001000	The function should search the system message-table resource(s) for the requested message. If this flag is specified with FORMAT_MESSAGE_FROM_HMODULE , the function searches the system message table if the message is not found in the module specified by
---	--

lpSource. This flag cannot be used with **FORMAT_MESSAGE_FROM_STRING**.

If this flag is specified, an application can pass the result of the [GetLastError](#) function to retrieve the message text for a system-defined error.

FORMAT_MESSAGE_IGNORE_INSERTS
0x00000200

Insert sequences in the message definition such as %1 are to be ignored and passed through to the output buffer unchanged. This flag is useful for fetching a message for later formatting. If this flag is set, the *Arguments* parameter is ignored.

The low-order byte of *dwFlags* can specify the maximum width of a formatted output line. The following are possible values of the low-order byte.

Value	Meaning
0	There are no output line width restrictions. The function stores line breaks that are in the message definition text into the output buffer.
FORMAT_MESSAGE_MAX_WIDTH_MASK 0x000000FF	The function ignores regular line breaks in the message definition text. The function stores hard-coded line breaks in the message definition text into the output buffer. The function generates no new line breaks.

If the low-order byte is a nonzero value other than **FORMAT_MESSAGE_MAX_WIDTH_MASK**, it specifies the maximum number of characters in an output line. The function ignores regular line breaks in the message definition text. The function never splits a string delimited by white space across a line break. The function stores hard-coded line breaks in the message definition text into the output buffer. Hard-coded line breaks are coded with the %n escape sequence.

[in, optional] *lpSource*

The location of the message definition. The type of this parameter depends upon the settings in the *dwFlags* parameter.

<i>dwFlags</i> Setting	Meaning
FORMAT_MESSAGE_FROM_HMODULE 0x00000800	A handle to the module that contains the message table to search.

FORMAT_MESSAGE_FROM_STRING	Pointer to a string that consists of unformatted message text. It will be scanned for inserts and formatted accordingly.
0x00000400	

If neither of these flags is set in *dwFlags*, then *lpSource* is ignored.

[in] dwMessageId

The message identifier for the requested message. This parameter is ignored if *dwFlags* includes **FORMAT_MESSAGE_FROM_STRING**.

[in] dwLanguageId

The [language identifier](#) for the requested message. This parameter is ignored if *dwFlags* includes **FORMAT_MESSAGE_FROM_STRING**.

If you pass a specific **LANGID** in this parameter, **FormatMessage** will return a message for that **LANGID** only. If the function cannot find a message for that **LANGID**, it sets Last-Error to **ERROR_RESOURCE_LANG_NOT_FOUND**. If you pass in zero, **FormatMessage** looks for a message for **LANGIDs** in the following order:

1. Language neutral
2. Thread **LANGID**, based on the thread's locale value
3. User default **LANGID**, based on the user's default locale value
4. System default **LANGID**, based on the system default locale value
5. US English

If **FormatMessage** does not locate a message for any of the preceding **LANGIDs**, it returns any language message string that is present. If that fails, it returns **ERROR_RESOURCE_LANG_NOT_FOUND**.

[out] lpBuffer

A pointer to a buffer that receives the null-terminated string that specifies the formatted message. If *dwFlags* includes **FORMAT_MESSAGE_ALLOCATE_BUFFER**, the function allocates a buffer using the [LocalAlloc](#) function, and places the pointer to the buffer at the address specified in *lpBuffer*.

This buffer cannot be larger than 64K bytes.

[in] nSize

If the **FORMAT_MESSAGE_ALLOCATE_BUFFER** flag is not set, this parameter specifies the size of the output buffer, in **TCHARs**. If **FORMAT_MESSAGE_ALLOCATE_BUFFER** is

set, this parameter specifies the minimum number of TCHARs to allocate for an output buffer.

The output buffer cannot be larger than 64K bytes.

[in, optional] Arguments

An array of values that are used as insert values in the formatted message. A %1 in the format string indicates the first value in the *Arguments* array; a %2 indicates the second argument; and so on.

The interpretation of each value depends on the formatting information associated with the insert in the message definition. The default is to treat each value as a pointer to a null-terminated string.

By default, the *Arguments* parameter is of type **va_list***, which is a language- and implementation-specific data type for describing a variable number of arguments. The state of the **va_list** argument is undefined upon return from the function. To use the **va_list** again, destroy the variable argument list pointer using **va_end** and reinitialize it with **va_start**.

If you do not have a pointer of type **va_list***, then specify the **FORMAT_MESSAGE_ARGUMENT_ARRAY** flag and pass a pointer to an array of **DWORD_PTR** values; those values are input to the message formatted as the insert values. Each insert must have a corresponding element in the array.

Return value

If the function succeeds, the return value is the number of TCHARs stored in the output buffer, excluding the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Within the message text, several escape sequences are supported for dynamically formatting the message. These escape sequences and their meanings are shown in the following tables. All escape sequences start with the percent character (%).

Escape sequence	Meaning

%0	Terminates a message text line without a trailing new line character. This escape sequence can be used to build up long lines or to terminate the message itself without a trailing new line character. It is useful for prompt messages.
%n!format string!	<p>Identifies an insert sequence. The value of <i>n</i> can be in the range from 1 through 99. The format string (which must be surrounded by exclamation marks) is optional and defaults to <code>!s!</code> if not specified. For more information, see Format Specification Fields.</p> <p>The format string can include a width and precision specifier for strings and a width specifier for integers. Use an asterisk (<code>*</code>) to specify the width and precision. For example, <code>%1!*s!</code> or <code>%1!*u!</code>.</p> <p>If you do not use the width and precision specifiers, the insert numbers correspond directly to the input arguments. For example, if the source string is <code>"%1 %2 %1"</code> and the input arguments are "Bill" and "Bob", the formatted output string is "Bill Bob Bill".</p> <p>However, if you use a width and precision specifier, the insert numbers do not correspond directly to the input arguments. For example, the insert numbers for the previous example could change to <code>"%1!*.*s! %4 %5!*s!"</code>.</p> <p>The insert numbers depend on whether you use an arguments array (<code>FORMAT_MESSAGE_ARGUMENT_ARRAY</code>) or a <code>va_list</code>. For an arguments array, the next insert number is <i>n</i>+2 if the previous format string contained one asterisk and is <i>n</i>+3 if two asterisks were specified. For a <code>va_list</code>, the next insert number is <i>n</i>+1 if the previous format string contained one asterisk and is <i>n</i>+2 if two asterisks were specified.</p> <p>If you want to repeat "Bill", as in the previous example, the arguments must include "Bill" twice. For example, if the source string is <code>"%1!*.*s! %4 %5!*s!"</code>, the arguments could be, 4, 2, Bill, Bob, 6, Bill (if using the <code>FORMAT_MESSAGE_ARGUMENT_ARRAY</code> flag). The formatted string would then be " Bi Bob Bill".</p> <p>Repeating insert numbers when the source string contains width and precision specifiers may not yield the intended results. If you replaced <code>%5</code> with <code>%1</code>, the function would try to print a string at address 6 (likely resulting in an access violation).</p> <p>Floating-point format specifiers—<code>e</code>, <code>E</code>, <code>f</code>, and <code>g</code>—are not supported. The workaround is to use the StringCchPrintf function to format the floating-point number into a temporary buffer, then use that buffer as the insert string.</p> <p>Inserts that use the <code>l64</code> prefix are treated as two 32-bit arguments. They must be used before subsequent arguments are used. Note that it may be easier for you to use StringCchPrintf instead of this prefix.</p>

Any other nondigit character following a percent character is formatted in the output message without the percent character. Following are some examples.

Format	Resulting output
string	
%%	A single percent sign.
%b	A single space. This format string can be used to ensure the appropriate number of trailing spaces in a message text line.
%.	A single period. This format string can be used to include a single period at the beginning of a line without terminating the message text definition.
%!	A single exclamation point. This format string can be used to include an exclamation point immediately after an insert without its being mistaken for the beginning of a format string.
%n	A hard line break when the format string occurs at the end of a line. This format string is useful when FormatMessage is supplying regular line breaks so the message fits in a certain width.
%r	A hard carriage return without a trailing newline character.
%t	A single tab.

Security Remarks

If this function is called without **FORMAT_MESSAGE_IGNORE_INSERTS**, the *Arguments* parameter must contain enough parameters to satisfy all insertion sequences in the message string, and they must be of the correct type. Therefore, do not use untrusted or unknown message strings with inserts enabled because they can contain more insertion sequences than *Arguments* provides, or those that may be of the wrong type. In particular, it is unsafe to take an arbitrary system error code returned from an API and use **FORMAT_MESSAGE_FROM_SYSTEM** without **FORMAT_MESSAGE_IGNORE_INSERTS**.

Examples

The **FormatMessage** function can be used to obtain error message strings for the system error codes returned by [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#).

The following example shows how to use an argument array and the width and precision specifiers.

C++

```

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

void main(void)
{
    LPWSTR pMessage = L"%1!.*s! %4 %5!*s!";
    DWORD_PTR pArgs[] = { (DWORD_PTR)4, (DWORD_PTR)2, (DWORD_PTR)L"Bill",
// %1!.*s! refers back to the first insertion string in pMessage
// %4 refers back to the second insertion string in pMessage
// %5!*s! refers back to the third insertion string in pMessage
        (DWORD_PTR)L"Bob",
        (DWORD_PTR)6, (DWORD_PTR)L"Bill" };
    const DWORD size = 100+1;
    WCHAR buffer[size];

    if (!FormatMessage(FORMAT_MESSAGE_FROM_STRING |
FORMAT_MESSAGE_ARGUMENT_ARRAY,
                    pMessage,
                    0,
                    0,
                    buffer,
                    size,
                    (va_list*)pArgs))
    {
        wprintf(L"Format message failed with 0x%x\n", GetLastError());
        return;
    }

    // Buffer contains " Bi Bob Bill".
    wprintf(L"Formatted message: %s\n", buffer);
}

```

The following example shows how to implement the previous example using `va_list`.

C++

```

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

LPWSTR GetFormattedMessage(LPWSTR pMessage, ...);

```

```

void main(void)
{
    LPWSTR pBuffer = NULL;
    LPWSTR pMessage = L"%1!*.*s! %3 %4!*s!";
    // The variable length arguments correspond directly to the format
    // strings in pMessage.
    pBuffer = GetFormattedMessage(pMessage, 4, 2, L"Bill", L"Bob", 6,
        L"Bill");
    if (pBuffer)
    {
        // Buffer contains " Bi Bob Bill".
        wprintf(L"Formatted message: %s\n", pBuffer);
        LocalFree(pBuffer);
    }
    else
    {
        wprintf(L"Format message failed with 0x%x\n", GetLastError());
    }
}

// Formats a message string using the specified message and variable
// list of arguments.
LPWSTR GetFormattedMessage(LPWSTR pMessage, ...)
{
    LPWSTR pBuffer = NULL;

    va_list args = NULL;
    va_start(args, pMessage);

    FormatMessage(FORMAT_MESSAGE_FROM_STRING |
                  FORMAT_MESSAGE_ALLOCATE_BUFFER,
                  pMessage,
                  0,
                  0,
                  (LPWSTR)&pBuffer,
                  0,
                  &args);

    va_end(args);

    return pBuffer;
}

```

Requirements

Minimum supported client

Windows XP [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Error Handling Functions](#)

[Message Compiler](#)

[Message Tables](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FormatMessageA function (winbase.h)

Article 02/09/2023

Formats a message string. The function requires a message definition as input. The message definition can come from a buffer passed into the function. It can come from a message table resource in an already-loaded module. Or the caller can ask the function to search the system's message table resource(s) for the message definition. The function finds the message definition in a message table resource based on a message identifier and a language identifier. The function copies the formatted message text to an output buffer, processing any embedded insert sequences if requested.

Syntax

C++

```
DWORD FormatMessageA(
    [in]          DWORD   dwFlags,
    [in, optional] LPCVOID lpSource,
    [in]          DWORD   dwMessageId,
    [in]          DWORD   dwLanguageId,
    [out]         LPSTR   lpBuffer,
    [in]          DWORD   nSize,
    [in, optional] va_list *Arguments
);
```

Parameters

[in] dwFlags

The formatting options, and how to interpret the *lpSource* parameter. The low-order byte of *dwFlags* specifies how the function handles line breaks in the output buffer. The low-order byte can also specify the maximum width of a formatted output line.

This parameter can be one or more of the following values.

Value	Meaning
FORMAT_MESSAGE_ALLOCATE_BUFFER 0x00000100	The function allocates a buffer large enough to hold the formatted message, and places a pointer to the allocated buffer at the address specified by <i>lpBuffer</i> . The <i>lpBuffer</i> parameter is a pointer to an LPTSTR ; you must cast the pointer to an LPTSTR (for example, <code>(LPTSTR)&lpBuffer</code>). The <i>nSize</i> parameter specifies the

minimum number of TCHARs to allocate for an output message buffer. The caller should use the [LocalFree](#) function to free the buffer when it is no longer needed.

If the length of the formatted message exceeds 128K bytes, then **FormatMessage** will fail and a subsequent call to [GetLastError](#) will return **ERROR_MORE_DATA**.

In previous versions of Windows, this value was not available for use when compiling Windows Store apps. As of Windows 10 this value can be used.

Windows Server 2003 and Windows XP:

If the length of the formatted message exceeds 128K bytes, then **FormatMessage** will not automatically fail with an error of **ERROR_MORE_DATA**.

FORMAT_MESSAGE_ARGUMENT_ARRAY 0x00002000	The <i>Arguments</i> parameter is not a va_list structure, but is a pointer to an array of values that represent the arguments.
--	--

This flag cannot be used with 64-bit integer values. If you are using a 64-bit integer, you must use the **va_list** structure.

FORMAT_MESSAGE_FROM_HMODULE 0x00000800	The <i>lpSource</i> parameter is a module handle containing the message-table resource(s) to search. If this <i>lpSource</i> handle is NULL , the current process's application image file will be searched. This flag cannot be used with FORMAT_MESSAGE_FROM_STRING . If the module has no message table resource, the function fails with ERROR_RESOURCE_TYPE_NOT_FOUND .
--	--

FORMAT_MESSAGE_FROM_STRING 0x00000400	The <i>lpSource</i> parameter is a pointer to a null-terminated string that contains a message definition. The message definition may contain insert sequences, just as the message text in a message table resource may. This flag cannot be used with FORMAT_MESSAGE_FROM_HMODULE or FORMAT_MESSAGE_FROM_SYSTEM .
---	---

FORMAT_MESSAGE_FROM_SYSTEM 0x00001000	The function should search the system message-table resource(s) for the requested message. If this flag is specified with FORMAT_MESSAGE_FROM_HMODULE , the function searches the system message table if the message is not found in the module specified by
---	--

lpSource. This flag cannot be used with **FORMAT_MESSAGE_FROM_STRING**.

If this flag is specified, an application can pass the result of the [GetLastError](#) function to retrieve the message text for a system-defined error.

FORMAT_MESSAGE_IGNORE_INSERTS 0x00000200	Insert sequences in the message definition such as %1 are to be ignored and passed through to the output buffer unchanged. This flag is useful for fetching a message for later formatting. If this flag is set, the <i>Arguments</i> parameter is ignored.
--	---

The low-order byte of *dwFlags* can specify the maximum width of a formatted output line. The following are possible values of the low-order byte.

Value	Meaning
0	There are no output line width restrictions. The function stores line breaks that are in the message definition text into the output buffer.
FORMAT_MESSAGE_MAX_WIDTH_MASK 0x000000FF	The function ignores regular line breaks in the message definition text. The function stores hard-coded line breaks in the message definition text into the output buffer. The function generates no new line breaks.

If the low-order byte is a nonzero value other than **FORMAT_MESSAGE_MAX_WIDTH_MASK**, it specifies the maximum number of characters in an output line. The function ignores regular line breaks in the message definition text. The function never splits a string delimited by white space across a line break. The function stores hard-coded line breaks in the message definition text into the output buffer. Hard-coded line breaks are coded with the %n escape sequence.

[in, optional] *lpSource*

The location of the message definition. The type of this parameter depends upon the settings in the *dwFlags* parameter.

<i>dwFlags</i> Setting	Meaning
FORMAT_MESSAGE_FROM_HMODULE 0x00000800	A handle to the module that contains the message table to search.

FORMAT_MESSAGE_FROM_STRING 0x00000400	Pointer to a string that consists of unformatted message text. It will be scanned for inserts and formatted accordingly.
---	--

If neither of these flags is set in *dwFlags*, then *lpSource* is ignored.

[in] dwMessageId

The message identifier for the requested message. This parameter is ignored if *dwFlags* includes **FORMAT_MESSAGE_FROM_STRING**.

[in] dwLanguageId

The [language identifier](#) for the requested message. This parameter is ignored if *dwFlags* includes **FORMAT_MESSAGE_FROM_STRING**.

If you pass a specific **LANGID** in this parameter, **FormatMessage** will return a message for that **LANGID** only. If the function cannot find a message for that **LANGID**, it sets Last-Error to **ERROR_RESOURCE_LANG_NOT_FOUND**. If you pass in zero, **FormatMessage** looks for a message for **LANGIDs** in the following order:

1. Language neutral
2. Thread **LANGID**, based on the thread's locale value
3. User default **LANGID**, based on the user's default locale value
4. System default **LANGID**, based on the system default locale value
5. US English

If **FormatMessage** does not locate a message for any of the preceding **LANGIDs**, it returns any language message string that is present. If that fails, it returns **ERROR_RESOURCE_LANG_NOT_FOUND**.

[out] lpBuffer

A pointer to a buffer that receives the null-terminated string that specifies the formatted message. If *dwFlags* includes **FORMAT_MESSAGE_ALLOCATE_BUFFER**, the function allocates a buffer using the [LocalAlloc](#) function, and places the pointer to the buffer at the address specified in *lpBuffer*.

This buffer cannot be larger than 64K bytes.

[in] nSize

If the **FORMAT_MESSAGE_ALLOCATE_BUFFER** flag is not set, this parameter specifies the size of the output buffer, in **TCHARs**. If **FORMAT_MESSAGE_ALLOCATE_BUFFER** is

set, this parameter specifies the minimum number of TCHARs to allocate for an output buffer.

The output buffer cannot be larger than 64K bytes.

[in, optional] Arguments

An array of values that are used as insert values in the formatted message. A %1 in the format string indicates the first value in the *Arguments* array; a %2 indicates the second argument; and so on.

The interpretation of each value depends on the formatting information associated with the insert in the message definition. The default is to treat each value as a pointer to a null-terminated string.

By default, the *Arguments* parameter is of type **va_list***, which is a language- and implementation-specific data type for describing a variable number of arguments. The state of the **va_list** argument is undefined upon return from the function. To use the **va_list** again, destroy the variable argument list pointer using **va_end** and reinitialize it with **va_start**.

If you do not have a pointer of type **va_list***, then specify the **FORMAT_MESSAGE_ARGUMENT_ARRAY** flag and pass a pointer to an array of **DWORD_PTR** values; those values are input to the message formatted as the insert values. Each insert must have a corresponding element in the array.

Return value

If the function succeeds, the return value is the number of TCHARs stored in the output buffer, excluding the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Within the message text, several escape sequences are supported for dynamically formatting the message. These escape sequences and their meanings are shown in the following tables. All escape sequences start with the percent character (%).

Escape sequence	Meaning

%0	Terminates a message text line without a trailing new line character. This escape sequence can be used to build up long lines or to terminate the message itself without a trailing new line character. It is useful for prompt messages.
%n!format string!	<p>Identifies an insert sequence. The value of <i>n</i> can be in the range from 1 through 99. The format string (which must be surrounded by exclamation marks) is optional and defaults to <code>!s!</code> if not specified. For more information, see Format Specification Fields.</p> <p>The format string can include a width and precision specifier for strings and a width specifier for integers. Use an asterisk (<code>*</code>) to specify the width and precision. For example, <code>%1!*s!</code> or <code>%1!*u!</code>.</p> <p>If you do not use the width and precision specifiers, the insert numbers correspond directly to the input arguments. For example, if the source string is <code>"%1 %2 %1"</code> and the input arguments are "Bill" and "Bob", the formatted output string is "Bill Bob Bill".</p> <p>However, if you use a width and precision specifier, the insert numbers do not correspond directly to the input arguments. For example, the insert numbers for the previous example could change to <code>"%1!*.*s! %4 %5!*s!"</code>.</p> <p>The insert numbers depend on whether you use an arguments array (<code>FORMAT_MESSAGE_ARGUMENT_ARRAY</code>) or a <code>va_list</code>. For an arguments array, the next insert number is <i>n</i>+2 if the previous format string contained one asterisk and is <i>n</i>+3 if two asterisks were specified. For a <code>va_list</code>, the next insert number is <i>n</i>+1 if the previous format string contained one asterisk and is <i>n</i>+2 if two asterisks were specified.</p> <p>If you want to repeat "Bill", as in the previous example, the arguments must include "Bill" twice. For example, if the source string is <code>"%1!*.*s! %4 %5!*s!"</code>, the arguments could be, 4, 2, Bill, Bob, 6, Bill (if using the <code>FORMAT_MESSAGE_ARGUMENT_ARRAY</code> flag). The formatted string would then be " Bi Bob Bill".</p> <p>Repeating insert numbers when the source string contains width and precision specifiers may not yield the intended results. If you replaced <code>%5</code> with <code>%1</code>, the function would try to print a string at address 6 (likely resulting in an access violation).</p> <p>Floating-point format specifiers—<code>e</code>, <code>E</code>, <code>f</code>, and <code>g</code>—are not supported. The workaround is to use the StringCchPrintf function to format the floating-point number into a temporary buffer, then use that buffer as the insert string.</p> <p>Inserts that use the <code>l64</code> prefix are treated as two 32-bit arguments. They must be used before subsequent arguments are used. Note that it may be easier for you to use StringCchPrintf instead of this prefix.</p>

Any other nondigit character following a percent character is formatted in the output message without the percent character. Following are some examples.

Format	Resulting output
string	
%%	A single percent sign.
%space	A single space. This format string can be used to ensure the appropriate number of trailing spaces in a message text line.
%.	A single period. This format string can be used to include a single period at the beginning of a line without terminating the message text definition.
%!	A single exclamation point. This format string can be used to include an exclamation point immediately after an insert without its being mistaken for the beginning of a format string.
%n	A hard line break when the format string occurs at the end of a line. This format string is useful when FormatMessage is supplying regular line breaks so the message fits in a certain width.
%r	A hard carriage return without a trailing newline character.
%t	A single tab.

Security Remarks

If this function is called without **FORMAT_MESSAGE_IGNORE_INSERTS**, the *Arguments* parameter must contain enough parameters to satisfy all insertion sequences in the message string, and they must be of the correct type. Therefore, do not use untrusted or unknown message strings with inserts enabled because they can contain more insertion sequences than *Arguments* provides, or those that may be of the wrong type. In particular, it is unsafe to take an arbitrary system error code returned from an API and use **FORMAT_MESSAGE_FROM_SYSTEM** without **FORMAT_MESSAGE_IGNORE_INSERTS**.

Examples

The **FormatMessage** function can be used to obtain error message strings for the system error codes returned by [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#).

The following example shows how to use an argument array and the width and precision specifiers.

C++

```

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

void main(void)
{
    LPWSTR pMessage = L"%1!.*s! %4 %5!*s!";
    DWORD_PTR pArgs[] = { (DWORD_PTR)4, (DWORD_PTR)2, (DWORD_PTR)L"Bill",
// %1!.*s! refers back to the first insertion string in pMessage
        (DWORD_PTR)L"Bob",
// %4 refers back to the second insertion string in pMessage
        (DWORD_PTR)6, (DWORD_PTR)L"Bill" };
// %5!*s! refers back to the third insertion string in pMessage
    const DWORD size = 100+1;
    WCHAR buffer[size];

    if (!FormatMessage(FORMAT_MESSAGE_FROM_STRING |
FORMAT_MESSAGE_ARGUMENT_ARRAY,
                    pMessage,
                    0,
                    0,
                    buffer,
                    size,
                    (va_list*)pArgs))
    {
        wprintf(L"Format message failed with 0x%x\n", GetLastError());
        return;
    }

    // Buffer contains " Bi Bob Bill".
    wprintf(L"Formatted message: %s\n", buffer);
}

```

The following example shows how to implement the previous example using `va_list`.

C++

```

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

LPWSTR GetFormattedMessage(LPWSTR pMessage, ...);

```

```

void main(void)
{
    LPWSTR pBuffer = NULL;
    LPWSTR pMessage = L"%1!*.*s! %3 %4!*s!";
    // The variable length arguments correspond directly to the format
    // strings in pMessage.
    pBuffer = GetFormattedMessage(pMessage, 4, 2, L"Bill", L"Bob", 6,
        L"Bill");
    if (pBuffer)
    {
        // Buffer contains " Bi Bob Bill".
        wprintf(L"Formatted message: %s\n", pBuffer);
        LocalFree(pBuffer);
    }
    else
    {
        wprintf(L"Format message failed with 0x%x\n", GetLastError());
    }
}

// Formats a message string using the specified message and variable
// list of arguments.
LPWSTR GetFormattedMessage(LPWSTR pMessage, ...)
{
    LPWSTR pBuffer = NULL;

    va_list args = NULL;
    va_start(args, pMessage);

    FormatMessage(FORMAT_MESSAGE_FROM_STRING |
                  FORMAT_MESSAGE_ALLOCATE_BUFFER,
                  pMessage,
                  0,
                  0,
                  (LPWSTR)&pBuffer,
                  0,
                  &args);

    va_end(args);

    return pBuffer;
}

```

ⓘ Note

The winbase.h header defines FormatMessage as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with

code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

System Requirements	
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Error Handling Functions](#)

[Message Compiler](#)

[Message Tables](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

FormatMessageW function (winbase.h)

Article 02/09/2023

Formats a message string. The function requires a message definition as input. The message definition can come from a buffer passed into the function. It can come from a message table resource in an already-loaded module. Or the caller can ask the function to search the system's message table resource(s) for the message definition. The function finds the message definition in a message table resource based on a message identifier and a language identifier. The function copies the formatted message text to an output buffer, processing any embedded insert sequences if requested.

Syntax

C++

```
DWORD FormatMessageW(
    [in]           DWORD   dwFlags,
    [in, optional] LPCVOID lpSource,
    [in]           DWORD   dwMessageId,
    [in]           DWORD   dwLanguageId,
    [out]          LPWSTR  lpBuffer,
    [in]           DWORD   nSize,
    [in, optional] va_list *Arguments
);
```

Parameters

[in] dwFlags

The formatting options, and how to interpret the *lpSource* parameter. The low-order byte of *dwFlags* specifies how the function handles line breaks in the output buffer. The low-order byte can also specify the maximum width of a formatted output line.

This parameter can be one or more of the following values.

Value	Meaning
FORMAT_MESSAGE_ALLOCATE_BUFFER 0x00000100	The function allocates a buffer large enough to hold the formatted message, and places a pointer to the allocated buffer at the address specified by <i>lpBuffer</i> . The <i>lpBuffer</i> parameter is a pointer to an LPTSTR ; you must cast the pointer to an LPTSTR (for example, <code>(LPTSTR)&lpBuffer</code>). The <i>nSize</i> parameter specifies the

minimum number of TCHARs to allocate for an output message buffer. The caller should use the [LocalFree](#) function to free the buffer when it is no longer needed.

If the length of the formatted message exceeds 128K bytes, then **FormatMessage** will fail and a subsequent call to [GetLastError](#) will return **ERROR_MORE_DATA**.

In previous versions of Windows, this value was not available for use when compiling Windows Store apps. As of Windows 10 this value can be used.

Windows Server 2003 and Windows XP:

If the length of the formatted message exceeds 128K bytes, then **FormatMessage** will not automatically fail with an error of **ERROR_MORE_DATA**.

FORMAT_MESSAGE_ARGUMENT_ARRAY 0x00002000	The <i>Arguments</i> parameter is not a va_list structure, but is a pointer to an array of values that represent the arguments.
--	--

This flag cannot be used with 64-bit integer values. If you are using a 64-bit integer, you must use the **va_list** structure.

FORMAT_MESSAGE_FROM_HMODULE 0x00000800	The <i>lpSource</i> parameter is a module handle containing the message-table resource(s) to search. If this <i>lpSource</i> handle is NULL , the current process's application image file will be searched. This flag cannot be used with FORMAT_MESSAGE_FROM_STRING . If the module has no message table resource, the function fails with ERROR_RESOURCE_TYPE_NOT_FOUND .
--	--

FORMAT_MESSAGE_FROM_STRING 0x00000400	The <i>lpSource</i> parameter is a pointer to a null-terminated string that contains a message definition. The message definition may contain insert sequences, just as the message text in a message table resource may. This flag cannot be used with FORMAT_MESSAGE_FROM_HMODULE or FORMAT_MESSAGE_FROM_SYSTEM .
---	---

FORMAT_MESSAGE_FROM_SYSTEM 0x00001000	The function should search the system message-table resource(s) for the requested message. If this flag is specified with FORMAT_MESSAGE_FROM_HMODULE , the function searches the system message table if the message is not found in the module specified by
---	--

lpSource. This flag cannot be used with **FORMAT_MESSAGE_FROM_STRING**.

If this flag is specified, an application can pass the result of the [GetLastError](#) function to retrieve the message text for a system-defined error.

FORMAT_MESSAGE_IGNORE_INSERTS 0x00000200	Insert sequences in the message definition such as %1 are to be ignored and passed through to the output buffer unchanged. This flag is useful for fetching a message for later formatting. If this flag is set, the <i>Arguments</i> parameter is ignored.
--	---

The low-order byte of *dwFlags* can specify the maximum width of a formatted output line. The following are possible values of the low-order byte.

Value	Meaning
0	There are no output line width restrictions. The function stores line breaks that are in the message definition text into the output buffer.
FORMAT_MESSAGE_MAX_WIDTH_MASK 0x000000FF	The function ignores regular line breaks in the message definition text. The function stores hard-coded line breaks in the message definition text into the output buffer. The function generates no new line breaks.

If the low-order byte is a nonzero value other than **FORMAT_MESSAGE_MAX_WIDTH_MASK**, it specifies the maximum number of characters in an output line. The function ignores regular line breaks in the message definition text. The function never splits a string delimited by white space across a line break. The function stores hard-coded line breaks in the message definition text into the output buffer. Hard-coded line breaks are coded with the %n escape sequence.

[in, optional] *lpSource*

The location of the message definition. The type of this parameter depends upon the settings in the *dwFlags* parameter.

<i>dwFlags</i> Setting	Meaning
FORMAT_MESSAGE_FROM_HMODULE 0x00000800	A handle to the module that contains the message table to search.

FORMAT_MESSAGE_FROM_STRING	Pointer to a string that consists of unformatted message text. It will be scanned for inserts and formatted accordingly.
0x00000400	

If neither of these flags is set in *dwFlags*, then *lpSource* is ignored.

[in] dwMessageId

The message identifier for the requested message. This parameter is ignored if *dwFlags* includes **FORMAT_MESSAGE_FROM_STRING**.

[in] dwLanguageId

The [language identifier](#) for the requested message. This parameter is ignored if *dwFlags* includes **FORMAT_MESSAGE_FROM_STRING**.

If you pass a specific **LANGID** in this parameter, **FormatMessage** will return a message for that **LANGID** only. If the function cannot find a message for that **LANGID**, it sets Last-Error to **ERROR_RESOURCE_LANG_NOT_FOUND**. If you pass in zero, **FormatMessage** looks for a message for **LANGIDs** in the following order:

1. Language neutral
2. Thread **LANGID**, based on the thread's locale value
3. User default **LANGID**, based on the user's default locale value
4. System default **LANGID**, based on the system default locale value
5. US English

If **FormatMessage** does not locate a message for any of the preceding **LANGIDs**, it returns any language message string that is present. If that fails, it returns **ERROR_RESOURCE_LANG_NOT_FOUND**.

[out] lpBuffer

A pointer to a buffer that receives the null-terminated string that specifies the formatted message. If *dwFlags* includes **FORMAT_MESSAGE_ALLOCATE_BUFFER**, the function allocates a buffer using the [LocalAlloc](#) function, and places the pointer to the buffer at the address specified in *lpBuffer*.

This buffer cannot be larger than 64K bytes.

[in] nSize

If the **FORMAT_MESSAGE_ALLOCATE_BUFFER** flag is not set, this parameter specifies the size of the output buffer, in **TCHARs**. If **FORMAT_MESSAGE_ALLOCATE_BUFFER** is

set, this parameter specifies the minimum number of TCHARs to allocate for an output buffer.

The output buffer cannot be larger than 64K bytes.

[in, optional] Arguments

An array of values that are used as insert values in the formatted message. A %1 in the format string indicates the first value in the *Arguments* array; a %2 indicates the second argument; and so on.

The interpretation of each value depends on the formatting information associated with the insert in the message definition. The default is to treat each value as a pointer to a null-terminated string.

By default, the *Arguments* parameter is of type **va_list***, which is a language- and implementation-specific data type for describing a variable number of arguments. The state of the **va_list** argument is undefined upon return from the function. To use the **va_list** again, destroy the variable argument list pointer using **va_end** and reinitialize it with **va_start**.

If you do not have a pointer of type **va_list***, then specify the **FORMAT_MESSAGE_ARGUMENT_ARRAY** flag and pass a pointer to an array of **DWORD_PTR** values; those values are input to the message formatted as the insert values. Each insert must have a corresponding element in the array.

Return value

If the function succeeds, the return value is the number of TCHARs stored in the output buffer, excluding the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Within the message text, several escape sequences are supported for dynamically formatting the message. These escape sequences and their meanings are shown in the following tables. All escape sequences start with the percent character (%).

Escape sequence	Meaning

%0	Terminates a message text line without a trailing new line character. This escape sequence can be used to build up long lines or to terminate the message itself without a trailing new line character. It is useful for prompt messages.
%n!format string!	<p>Identifies an insert sequence. The value of <i>n</i> can be in the range from 1 through 99. The format string (which must be surrounded by exclamation marks) is optional and defaults to <code>!s!</code> if not specified. For more information, see Format Specification Fields.</p> <p>The format string can include a width and precision specifier for strings and a width specifier for integers. Use an asterisk (<code>*</code>) to specify the width and precision. For example, <code>%1!*s!</code> or <code>%1!*u!</code>.</p> <p>If you do not use the width and precision specifiers, the insert numbers correspond directly to the input arguments. For example, if the source string is <code>"%1 %2 %1"</code> and the input arguments are "Bill" and "Bob", the formatted output string is "Bill Bob Bill".</p> <p>However, if you use a width and precision specifier, the insert numbers do not correspond directly to the input arguments. For example, the insert numbers for the previous example could change to <code>"%1!*.*s! %4 %5!*s!"</code>.</p> <p>The insert numbers depend on whether you use an arguments array (<code>FORMAT_MESSAGE_ARGUMENT_ARRAY</code>) or a <code>va_list</code>. For an arguments array, the next insert number is <i>n</i>+2 if the previous format string contained one asterisk and is <i>n</i>+3 if two asterisks were specified. For a <code>va_list</code>, the next insert number is <i>n</i>+1 if the previous format string contained one asterisk and is <i>n</i>+2 if two asterisks were specified.</p> <p>If you want to repeat "Bill", as in the previous example, the arguments must include "Bill" twice. For example, if the source string is <code>"%1!*.*s! %4 %5!*s!"</code>, the arguments could be, 4, 2, Bill, Bob, 6, Bill (if using the <code>FORMAT_MESSAGE_ARGUMENT_ARRAY</code> flag). The formatted string would then be " Bi Bob Bill".</p> <p>Repeating insert numbers when the source string contains width and precision specifiers may not yield the intended results. If you replaced <code>%5</code> with <code>%1</code>, the function would try to print a string at address 6 (likely resulting in an access violation).</p> <p>Floating-point format specifiers—<code>e</code>, <code>E</code>, <code>f</code>, and <code>g</code>—are not supported. The workaround is to use the StringCchPrintf function to format the floating-point number into a temporary buffer, then use that buffer as the insert string.</p> <p>Inserts that use the <code>l64</code> prefix are treated as two 32-bit arguments. They must be used before subsequent arguments are used. Note that it may be easier for you to use StringCchPrintf instead of this prefix.</p>

Any other nondigit character following a percent character is formatted in the output message without the percent character. Following are some examples.

Format	Resulting output
string	
%%	A single percent sign.
%space	A single space. This format string can be used to ensure the appropriate number of trailing spaces in a message text line.
%.	A single period. This format string can be used to include a single period at the beginning of a line without terminating the message text definition.
%!	A single exclamation point. This format string can be used to include an exclamation point immediately after an insert without its being mistaken for the beginning of a format string.
%n	A hard line break when the format string occurs at the end of a line. This format string is useful when FormatMessage is supplying regular line breaks so the message fits in a certain width.
%r	A hard carriage return without a trailing newline character.
%t	A single tab.

Security Remarks

If this function is called without **FORMAT_MESSAGE_IGNORE_INSERTS**, the *Arguments* parameter must contain enough parameters to satisfy all insertion sequences in the message string, and they must be of the correct type. Therefore, do not use untrusted or unknown message strings with inserts enabled because they can contain more insertion sequences than *Arguments* provides, or those that may be of the wrong type. In particular, it is unsafe to take an arbitrary system error code returned from an API and use **FORMAT_MESSAGE_FROM_SYSTEM** without **FORMAT_MESSAGE_IGNORE_INSERTS**.

Examples

The **FormatMessage** function can be used to obtain error message strings for the system error codes returned by [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#).

The following example shows how to use an argument array and the width and precision specifiers.

C++

```

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

void main(void)
{
    LPWSTR pMessage = L"%1!.*s! %4 %5!*s!";
    DWORD_PTR pArgs[] = { (DWORD_PTR)4, (DWORD_PTR)2, (DWORD_PTR)L"Bill",
// %1!.*s! refers back to the first insertion string in pMessage
        (DWORD_PTR)L"Bob",
// %4 refers back to the second insertion string in pMessage
        (DWORD_PTR)6, (DWORD_PTR)L"Bill" };
// %5!*s! refers back to the third insertion string in pMessage
    const DWORD size = 100+1;
    WCHAR buffer[size];

    if (!FormatMessage(FORMAT_MESSAGE_FROM_STRING |
FORMAT_MESSAGE_ARGUMENT_ARRAY,
                    pMessage,
                    0,
                    0,
                    buffer,
                    size,
                    (va_list*)pArgs))
    {
        wprintf(L"Format message failed with 0x%x\n", GetLastError());
        return;
    }

    // Buffer contains " Bi Bob Bill".
    wprintf(L"Formatted message: %s\n", buffer);
}

```

The following example shows how to implement the previous example using `va_list`.

C++

```

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <stdio.h>

LPWSTR GetFormattedMessage(LPWSTR pMessage, ...);

```

```

void main(void)
{
    LPWSTR pBuffer = NULL;
    LPWSTR pMessage = L"%1!*.*s! %3 %4!*s!";
    // The variable length arguments correspond directly to the format
    // strings in pMessage.
    pBuffer = GetFormattedMessage(pMessage, 4, 2, L"Bill", L"Bob", 6,
        L"Bill");
    if (pBuffer)
    {
        // Buffer contains " Bi Bob Bill".
        wprintf(L"Formatted message: %s\n", pBuffer);
        LocalFree(pBuffer);
    }
    else
    {
        wprintf(L"Format message failed with 0x%x\n", GetLastError());
    }
}

// Formats a message string using the specified message and variable
// list of arguments.
LPWSTR GetFormattedMessage(LPWSTR pMessage, ...)
{
    LPWSTR pBuffer = NULL;

    va_list args = NULL;
    va_start(args, pMessage);

    FormatMessage(FORMAT_MESSAGE_FROM_STRING |
                  FORMAT_MESSAGE_ALLOCATE_BUFFER,
                  pMessage,
                  0,
                  0,
                  (LPWSTR)&pBuffer,
                  0,
                  &args);

    va_end(args);

    return pBuffer;
}

```

ⓘ Note

The winbase.h header defines FormatMessage as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with

code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

System Requirements	
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Error Handling Functions](#)

[Message Compiler](#)

[Message Tables](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetActiveProcessorCount function (winbase.h)

Article 10/13/2021

Returns the number of active processors in a processor group or in the system.

Syntax

C++

```
DWORD GetActiveProcessorCount(
    [in] WORD GroupNumber
);
```

Parameters

[in] GroupNumber

The processor group number. If this parameter is ALL_PROCESSOR_GROUPS, the function returns the number of active processors in the system.

Return value

If the function succeeds, the return value is the number of active processors in the specified group.

If the function fails, the return value is zero. To get extended error information, use [GetLastError](#).

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetActiveProcessorGroupCount function (winbase.h)

Article 06/29/2021

Returns the number of active processor groups in the system.

Syntax

C++

```
WORD GetActiveProcessorGroupCount();
```

Return value

If the function succeeds, the return value is the number of active processor groups in the system.

If the function fails, the return value is zero.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetApplicationRecoveryCallback function (winbase.h)

Article 10/13/2021

Retrieves a pointer to the callback routine registered for the specified process. The address returned is in the virtual address space of the process.

Syntax

C++

```
HRESULT GetApplicationRecoveryCallback(
    [in]  HANDLE                 hProcess,
    [out] APPLICATION_RECOVERY_CALLBACK *pRecoveryCallback,
    [out] PVOID                  *ppvParameter,
    [out] PDWORD                pdwPingInterval,
    [out] PDWORD                pdwFlags
);
```

Parameters

[in] hProcess

A handle to the process. This handle must have the PROCESS_VM_READ access right.

[out] pRecoveryCallback

A pointer to the recovery callback function. For more information, see [ApplicationRecoveryCallback](#).

[out] ppvParameter

A pointer to the callback parameter.

[out] pdwPingInterval

The recovery ping interval, in 100-nanosecond intervals.

[out] pdwFlags

Reserved for future use.

Return value

This function returns `S_OK` on success or one of the following error codes.

Return code	Description
<code>S_FALSE</code>	The application did not register for recovery.
<code>E_INVALIDARG</code>	One or more parameters are not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[RegisterApplicationRecoveryCallback](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetApplicationRestartSettings function (winbase.h)

Article 10/13/2021

Retrieves the restart information registered for the specified process.

Syntax

C++

```
HRESULT GetApplicationRestartSettings(
    [in]          HANDLE hProcess,
    [out, optional] PWSTR  pwzCommandline,
    [in, out]      PDWORD pcchSize,
    [out, optional] PDWORD pdwFlags
);
```

Parameters

[in] hProcess

A handle to the process. This handle must have the PROCESS_VM_READ access right.

[out, optional] pwzCommandline

A pointer to a buffer that receives the restart command line specified by the application when it called the [RegisterApplicationRestart](#) function. The maximum size of the command line, in characters, is RESTART_MAX_CMD_LINE. Can be NULL if *pcchSize* is zero.

[in, out] pcchSize

On input, specifies the size of the *pwzCommandLine* buffer, in characters.

If the buffer is not large enough to receive the command line, the function fails with `HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER)` and sets this parameter to the required buffer size, in characters.

On output, specifies the size of the buffer that was used.

To determine the required buffer size, set *pwzCommandLine* to NULL and this parameter to zero. The size includes one for the null-terminator character. Note that the function

returns S_OK, not HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER) in this case.

```
[out, optional] pdwFlags
```

A pointer to a variable that receives the flags specified by the application when it called the [RegisterApplicationRestart](#) function.

Return value

This function returns S_OK on success or one of the following error codes.

Return code	Description
E_INVALIDARG	One or more parameters are not valid.
HRESULT_FROM_WIN32(ERROR_NOT_FOUND)	The application did not register for restart.
HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER)	The <i>pwzCommandLine</i> buffer is too small. The function returns the required buffer size in <i>pcchSize</i> . Use the required size to reallocate the buffer.

Remarks

This information is available only for the current process; you cannot call this function after your program is restarted to get the restart command line. To get the command line after a restart, access the *argv* parameter of your **main** function.

Examples

The following example shows how to get the restart settings specified when you called the [RegisterApplicationRestart](#) function.

C++

```
#include <windows.h>
#include <stdio.h>

void wmain(int argc, WCHAR* argv[])
{
    HRESULT hr = S_OK;
```

```
WCHAR wsCommandLine[RESTART_MAX_CMD_LINE + 1];
DWORD cchCmdLine = sizeof(wsCommandLine) / sizeof(WCHAR);
DWORD dwFlags = 0;
LPWSTR pwsCmdLine = NULL;
UNREFERENCED_PARAMETER(argv);
UNREFERENCED_PARAMETER(argc);

wprintf(L"Registering for restart...\n");
hr = RegisterApplicationRestart(L"/restart -f .\\filename.ext", 0);
if (FAILED(hr))
{
    wprintf(L"RegisterApplicationRestart failed, 0x%x\n", hr);
    goto cleanup;
}

wprintf(L"Get restart command line using static buffer...\n");
hr = GetApplicationRestartSettings(GetCurrentProcess(), wsCommandLine,
&cchCmdLine, &dwFlags);
if (FAILED(hr))
{
    wprintf(L"GetApplicationRestartSettings failed, 0x%x\n", hr);
    goto cleanup;
}

wprintf(L"Command line: %s\n", wsCommandLine);

wprintf(L"\nGet settings using dynamic buffer...\n");

cchCmdLine = 0;

// Returns S_OK instead of ERROR_INSUFFICIENT_BUFFER when pBuffer is
// NULL and size is 0.
hr = GetApplicationRestartSettings(GetCurrentProcess(),
(PWSTR)pwsCmdLine, &cchCmdLine, &dwFlags);
if (SUCCEEDED(hr))
{
    pwsCmdLine = (LPWSTR)malloc(cchCmdLine * sizeof(WCHAR));

    if (pwsCmdLine)
    {
        hr = GetApplicationRestartSettings(GetCurrentProcess(),
(PWSTR)pwsCmdLine, &cchCmdLine, &dwFlags);
        if (FAILED(hr))
        {
            wprintf(L"GetApplicationRestartSettings failed with 0x%x\n",
hr);
            goto cleanup;
        }

        wprintf(L"Command line: %s\n", pwsCmdLine);
    }
    else
    {
        wprintf(L"Allocating the command-line buffer failed.\n");
    }
}
```

```
    }
    else
    {
        if (hr != HRESULT_FROM_WIN32(ERROR_NOT_FOUND)) // Not a restart.
        {
            wprintf(L"GetApplicationRestartSettings failed with 0x%x\n",
hr);
            goto cleanup;
        }
    }

cleanup:

    if (pwsCmdLine)
        free(pwsCmdLine);
}
```

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[RegisterApplicationRestart](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetAtomNameA function (winbase.h)

Article 02/09/2023

Retrieves a copy of the character string associated with the specified local atom.

Syntax

C++

```
UINT GetAtomNameA(
    [in] ATOM nAtom,
    [out] LPSTR lpBuffer,
    [in] int nSize
);
```

Parameters

[in] nAtom

Type: **ATOM**

The local atom that identifies the character string to be retrieved.

[out] lpBuffer

Type: **LPTSTR**

The character string.

[in] nSize

Type: **int**

The size, in characters, of the buffer.

Return value

Type: **UINT**

If the function succeeds, the return value is the length of the string copied to the buffer, in characters, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Security Considerations

Using this function incorrectly might compromise the security of your program. Incorrect use of this function includes not correctly specifying the size of the *lpBuffer* parameter.

Note

The winbase.h header defines GetAtomName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[MAKEINTATOM](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetAtomNameW function (winbase.h)

Article 02/09/2023

Retrieves a copy of the character string associated with the specified local atom.

Syntax

C++

```
UINT GetAtomNameW(
    [in] ATOM nAtom,
    [out] LPWSTR lpBuffer,
    [in] int nSize
);
```

Parameters

[in] nAtom

Type: **ATOM**

The local atom that identifies the character string to be retrieved.

[out] lpBuffer

Type: **LPTSTR**

The character string.

[in] nSize

Type: **int**

The size, in characters, of the buffer.

Return value

Type: **UINT**

If the function succeeds, the return value is the length of the string copied to the buffer, in characters, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Security Considerations

Using this function incorrectly might compromise the security of your program. Incorrect use of this function includes not correctly specifying the size of the *lpBuffer* parameter.

Note

The winbase.h header defines GetAtomName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[MAKEINTATOM](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetBinaryTypeA function (winbase.h)

Article06/01/2023

Determines whether a file is an executable (.exe) file, and if so, which subsystem runs the executable file.

Syntax

C++

```
BOOL GetBinaryTypeA(  
    [in]  LPCSTR  lpApplicationName,  
    [out] LPDWORD lpBinaryType  
);
```

Parameters

[in] *lpApplicationName*

The full path of the file whose executable type is to be determined.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[out] *lpBinaryType*

A pointer to a variable to receive information about the executable type of the file specified by *lpApplicationName*. The following constants are defined.

Value	Meaning
SCS_32BIT_BINARY	A 32-bit Windows-based application
0	

SCS_64BIT_BINARY	A 64-bit Windows-based application.
6	
SCS_DOS_BINARY	An MS-DOS – based application
1	
SCS_OS216_BINARY	A 16-bit OS/2-based application
5	
SCS_PIF_BINARY	A PIF file that executes an MS-DOS – based application
3	
SCS_POSIX_BINARY	A POSIX – based application
4	
SCS_WOW_BINARY	A 16-bit Windows-based application
2	

Return value

If the file is executable, the return value is nonzero. The function sets the variable pointed to by *lpBinaryType* to indicate the file's executable type.

If the file is not executable, or if the function fails, the return value is zero. To get extended error information, call [GetLastError](#). If the file is a DLL, the last error code is **ERROR_BAD_EXE_FORMAT**.

Remarks

As an alternative, you can obtain the same information by calling the [SHGetFileInfo](#) function, passing the **SHGFI_EXETYPE** flag in the *uFlags* parameter.

Symbolic link behavior—If the path points to a symbolic link, the target file is used.

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

ⓘ Note

The `winbase.h` header defines `GetBinaryType` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[File Management Functions](#)

[SHGetFileInfo](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetBinaryTypeW function (winbase.h)

Article06/01/2023

Determines whether a file is an executable (.exe) file, and if so, which subsystem runs the executable file.

Syntax

C++

```
BOOL GetBinaryTypeW(
    [in]  LPCWSTR lpApplicationName,
    [out] LPDWORD lpBinaryType
);
```

Parameters

[in] *lpApplicationName*

The full path of the file whose executable type is to be determined.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[out] *lpBinaryType*

A pointer to a variable to receive information about the executable type of the file specified by *lpApplicationName*. The following constants are defined.

Value	Meaning
SCS_32BIT_BINARY	A 32-bit Windows-based application
0	

SCS_64BIT_BINARY	A 64-bit Windows-based application.
6	
SCS_DOS_BINARY	An MS-DOS – based application
1	
SCS_OS216_BINARY	A 16-bit OS/2-based application
5	
SCS_PIF_BINARY	A PIF file that executes an MS-DOS – based application
3	
SCS_POSIX_BINARY	A POSIX – based application
4	
SCS_WOW_BINARY	A 16-bit Windows-based application
2	

Return value

If the file is executable, the return value is nonzero. The function sets the variable pointed to by *lpBinaryType* to indicate the file's executable type.

If the file is not executable, or if the function fails, the return value is zero. To get extended error information, call [GetLastError](#). If the file is a DLL, the last error code is **ERROR_BAD_EXE_FORMAT**.

Remarks

As an alternative, you can obtain the same information by calling the [SHGetFileInfo](#) function, passing the **SHGFI_EXETYPE** flag in the *uFlags* parameter.

Symbolic link behavior—If the path points to a symbolic link, the target file is used.

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

ⓘ Note

The winbase.h header defines GetBinaryType as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[SHGetFileInfo](#)

[Symbolic Links](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCommConfig function (winbase.h)

Article10/13/2021

Retrieves the current configuration of a communications device.

To retrieve the default configuration settings from the device manager, use the [GetDefaultCommConfig](#) function.

Syntax

C++

```
BOOL GetCommConfig(
    [in]      HANDLE     hCommDev,
    [out]     LPCOMMCONFIG lpCC,
    [in, out] LPDWORD    lpdwSize
);
```

Parameters

[in] hCommDev

A handle to the open communications device. The [CreateFile](#) function returns this handle.

[out] lpCC

A pointer to a buffer that receives a [COMMCONFIG](#) structure.

[in, out] lpdwSize

The size, in bytes, of the buffer pointed to by *lpCC*. When the function returns, the variable contains the number of bytes copied if the function succeeds, or the number of bytes required if the buffer was too small.

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, use the [GetLastError](#) function.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[GetDefaultCommConfig](#)

[SetCommConfig](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCommMask function (winbase.h)

Article10/13/2021

Retrieves the value of the event mask for a specified communications device.

Syntax

C++

```
BOOL GetCommMask(
    [in] HANDLE hFile,
    [out] LPDWORD lpEvtMask
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[out] lpEvtMask

A pointer to the variable that receives a mask of events that are currently enabled. This parameter can be one or more of the following values.

Value	Meaning
EV_BREAK 0x0040	A break was detected on input.
EV_CTS 0x0008	The CTS (clear-to-send) signal changed state.
EV_DSR 0x0010	The DSR (data-set-ready) signal changed state.
EV_ERR 0x0080	A line-status error occurred. Line-status errors are CE_FRAME , CE_OVERRUN , and CE_RXPARITY .
EV_EVENT1 0x0800	An event of the first provider-specific type occurred.
EV_EVENT2 0x1000	An event of the second provider-specific type occurred.

EV_PERR 0x0200	A printer error occurred.
EV_RING 0x0100	A ring indicator was detected.
EV_RLSD 0x0020	The RLSD (receive-line-signal-detect) signal changed state.
EV_RX80FULL 0x0400	The receive buffer is 80 percent full.
EV_RXCHAR 0x0001	A character was received and placed in the input buffer.
EV_RXFLAG 0x0002	The event character was received and placed in the input buffer. The event character is specified in the device's DCB structure, which is applied to a serial port by using the SetCommState function.
EV_TXEMPTY 0x0004	The last character in the output buffer was sent.

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

The [GetCommMask](#) function uses a mask variable to indicate the set of events that can be monitored for a particular communications resource. A handle to the communications resource can be specified in a call to the [WaitCommEvent](#) function, which waits for one of the events to occur. To modify the event mask of a communications resource, use the [SetCommMask](#) function.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[DCB](#)

[SetCommMask](#)

[WaitCommEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCommModemStatus function (winbase.h)

Article 10/13/2021

Retrieves the modem control-register values.

Syntax

C++

```
BOOL GetCommModemStatus(
    [in] HANDLE hFile,
    [out] LPDWORD lpModemStat
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[out] lpModemStat

A pointer to a variable that receives the current state of the modem control-register values. This parameter can be one or more of the following values.

Value	Meaning
MS_CTS_ON 0x0010	The CTS (clear-to-send) signal is on.
MS_DSR_ON 0x0020	The DSR (data-set-ready) signal is on.
MS_RING_ON 0x0040	The ring indicator signal is on.
MS_RLSD_ON 0x0080	The RLSD (receive-line-signal-detect) signal is on.

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

The **GetCommModemStatus** function is useful when you are using the [WaitCommEvent](#) function to monitor the CTS, RLSD, DSR, or ring indicator signals. To detect when these signals change state, use [WaitCommEvent](#) and then use **GetCommModemStatus** to determine the state after a change occurs.

The function fails if the hardware does not support the control-register values.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[WaitCommEvent](#)

Feedback



Was this page helpful? [Yes](#) | [No](#)

Get help at Microsoft Q&A

GetCommPorts function (winbase.h)

Article10/13/2021

Gets an array that contains the well-formed COM ports.

This function obtains the COM port numbers from the **HKLM\Hardware\DeviceMap\SERIALCOMM** registry key and then writes them to a caller-supplied array. If the array is too small, the function gets the necessary size.

Note If new entries are added to the registry key, the necessary size can change between API calls.

Syntax

C++

```
ULONG GetCommPorts(
    [out] PULONG lpPortNumbers,
    [in]  ULONG uPortNumbersCount,
    [out] PULONG puPortNumbersFound
);
```

Parameters

[out] *lpPortNumbers*

An array for the port numbers.

[in] *uPortNumbersCount*

The length of the array in the *lpPortNumbers* parameter.

[out] *puPortNumbersFound*

The number of port numbers written to the *lpPortNumbers* or the length of the array required for the port numbers.

Return value

Return code	Description
ERROR_SUCCESS	The call succeeded. The <i>lpPortNumbers</i> array was large enough for the result.
ERROR_MORE_DATA	The <i>lpPortNumbers</i> array was too small to contain all available port numbers.
ERROR_FILE_NOT_FOUND	There are no comm ports available.

Requirements

Minimum supported client	Windows 10, version 1803 [desktop apps UWP apps]
Minimum supported server	Windows Server, version 1709 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	OneCore.lib
DLL	KernelBase.dll

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCommProperties function (winbase.h)

Article10/13/2021

Retrieves information about the communications properties for a specified communications device.

Syntax

C++

```
BOOL GetCommProperties(
    [in] HANDLE     hFile,
    [out] LPCOMMPROP lpCommProp
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[out] lpCommProp

A pointer to a [COMMPROP](#) structure in which the communications properties information is returned. This information can be used in subsequent calls to the [SetCommState](#), [SetCommTimeouts](#), or [SetupComm](#) function to configure the communications device.

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

The **GetCommProperties** function returns information from a device driver about the configuration settings that are supported by the driver.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMPROP](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[SetCommState](#)

[SetCommTimeouts](#)

[SetupComm](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCommState function (winbase.h)

Article10/13/2021

Retrieves the current control settings for a specified communications device.

Syntax

C++

```
BOOL GetCommState(
    [in]      HANDLE hFile,
    [in, out] LPDCB  lpDCB
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in, out] lpDCB

A pointer to a [DCB](#) structure that receives the control settings information.

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](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[DCB](#)

[SetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCommTimeouts function (winbase.h)

Article10/13/2021

Retrieves the time-out parameters for all read and write operations on a specified communications device.

Syntax

C++

```
BOOL GetCommTimeouts(
    [in]   HANDLE          hFile,
    [out]  LPCOMMTIMEOUTS lpCommTimeouts
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[out] lpCommTimeouts

A pointer to a [COMMTIMEOUTS](#) structure in which the time-out information is returned.

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

For more information about time-out values for communications devices, see the [SetCommTimeouts](#) function.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMTIMEOUTS](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[SetCommTimeouts](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCompressedFileSizeTransactedA function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Retrieves the actual number of bytes of disk storage used to store a specified file as a transacted operation. If the file is located on a volume that supports compression and the file is compressed, the value obtained is the compressed size of the specified file. If the file is located on a volume that supports sparse files and the file is a sparse file, the value obtained is the sparse size of the specified file.

Syntax

C++

```
DWORD GetCompressedFileSizeTransactedA(
    [in]          LPCSTR  lpFileName,
    [out, optional] LPDWORD lpFileSizeHigh,
    [in]          HANDLE   hTransaction
);
```

Parameters

`[in] lpFileName`

The name of the file.

Do not specify the name of a file on a nonseeking device, such as a pipe or a communications device, as its file size has no meaning.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE`.

`[out, optional] lpFileSizeHigh`

A pointer to a variable that receives the high-order **DWORD** of the compressed file size. The function's return value is the low-order **DWORD** of the compressed file size.

This parameter can be **NULL** if the high-order **DWORD** of the compressed file size is not needed. Files less than 4 gigabytes in size do not need the high-order **DWORD**.

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is the low-order **DWORD** of the actual number of bytes of disk storage used to store the specified file, and if *lpFileSizeHigh* is non-**NULL**, the function puts the high-order **DWORD** of that actual value into the **DWORD** pointed to by that parameter. This is the compressed file size for compressed files, the actual file size for noncompressed files.

If the function fails, and *lpFileSizeHigh* is **NULL**, the return value is **INVALID_FILE_SIZE**. To get extended error information, call [GetLastError](#).

If the return value is **INVALID_FILE_SIZE** and *lpFileSizeHigh* is non-**NULL**, an application must call [GetLastError](#) to determine whether the function has succeeded (value is **NO_ERROR**) or failed (value is other than **NO_ERROR**).

Remarks

An application can determine whether a volume is compressed by calling [GetVolumeInformation](#), then checking the status of the **FS_VOL_IS_COMPRESSED** flag in the **DWORD** value pointed to by that function's *lpFileSystemFlags* parameter.

If the file is not located on a volume that supports compression or sparse files, or if the file is not compressed or a sparse file, the value obtained is the actual file size, the same as the value returned by a call to [GetFileSize](#).

Symbolic links: If the path points to a symbolic link, the function returns the file size of the target.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported

Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The `winbase.h` header defines `GetCompressedFileSizeTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[File Compression and Decompression](#)

[File Management Functions](#)

[GetFileSize](#)

[GetVolumeInformation](#)

[Symbolic Links](#)

[Transaction Management](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCompressedFileSizeTransactedW function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Retrieves the actual number of bytes of disk storage used to store a specified file as a transacted operation. If the file is located on a volume that supports compression and the file is compressed, the value obtained is the compressed size of the specified file. If the file is located on a volume that supports sparse files and the file is a sparse file, the value obtained is the sparse size of the specified file.

Syntax

C++

```
DWORD GetCompressedFileSizeTransactedW(
    [in]          LPCWSTR lpFileName,
    [out, optional] LPDWORD lpFileSizeHigh,
    [in]          HANDLE hTransaction
);
```

Parameters

`[in] lpFileName`

The name of the file.

Do not specify the name of a file on a nonseeking device, such as a pipe or a communications device, as its file size has no meaning.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE`.

`[out, optional] lpFileSizeHigh`

A pointer to a variable that receives the high-order **DWORD** of the compressed file size. The function's return value is the low-order **DWORD** of the compressed file size.

This parameter can be **NULL** if the high-order **DWORD** of the compressed file size is not needed. Files less than 4 gigabytes in size do not need the high-order **DWORD**.

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is the low-order **DWORD** of the actual number of bytes of disk storage used to store the specified file, and if *lpFileSizeHigh* is non-**NULL**, the function puts the high-order **DWORD** of that actual value into the **DWORD** pointed to by that parameter. This is the compressed file size for compressed files, the actual file size for noncompressed files.

If the function fails, and *lpFileSizeHigh* is **NULL**, the return value is **INVALID_FILE_SIZE**. To get extended error information, call [GetLastError](#).

If the return value is **INVALID_FILE_SIZE** and *lpFileSizeHigh* is non-**NULL**, an application must call [GetLastError](#) to determine whether the function has succeeded (value is **NO_ERROR**) or failed (value is other than **NO_ERROR**).

Remarks

An application can determine whether a volume is compressed by calling [GetVolumeInformation](#), then checking the status of the **FS_VOL_IS_COMPRESSED** flag in the **DWORD** value pointed to by that function's *lpFileSystemFlags* parameter.

If the file is not located on a volume that supports compression or sparse files, or if the file is not compressed or a sparse file, the value obtained is the actual file size, the same as the value returned by a call to [GetFileSize](#).

Symbolic links: If the path points to a symbolic link, the function returns the file size of the target.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported

Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The `winbase.h` header defines `GetCompressedFileSizeTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[File Compression and Decompression](#)

[File Management Functions](#)

[GetFileSize](#)

[GetVolumeInformation](#)

[Symbolic Links](#)

[Transaction Management](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetComputerNameA function (winbase.h)

Article02/09/2023

Retrieves the NetBIOS name of the local computer. This name is established at system startup, when the system reads it from the registry.

GetComputerName retrieves only the NetBIOS name of the local computer. To retrieve the DNS host name, DNS domain name, or the fully qualified DNS name, call the [GetComputerNameEx](#) function. Additional information is provided by the [IADsADSystemInfo](#) interface.

The behavior of this function can be affected if the local computer is a node in a cluster. For more information, see [ResUtilGetEnvironmentWithNetName](#) and [UseNetworkName](#).

Syntax

C++

```
BOOL GetComputerNameA(
    [out]     LPSTR    lpBuffer,
    [in, out] LPDWORD nSize
);
```

Parameters

[out] *lpBuffer*

A pointer to a buffer that receives the computer name or the cluster virtual server name. The buffer size should be large enough to contain MAX_COMPUTERNAME_LENGTH + 1 characters.

[in, out] *nSize*

On input, specifies the size of the buffer, in TCHARs. On output, the number of TCHARs copied to the destination buffer, not including the terminating null character.

If the buffer is too small, the function fails and [GetLastError](#) returns [ERROR_BUFFER_OVERFLOW](#). The *lpnSize* parameter specifies the size of the buffer required, including the terminating null character.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The [GetComputerName](#) function retrieves the NetBIOS name established at system startup. Name changes made by the [SetComputerName](#) or [SetComputerNameEx](#) functions do not take effect until the user restarts the computer.

If the caller is running under a client session, this function returns the server name. To retrieve the client name, use the [WTSQuerySessionInformation](#) function.

Examples

For an example, see [Getting System Information](#).

ⓘ Note

The winbase.h header defines GetComputerName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[Computer Names](#)

[GetComputerNameEx](#)

[SetComputerName](#)

[SetComputerNameEx](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetComputerNameW function (winbase.h)

Article02/09/2023

Retrieves the NetBIOS name of the local computer. This name is established at system startup, when the system reads it from the registry.

GetComputerName retrieves only the NetBIOS name of the local computer. To retrieve the DNS host name, DNS domain name, or the fully qualified DNS name, call the [GetComputerNameEx](#) function. Additional information is provided by the [IADsADSystemInfo](#) interface.

The behavior of this function can be affected if the local computer is a node in a cluster. For more information, see [ResUtilGetEnvironmentWithNetName](#) and [UseNetworkName](#).

Syntax

C++

```
BOOL GetComputerNameW(
    [out]     LPWSTR lpBuffer,
    [in, out] LPDWORD nSize
);
```

Parameters

[out] *lpBuffer*

A pointer to a buffer that receives the computer name or the cluster virtual server name. The buffer size should be large enough to contain MAX_COMPUTERNAME_LENGTH + 1 characters.

[in, out] *nSize*

On input, specifies the size of the buffer, in TCHARs. On output, the number of TCHARs copied to the destination buffer, not including the terminating null character.

If the buffer is too small, the function fails and [GetLastError](#) returns [ERROR_BUFFER_OVERFLOW](#). The *lpnSize* parameter specifies the size of the buffer required, including the terminating null character.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The [GetComputerName](#) function retrieves the NetBIOS name established at system startup. Name changes made by the [SetComputerName](#) or [SetComputerNameEx](#) functions do not take effect until the user restarts the computer.

If the caller is running under a client session, this function returns the server name. To retrieve the client name, use the [WTSQuerySessionInformation](#) function.

Examples

For an example, see [Getting System Information](#).

ⓘ Note

The winbase.h header defines GetComputerName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[Computer Names](#)

[GetComputerNameEx](#)

[SetComputerName](#)

[SetComputerNameEx](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCurrentActCtx function (winbase.h)

Article10/13/2021

The **GetCurrentActCtx** function returns the handle to the active activation context of the calling thread.

Syntax

C++

```
BOOL GetCurrentActCtx(  
    [out] HANDLE *lphActCtx  
);
```

Parameters

[out] lphActCtx

Pointer to the returned [ACTCTX](#) structure that contains information on the active activation context.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

The calling thread is responsible for releasing the handle of the returned activation context. This function can return a null handle if no activation contexts have been activated by this thread. This is not an error.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCurrentDirectory function (winbase.h)

Article 10/13/2021

Retrieves the current directory for the current process.

Syntax

C++

```
DWORD GetCurrentDirectory(
    [in]  DWORD  nBufferLength,
    [out] LPTSTR lpBuffer
);
```

Parameters

[in] `nBufferLength`

The length of the buffer for the current directory string, in TCHARs. The buffer length must include room for a terminating null character.

[out] `lpBuffer`

A pointer to the buffer that receives the current directory string. This null-terminated string specifies the absolute path to the current directory.

To determine the required buffer size, set this parameter to **NULL** and the *nBufferLength* parameter to 0.

Return value

If the function succeeds, the return value specifies the number of characters that are written to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

If the buffer that is pointed to by *lpBuffer* is not large enough, the return value specifies the required size of the buffer, in characters, including the null-terminating character.

Remarks

Each process has a single current directory that consists of two parts:

- A disk designator that is either a drive letter followed by a colon, or a server name followed by a share name (`\servername\sharename`)
- A directory on the disk designator

To set the current directory, use the [SetCurrentDirectory](#) function.

Multithreaded applications and shared library code should not use the [GetCurrentDirectory](#) function and should avoid using relative path names. The current directory state written by the [SetCurrentDirectory](#) function is stored as a global variable in each process, therefore multithreaded applications cannot reliably use this value without possible data corruption from other threads that may also be reading or setting this value. This limitation also applies to the [SetCurrentDirectory](#) and [GetFullPathName](#) functions. The exception being when the application is guaranteed to be running in a single thread, for example parsing file names from the command line argument string in the main thread prior to creating any additional threads. Using relative path names in multithreaded applications or shared library code can yield unpredictable results and is not supported.

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 an example, see [Changing the Current Directory](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[.CreateDirectory](#)

[Directory Management Functions](#)

[GetSystemDirectory](#)

[GetWindowsDirectory](#)

[RemoveDirectory](#)

[SetCurrentDirectory](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCurrentHwProfileA function (winbase.h)

Article02/09/2023

Retrieves information about the current hardware profile for the local computer.

Syntax

C++

```
BOOL GetCurrentHwProfileA(
    [out] LPHW_PROFILE_INFOA lpHwProfileInfo
);
```

Parameters

[out] lpHwProfileInfo

A pointer to an [HW_PROFILE_INFO](#) structure that receives information about the current hardware profile.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The [GetCurrentHwProfile](#) function retrieves the display name and globally unique identifier (GUID) string for the hardware profile. The function also retrieves the reported docking state for portable computers with docking stations.

The system generates a GUID for each hardware profile and stores it as a string in the registry. You can use [GetCurrentHwProfile](#) to retrieve the GUID string to use as a registry subkey under your application's configuration settings key in [HKEY_CURRENT_USER](#). This enables you to store each user's settings for each hardware profile. For example, the Colors control panel application could use the subkey to store

each user's color preferences for different hardware profiles, such as profiles for the docked and undocked states. Applications that use this functionality can check the current hardware profile when they start up, and update their settings accordingly.

Applications can also update their settings when a system device message, such as [DBT_CONFIGCHANGED](#), indicates that the hardware profile has changed.

To compile an application that uses this function, define the _WIN32_WINNT macro as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Examples

C++

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void main(void)
{
    HW_PROFILE_INFO    HwProfInfo;
    if (!GetCurrentHwProfile(&HwProfInfo))
    {
        _tprintf(TEXT("GetCurrentHwProfile failed with error %lx\n"),
                 GetLastError());
        return;
    }
    _tprintf(TEXT("DockInfo = %d\n"), HwProfInfo.dwDockInfo);
    _tprintf(TEXT("Profile Guid = %s\n"), HwProfInfo.szHwProfileGuid);
    _tprintf(TEXT("Friendly Name = %s\n"), HwProfInfo.szHwProfileName);
}
```

ⓘ Note

The winbase.h header defines GetCurrentHwProfile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-hwprof-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[DBT_CONFIGCHANGED](#)

[HW_PROFILE_INFO](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCurrentHwProfileW function (winbase.h)

Article02/09/2023

Retrieves information about the current hardware profile for the local computer.

Syntax

C++

```
BOOL GetCurrentHwProfileW(
    [out] LPHW_PROFILE_INFOW lpHwProfileInfo
);
```

Parameters

[out] lpHwProfileInfo

A pointer to an [HW_PROFILE_INFO](#) structure that receives information about the current hardware profile.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The **GetCurrentHwProfile** function retrieves the display name and globally unique identifier (GUID) string for the hardware profile. The function also retrieves the reported docking state for portable computers with docking stations.

The system generates a GUID for each hardware profile and stores it as a string in the registry. You can use **GetCurrentHwProfile** to retrieve the GUID string to use as a registry subkey under your application's configuration settings key in **HKEY_CURRENT_USER**. This enables you to store each user's settings for each hardware profile. For example, the Colors control panel application could use the subkey to store

each user's color preferences for different hardware profiles, such as profiles for the docked and undocked states. Applications that use this functionality can check the current hardware profile when they start up, and update their settings accordingly.

Applications can also update their settings when a system device message, such as [DBT_CONFIGCHANGED](#), indicates that the hardware profile has changed.

To compile an application that uses this function, define the _WIN32_WINNT macro as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Examples

C++

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void main(void)
{
    HW_PROFILE_INFO    HwProfInfo;
    if (!GetCurrentHwProfile(&HwProfInfo))
    {
        _tprintf(TEXT("GetCurrentHwProfile failed with error %lx\n"),
                 GetLastError());
        return;
    }
    _tprintf(TEXT("DockInfo = %d\n"), HwProfInfo.dwDockInfo);
    _tprintf(TEXT("Profile Guid = %s\n"), HwProfInfo.szHwProfileGuid);
    _tprintf(TEXT("Friendly Name = %s\n"), HwProfInfo.szHwProfileName);
}
```

ⓘ Note

The winbase.h header defines GetCurrentHwProfile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-hwprof-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[DBT_CONFIGCHANGED](#)

[HW_PROFILE_INFO](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetCurrentUmsThread function (winbase.h)

Article 03/18/2022

Returns the user-mode scheduling (UMS) thread context of the calling UMS thread.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
PUMS_CONTEXT GetCurrentUmsThread();
```

Return value

The function returns a pointer to the UMS thread context of the calling thread.

If calling thread is not a UMS thread, the function returns NULL. To get extended error information, call [GetLastError](#).

Remarks

The `GetCurrentUmsThread` function can be called for a UMS scheduler thread or UMS worker thread.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetDefaultCommConfigA function (winbase.h)

Article 02/09/2023

Retrieves the default configuration for the specified communications device.

Syntax

C++

```
BOOL GetDefaultCommConfigA(
    [in]      LPCSTR      lpszName,
    [out]     LPCOMMCONFIG lpCC,
    [in, out] LPDWORD     lpdwSize
);
```

Parameters

[in] lpszName

The name of the device. For example, COM1 through COM9 are serial ports and LPT1 through LPT9 are parallel ports.

[out] lpCC

A pointer to a buffer that receives a [COMMCONFIG](#) structure.

[in, out] lpdwSize

A pointer to a variable that specifies the size of the buffer pointed to by *lpCC*, in bytes. Upon return, the variable contains the number of bytes copied if the function succeeds, or the number of bytes required if the buffer was too small.

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, use the [GetLastError](#) function.

Remarks

ⓘ Note

The winbase.h header defines GetDefaultCommConfig as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

[SetDefaultCommConfig](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetDefaultCommConfigW function (winbase.h)

Article02/09/2023

Retrieves the default configuration for the specified communications device.

Syntax

C++

```
BOOL GetDefaultCommConfigW(
    [in]      LPCWSTR     lpszName,
    [out]     LPCOMMCONFIG lpCC,
    [in, out] LPDWORD     lpdwSize
);
```

Parameters

[in] lpszName

The name of the device. For example, COM1 through COM9 are serial ports and LPT1 through LPT9 are parallel ports.

[out] lpCC

A pointer to a buffer that receives a [COMMCONFIG](#) structure.

[in, out] lpdwSize

A pointer to a variable that specifies the size of the buffer pointed to by *lpCC*, in bytes. Upon return, the variable contains the number of bytes copied if the function succeeds, or the number of bytes required if the buffer was too small.

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, use the [GetLastError](#) function.

Remarks

ⓘ Note

The winbase.h header defines GetDefaultCommConfig as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

[SetDefaultCommConfig](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetDevicePowerState function (winbase.h)

Article 10/13/2021

Retrieves the current power state of the specified device. This function cannot be used to query the power state of a display device.

Syntax

C++

```
BOOL GetDevicePowerState(  
    [in]  HANDLE hDevice,  
    [out] BOOL   *pfOn  
>;
```

Parameters

[in] hDevice

A handle to an object on the device, such as a file or socket, or a handle to the device itself.

[out] pfOn

A pointer to the variable that receives the [power state](#). This value is **TRUE** if the device is in the working state. Otherwise, it is **FALSE**.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

An application can use **GetDevicePowerState** to determine whether a device is in the working state or a low-power state. If the device is in a low-power state, accessing the

device may cause it to either queue or fail any I/O requests, or transition the device into the working state. The exact behavior depends on the implementation of the device.

To ensure maximum battery life on a laptop computer, use **GetDevicePowerState** to reduce power consumption. For example, if a disk is currently powered down, accessing the disk will cause it to spin up, resulting in increased power consumption and reduced battery life.

Applications should defer or limit access to devices wherever possible while the system is running on battery power. To determine whether the system is running on battery power, and the remaining battery life, use the [GetSystemPowerStatus](#) function.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetSystemPowerStatus](#)

[Power Management Functions](#)

[System Power Status](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetDllDirectoryA function (winbase.h)

Article02/09/2023

Retrieves the application-specific portion of the search path used to locate DLLs for the application.

Syntax

C++

```
DWORD GetDllDirectoryA(  
    [in]  DWORD nBufferLength,  
    [out] LPSTR lpBuffer  
);
```

Parameters

[in] *nBufferLength*

The size of the output buffer, in characters.

[out] *lpBuffer*

A pointer to a buffer that receives the application-specific portion of the search path.

Return value

If the function succeeds, the return value is the length of the string copied to *lpBuffer*, in characters, not including the terminating null character. If the return value is greater than *nBufferLength*, it specifies the size of the buffer required for the path.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

To compile an application that uses this function, define `_WIN32_WINNT` as `0x0502` or later. For more information, see [Using the Windows Headers](#).

 Note

The winbase.h header defines GetDIIIDirectory as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Dynamic-Link Library Search Order](#)

[SetDIIIDirectory](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetDllDirectoryW function (winbase.h)

Article02/09/2023

Retrieves the application-specific portion of the search path used to locate DLLs for the application.

Syntax

C++

```
DWORD GetDllDirectoryW(
    [in]  DWORD  nBufferLength,
    [out] LPWSTR lpBuffer
);
```

Parameters

[in] *nBufferLength*

The size of the output buffer, in characters.

[out] *lpBuffer*

A pointer to a buffer that receives the application-specific portion of the search path.

Return value

If the function succeeds, the return value is the length of the string copied to *lpBuffer*, in characters, not including the terminating null character. If the return value is greater than *nBufferLength*, it specifies the size of the buffer required for the path.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

To compile an application that uses this function, define `_WIN32_WINNT` as `0x0502` or later. For more information, see [Using the Windows Headers](#).

 Note

The winbase.h header defines GetDIIIDirectory as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Dynamic-Link Library Search Order](#)

[SetDIIIDirectory](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetEnabledXStateFeatures function (winbase.h)

Article 06/29/2021

Gets a mask of enabled XState features on x86 or x64 processors.

The definition of XState feature bits are processor vendor specific. Please refer to the relevant processor reference manuals for additional information on a particular feature.

Syntax

C++

```
DWORD64 GetEnabledXStateFeatures();
```

Return value

This function returns a bitmask in which each bit represents an XState feature that is enabled on the system.

Remarks

An application should call this function to determine what features are present and enabled on the system before using an XState processor feature or attempting to manipulate XState contexts. Bits 0 and 1 refer to the X87 FPU and the presence of SSE registers, respectively. The meanings of specific feature bits beyond 0 and 1 are defined in the Programmer Reference Manuals released by the processor vendors.

Note Not all features supported by a processor may be enabled on the system. Using a feature which is not enabled may result in exceptions or undefined behavior.

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The [AVX API](#) is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1. Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this

documentation and get pointers to them using [GetModuleHandle](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Intel AVX](#)

[Working with XState Context](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetEnvironmentVariable function (winbase.h)

Article09/23/2022

Retrieves the contents of the specified variable from the environment block of the calling process.

Syntax

C++

```
DWORD GetEnvironmentVariable(
    [in, optional] LPCTSTR lpName,
    [out, optional] LPTSTR lpBuffer,
    [in]           DWORD   nSize
);
```

Parameters

[in, optional] *lpName*

The name of the environment variable.

[out, optional] *lpBuffer*

A pointer to a buffer that receives the contents of the specified environment variable as a null-terminated string. An environment variable has a maximum size limit of 32,767 characters, including the null-terminating character.

[in] *nSize*

The size of the buffer pointed to by the *lpBuffer* parameter, including the null-terminating character, in characters.

Return value

If the function succeeds, the return value is the number of characters stored in the buffer pointed to by *lpBuffer*, not including the terminating null character.

If *lpBuffer* is not large enough to hold the data, the return value is the buffer size, in characters, required to hold the string and its terminating null character and the contents of *lpBuffer* are undefined.

If the function fails, the return value is zero. If the specified environment variable was not found in the environment block, [GetLastError](#) returns ERROR_ENVVAR_NOT_FOUND.

Remarks

This function can retrieve either a system environment variable or a user environment variable.

Examples

For an example, see [Changing Environment Variables](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Environment Variables](#)

[GetEnvironmentStrings](#)

[SetEnvironmentVariable](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetEventLogInformation function (winbase.h)

Article 10/13/2021

Retrieves information about the specified event log.

Syntax

C++

```
BOOL GetEventLogInformation(
    [in] HANDLE hEventLog,
    [in] DWORD dwInfoLevel,
    [out] LPVOID lpBuffer,
    [in] DWORD cbBufSize,
    [out] LPDWORD pcbBytesNeeded
);
```

Parameters

[in] `hEventLog`

A handle to the event log. The [OpenEventLog](#) or [RegisterEventSource](#) function returns this handle.

[in] `dwInfoLevel`

The level of event log information to return.

This parameter can be the following value.

Value	Meaning
<code>EVENTLOG_FULL_INFO</code>	Indicate whether the specified log is full. The <code>lpBuffer</code> parameter will contain an EVENTLOG_FULL_INFORMATION structure.

[out] `lpBuffer`

An application-allocated buffer that receives the event log information. The format of this data depends on the value of the `dwInfoLevel` parameter.

[in] cbBufSize

The size of the *lpBuffer* buffer, in bytes.

[out] pcbBytesNeeded

The function sets this parameter to the required buffer size for the requested information, regardless of whether the function succeeds. Use this value if the function fails with **ERROR_INSUFFICIENT_BUFFER** to allocate a buffer of the correct 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](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[EVENTLOG_FULL_INFORMATION](#)

[Event Logging Functions](#)

[OpenEventLog](#)

[RegisterEventSource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFileAttributesTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Retrieves file system attributes for a specified file or directory as a transacted operation.

Syntax

C++

```
BOOL GetFileAttributesTransactedA(
    [in]    LPCSTR             lpFileName,
    [in]    GET_FILEEX_INFO_LEVELS fInfoLevelId,
    [out]   LPVOID              lpFileInformation,
    [in]    HANDLE              hTransaction
);
```

Parameters

[in] lpFileName

The name of the file or directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The file or directory must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] fInfoLevelId

The level of attribute information to retrieve.

This parameter can be the following value from the [GET_FILEEX_INFO_LEVELS](#) enumeration.

Value	Meaning
GetFileExInfoStandard	The <i>lpFileInformation</i> parameter is a WIN32_FILE_ATTRIBUTE_DATA structure.

[out] lpFileInformation

A pointer to a buffer that receives the attribute information.

The type of attribute information that is stored into this buffer is determined by the value of *fInfoLevelId*. If the *fInfoLevelId* parameter is **GetFileExInfoStandard** then this parameter points to a [WIN32_FILE_ATTRIBUTE_DATA](#) structure

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

Remarks

When **GetFileAttributesTransacted** is called on a directory that is a mounted folder, it returns the attributes of the directory, not those of the root directory in the volume that the mounted folder associates with the directory. To obtain the file attributes of the associated volume, call [GetVolumeNameForVolumeMountPoint](#) to obtain the name of the associated volume. Then use the resulting name in a call to **GetFileAttributesTransacted**. The results are the attributes of the root directory on the associated volume.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Symbolic links: If the path points to a symbolic link, the function returns attributes for the symbolic link.

Transacted Operations

If a file is open for modification in a transaction, no other thread can open the file for modification until the transaction is committed. Conversely, if a file is open for modification outside of a transaction, no transacted thread can open the file for modification until the non-transacted handle is closed. If a non-transacted thread has a handle opened to modify a file, a call to **GetFileAttributesTransacted** for that file will fail with an **ERROR_TRANSACTIONAL_CONFLICT** error.

ⓘ Note

The winbase.h header defines **GetFileAttributesTransacted** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see **Conventions for Function Prototypes**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeviceIoControl](#)

[File Attribute Constants](#)

[File Management Functions](#)

[FindFirstFileTransacted](#)

[FindNextFile](#)

[GET_FILEEX_INFO_LEVELS](#)

[SetFileAttributesTransacted](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFileAttributesTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Retrieves file system attributes for a specified file or directory as a transacted operation.

Syntax

C++

```
BOOL GetFileAttributesTransactedW(
    [in]  LPCWSTR             lpFileName,
    [in]  GET_FILEEX_INFO_LEVELS fInfoLevelId,
    [out] LPVOID               lpFileInformation,
    [in]  HANDLE                hTransaction
);
```

Parameters

[in] lpFileName

The name of the file or directory.

The file or directory must reside on the local computer; otherwise, the function fails and the last error code is set to `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE`.

By default, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, prepend "`\?\`" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the `MAX_PATH` limitation without prepending "`\?\`". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] fInfoLevelId

The level of attribute information to retrieve.

This parameter can be the following value from the [GET_FILEEX_INFO_LEVELS](#) enumeration.

Value	Meaning
GetFileExInfoStandard	The <i>lpFileInformation</i> parameter is a WIN32_FILE_ATTRIBUTE_DATA structure.

[out] lpFileInformation

A pointer to a buffer that receives the attribute information.

The type of attribute information that is stored into this buffer is determined by the value of *fInfoLevelId*. If the *fInfoLevelId* parameter is **GetFileExInfoStandard** then this parameter points to a [WIN32_FILE_ATTRIBUTE_DATA](#) structure

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

Remarks

When **GetFileAttributesTransacted** is called on a directory that is a mounted folder, it returns the attributes of the directory, not those of the root directory in the volume that the mounted folder associates with the directory. To obtain the file attributes of the associated volume, call [GetVolumeNameForVolumeMountPoint](#) to obtain the name of the associated volume. Then use the resulting name in a call to **GetFileAttributesTransacted**. The results are the attributes of the root directory on the associated volume.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Symbolic links: If the path points to a symbolic link, the function returns attributes for the symbolic link.

Transacted Operations

If a file is open for modification in a transaction, no other thread can open the file for modification until the transaction is committed. Conversely, if a file is open for modification outside of a transaction, no transacted thread can open the file for modification until the non-transacted handle is closed. If a non-transacted thread has a handle opened to modify a file, a call to **GetFileAttributesTransacted** for that file will fail with an **ERROR_TRANSACTIONAL_CONFLICT** error.

ⓘ Note

The winbase.h header defines **GetFileAttributesTransacted** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see **Conventions for Function Prototypes**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeviceIoControl](#)

[File Attribute Constants](#)

[File Management Functions](#)

[FindFirstFileTransacted](#)

[FindNextFile](#)

[GET_FILEEX_INFO_LEVELS](#)

[SetFileAttributesTransacted](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFileBandwidthReservation function (winbase.h)

Article 10/13/2021

Retrieves the bandwidth reservation properties of the volume on which the specified file resides.

Syntax

C++

```
BOOL GetFileBandwidthReservation(
    [in] HANDLE hFile,
    [out] LPDWORD lpPeriodMilliseconds,
    [out] LPDWORD lpBytesPerPeriod,
    [out] LPBOOL pDiscardable,
    [out] LPDWORD lpTransferSize,
    [out] LPDWORD lpNumOutstandingRequests
);
```

Parameters

[in] hFile

A handle to the file.

[out] lpPeriodMilliseconds

A pointer to a variable that receives the period of the reservation, in milliseconds. The period is the time from which the I/O is issued to the kernel until the time the I/O should be completed. If no bandwidth has been reserved for this handle, then the value returned is the minimum reservation period supported for this volume.

[out] lpBytesPerPeriod

A pointer to a variable that receives the maximum number of bytes per period that can be reserved on the volume. If no bandwidth has been reserved for this handle, then the value returned is the maximum number of bytes per period supported for the volume.

[out] pDiscardable

TRUE if I/O should be completed with an error if a driver is unable to satisfy an I/O operation before the period expires. **FALSE** if the underlying subsystem does not support failing in this manner.

[out] lpTransferSize

The minimum size of any individual I/O request that may be issued by the application. All I/O requests should be multiples of *TransferSize*. If no bandwidth has been reserved for this handle, then the value returned is the minimum transfer size supported for this volume.

[out] lpNumOutstandingRequests

The number of *TransferSize* chunks allowed to be outstanding with the operating system.

Return value

Returns nonzero if successful or zero otherwise.

To get extended error information, call [GetLastError](#).

Remarks

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	Yes

Requirements

Minimum supported client	Windows Vista [desktop apps only]
--------------------------	-----------------------------------

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[SetFileBandwidthReservation](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFileInformationByHandleEx function (winbase.h)

Article07/27/2022

Retrieves file information for the specified file.

For a more basic version of this function for desktop apps, see [GetFileInformationByHandle](#).

To set file information using a file handle, see [SetFileInformationByHandle](#).

Syntax

C++

```
BOOL GetFileInformationByHandleEx(
    [in] HANDLE             hFile,
    [in] FILE_INFO_BY_HANDLE_CLASS FileInformationClass,
    [out] LPVOID            lpFileInformation,
    [in] DWORD              dwBufferSize
);
```

Parameters

[in] hFile

A handle to the file that contains the information to be retrieved.

This handle should not be a pipe handle.

[in] FileInformationClass

A [FILE_INFO_BY_HANDLE_CLASS](#) enumeration value that specifies the type of information to be retrieved.

For a table of valid values, see the Remarks section.

[out] lpFileInformation

A pointer to the buffer that receives the requested file information. The structure that is returned corresponds to the class that is specified by *FileInformationClass*. For a table of valid structure types, see the Remarks section.

[in] dwBufferSize

The size of the *lpFileInformation* buffer, in bytes.

Return value

If the function succeeds, the return value is nonzero and file information data is contained in the buffer pointed to by the *lpFileInformation* parameter.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If *FileInfoClass* is **FileStreamInfo** and the calls succeed but no streams are returned, the error that is returned by [GetLastError](#) is **ERROR_HANDLE_EOF**.

Certain file information classes behave slightly differently on different operating system releases. These classes are supported by the underlying drivers, and any information they return is subject to change between operating system releases.

The following table shows the valid file information class types and their corresponding data structure types for use with this function.

FileInfoClass value	lpFileInformation type
FileBasicInfo (0)	FILE_BASIC_INFO
FileStandardInfo (1)	FILE_STANDARD_INFO
FileNameInfo (2)	FILE_NAME_INFO
FileStreamInfo (7)	FILE_STREAM_INFO
FileCompressionInfo (8)	FILE_COMPRESSION_INFO
FileAttributeTagInfo (9)	FILE_ATTRIBUTE_TAG_INFO
FileIdBothDirectoryInfo (0xa)	FILE_ID_BOTH_DIR_INFO
FileIdBothDirectoryRestartInfo (0xb)	FILE_ID_BOTH_DIR_INFO
FileRemoteProtocolInfo (0xd)	FILE_REMOTE_PROTOCOL_INFO
FileFullDirectoryInfo (0xe)	FILE_FULL_DIR_INFO
FileFullDirectoryRestartInfo (0xf)	FILE_FULL_DIR_INFO

FileStorageInfo (0x10)	FILE_STORAGE_INFO
FileAlignmentInfo (0x11)	FILE_ALIGNMENT_INFO
FileInfo (0x12)	FILE_ID_INFO
FileIdExtdDirectoryInfo (0x13)	FILE_ID_EXTD_DIR_INFO
FileIdExtdDirectoryRestartInfo (0x14)	FILE_ID_EXTD_DIR_INFO

Transacted Operations

If there is a transaction bound to the thread at the time of the call, then the function returns the compressed file size of the isolated file view. 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

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib; FileExtd.lib on Windows Server 2003 and Windows XP
DLL	Kernel32.dll

Redistributable

Windows SDK on Windows Server 2003 and Windows XP.

See also

[FILE_INFO_BY_HANDLE_CLASS](#)

[File Management Functions](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFileSecurityA function (winbase.h)

Article07/27/2022

The **GetFileSecurity** function obtains specified information about the security of a file or directory. The information obtained is constrained by the caller's access rights and [privileges](#).

The [GetNamedSecurityInfo](#) function provides functionality similar to **GetFileSecurity** for files as well as other types of objects.

Syntax

C++

```
BOOL GetFileSecurityA(
    [in]          LPCSTR      lpFileName,
    [in]          SECURITY_INFORMATION RequestedInformation,
    [out, optional] PSECURITY_DESCRIPTOR pSecurityDescriptor,
    [in]          DWORD       nLength,
    [out]         LPDWORD     lpnLengthNeeded
);
```

Parameters

[in] `lpFileName`

A pointer to a null-terminated string that specifies the file or directory for which security information is retrieved.

[in] `RequestedInformation`

A [SECURITY_INFORMATION](#) value that identifies the security information being requested.

[out, optional] `pSecurityDescriptor`

A pointer to a buffer that receives a copy of the [security descriptor](#) of the object specified by the *lpFileName* parameter. The calling process must have permission to view the specified aspects of the object's security status. The [SECURITY_DESCRIPTOR](#) structure is returned in [self-relative security descriptor](#) format.

[in] `nLength`

Specifies the size, in bytes, of the buffer pointed to by the *pSecurityDescriptor* parameter.

[out] *lpnLengthNeeded*

A pointer to the variable that receives the number of bytes necessary to store the complete [security descriptor](#). If the returned number of bytes is less than or equal to *nLength*, the entire security descriptor is returned in the output buffer; otherwise, none of the descriptor is returned.

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

To read the owner, group, or [DACL](#) from the security descriptor for the specified file or directory, the DACL for the file or directory must grant READ_CONTROL access to the caller, or the caller must be the owner of the file or directory.

To read the [SACL](#) of a file or directory, the SE_SECURITY_NAME privilege must be enabled for the calling process.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[GetKernelObjectSecurity](#)

[GetNamedSecurityInfo](#)

[GetPrivateObjectSecurity](#)

[GetUserObjectSecurity](#)

[Low-level Access Control](#)

[Low-level Access Control Functions](#)

[SECURITY_DESCRIPTOR](#)

[SECURITY_INFORMATION](#)

[SetFileSecurity](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFirmwareEnvironmentVariableA function (winbase.h)

Article 02/09/2023

Retrieves the value of the specified firmware environment variable.

Syntax

C++

```
DWORD GetFirmwareEnvironmentVariableA(
    [in]  LPCSTR lpName,
    [in]  LPCSTR lpGuid,
    [out] PVOID  pBuffer,
    [in]  DWORD   nSize
);
```

Parameters

[in] lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

[in] lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}" where 'x' represents a hexadecimal value.

[out] pBuffer

A pointer to a buffer that receives the value of the specified firmware environment variable.

[in] nSize

The size of the *pBuffer* buffer, in bytes.

Return value

If the function succeeds, the return value is the number of bytes stored in the *pBuffer* buffer.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include `ERROR_INVALID_FUNCTION`.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write Unified Extensible Firmware Interface (UEFI) firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

To read a firmware environment variable, the user account that the app is running under must have the `SE_SYSTEM_ENVIRONMENT_NAME` privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

Starting with Windows 10, version 1803, reading Unified Extensible Firmware Interface (UEFI) variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI variables from UMDF drivers is not supported.

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The [GetFirmwareEnvironmentVariable](#) function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with `ERROR_INVALID_FUNCTION`. On a UEFI-based system, the function will fail with an error specific to the firmware, such as `ERROR_NOACCESS`, to indicate that the dummy GUID namespace does not exist.

If you are creating a backup application, you can use this function to save all the boot settings for the system so they can be restored using the [SetFirmwareEnvironmentVariable](#) function if needed.

[GetFirmwareEnvironmentVariable](#) is the user-mode equivalent of the [ExGetFirmwareEnvironmentVariable](#) kernel-mode routine.

 **Note**

The winbase.h header defines GetFirmwareEnvironmentVariable as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariableEx](#)

[SetFirmwareEnvironmentVariable](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetFirmwareEnvironmentVariableExA function (winbase.h)

Article 02/09/2023

Retrieves the value of the specified firmware environment variable and its attributes.

Syntax

C++

```
DWORD GetFirmwareEnvironmentVariableExA(
    LPCSTR lpName,
    LPCSTR lpGuid,
    PVOID pBuffer,
    DWORD nSize,
    PDWORD pdwAttribubutes
);
```

Parameters

lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}" where 'x' represents a hexadecimal value. The pointer must not be **NULL**.

pBuffer

A pointer to a buffer that receives the value of the specified firmware environment variable.

nSize

The size of the *pValue* buffer, in bytes.

pdwAttribubutes

Bitmask identifying UEFI variable attributes associated with the variable. See [SetFirmwareEnvironmentVariableEx](#) for the bitmask definition.

Return value

If the function succeeds, the return value is the number of bytes stored in the *pValue* buffer.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include `ERROR_INVALID_FUNCTION`.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write UEFI firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

To read a UEFI firmware environment variable, the user account that the app is running under must have the `SE_SYSTEM_ENVIRONMENT_NAME` privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

Starting with Windows 10, version 1803, reading Unified Extensible Firmware Interface (UEFI) variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI variables from UMDF drivers is not supported.

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The [GetFirmwareEnvironmentVariableEx](#) function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with `ERROR_INVALID_FUNCTION`. On a UEFI-based system, the function will fail with an error

specific to the firmware, such as ERROR_NOACCESS, to indicate that the dummy GUID namespace does not exist.

If you are creating a backup application, you can use this function to save all the boot settings for the system so they can be restored using the [SetFirmwareEnvironmentVariable](#) function if needed.

ⓘ Note

The winbase.h header defines GetFirmwareEnvironmentVariableEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariable](#)

[SetFirmwareEnvironmentVariableEx](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetFirmwareEnvironmentVariableExW function (winbase.h)

Article 02/09/2023

Retrieves the value of the specified firmware environment variable and its attributes.

Syntax

C++

```
DWORD GetFirmwareEnvironmentVariableExW(
    LPCWSTR lpName,
    LPCWSTR lpGuid,
    PVOID    pBuffer,
    DWORD    nSize,
    PDWORD   pdwAttribubutes
);
```

Parameters

lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}" where 'x' represents a hexadecimal value. The pointer must not be **NULL**.

pBuffer

A pointer to a buffer that receives the value of the specified firmware environment variable.

nSize

The size of the *pValue* buffer, in bytes.

pdwAttribubutes

Bitmask identifying UEFI variable attributes associated with the variable. See [SetFirmwareEnvironmentVariableEx](#) for the bitmask definition.

Return value

If the function succeeds, the return value is the number of bytes stored in the *pValue* buffer.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include `ERROR_INVALID_FUNCTION`.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write UEFI firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

To read a UEFI firmware environment variable, the user account that the app is running under must have the `SE_SYSTEM_ENVIRONMENT_NAME` privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

Starting with Windows 10, version 1803, reading Unified Extensible Firmware Interface (UEFI) variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI variables from UMDF drivers is not supported.

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The [GetFirmwareEnvironmentVariableEx](#) function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with `ERROR_INVALID_FUNCTION`. On a UEFI-based system, the function will fail with an error

specific to the firmware, such as ERROR_NOACCESS, to indicate that the dummy GUID namespace does not exist.

If you are creating a backup application, you can use this function to save all the boot settings for the system so they can be restored using the [SetFirmwareEnvironmentVariable](#) function if needed.

ⓘ Note

The winbase.h header defines GetFirmwareEnvironmentVariableEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariable](#)

[SetFirmwareEnvironmentVariableEx](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetFirmwareEnvironmentVariableW function (winbase.h)

Article 02/09/2023

Retrieves the value of the specified firmware environment variable.

Syntax

C++

```
DWORD GetFirmwareEnvironmentVariableW(
    [in]  LPCWSTR lpName,
    [in]  LPCWSTR lpGuid,
    [out] PVOID    pBuffer,
    [in]  DWORD    nSize
);
```

Parameters

[in] lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

[in] lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}" where 'x' represents a hexadecimal value.

[out] pBuffer

A pointer to a buffer that receives the value of the specified firmware environment variable.

[in] nSize

The size of the *pBuffer* buffer, in bytes.

Return value

If the function succeeds, the return value is the number of bytes stored in the *pBuffer* buffer.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include `ERROR_INVALID_FUNCTION`.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write Unified Extensible Firmware Interface (UEFI) firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

To read a firmware environment variable, the user account that the app is running under must have the `SE_SYSTEM_ENVIRONMENT_NAME` privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

Starting with Windows 10, version 1803, reading Unified Extensible Firmware Interface (UEFI) variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI variables from UMDF drivers is not supported.

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The [GetFirmwareEnvironmentVariable](#) function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with `ERROR_INVALID_FUNCTION`. On a UEFI-based system, the function will fail with an error specific to the firmware, such as `ERROR_NOACCESS`, to indicate that the dummy GUID namespace does not exist.

If you are creating a backup application, you can use this function to save all the boot settings for the system so they can be restored using the [SetFirmwareEnvironmentVariable](#) function if needed.

[GetFirmwareEnvironmentVariable](#) is the user-mode equivalent of the [ExGetFirmwareEnvironmentVariable](#) kernel-mode routine.

 **Note**

The winbase.h header defines GetFirmwareEnvironmentVariable as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariableEx](#)

[SetFirmwareEnvironmentVariable](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetFirmwareType function (winbase.h)

Article10/13/2021

Retrieves the firmware type of the local computer.

Syntax

C++

```
BOOL GetFirmwareType(
    [in, out] PFIRMWARE_TYPE FirmwareType
);
```

Parameters

[in, out] FirmwareType

A pointer to a [FIRMWARE_TYPE](#) enumeration.

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 the [GetLastError](#) function.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[FIRMWARE_TYPE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFullPathNameTransactedA function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Retrieves the full path and file name of the specified file as a transacted operation.

To perform this operation without transactions, use the [GetFullPathName](#) function.

For more information about file and path names, see [File Names, Paths, and Namespaces](#).

Syntax

C++

```
DWORD GetFullPathNameTransactedA(
    [in]  LPCSTR lpFileName,
    [in]  DWORD   nBufferLength,
    [out] LPSTR   lpBuffer,
    [out] LPSTR   *lpFilePart,
    [in]  HANDLE  hTransaction
);
```

Parameters

[in] lpFileName

The name of the file.

This string can use short (the 8.3 form) or long file names. This string can be a share or volume name.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] nBufferLength

The size of the buffer to receive the null-terminated string for the drive and path, in **TCHARs**.

[out] *lpBuffer*

A pointer to a buffer that receives the null-terminated string for the drive and path.

[out] *lpFilePart*

A pointer to a buffer that receives the address (in *lpBuffer*) of the final file name component in the path. Specify **NULL** if you do not need to receive this information.

If *lpBuffer* points to a directory and not a file, *lpFilePart* receives 0 (zero).

[in] *hTransaction*

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to *lpBuffer*, not including the terminating null character.

If the *lpBuffer* buffer is too small to contain the path, the return value is the size, in **TCHARs**, of the buffer that is required to hold the path and the terminating null character.

If the function fails for any other reason, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

GetFullPathNameTransacted merges the name of the current drive and directory with a specified file name to determine the full path and file name of a specified file. It also calculates the address of the file name portion of the full path and file name. This function does not verify that the resulting path and file name are valid, or that they see an existing file on the associated volume.

Share and volume names are valid input for *lpFileName*. For example, the following list identifies the returned path and file names if test-2 is a remote computer and U: is a network mapped drive:

- If you specify "\\test-2\q\$\lh" the path returned is "\\test-2\q\$\lh"
- If you specify "\\?\UNC\test-2\q\$\lh" the path returned is "\\?\UNC\test-2\q\$\lh"

- If you specify "U:" the path returned is "U:\\"

GetFullPathNameTransacted does not convert the specified file name, *lpFileName*. If the specified file name exists, you can use [GetLongPathNameTransacted](#), [GetLongPathName](#), or [GetShortPathName](#) to convert to long or short path names, respectively.

If the return value is greater than the value specified in *nBufferLength*, you can call the function again with a buffer that is large enough to hold the path. For an example of this case as well as using zero length buffer for dynamic allocation, see the Example Code section.

Note Although the return value in this case is a length that includes the terminating null character, the return value on success does not include the terminating null character in the count.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Note

The winbase.h header defines **GetFullPathNameTransacted** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetFullPathName](#)

[GetLongPathNameTransacted](#)

[GetShortPathName](#)

[GetTempPath](#)

[SearchPath](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetFullPathNameTransactedW function (winbase.h)

Article02/09/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Retrieves the full path and file name of the specified file as a transacted operation.

To perform this operation without transactions, use the [GetFullPathName](#) function.

For more information about file and path names, see [File Names, Paths, and Namespaces](#).

Syntax

C++

```
DWORD GetFullPathNameTransactedW(
    [in]  LPCWSTR lpFileName,
    [in]  DWORD    nBufferLength,
    [out] LPWSTR   lpBuffer,
    [out] LPWSTR   *lpFilePart,
    [in]  HANDLE   hTransaction
);
```

Parameters

[in] lpFileName

The name of the file.

This string can use short (the 8.3 form) or long file names. This string can be a share or volume name.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] nBufferLength

The size of the buffer to receive the null-terminated string for the drive and path, in **TCHARs**.

[out] *lpBuffer*

A pointer to a buffer that receives the null-terminated string for the drive and path.

[out] *lpFilePart*

A pointer to a buffer that receives the address (in *lpBuffer*) of the final file name component in the path. Specify **NULL** if you do not need to receive this information.

If *lpBuffer* points to a directory and not a file, *lpFilePart* receives 0 (zero).

[in] *hTransaction*

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is the length, in **TCHARs**, of the string copied to *lpBuffer*, not including the terminating null character.

If the *lpBuffer* buffer is too small to contain the path, the return value is the size, in **TCHARs**, of the buffer that is required to hold the path and the terminating null character.

If the function fails for any other reason, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

GetFullPathNameTransacted merges the name of the current drive and directory with a specified file name to determine the full path and file name of a specified file. It also calculates the address of the file name portion of the full path and file name. This function does not verify that the resulting path and file name are valid, or that they see an existing file on the associated volume.

Share and volume names are valid input for *lpFileName*. For example, the following list identifies the returned path and file names if test-2 is a remote computer and U: is a network mapped drive:

- If you specify "\\test-2\q\$\lh" the path returned is "\\test-2\q\$\lh"
- If you specify "\\?\UNC\test-2\q\$\lh" the path returned is "\\?\UNC\test-2\q\$\lh"

- If you specify "U:" the path returned is "U:\\"

GetFullPathNameTransacted does not convert the specified file name, *lpFileName*. If the specified file name exists, you can use [GetLongPathNameTransacted](#), [GetLongPathName](#), or [GetShortPathName](#) to convert to long or short path names, respectively.

If the return value is greater than the value specified in *nBufferLength*, you can call the function again with a buffer that is large enough to hold the path. For an example of this case as well as using zero length buffer for dynamic allocation, see the Example Code section.

Note Although the return value in this case is a length that includes the terminating null character, the return value on success does not include the terminating null character in the count.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The winbase.h header defines **GetFullPathNameTransacted** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetFullPathName](#)

[GetLongPathNameTransacted](#)

[GetShortPathName](#)

[GetTempPath](#)

[SearchPath](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetLogicalDriveStringsA function (winbase.h)

Article 07/27/2022

Fills a buffer with strings that specify valid drives in the system.

Syntax

C++

```
DWORD GetLogicalDriveStringsA(
    [in]  DWORD nBufferLength,
    [out] LPSTR lpBuffer
);
```

Parameters

[in] `nBufferLength`

The maximum size of the buffer pointed to by `lpBuffer`, in TCHARs. This size does not include the terminating null character. If this parameter is zero, `lpBuffer` is not used.

[out] `lpBuffer`

A pointer to a buffer that receives a series of null-terminated strings, one for each valid drive in the system, plus with an additional null character. Each string is a device name.

Return value

If the function succeeds, the return value is the length, in characters, of the strings copied to the buffer, not including the terminating null character. Note that an ANSI-ASCII null character uses one byte, but a Unicode (UTF-16) null character uses two bytes.

If the buffer is not large enough, the return value is greater than `nBufferLength`. It is the size of the buffer required to hold the drive strings.

If the function fails, the return value is zero. To get extended error information, use the [GetLastError](#) function.

Remarks

Each string in the buffer may be used wherever a root directory is required, such as for the [GetDriveType](#) and [GetDiskFreeSpace](#) functions.

This function returns a concatenation of the drives in the Global and Local MS-DOS Device namespaces. If a drive exists in both namespaces, this function will return the entry in the Local MS-DOS Device namespace. For more information, see [Defining an MS DOS Device Name](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

Examples

For an example, see [Obtaining a File Name From a File Handle](#).

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)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[GetDiskFreeSpace](#)

[GetDriveType](#)

[GetLogicalDrives](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetLongPathNameTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Converts the specified path to its long form as a transacted operation.

To perform this operation without a transaction, use the [GetLongPathName](#) function.

For more information about file and path names, see [Naming Files, Paths, and Namespaces](#).

Syntax

C++

```
DWORD GetLongPathNameTransactedA(
    [in]  LPCSTR lpszShortPath,
    [out] LPSTR  lpszLongPath,
    [in]  DWORD   cchBuffer,
    [in]  HANDLE  hTransaction
);
```

Parameters

[in] lpszShortPath

The path to be converted.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The path must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[out] *lpszLongPath*

A pointer to the buffer to receive the long path.

You can use the same buffer you used for the *lpszShortPath* parameter.

[in] *cchBuffer*

The size of the buffer *lpszLongPath* points to, in TCHARs.

[in] *hTransaction*

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is the length, in TCHARs, of the string copied to *lpszLongPath*, not including the terminating null character.

If the *lpBuffer* buffer is too small to contain the path, the return value is the size, in TCHARs, of the buffer that is required to hold the path and the terminating null character.

If the function fails for any other reason, such as if the file does not exist, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

On many file systems, a short file name contains a tilde (~) character. However, not all file systems follow this convention. Therefore, do not assume that you can skip calling [GetLongPathNameTransacted](#) if the path does not contain a tilde (~) character.

If a long path is not found, this function returns the name specified in the *lpszShortPath* parameter in the *lpszLongPath* parameter.

If the return value is greater than the value specified in *cchBuffer*, you can call the function again with a buffer that is large enough to hold the path. For an example of this case, see the Example Code section for [GetFullPathName](#).

Note Although the return value in this case is a length that includes the terminating null character, the return value on success does not include the terminating null character in the count.

It is possible to have access to a file or directory but not have access to some of the parent directories of that file or directory. As a result, **GetLongPathNameTransacted** may fail when it is unable to query the parent directory of a path component to determine the long name for that component. This check can be skipped for directory components that have file extensions longer than 3 characters, or total lengths longer than 12 characters. For more information, see the [Short vs. Long Names](#) section of [Naming Files, Paths, and Namespaces](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

① Note

The winbase.h header defines **GetLongPathNameTransacted** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetFullPathNameTransacted](#)

[GetShortPathName](#)

[Naming Files, Paths, and Namespaces](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetLongPathNameTransactedW function (winbase.h)

Article 06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Converts the specified path to its long form as a transacted operation.

To perform this operation without a transaction, use the [GetLongPathName](#) function.

For more information about file and path names, see [Naming Files, Paths, and Namespaces](#).

Syntax

C++

```
DWORD GetLongPathNameTransactedW(
    [in]  LPCWSTR lpszShortPath,
    [out] LPWSTR   lpszLongPath,
    [in]  DWORD    cchBuffer,
    [in]  HANDLE   hTransaction
);
```

Parameters

`[in] lpszShortPath`

The path to be converted.

The path must reside on the local computer; otherwise, the function fails and the last error code is set to `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE`.

By default, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, prepend "`\?\`" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[out] *lpszLongPath*

A pointer to the buffer to receive the long path.

You can use the same buffer you used for the *lpszShortPath* parameter.

[in] *cchBuffer*

The size of the buffer *lpszLongPath* points to, in TCHARs.

[in] *hTransaction*

A handle to the transaction. This handle is returned by the [CreateTransaction](#) function.

Return value

If the function succeeds, the return value is the length, in TCHARs, of the string copied to *lpszLongPath*, not including the terminating null character.

If the *lpBuffer* buffer is too small to contain the path, the return value is the size, in TCHARs, of the buffer that is required to hold the path and the terminating null character.

If the function fails for any other reason, such as if the file does not exist, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

On many file systems, a short file name contains a tilde (~) character. However, not all file systems follow this convention. Therefore, do not assume that you can skip calling [GetLongPathNameTransacted](#) if the path does not contain a tilde (~) character.

If a long path is not found, this function returns the name specified in the *lpszShortPath* parameter in the *lpszLongPath* parameter.

If the return value is greater than the value specified in *cchBuffer*, you can call the function again with a buffer that is large enough to hold the path. For an example of this

case, see the Example Code section for [GetFullPathName](#).

Note Although the return value in this case is a length that includes the terminating null character, the return value on success does not include the terminating null character in the count.

It is possible to have access to a file or directory but not have access to some of the parent directories of that file or directory. As a result, [GetLongPathNameTransacted](#) may fail when it is unable to query the parent directory of a path component to determine the long name for that component. This check can be skipped for directory components that have file extensions longer than 3 characters, or total lengths longer than 12 characters. For more information, see the [Short vs. Long Names](#) section of [Naming Files, Paths, and Namespaces](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

ⓘ Note

The winbase.h header defines GetLongPathNameTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetFullPathNameTransacted](#)

[GetShortPathName](#)

[Naming Files, Paths, and Namespaces](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetMailslotInfo function (winbase.h)

Article10/13/2021

Retrieves information about the specified mailslot.

Syntax

C++

```
BOOL GetMailslotInfo(
    [in]             HANDLE hMailslot,
    [out, optional] LPDWORD lpMaxMessageSize,
    [out, optional] LPDWORD lpNextSize,
    [out, optional] LPDWORD lpMessageCount,
    [out, optional] LPDWORD lpReadTimeout
);
```

Parameters

[in] hMailslot

A handle to a mailslot. The [CreateMailslot](#) function must create this handle.

[out, optional] lpMaxMessageSize

The maximum message size, in bytes, allowed for this mailslot. This value can be greater than or equal to the value specified in the *cbMaxMsg* parameter of the [CreateMailslot](#) function that created the mailslot. This parameter can be **NULL**.

[out, optional] lpNextSize

The size of the next message, in bytes. The following value has special meaning.

Value	Meaning
MAILSLOT_NO_MESSAGE ((DWORD)-1)	There is no next message.

This parameter can be **NULL**.

[out, optional] lpMessageCount

The total number of messages waiting to be read, when the function returns. This parameter can be **NULL**.

[out, optional] lpReadTimeout

The amount of time, in milliseconds, a read operation can wait for a message to be written to the mailslot before a time-out occurs. This parameter is filled in when the function returns. This parameter can be **NULL**.

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](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateMailslot](#)

[Mailslot Functions](#)

[Mailslots Overview](#)

[SetMailslotInfo](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetMaximumProcessorCount function (winbase.h)

Article 10/13/2021

Returns the maximum number of logical processors that a processor group or the system can have.

Syntax

C++

```
DWORD GetMaximumProcessorCount(  
    [in] WORD GroupNumber  
);
```

Parameters

[in] GroupNumber

The processor group number. If this parameter is ALL_PROCESSOR_GROUPS, the function returns the maximum number of processors that the system can have.

Return value

If the function succeeds, the return value is the maximum number of processors that the specified group can have.

If the function fails, the return value is zero. To get extended error information, use [GetLastError](#).

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetMaximumProcessorGroupCount function (winbase.h)

Article06/29/2021

Returns the maximum number of processor groups that the system can have.

Syntax

C++

```
WORD GetMaximumProcessorGroupCount();
```

Return value

If the function succeeds, the return value is the maximum number of processor groups that the system can have.

If the function fails, the return value is zero.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNamedPipeClientComputerNameA function (winbase.h)

Article 02/02/2023

Retrieves the client computer name for the specified named pipe.

Syntax

C++

```
BOOL GetNamedPipeClientComputerNameA(
    [in]  HANDLE Pipe,
    [out] LPSTR ClientComputerName,
    [in]  ULONG ClientComputerNameLength
);
```

Parameters

[in] Pipe

A handle to an instance of a named pipe. This handle must be created by the [CreateNamedPipe](#) function.

[out] ClientComputerName

The computer name.

[in] ClientComputerNameLength

The size of the *ClientComputerName* buffer, in bytes.

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 the [GetLastError](#) function.

Remarks

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\pipe\LOCAL\` for the pipe name.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateNamedPipe](#)

[Pipe Functions](#)

[Pipes Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNamedPipeClientProcessId function (winbase.h)

Article 02/02/2023

Retrieves the client process identifier for the specified named pipe.

Syntax

C++

```
BOOL GetNamedPipeClientProcessId(
    [in]  HANDLE Pipe,
    [out] PULONG ClientProcessId
);
```

Parameters

[in] Pipe

A handle to an instance of a named pipe. This handle must be created by the [CreateNamedPipe](#) function.

[out] ClientProcessId

The process identifier.

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 the [GetLastError](#) function.

Remarks

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\.\pipe\LOCAL\` for the pipe name.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateNamedPipe](#)

[GetNamedPipeServerProcessId](#)

[Pipe Functions](#)

[Pipes Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNamedPipeClientSessionId function (winbase.h)

Article 02/02/2023

Retrieves the client session identifier for the specified named pipe.

Syntax

C++

```
BOOL GetNamedPipeClientSessionId(
    [in]  HANDLE Pipe,
    [out] PULONG ClientSessionId
);
```

Parameters

[in] Pipe

A handle to an instance of a named pipe. This handle must be created by the [CreateNamedPipe](#) function.

[out] ClientSessionId

The session identifier.

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 the [GetLastError](#) function.

Remarks

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\.\pipe\LOCAL\` for the pipe name.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateNamedPipe](#)

[GetNamedPipeServerSessionId](#)

[Pipe Functions](#)

[Pipes Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNamedPipeHandleStateA function (winbase.h)

Article 02/02/2023

Retrieves information about a specified named pipe. The information returned can vary during the lifetime of an instance of the named pipe.

Syntax

C++

```
BOOL GetNamedPipeHandleStateA(
    [in]             HANDLE hNamedPipe,
    [out, optional] LPDWORD lpState,
    [out, optional] LPDWORD lpCurInstances,
    [out, optional] LPDWORD lpMaxCollectionCount,
    [out, optional] LPDWORD lpCollectDataTimeout,
    [out, optional] LPSTR  lpUserName,
    [in]             DWORD  nMaxUserNameSize
);
```

Parameters

[in] hNamedPipe

A handle to the named pipe for which information is wanted. The handle must have GENERIC_READ access for a read-only or read/write pipe, or it must have GENERIC_WRITE and FILE_READ_ATTRIBUTES access for a write-only pipe.

This parameter can also be a handle to an anonymous pipe, as returned by the [CreatePipe](#) function.

[out, optional] lpState

A pointer to a variable that indicates the current state of the handle. This parameter can be **NULL** if this information is not needed. Either or both of the following values can be specified.

Value	Meaning
PIPE_NOWAIT 0x00000001	The pipe handle is in nonblocking mode. If this flag is not specified, the pipe handle is in blocking mode.

PIPE_READMODE_MESSAGE
0x00000002

The pipe handle is in message-read mode. If this flag is not specified, the pipe handle is in byte-read mode.

[out, optional] *lpCurInstances*

A pointer to a variable that receives the number of current pipe instances. This parameter can be **NULL** if this information is not required.

[out, optional] *lpMaxCollectionCount*

A pointer to a variable that receives the maximum number of bytes to be collected on the client's computer before transmission to the server. This parameter must be **NULL** if the specified pipe handle is to the server end of a named pipe or if client and server processes are on the same computer. This parameter can be **NULL** if this information is not required.

[out, optional] *lpCollectDataTimeout*

A pointer to a variable that receives the maximum time, in milliseconds, that can pass before a remote named pipe transfers information over the network. This parameter must be **NULL** if the specified pipe handle is to the server end of a named pipe or if client and server processes are on the same computer. This parameter can be **NULL** if this information is not required.

[out, optional] *lpUserName*

A pointer to a buffer that receives the user name string associated with the client application. The server can only retrieve this information if the client opened the pipe with SECURITY_IMPERSONATION access.

This parameter must be **NULL** if the specified pipe handle is to the client end of a named pipe. This parameter can be **NULL** if this information is not required.

[in] *nMaxUserNameSize*

The size of the buffer specified by the *lpUserName* parameter, in TCHARs. This parameter is ignored if *lpUserName* is **NULL**.

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

The [GetNamedPipeHandleState](#) function returns successfully even if all of the pointers passed to it are **NULL**.

To set the pipe handle state, use the [SetNamedPipeHandleState](#) function.

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\.\pipe\LOCAL\` for the pipe name.

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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Pipe Functions](#)

[Pipes Overview](#)

[SetNamedPipeHandleState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNamedPipeServerProcessId function (winbase.h)

Article 02/02/2023

Retrieves the server process identifier for the specified named pipe.

Syntax

C++

```
BOOL GetNamedPipeServerProcessId(
    [in] HANDLE Pipe,
    [out] PULONG ServerProcessId
);
```

Parameters

[in] Pipe

A handle to an instance of a named pipe. This handle must be created by the [CreateNamedPipe](#) function.

[out] ServerProcessId

The process identifier.

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 the [GetLastError](#) function.

Remarks

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\.\pipe\LOCAL\` for the pipe name.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateNamedPipe](#)

[GetNamedPipeClientProcessId](#)

[Pipe Functions](#)

[Pipes Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNamedPipeServerSessionId function (winbase.h)

Article 02/02/2023

Retrieves the server session identifier for the specified named pipe.

Syntax

C++

```
BOOL GetNamedPipeServerSessionId(
    [in]  HANDLE Pipe,
    [out] PULONG ServerSessionId
);
```

Parameters

[in] Pipe

A handle to an instance of a named pipe. This handle must be created by the [CreateNamedPipe](#) function.

[out] ServerSessionId

The session identifier.

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 the [GetLastError](#) function.

Remarks

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\.\pipe\LOCAL\` for the pipe name.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateNamedPipe](#)

[GetNamedPipeClientSessionId](#)

[Pipe Functions](#)

[Pipes Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNextUmsListItem function (winbase.h)

Article03/18/2022

Returns the next user-mode scheduling (UMS) thread context in a list of thread contexts.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
PUMS_CONTEXT GetNextUmsListItem(  
    [in, out] PUMS_CONTEXT UmsContext  
)
```

Parameters

`[in, out] UmsContext`

A pointer to a UMS context in a list of thread contexts. This list is retrieved by the [DequeueUmsCompletionListItems](#) function.

Return value

If the function succeeds, it returns a pointer to the next thread context in the list.

If there is no thread context after the context specified by the *UmsContext* parameter, the function returns NULL. To get extended error information, call [GetLastError](#).

Requirements

Minimum supported client

Windows 7 (64-bit only) [desktop apps only]

Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[DequeueUmsCompletionListItems](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaAvailableMemoryNode function (winbase.h)

Article10/13/2021

Retrieves the amount of memory available in the specified node.

Use the [GetNumaAvailableMemoryNodeEx](#) function to specify the node as a **USHORT** value.

Syntax

C++

```
BOOL GetNumaAvailableMemoryNode(
    [in] UCHAR     Node,
    [out] PULONGLONG AvailableBytes
);
```

Parameters

[in] Node

The number of the node.

[out] AvailableBytes

The amount of available memory for the node, in bytes.

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

The **GetNumaAvailableMemoryNode** function returns the amount of memory consumed by free and zeroed pages on the specified node. On systems with more than one node, this memory does not include standby pages. Therefore, the sum of the

available memory values for all nodes in the system is equal to the value of the Free & Zero Page List Bytes memory performance counter. On systems with only one node, the value returned by `GetNumaAvailableMemoryNode` includes standby pages and is equal to the value of the Available Bytes memory performance counter. For more information about performance counters, see [Memory Performance Information](#).

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetNumaAvailableMemoryNodeEx](#)

[NUMA Support](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaAvailableMemoryNodeEx function (winbase.h)

Article 10/13/2021

Retrieves the amount of memory that is available in a node specified as a **USHORT** value.

Syntax

C++

```
BOOL GetNumaAvailableMemoryNodeEx(
    [in] USHORT     Node,
    [out] PULONGLONG AvailableBytes
);
```

Parameters

[in] Node

The number of the node.

[out] AvailableBytes

The amount of available memory for the node, in bytes.

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

The **GetNumaAvailableMemoryNodeEx** function returns the amount of memory consumed by free and zeroed pages on the specified node. On systems with more than one node, this memory does not include standby pages. Therefore, the sum of the available memory values for all nodes in the system is equal to the value of the Free &

Zero Page List Bytes memory performance counter. On systems with only one node, the value returned by [GetNumaAvailableMemoryNode](#) includes standby pages and is equal to the value of the Available Bytes memory performance counter. For more information about performance counters, see [Memory Performance Information](#).

The only difference between the [GetNumaAvailableMemoryNodeEx](#) function and the [GetNumaAvailableMemoryNode](#) function is the data type of the *Node* parameter.

To compile an application that uses this function, set `_WIN32_WINNT >= 0x0601`. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetNumaAvailableMemoryNode](#)

[NUMA Support](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaNodeNumberFromHandle function (winbase.h)

Article10/13/2021

Retrieves the NUMA node associated with the file or I/O device represented by the specified file handle.

Syntax

C++

```
BOOL GetNumaNodeNumberFromHandle(
    [in] HANDLE hFile,
    [out] PUSHORT NodeNumber
);
```

Parameters

[in] `hFile`

A handle to a file or I/O device. Examples of I/O devices include files, file streams, volumes, physical disks, and sockets. For more information, see the [CreateFile](#) function.

[out] `NodeNumber`

A pointer to a variable to receive the number of the NUMA node associated with the specified file handle.

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, use [GetLastError](#).

Remarks

If the specified handle does not have a node associated with it, the function returns FALSE. The value of the `NodeNumber` parameter is undetermined and should not be

used.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[NUMA Support](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaNodeProcessorMask function (winbase.h)

Article 10/13/2021

Retrieves the processor mask for the specified node.

Syntax

C++

```
BOOL GetNumaNodeProcessorMask(
    [in] UCHAR Node,
    [out] PULONGLONG ProcessorMask
);
```

Parameters

[in] Node

The number of the node.

[out] ProcessorMask

The processor mask for the node. A processor mask is a bit vector in which each bit represents a processor and whether it is in the node.

If the node has no processors configured, the processor mask is zero.

On systems with more than 64 processors, this parameter is set to the processor mask for the node only if the node is in the same [processor group](#) as the calling thread. Otherwise, the parameter is set to zero.

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

To retrieve the highest numbered node in the system, use the [GetNumaHighestNodeNumber](#) function. Note that this number is not guaranteed to equal the total number of nodes in the system.

To ensure that all threads for your process run on the same node, use the [SetProcessAffinityMask](#) function with a process affinity mask that specifies processors in the same node.

Use the [GetNumaNodeProcessorMaskEx](#) function to retrieve the processor mask for a node in any processor group.

 **Note**

Starting with *TBD Release Iron*, the behavior of this and other NUMA functions has been modified to better support systems with nodes containing more than 64 processors. For more information about this change, including information about enabling the old behavior of this API, see [NUMA Support](#).

Behavior starting with TBD Release Iron

Each node is assigned a primary group by the system. The mask returned by [GetNumaNodeProcessorMask](#) is for the node's primary group and is only returned if the calling thread belongs to that group.

Behavior in previous versions

The mask for the specified node is returned.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[GetNumaNodeProcessorMaskEx](#)

[GetNumaProcessorNode](#)

[NUMA Support](#)

[Process and Thread Functions](#)

[SetProcessAffinityMask](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaProcessorNode function (winbase.h)

Article 10/13/2021

Retrieves the node number for the specified processor.

Use the [GetNumaProcessorNodeEx](#) function to specify a processor group and retrieve the node number as a **USHORT** value.

Syntax

C++

```
BOOL GetNumaProcessorNode(
    [in] UCHAR Processor,
    [out] PUCHAR NodeNumber
);
```

Parameters

[in] Processor

The processor number.

On a system with more than 64 logical processors, the processor number is relative to the [processor group](#) that contains the processor on which the calling thread is running.

[out] NodeNumber

The node number. If the processor does not exist, this parameter is 0xFF.

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

To retrieve the list of processors on the system, use the [GetProcessAffinityMask](#) function.

Examples

For an example, see [Allocating Memory from a NUMA Node](#).

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetNumaNodeProcessorMask](#)

[GetNumaProcessorNodeEx](#)

[GetNumaProximityNode](#)

[GetProcessAffinityMask](#)

[NUMA Support](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaProcessorNodeEx function (winbase.h)

Article 10/13/2021

Retrieves the node number as a **USHORT** value for the specified logical processor.

Syntax

C++

```
BOOL GetNumaProcessorNodeEx(
    [in] PPROCESSOR_NUMBER Processor,
    [out] PUSHORT           NodeNumber
);
```

Parameters

[in] Processor

A pointer to a **PROCESSOR_NUMBER** structure that represents the logical processor and the processor group to which it is assigned.

[out] NodeNumber

A pointer to a variable to receive the node number. If the specified processor does not exist, this parameter is set to MAXUSHORT.

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

To compile an application that uses this function, set `_WIN32_WINNT >= 0x0601`. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetNumaProcessorNode](#)

[PROCESSOR_NUMBER](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumaProximityNode function (winbase.h)

Article 08/23/2022

Retrieves the NUMA node number that corresponds to the specified proximity domain identifier.

Use the [GetNumaProximityNodeEx](#) function to retrieve the node number as a **USHORT** value.

Syntax

C++

```
BOOL GetNumaProximityNode(
    [in] ULONG ProximityId,
    [out] P UCHAR NodeNumber
);
```

Parameters

[in] ProximityId

The proximity domain identifier of the node.

[out] NodeNumber

The node number. If the processor does not exist, this parameter is 0xFF.

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

A proximity domain identifier is an index to a NUMA node on a NUMA system.

Proximity domain identifiers are found in the ACPI System Resource Affinity Table

(SRAT), where they are used to associate processors and memory regions with a particular NUMA node. Proximity domain identifiers are also found in the ACPI namespace, where they are used to associate a device with a particular NUMA node. Proximity domain identifiers are typically used only by management applications provided by system manufacturers. Windows does not use proximity domain identifiers to identify NUMA nodes; instead, it assigns a unique number to each NUMA node in the system.

The relative distance between nodes on a system is stored in the ACPI System Locality Distance Information Table (SLIT), which is not exposed by any Windows functions. For more information about ACPI tables, see the [ACPI specifications](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetNumaProcessorNode](#)

[GetNumaProximityNodeEx](#)

[NUMA Support](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetNumberOfEventLogRecords function (winbase.h)

Article 10/13/2021

Retrieves the number of records in the specified event log.

Syntax

C++

```
BOOL GetNumberOfEventLogRecords(
    [in] HANDLE hEventLog,
    [out] PDWORD NumberOfRecords
);
```

Parameters

[in] hEventLog

A handle to the open event log. The [OpenEventLog](#) or [OpenBackupEventLog](#) function returns this handle.

[out] NumberOfRecords

A pointer to a variable that receives the number of records in the specified event log.

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

The oldest record in an event log is not necessarily record number 1. To determine the oldest record number in an event log, use the [GetOldestEventLogRecord](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-l1-1-1 (introduced in Windows 10, version 10.0.10240)

See also

[Event Logging Functions](#)

[GetOldestEventLogRecord](#)

[OpenBackupEventLog](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetOldestEventLogRecord function (winbase.h)

Article 10/13/2021

Retrieves the absolute record number of the oldest record in the specified event log.

Syntax

C++

```
BOOL GetOldestEventLogRecord(
    [in] HANDLE hEventLog,
    [out] PDWORD OldestRecord
);
```

Parameters

[in] `hEventLog`

A handle to the open event log. The [OpenEventLog](#) or [OpenBackupEventLog](#) function returns this handle.

[out] `OldestRecord`

A pointer to a variable that receives the absolute record number of the oldest record in the specified event log.

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

The oldest record in an event log is not necessarily record number 1. For more information, see [Event Log Records](#).

Examples

For an example, see [Querying for Event Information](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-l1-1-1 (introduced in Windows 10, version 10.0.10240)

See also

[Event Logging Functions](#)

[GetNumberOfEventLogRecords](#)

[OpenBackupEventLog](#)

[OpenEventLog](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileInt function (winbase.h)

Article09/22/2022

Retrieves an integer associated with a key in the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
UINT GetPrivateProfileInt(
    [in] LPCTSTR lpAppName,
    [in] LPCTSTR lpKeyName,
    [in] INT     nDefault,
    [in] LPCTSTR lpFileName
);
```

Parameters

[in] lpAppName

The name of the section in the initialization file.

[in] lpKeyName

The name of the key whose value is to be retrieved. This value is in the form of a string; the **GetPrivateProfileInt** function converts the string into an integer and returns the integer.

[in] nDefault

The default value to return if the key name cannot be found in the initialization file.

[in] lpFileName

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value is the integer equivalent of the string following the specified key name in the specified initialization file. If the key is not found, the return value is the specified default value.

Remarks

The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the section name specified by the *lpAppName* parameter. A section in the initialization file must have the following form:

syntax

```
[section]
key=value
.
.
.
```

The **GetPrivateProfileInt** function is not case-sensitive; the strings in *lpAppName* and *lpKeyName* can be a combination of uppercase and lowercase letters.

An application can use the [GetProfileInt](#) function to retrieve an integer value from the Win.ini file.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileInt](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetPrivateProfileIntA function (winbase.h)

Article02/09/2023

Retrieves an integer associated with a key in the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
UINT GetPrivateProfileIntA(
    [in] LPCSTR lpAppName,
    [in] LPCSTR lpKeyName,
    [in] INT     nDefault,
    [in] LPCSTR lpFileName
);
```

Parameters

[in] lpAppName

The name of the section in the initialization file.

[in] lpKeyName

The name of the key whose value is to be retrieved. This value is in the form of a string; the **GetPrivateProfileInt** function converts the string into an integer and returns the integer.

[in] nDefault

The default value to return if the key name cannot be found in the initialization file.

[in] lpFileName

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value is the integer equivalent of the string following the specified key name in the specified initialization file. If the key is not found, the return value is the specified default value.

Remarks

The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the section name specified by the *lpAppName* parameter. A section in the initialization file must have the following form:

syntax

```
[section]
key=value
.
.
.
```

The **GetPrivateProfileInt** function is not case-sensitive; the strings in *lpAppName* and *lpKeyName* can be a combination of uppercase and lowercase letters.

An application can use the [GetProfileInt](#) function to retrieve an integer value from the Win.ini file.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping`

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.

2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines GetPrivateProfileInt as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileInt](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileIntW function (winbase.h)

Article02/09/2023

Retrieves an integer associated with a key in the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
UINT GetPrivateProfileIntW(
    [in] LPCWSTR lpAppName,
    [in] LPCWSTR lpKeyName,
    [in] INT     nDefault,
    [in] LPCWSTR lpFileName
);
```

Parameters

[in] lpAppName

The name of the section in the initialization file.

[in] lpKeyName

The name of the key whose value is to be retrieved. This value is in the form of a string; the **GetPrivateProfileInt** function converts the string into an integer and returns the integer.

[in] nDefault

The default value to return if the key name cannot be found in the initialization file.

[in] lpFileName

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value is the integer equivalent of the string following the specified key name in the specified initialization file. If the key is not found, the return value is the specified default value.

Remarks

The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the section name specified by the *lpAppName* parameter. A section in the initialization file must have the following form:

syntax

```
[section]
key=value
.
.
.
```

The **GetPrivateProfileInt** function is not case-sensitive; the strings in *lpAppName* and *lpKeyName* can be a combination of uppercase and lowercase letters.

An application can use the [GetProfileInt](#) function to retrieve an integer value from the Win.ini file.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping`

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.

2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines GetPrivateProfileInt as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileInt](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileSection function (winbase.h)

Article09/22/2022

Retrieves all the keys and values for the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit applications written for Windows. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileSection(
    [in] LPCTSTR lpAppName,
    [out] LPTSTR lpReturnedString,
    [in] DWORD nSize,
    [in] LPCTSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section in the initialization file.

[out] *lpReturnedString*

A pointer to a buffer that receives the key name and value pairs associated with the named section. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

[in] *nSize*

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

The maximum profile section size is 32,767 characters.

[in] *lpFileName*

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value specifies the number of characters copied to the buffer, not including the terminating null character. If the buffer is not large enough to contain all the key name and value pairs associated with the named section, the return value is equal to *nSize* minus two.

Remarks

The data in the buffer pointed to by the *lpReturnedString* parameter consists of one or more null-terminated strings, followed by a final null character. Each string has the following format:

key=string

The **GetPrivateProfileSection** function is not case-sensitive; the string pointed to by the *lpAppName* parameter can be a combination of uppercase and lowercase letters.

This operation is atomic; no updates to the specified initialization file are allowed while the key name and value pairs for the section are being copied to the buffer pointed to by the *lpReturnedString* parameter.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this

name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Comments (any line that starts with a semicolon) are stripped out and not returned in the *lpReturnedString* buffer.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSectionNames](#)

[GetProfileSection](#)

[WritePrivateProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileSectionA function (winbase.h)

Article02/09/2023

Retrieves all the keys and values for the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit applications written for Windows. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileSection(
    [in]  LPCSTR lpAppName,
    [out] LPSTR  lpReturnedString,
    [in]  DWORD   nSize,
    [in]  LPCSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section in the initialization file.

[out] *lpReturnedString*

A pointer to a buffer that receives the key name and value pairs associated with the named section. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

[in] *nSize*

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

The maximum profile section size is 32,767 characters.

[in] *lpFileName*

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value specifies the number of characters copied to the buffer, not including the terminating null character. If the buffer is not large enough to contain all the key name and value pairs associated with the named section, the return value is equal to *nSize* minus two.

Remarks

The data in the buffer pointed to by the *lpReturnedString* parameter consists of one or more null-terminated strings, followed by a final null character. Each string has the following format:

key=string

The **GetPrivateProfileSection** function is not case-sensitive; the string pointed to by the *lpAppName* parameter can be a combination of uppercase and lowercase letters.

This operation is atomic; no updates to the specified initialization file are allowed while the key name and value pairs for the section are being copied to the buffer pointed to by the *lpReturnedString* parameter.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this

name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Comments (any line that starts with a semicolon) are stripped out and not returned in the *lpReturnedString* buffer.

Note

The winbase.h header defines GetPrivateProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSectionNames](#)

[GetProfileSection](#)

[WritePrivateProfileSection](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetPrivateProfileSectionNames function (winbase.h)

Article 09/22/2022

Retrieves the names of all sections in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileSectionNames(
    [out] LPTSTR lpszReturnBuffer,
    [in]  DWORD   nSize,
    [in]  LPCTSTR lpFileName
);
```

Parameters

[out] `lpszReturnBuffer`

A pointer to a buffer that receives the section names associated with the named file. The buffer is filled with one or more **null**-terminated strings; the last string is followed by a second **null** character.

[in] `nSize`

The size of the buffer pointed to by the *lpszReturnBuffer* parameter, in characters.

[in] `lpFileName`

The name of the initialization file. If this parameter is **NULL**, the function searches the Win.ini file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value specifies the number of characters copied to the specified buffer, not including the terminating **null** character. If the buffer is not large enough to contain all the section names associated with the specified initialization file, the return value is equal to the size specified by *nSize* minus two.

Remarks

This operation is atomic; no updates to the initialization file are allowed while the section names are being copied to the buffer.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows`

`NT\CurrentVersion\IniFileMapping`

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

[WritePrivateProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileSectionNamesA function (winbase.h)

Article 02/09/2023

Retrieves the names of all sections in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileSectionNamesA(
    [out] LPSTR lpszReturnBuffer,
    [in]  DWORD  nSize,
    [in]  LPCSTR lpFileName
);
```

Parameters

[out] `lpszReturnBuffer`

A pointer to a buffer that receives the section names associated with the named file. The buffer is filled with one or more **null**-terminated strings; the last string is followed by a second **null** character.

[in] `nSize`

The size of the buffer pointed to by the *lpszReturnBuffer* parameter, in characters.

[in] `lpFileName`

The name of the initialization file. If this parameter is **NULL**, the function searches the Win.ini file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value specifies the number of characters copied to the specified buffer, not including the terminating **null** character. If the buffer is not large enough to contain all the section names associated with the specified initialization file, the return value is equal to the size specified by *nSize* minus two.

Remarks

This operation is atomic; no updates to the initialization file are allowed while the section names are being copied to the buffer.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows`

`NT\CurrentVersion\IniFileMapping`

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines GetPrivateProfileSectionNames as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileSectionNamesW function (winbase.h)

Article 02/09/2023

Retrieves the names of all sections in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileSectionNamesW(
    [out] LPWSTR lpszReturnBuffer,
    [in]  DWORD   nSize,
    [in]  LPCWSTR lpFileName
);
```

Parameters

[out] `lpszReturnBuffer`

A pointer to a buffer that receives the section names associated with the named file. The buffer is filled with one or more **null**-terminated strings; the last string is followed by a second **null** character.

[in] `nSize`

The size of the buffer pointed to by the *lpszReturnBuffer* parameter, in characters.

[in] `lpFileName`

The name of the initialization file. If this parameter is **NULL**, the function searches the Win.ini file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value specifies the number of characters copied to the specified buffer, not including the terminating **null** character. If the buffer is not large enough to contain all the section names associated with the specified initialization file, the return value is equal to the size specified by *nSize* minus two.

Remarks

This operation is atomic; no updates to the initialization file are allowed while the section names are being copied to the buffer.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <**No Name**>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <**No Name**>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines GetPrivateProfileSectionNames as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileSectionW function (winbase.h)

Article02/09/2023

Retrieves all the keys and values for the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit applications written for Windows. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileSection(
    [in]  LPCWSTR lpAppName,
    [out] LPWSTR  lpReturnedString,
    [in]  DWORD   nSize,
    [in]  LPCWSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section in the initialization file.

[out] *lpReturnedString*

A pointer to a buffer that receives the key name and value pairs associated with the named section. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

[in] *nSize*

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

The maximum profile section size is 32,767 characters.

[in] *lpFileName*

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value specifies the number of characters copied to the buffer, not including the terminating null character. If the buffer is not large enough to contain all the key name and value pairs associated with the named section, the return value is equal to *nSize* minus two.

Remarks

The data in the buffer pointed to by the *lpReturnedString* parameter consists of one or more null-terminated strings, followed by a final null character. Each string has the following format:

key=string

The **GetPrivateProfileSection** function is not case-sensitive; the string pointed to by the *lpAppName* parameter can be a combination of uppercase and lowercase letters.

This operation is atomic; no updates to the specified initialization file are allowed while the key name and value pairs for the section are being copied to the buffer pointed to by the *lpReturnedString* parameter.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this

name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Comments (any line that starts with a semicolon) are stripped out and not returned in the *lpReturnedString* buffer.

Note

The winbase.h header defines GetPrivateProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSectionNames](#)

[GetProfileSection](#)

[WritePrivateProfileSection](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetPrivateProfileString function (winbase.h)

Article09/22/2022

Retrieves a string from the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileString(
    [in] LPCTSTR lpAppName,
    [in] LPCTSTR lpKeyName,
    [in] LPCTSTR lpDefault,
    [out] LPTSTR lpReturnedString,
    [in] DWORD nSize,
    [in] LPCTSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section containing the key name. If this parameter is **NULL**, the **GetPrivateProfileString** function copies all section names in the file to the supplied buffer.

[in] *lpKeyName*

The name of the key whose associated string is to be retrieved. If this parameter is **NULL**, all key names in the section specified by the *lpAppName* parameter are copied to the buffer specified by the *lpReturnedString* parameter.

[in] *lpDefault*

A default string. If the *lpKeyName* key cannot be found in the initialization file, **GetPrivateProfileString** copies the default string to the *lpReturnedString* buffer.

If this parameter is **NULL**, the default is an empty string, "".

Avoid specifying a default string with trailing blank characters. The function inserts a **null** character in the *lpReturnedString* buffer to strip any trailing blanks.

[out] lpReturnedString

A pointer to the buffer that receives the retrieved string.

[in] nSize

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

[in] lpFileName

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value is the number of characters copied to the buffer, not including the terminating **null** character.

If neither *lpAppName* nor *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold the requested string, the string is truncated and followed by a **null** character, and the return value is equal to *nSize* minus one.

If either *lpAppName* or *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters. In this case, the return value is equal to *nSize* minus two.

In the event the initialization file specified by *lpFileName* is not found, or contains invalid values, calling **GetLastError** will return '0x2' (File Not Found). To retrieve extended error information, call [GetLastError](#).

Remarks

The **GetPrivateProfileString** function searches the specified initialization file for a key that matches the name specified by the *lpKeyName* parameter under the section heading specified by the *lpAppName* parameter. If it finds the key, the function copies the corresponding string to the buffer. If the key does not exist, the function copies the

default character string specified by the *lpDefault* parameter. A section in the initialization file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

If *lpAppName* is **NULL**, **GetPrivateProfileString** copies all section names in the specified file to the supplied buffer. If *lpKeyName* is **NULL**, the function copies all key names in the specified section to the supplied buffer. An application can use this method to enumerate all of the sections and keys in a file. In either case, each string is followed by a **null** character and the final string is followed by a second **null** character. If the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters.

If the string associated with *lpKeyName* is enclosed in single or double quotation marks, the marks are discarded when the **GetPrivateProfileString** function retrieves the string.

The **GetPrivateProfileString** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

To retrieve a string from the Win.ini file, use the [GetProfileString](#) function.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Example

The following example demonstrates the use of **GetPrivateProfileString**.

C++

```
// Gets a profile string called "Preferred line" and converts it to an int.
GetPrivateProfileString (
    "Preference",
    "Preferred Line",
    gszNULL,
    szBuffer,
    MAXBUFSIZE,
    gszINIfilename
);
```

```
// if szBuffer is not empty.  
if ( lstrcmp ( gszNULL, szBuffer ) )  
{  
    dwPreferredPLID = (DWORD) atoi( szBuffer );  
}  
else  
{  
    dwPreferredPLID = (DWORD) -1;  
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileString](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileStringA function (winbase.h)

Article02/09/2023

Retrieves a string from the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileStringA(
    [in]  LPCSTR lpAppName,
    [in]  LPCSTR lpKeyName,
    [in]  LPCSTR lpDefault,
    [out] LPSTR  lpReturnedString,
    [in]  DWORD   nSize,
    [in]  LPCSTR lpFileName
);
```

Parameters

[in] `lpAppName`

The name of the section containing the key name. If this parameter is **NULL**, the **GetPrivateProfileString** function copies all section names in the file to the supplied buffer.

[in] `lpKeyName`

The name of the key whose associated string is to be retrieved. If this parameter is **NULL**, all key names in the section specified by the *lpAppName* parameter are copied to the buffer specified by the *lpReturnedString* parameter.

[in] `lpDefault`

A default string. If the *lpKeyName* key cannot be found in the initialization file, **GetPrivateProfileString** copies the default string to the *lpReturnedString* buffer.

If this parameter is **NULL**, the default is an empty string, "".

Avoid specifying a default string with trailing blank characters. The function inserts a **null** character in the *lpReturnedString* buffer to strip any trailing blanks.

[out] lpReturnedString

A pointer to the buffer that receives the retrieved string.

[in] nSize

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

[in] lpFileName

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value is the number of characters copied to the buffer, not including the terminating **null** character.

If neither *lpAppName* nor *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold the requested string, the string is truncated and followed by a **null** character, and the return value is equal to *nSize* minus one.

If either *lpAppName* or *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters. In this case, the return value is equal to *nSize* minus two.

In the event the initialization file specified by *lpFileName* is not found, or contains invalid values, this function will set **errno** with a value of '0x2' (File Not Found). To retrieve extended error information, call [GetLastError](#).

Remarks

The **GetPrivateProfileString** function searches the specified initialization file for a key that matches the name specified by the *lpKeyName* parameter under the section heading specified by the *lpAppName* parameter. If it finds the key, the function copies the corresponding string to the buffer. If the key does not exist, the function copies the

default character string specified by the *lpDefault* parameter. A section in the initialization file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

If *lpAppName* is **NULL**, **GetPrivateProfileString** copies all section names in the specified file to the supplied buffer. If *lpKeyName* is **NULL**, the function copies all key names in the specified section to the supplied buffer. An application can use this method to enumerate all of the sections and keys in a file. In either case, each string is followed by a **null** character and the final string is followed by a second **null** character. If the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters.

If the string associated with *lpKeyName* is enclosed in single or double quotation marks, the marks are discarded when the **GetPrivateProfileString** function retrieves the string.

The **GetPrivateProfileString** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

To retrieve a string from the Win.ini file, use the [GetProfileString](#) function.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Example

The following example demonstrates the use of **GetPrivateProfileString**.

C++

```
// Gets a profile string called "Preferred line" and converts it to an int.
GetPrivateProfileString (
    "Preference",
    "Preferred Line",
    gszNULL,
    szBuffer,
    MAXBUFSIZE,
    gszINIfilename
);
```

```
// if szBuffer is not empty.  
if ( lstrcmp ( gszNULL, szBuffer ) )  
{  
    dwPreferredPLID = (DWORD) atoi( szBuffer );  
}  
else  
{  
    dwPreferredPLID = (DWORD) -1;  
}
```

ⓘ Note

The winbase.h header defines GetPrivateProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileString](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetPrivateProfileStringW function (winbase.h)

Article02/09/2023

Retrieves a string from the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetPrivateProfileStringW(
    [in]  LPCWSTR lpAppName,
    [in]  LPCWSTR lpKeyName,
    [in]  LPCWSTR lpDefault,
    [out] LPWSTR  lpReturnedString,
    [in]  DWORD   nSize,
    [in]  LPCWSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section containing the key name. If this parameter is **NULL**, the **GetPrivateProfileString** function copies all section names in the file to the supplied buffer.

[in] *lpKeyName*

The name of the key whose associated string is to be retrieved. If this parameter is **NULL**, all key names in the section specified by the *lpAppName* parameter are copied to the buffer specified by the *lpReturnedString* parameter.

[in] *lpDefault*

A default string. If the *lpKeyName* key cannot be found in the initialization file, **GetPrivateProfileString** copies the default string to the *lpReturnedString* buffer.

If this parameter is **NULL**, the default is an empty string, "".

Avoid specifying a default string with trailing blank characters. The function inserts a **null** character in the *lpReturnedString* buffer to strip any trailing blanks.

[out] lpReturnedString

A pointer to the buffer that receives the retrieved string.

[in] nSize

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

[in] lpFileName

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

The return value is the number of characters copied to the buffer, not including the terminating **null** character.

If neither *lpAppName* nor *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold the requested string, the string is truncated and followed by a **null** character, and the return value is equal to *nSize* minus one.

If either *lpAppName* or *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters. In this case, the return value is equal to *nSize* minus two.

In the event the initialization file specified by *lpFileName* is not found, or contains invalid values, this function will set **errno** with a value of '0x2' (File Not Found). To retrieve extended error information, call [GetLastError](#).

Remarks

The **GetPrivateProfileString** function searches the specified initialization file for a key that matches the name specified by the *lpKeyName* parameter under the section heading specified by the *lpAppName* parameter. If it finds the key, the function copies the corresponding string to the buffer. If the key does not exist, the function copies the

default character string specified by the *lpDefault* parameter. A section in the initialization file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

If *lpAppName* is **NULL**, **GetPrivateProfileString** copies all section names in the specified file to the supplied buffer. If *lpKeyName* is **NULL**, the function copies all key names in the specified section to the supplied buffer. An application can use this method to enumerate all of the sections and keys in a file. In either case, each string is followed by a **null** character and the final string is followed by a second **null** character. If the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters.

If the string associated with *lpKeyName* is enclosed in single or double quotation marks, the marks are discarded when the **GetPrivateProfileString** function retrieves the string.

The **GetPrivateProfileString** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

To retrieve a string from the Win.ini file, use the [GetProfileString](#) function.

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.

3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Example

The following example demonstrates the use of **GetPrivateProfileString**.

C++

```
// Gets a profile string called "Preferred line" and converts it to an int.
GetPrivateProfileString (
    "Preference",
    "Preferred Line",
    gszNULL,
    szBuffer,
    MAXBUFSIZE,
    gszINIfilename
);
```

```
// if szBuffer is not empty.  
if ( lstrcmp ( gszNULL, szBuffer ) )  
{  
    dwPreferredPLID = (DWORD) atoi( szBuffer );  
}  
else  
{  
    dwPreferredPLID = (DWORD) -1;  
}
```

ⓘ Note

The winbase.h header defines GetPrivateProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileString](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetPrivateProfileStruct function (winbase.h)

Article09/22/2022

Retrieves the data associated with a key in the specified section of an initialization file. As it retrieves the data, the function calculates a checksum and compares it with the checksum calculated by the [WritePrivateProfileStruct](#) function when the data was added to the file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL GetPrivateProfileStruct(
    [in] LPCTSTR lpszSection,
    [in] LPCTSTR lpszKey,
    [out] LPVOID lpStruct,
    [in] UINT     uSizeStruct,
    [in] LPCTSTR szFile
);
```

Parameters

`[in] lpszSection`

The name of the section in the initialization file.

`[in] lpszKey`

The name of the key whose data is to be retrieved.

`[out] lpStruct`

A pointer to the buffer that receives the data associated with the file, section, and key names.

[in] *uSizeStruct*

The size of the buffer pointed to by the *lpStruct* parameter, in bytes.

[in] *szFile*

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=data
.
.
.
```

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say MyFile.ini, under **IniFileMapping**.
2. Look for the section name specified by *lpAppName*. This will be a named value under **myfile.ini**, or a subkey of **myfile.ini**, or will not exist.

3. If the section name specified by *lpAppName* is a named value under **myfile.ini**, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey of **myfile.ini**, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey under **myfile.ini**, then there will be an unnamed value (shown as <No Name>) under **myfile.ini** that specifies the default location in the registry where you will find the keys for the section.
6. If there is no **myfile.ini** subkey, or if it does not contain an entry for the section name, then look for the actual MyFile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[WritePrivateProfileStruct](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetPrivateProfileStructA function (winbase.h)

Article02/09/2023

Retrieves the data associated with a key in the specified section of an initialization file. As it retrieves the data, the function calculates a checksum and compares it with the checksum calculated by the [WritePrivateProfileStruct](#) function when the data was added to the file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL GetPrivateProfileStructA(
    [in]  LPCSTR lpszSection,
    [in]  LPCSTR lpszKey,
    [out] LPVOID lpStruct,
    [in]  UINT   uSizeStruct,
    [in]  LPCSTR szFile
);
```

Parameters

[in] lpszSection

The name of the section in the initialization file.

[in] lpszKey

The name of the key whose data is to be retrieved.

[out] lpStruct

A pointer to the buffer that receives the data associated with the file, section, and key names.

[in] *uSizeStruct*

The size of the buffer pointed to by the *lpStruct* parameter, in bytes.

[in] *szFile*

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=data
.
.
.
```

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say MyFile.ini, under **IniFileMapping**.
2. Look for the section name specified by *lpAppName*. This will be a named value under **myfile.ini**, or a subkey of **myfile.ini**, or will not exist.

3. If the section name specified by *lpAppName* is a named value under **myfile.ini**, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey of **myfile.ini**, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey under **myfile.ini**, then there will be an unnamed value (shown as **<No Name>**) under **myfile.ini** that specifies the default location in the registry where you will find the keys for the section.
6. If there is no **myfile.ini** subkey, or if it does not contain an entry for the section name, then look for the actual MyFile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines GetPrivateProfileStruct as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[WritePrivateProfileStruct](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetPrivateProfileStructW function (winbase.h)

Article02/09/2023

Retrieves the data associated with a key in the specified section of an initialization file. As it retrieves the data, the function calculates a checksum and compares it with the checksum calculated by the [WritePrivateProfileStruct](#) function when the data was added to the file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL GetPrivateProfileStructW(
    [in]  LPCWSTR lpszSection,
    [in]  LPCWSTR lpszKey,
    [out] LPVOID  lpStruct,
    [in]  UINT     uSizeStruct,
    [in]  LPCWSTR szFile
);
```

Parameters

`[in] lpszSection`

The name of the section in the initialization file.

`[in] lpszKey`

The name of the key whose data is to be retrieved.

`[out] lpStruct`

A pointer to the buffer that receives the data associated with the file, section, and key names.

[in] *uSizeStruct*

The size of the buffer pointed to by the *lpStruct* parameter, in bytes.

[in] *szFile*

The name of the initialization file. If this parameter does not contain a full path to the file, the system searches for the file in the Windows directory.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=data
.
.
.
```

The system maps most .ini file references to the registry, using the mapping defined under the following registry

key:HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\IniFileMapping

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In these cases, the function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file, say MyFile.ini, under **IniFileMapping**.
2. Look for the section name specified by *lpAppName*. This will be a named value under **myfile.ini**, or a subkey of **myfile.ini**, or will not exist.

3. If the section name specified by *lpAppName* is a named value under **myfile.ini**, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey of **myfile.ini**, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey under **myfile.ini**, then there will be an unnamed value (shown as **<No Name>**) under **myfile.ini** that specifies the default location in the registry where you will find the keys for the section.
6. If there is no **myfile.ini** subkey, or if it does not contain an entry for the section name, then look for the actual MyFile.ini on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines GetPrivateProfileStruct as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[WritePrivateProfileStruct](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetProcessAffinityMask function (winbase.h)

Article01/27/2022

Retrieves the process affinity mask for the specified process and the system affinity mask for the system.

Syntax

C++

```
BOOL GetProcessAffinityMask(
    [in] HANDLE     hProcess,
    [out] PDWORD_PTR lpProcessAffinityMask,
    [out] PDWORD_PTR lpSystemAffinityMask
);
```

Parameters

[in] hProcess

A handle to the process whose affinity mask is desired.

This handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

Windows Server 2003 and Windows XP: The handle must have the **PROCESS_QUERY_INFORMATION** access right.

[out] lpProcessAffinityMask

A pointer to a variable that receives the affinity mask for the specified process.

[out] lpSystemAffinityMask

A pointer to a variable that receives the affinity mask for the system.

Return value

If the function succeeds, the return value is nonzero and the function sets the variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the appropriate affinity masks.

On a system with more than 64 processors, if the threads of the calling process are in a single [processor group](#), the function sets the variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the process affinity mask and the processor mask of active logical processors for that group. If the calling process contains threads in multiple groups, the function returns zero for both affinity masks.

If the function fails, the return value is zero, and the values of the variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* are undefined. To get extended error information, call [GetLastError](#).

Remarks

A process affinity mask is a bit vector in which each bit represents the processors that a process is allowed to run on. A system affinity mask is a bit vector in which each bit represents the processors that are configured into a system.

A process affinity mask is a subset of the system affinity mask. A process is only allowed to run on the processors configured into a system. Therefore, the process affinity mask cannot specify a 1 bit for a processor when the system affinity mask specifies a 0 bit for that processor.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all [processor groups](#), by default. The [GetProcessAffinityMask](#) function sets the *lpProcessAffinityMask* and *lpSystemAffinityMask* to the process and system processor masks over the process' primary group. If the process had explicitly set the affinity of one or more of its threads outside of the process' primary group, the function returns zero for both affinity masks. If, however, *hHandle* specifies a handle to the current process, the function always uses the calling thread's primary group (which by default is the same as the process' primary group) in order to set the *lpProcessAffinityMask* and *lpSystemAffinityMask*.

Requirements

Minimum supported client
Windows XP [desktop apps UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Multiple Processors](#)

[Process and Thread Functions](#)

[Processes](#)

[Processor Groups](#)

[SetProcessAffinityMask](#)

[SetThreadAffinityMask](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetProcessDEPPolicy function (winbase.h)

Article10/13/2021

Gets the data execution prevention (DEP) and DEP-ATL thunk emulation settings for the specified 32-bit process. **Windows XP with SP3:** Gets the DEP and DEP-ATL thunk emulation settings for the current process.

Syntax

C++

```
BOOL GetProcessDEPPolicy(
    [in]  HANDLE hProcess,
    [out] LPDWORD lpFlags,
    [out] PBOOL   lpPermanent
);
```

Parameters

[in] hProcess

A handle to the process. **PROCESS_QUERY_INFORMATION** privilege is required to get the DEP policy of a process.

Windows XP with SP3: The *hProcess* parameter is ignored.

[out] lpFlags

A DWORD that receives one or more of the following flags.

Value	Meaning
0	DEP is disabled for the specified process.
PROCESS_DEP_ENABLE 0x00000001	DEP is enabled for the specified process.
PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION 0x00000002	DEP-ATL thunk emulation is disabled for the specified process. For information about DEP-ATL thunk emulation, see SetProcessDEPPolicy .

[out] *lpPermanent*

TRUE if DEP is enabled or disabled permanently for the specified process; otherwise **FALSE**. If *lpPermanent* is **TRUE**, the current DEP setting persists for the life of the process and cannot be changed by calling [SetProcessDEPPolicy](#).

Return value

If the function succeeds, it returns **TRUE**.

If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call [GetLastError](#).

Remarks

[GetProcessDEPPolicy](#) is supported for 32-bit processes only. If this function is called on a 64-bit process, it fails with **ERROR_NOT_SUPPORTED**.

To compile an application that calls this function, define **_WIN32_WINNT** as 0x0600 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows Vista with SP1, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Data Execution Prevention](#)

[GetSystemDEPPolicy](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetProcessIoCounters function (winbase.h)

Article10/13/2021

Retrieves accounting information for all I/O operations performed by the specified process.

Syntax

C++

```
BOOL GetProcessIoCounters(
    [in] HANDLE     hProcess,
    [out] PIO_COUNTERS lpIoCounters
);
```

Parameters

[in] hProcess

A handle to the process. The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

Windows Server 2003 and Windows XP: The handle must have the **PROCESS_QUERY_INFORMATION** access right.

[out] lpIoCounters

A pointer to an [IO_COUNTERS](#) structure that receives the I/O accounting information for the process.

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](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[IO_COUNTERS](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetProfileIntA function (winbase.h)

Article02/09/2023

Retrieves an integer from a key in the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
UINT GetProfileIntA(
    [in] LPCSTR lpAppName,
    [in] LPCSTR lpKeyName,
    [in] INT     nDefault
);
```

Parameters

[in] lpAppName

The name of the section containing the key name.

[in] lpKeyName

The name of the key whose value is to be retrieved. This value is in the form of a string; the **GetProfileInt** function converts the string into an integer and returns the integer.

[in] nDefault

The default value to return if the key name cannot be found in the initialization file.

Return value

The return value is the integer equivalent of the string following the key name in Win.ini. If the function cannot find the key, the return value is the default value. If the value of the key is less than zero, the return value is zero.

Remarks

If the key name consists of digits followed by characters that are not numeric, the function returns only the value of the digits. For example, the function returns 102 for the following line: KeyName=102abc.

Windows Server 2003 and Windows XP/2000: Calls to profile functions may be mapped to the registry instead of to the initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following key:
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping

When the operation has been mapped, the **GetProfileInt** function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.

- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

ⓘ Note

The winbase.h header defines GetProfileInt as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileInt](#)

[WriteProfileString](#)

Feedback

Was this page helpful?  

Get help at Microsoft Q&A

GetProfileIntW function (winbase.h)

Article02/09/2023

Retrieves an integer from a key in the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
UINT GetProfileIntW(
    [in] LPCWSTR lpAppName,
    [in] LPCWSTR lpKeyName,
    [in] INT      nDefault
);
```

Parameters

[in] lpAppName

The name of the section containing the key name.

[in] lpKeyName

The name of the key whose value is to be retrieved. This value is in the form of a string; the **GetProfileInt** function converts the string into an integer and returns the integer.

[in] nDefault

The default value to return if the key name cannot be found in the initialization file.

Return value

The return value is the integer equivalent of the string following the key name in Win.ini. If the function cannot find the key, the return value is the default value. If the value of the key is less than zero, the return value is zero.

Remarks

If the key name consists of digits followed by characters that are not numeric, the function returns only the value of the digits. For example, the function returns 102 for the following line: KeyName=102abc.

Windows Server 2003 and Windows XP/2000: Calls to profile functions may be mapped to the registry instead of to the initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following key:
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping

When the operation has been mapped, the **GetProfileInt** function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.

- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines GetProfileInt as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileInt](#)

[WriteProfileString](#)

Feedback

Was this page helpful?  

Get help at Microsoft Q&A

GetProfileSectionA function (winbase.h)

Article 02/09/2023

Retrieves all the keys and values for the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetProfileSectionA(
    [in]  LPCSTR lpAppName,
    [out] LPSTR  lpReturnedString,
    [in]  DWORD   nSize
);
```

Parameters

[in] `lpAppName`

The name of the section in the Win.ini file.

[out] `lpReturnedString`

A pointer to a buffer that receives the keys and values associated with the named section. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

[in] `nSize`

The size of the buffer pointed to by the `lpReturnedString` parameter, in characters.

The maximum profile section size is 32,767 characters.

Return value

The return value specifies the number of characters copied to the specified buffer, not including the terminating null character. If the buffer is not large enough to contain all the keys and values associated with the named section, the return value is equal to the size specified by *nSize* minus two.

Remarks

The format of the returned keys and values is one or more null-terminated strings, followed by a final null character. Each string has the following form: *key=string*

The **GetProfileSection** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

This operation is atomic; no updates to the Win.ini file are allowed while the keys and values for the section are being copied to the buffer.

Windows Server 2003 and Windows XP/2000: Calls to profile functions may be mapped to the registry instead of to the initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following key:
`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping`.

When the operation has been mapped, the **GetProfileSection** function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that

specifies the default location in the registry where you will find the keys for the section.

6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines GetProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

[WriteProfileSection](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetProfileSectionW function (winbase.h)

Article02/09/2023

Retrieves all the keys and values for the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetProfileSectionW(
    [in]  LPCWSTR lpAppName,
    [out] LPWSTR  lpReturnedString,
    [in]  DWORD   nSize
);
```

Parameters

[in] `lpAppName`

The name of the section in the Win.ini file.

[out] `lpReturnedString`

A pointer to a buffer that receives the keys and values associated with the named section. The buffer is filled with one or more null-terminated strings; the last string is followed by a second null character.

[in] `nSize`

The size of the buffer pointed to by the `lpReturnedString` parameter, in characters.

The maximum profile section size is 32,767 characters.

Return value

The return value specifies the number of characters copied to the specified buffer, not including the terminating null character. If the buffer is not large enough to contain all the keys and values associated with the named section, the return value is equal to the size specified by *nSize* minus two.

Remarks

The format of the returned keys and values is one or more null-terminated strings, followed by a final null character. Each string has the following form: *key=string*

The **GetProfileSection** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

This operation is atomic; no updates to the Win.ini file are allowed while the keys and values for the section are being copied to the buffer.

Windows Server 2003 and Windows XP/2000: Calls to profile functions may be mapped to the registry instead of to the initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following key:
`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping`.

When the operation has been mapped, the **GetProfileSection** function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that

specifies the default location in the registry where you will find the keys for the section.

6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines GetProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

[WriteProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetProfileStringA function (winbase.h)

Article02/09/2023

Retrieves the string associated with a key in the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications, therefore this function should not be called from server code. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetProfileStringA(
    [in]  LPCSTR lpAppName,
    [in]  LPCSTR lpKeyName,
    [in]  LPCSTR lpDefault,
    [out] LPSTR  lpReturnedString,
    [in]  DWORD   nSize
);
```

Parameters

[in] *lpAppName*

The name of the section containing the key. If this parameter is **NULL**, the function copies all section names in the file to the supplied buffer.

[in] *lpKeyName*

The name of the key whose associated string is to be retrieved. If this parameter is **NULL**, the function copies all keys in the given section to the supplied buffer. Each string is followed by a **null** character, and the final string is followed by a second **null** character.

[in] *lpDefault*

A default string. If the *lpKeyName* key cannot be found in the initialization file, **GetProfileString** copies the default string to the *lpReturnedString* buffer. If this parameter is **NULL**, the default is an empty string, "".

Avoid specifying a default string with trailing blank characters. The function inserts a **null** character in the *lpReturnedString* buffer to strip any trailing blanks.

[out] *lpReturnedString*

A pointer to a buffer that receives the character string.

[in] *nSize*

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

Return value

The return value is the number of characters copied to the buffer, not including the **null**-terminating character.

If neither *lpAppName* nor *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold the requested string, the string is truncated and followed by a **null** character, and the return value is equal to *nSize* minus one.

If either *lpAppName* or *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters. In this case, the return value is equal to *nSize* minus two.

Remarks

If the string associated with the *lpKeyName* parameter is enclosed in single or double quotation marks, the marks are discarded when the **GetProfileString** function returns the string.

The **GetProfileString** function is not case-sensitive; the strings can contain a combination of uppercase and lowercase letters.

A section in the Win.ini file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

An application can use the [GetPrivateProfileString](#) function to retrieve a string from a specified initialization file.

The *lpDefault* parameter must point to a valid string, even if the string is empty (that is, even if its first character is a **null** character).

Windows Server 2003 and Windows XP/2000: Calls to profile functions may be mapped to the registry instead of to the initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following keys:

**HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping**

When the operation has been mapped, the [GetProfileString](#) function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.

- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines GetProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileString](#)

[WriteProfileString](#)

Feedback



Was this page helpful?  

Get help at Microsoft Q&A

GetProfileStringW function (winbase.h)

Article 02/09/2023

Retrieves the string associated with a key in the specified section of the Win.ini file.

Note This function is provided only for compatibility with 16-bit Windows-based applications, therefore this function should not be called from server code. Applications should store initialization information in the registry.

Syntax

C++

```
DWORD GetProfileStringW(
    [in]  LPCWSTR lpAppName,
    [in]  LPCWSTR lpKeyName,
    [in]  LPCWSTR lpDefault,
    [out] LPWSTR  lpReturnedString,
    [in]  DWORD   nSize
);
```

Parameters

[in] *lpAppName*

The name of the section containing the key. If this parameter is **NULL**, the function copies all section names in the file to the supplied buffer.

[in] *lpKeyName*

The name of the key whose associated string is to be retrieved. If this parameter is **NULL**, the function copies all keys in the given section to the supplied buffer. Each string is followed by a **null** character, and the final string is followed by a second **null** character.

[in] *lpDefault*

A default string. If the *lpKeyName* key cannot be found in the initialization file, **GetProfileString** copies the default string to the *lpReturnedString* buffer. If this parameter is **NULL**, the default is an empty string, "".

Avoid specifying a default string with trailing blank characters. The function inserts a **null** character in the *lpReturnedString* buffer to strip any trailing blanks.

[out] *lpReturnedString*

A pointer to a buffer that receives the character string.

[in] *nSize*

The size of the buffer pointed to by the *lpReturnedString* parameter, in characters.

Return value

The return value is the number of characters copied to the buffer, not including the **null**-terminating character.

If neither *lpAppName* nor *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold the requested string, the string is truncated and followed by a **null** character, and the return value is equal to *nSize* minus one.

If either *lpAppName* or *lpKeyName* is **NULL** and the supplied destination buffer is too small to hold all the strings, the last string is truncated and followed by two **null** characters. In this case, the return value is equal to *nSize* minus two.

Remarks

If the string associated with the *lpKeyName* parameter is enclosed in single or double quotation marks, the marks are discarded when the **GetProfileString** function returns the string.

The **GetProfileString** function is not case-sensitive; the strings can contain a combination of uppercase and lowercase letters.

A section in the Win.ini file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

An application can use the [GetPrivateProfileString](#) function to retrieve a string from a specified initialization file.

The *lpDefault* parameter must point to a valid string, even if the string is empty (that is, even if its first character is a **null** character).

Windows Server 2003 and Windows XP/2000: Calls to profile functions may be mapped to the registry instead of to the initialization files. This mapping occurs when the initialization file and section are specified in the registry under the following keys:

**HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping**

When the operation has been mapped, the [GetProfileString](#) function retrieves information from the registry, not from the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as **<No Name>**) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.

- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines GetProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileString](#)

[WriteProfileString](#)

Feedback



Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetShortPathNameA function (winbase.h)

Article06/01/2023

Retrieves the short path form of the specified path.

For more information about file and path names, see [Naming Files, Paths, and Namespaces](#).

Syntax

C++

```
DWORD GetShortPathNameA(
    [in]  LPCSTR lpszLongPath,
    [out] LPSTR  lpszShortPath,
    [in]  DWORD   cchBuffer
);
```

Parameters

`[in] lpszLongPath`

The path string.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

`[out] lpszShortPath`

A pointer to a buffer to receive the null-terminated short form of the path that *lpszLongPath* specifies.

Passing **NULL** for this parameter and zero for *cchBuffer* will always return the required buffer size for a specified *lpszLongPath*.

[in] *cchBuffer*

The size of the buffer that *lpszShortPath* points to, in **TCHARs**.

Set this parameter to zero if *lpszShortPath* is set to **NULL**.

Return value

If the function succeeds, the return value is the length, in **TCHARs**, of the string that is copied to *lpszShortPath*, not including the terminating null character.

If the *lpszShortPath* buffer is too small to contain the path, the return value is the size of the buffer, in **TCHARs**, that is required to hold the path and the terminating null character.

If the function fails for any other reason, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The path that the *lpszLongPath* parameter specifies does not have to be a full or long path. The short form can be longer than the specified path.

If the return value is greater than the value specified in the *cchBuffer* parameter, you can call the function again with a buffer that is large enough to hold the path. For an example of this case in addition to using zero-length buffer for dynamic allocation, see the Example Code section.

Note Although the return value in this case is a length that includes the terminating null character, the return value on success does not include the terminating null character in the count.

If the specified path is already in its short form and conversion is not needed, the function simply copies the specified path to the buffer specified by the *lpszShortPath* parameter.

You can set the *lpszShortPath* parameter to the same value as the *lpszLongPath* parameter; in other words, you can set the output buffer for the short path to the

address of the input path string. Always ensure that the *cchBuffer* parameter accurately represents the total size, in TCHARs, of this buffer.

You can obtain the long name of a file from the short name by calling the [GetLongPathName](#) function. Alternatively, where **GetLongPathName** is not available, you can call [FindFirstFile](#) on each component of the path to get the corresponding long name.

It is possible to have access to a file or directory but not have access to some of the parent directories of that file or directory. As a result, **GetShortPathName** may fail when it is unable to query the parent directory of a path component to determine the short name for that component. This check can be skipped for directory components that already meet the requirements of a short name. For more information, see the [Short vs. Long Names](#) section of [Naming Files, Paths, and Namespaces](#).

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)	No
Resilient File System (ReFS)	Yes

SMB 3.0 does not support short names on shares with continuous availability capability.

Resilient File System (ReFS) doesn't support short names. If you call **GetShortPathName** on a path that doesn't have any short names on-disk, the call will succeed, but will return the long-name path instead. This outcome is also possible with NTFS volumes because there's no guarantee that a short name will exist for a given long name.

Examples

For an example that uses **GetShortPathName**, see the Example Code section for [GetFullPathName](#).

The following C++ example shows how to use a dynamically allocated output buffer.

C++

```

//...
long      length = 0;
TCHAR*   buffer = NULL;

// First obtain the size needed by passing NULL and 0.

length = GetShortPathName(lpszPath, NULL, 0);
if (length == 0) ErrorExit(TEXT("GetShortPathName"));

// Dynamically allocate the correct size
// (terminating null char was included in length)

buffer = new TCHAR[length];

// Now simply call again using same long path.

length = GetShortPathName(lpszPath, buffer, length);
if (length == 0) ErrorExit(TEXT("GetShortPathName"));

_tprintf(TEXT("long name = %s shortname = %s"), lpszPath, buffer);

delete [] buffer;
///...

```

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[FindFirstFile](#)

[GetFullPathName](#)

[GetLongPathName](#)

[Naming Files, Paths, and Namespaces](#)

[SetFileShortName](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetSystemDEPPolicy function (winbase.h)

Article 06/29/2021

Gets the data execution prevention (DEP) policy setting for the system.

Syntax

C++

```
DEP_SYSTEM_POLICY_TYPE GetSystemDEPPolicy();
```

Return value

This function returns a value of type **DEP_SYSTEM_POLICY_TYPE**, which can be one of the following values.

Return code/value	Description
AlwaysOff 0	DEP is disabled for all parts of the system, regardless of hardware support for DEP. The processor runs in PAE mode with 32-bit versions of Windows unless PAE is disabled in the boot configuration data.
AlwaysOn 1	DEP is enabled for all parts of the system. All processes always run with DEP enabled. DEP cannot be explicitly disabled for selected applications. System compatibility fixes are ignored.
OptIn 2	On systems with processors that are capable of hardware-enforced DEP, DEP is automatically enabled only for operating system components. This is the default setting for client versions of Windows. DEP can be explicitly enabled for selected applications or the current process.
OptOut 3	DEP is automatically enabled for operating system components and all processes. This is the default setting for Windows Server versions. DEP can be explicitly disabled for selected applications or the current process. System compatibility fixes for DEP are in effect.

Remarks

The system-wide DEP policy is configured at boot time according to the policy setting in the boot configuration data. To change the system-wide DEP policy setting, use the [BCDEdit /set](#) command to set the **nx** boot entry option.

If the system DEP policy is OptIn or OptOut, DEP can be selectively enabled or disabled for the current process by calling the [SetProcessDEPPolicy](#) function. This function works only for 32-bit processes.

A user with administrative privileges can disable DEP for selected applications by using the **System** Control Panel application. If the system DEP policy is OptOut, DEP is disabled for these applications.

The Application Compatibility Toolkit can be used to create a list of individual applications that are exempt from DEP. If the system DEP policy is OptOut, DEP is automatically disabled for applications on the list.

Requirements

Minimum supported client	Windows Vista with SP1, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Data Execution Prevention](#)

[GetProcessDEPPolicy](#)

[GetSystemDEPPolicy](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetSystemPowerStatus function (winbase.h)

Article10/13/2021

Retrieves the power status of the system. The status indicates whether the system is running on AC or DC power, whether the battery is currently charging, how much battery life remains, and if battery saver is on or off.

Syntax

C++

```
BOOL GetSystemPowerStatus(  
    [out] LPSYSTEM_POWER_STATUS lpSystemPowerStatus  
);
```

Parameters

[out] `lpSystemPowerStatus`

A pointer to a `SYSTEM_POWER_STATUS` structure that receives status information.

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

To compile an application that uses this function, define the `_WIN32_WINNT` macro as `0x0400` or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Power Management Functions](#)

[SYSTEM_POWER_STATUS](#)

[System Power Status](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetSystemRegistryQuota function (winbase.h)

Article 10/13/2021

Retrieves the current size of the registry and the maximum size that the registry is allowed to attain on the system.

Syntax

C++

```
BOOL GetSystemRegistryQuota(
    [out, optional] PDWORD pdwQuotaAllowed,
    [out, optional] PDWORD pdwQuotaUsed
);
```

Parameters

[out, optional] pdwQuotaAllowed

A pointer to a variable that receives the maximum size that the registry is allowed to attain on this system, in bytes.

[out, optional] pdwQuotaUsed

A pointer to a variable that receives the current size of the registry, in bytes.

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

To compile an application that uses this function, define _WIN32_WINNT as 0x0501 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetTapeParameters function (winbase.h)

Article10/13/2021

The **GetTapeParameters** function retrieves information that describes the tape or the tape drive.

Syntax

C++

```
DWORD GetTapeParameters(
    [in] HANDLE hDevice,
    [in] DWORD dwOperation,
    [out] LPDWORD lpdwSize,
    [out] LPVOID lpTapeInformation
);
```

Parameters

[in] *hDevice*

Handle to the device about which information is sought. This handle is created by using the [CreateFile](#) function.

[in] *dwOperation*

Type of information requested. This parameter must be one of the following values.

Value	Meaning
GET_TAPE_DRIVE_INFORMATION 1	Retrieves information about the tape device.
GET_TAPE_MEDIA_INFORMATION 0	Retrieves information about the tape in the tape device.

[out] *lpdwSize*

Pointer to a variable that receives the size, in bytes, of the buffer specified by the *lpTapeInformation* parameter. If the buffer is too small, this parameter receives the required size.

[out] *lpTapeInformation*

Pointer to a structure that contains the requested information. If the *dwOperation* parameter is **GET_TAPE_MEDIA_INFORMATION**, *lpTapeInformation* points to a **TAPE_GET_MEDIA_PARAMETERS** structure.

If *dwOperation* is **GET_TAPE_DRIVE_INFORMATION**, *lpTapeInformation* points to a **TAPE_GET_DRIVE_PARAMETERS** structure.

Return value

If the function succeeds, the return value is **NO_ERROR**.

If the function fails, it can return one of the following error codes.

Error code	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED	A setmark was reached during an operation.

1103L	
1108L	ERROR_UNABLE_TO_LOCK_MEDIA An attempt to lock the ejection mechanism failed.
1109L	ERROR_UNABLE_TO_UNLOAD_MEDIA An attempt to unload the tape failed.
19L	ERROR_WRITE_PROTECT The media is write protected.

Remarks

The block size range values (maximum and minimum) returned by the **GetTapeParameters** function called with the *dwOperation* parameter set to the **GET_TAPE_DRIVE_INFORMATION** value will indicate system limits, not drive limits. However, it is the tape drive device and the media present in the drive that determine the true block size limits. Thus, an application may not be able to set all the block sizes mentioned in the range obtained by specifying **GET_TAPE_DRIVE_INFORMATION** in *dwOperation*.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[SetTapeParameters](#)

[TAPE_GET_DRIVE_PARAMETERS](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetTapePosition function (winbase.h)

Article 10/13/2021

The **GetTapePosition** function retrieves the current address of the tape, in logical or absolute blocks.

Syntax

C++

```
DWORD GetTapePosition(
    [in]  HANDLE  hDevice,
    [in]  DWORD   dwPositionType,
    [out] LPDWORD lpdwPartition,
    [out] LPDWORD lpdwOffsetLow,
    [out] LPDWORD lpdwOffsetHigh
);
```

Parameters

[in] `hDevice`

Handle to the device on which to get the tape position. This handle is created by using [CreateFile](#).

[in] `dwPositionType`

Type of address to obtain. This parameter can be one of the following values.

Value	Meaning
<code>TAPE_ABSOLUTE_POSITION</code> 0L	The <code>lpdwOffsetLow</code> and <code>lpdwOffsetHigh</code> parameters receive the device-specific block address. The <code>dwPartition</code> parameter receives zero.
<code>TAPE_LOGICAL_POSITION</code> 1L	The <code>lpdwOffsetLow</code> and <code>lpdwOffsetHigh</code> parameters receive the logical block address. The <code>dwPartition</code> parameter receives the logical tape partition.

[out] `lpdwPartition`

Pointer to a variable that receives the number of the current tape partition. Partitions are numbered logically from 1 through n, where 1 is the first partition on the tape and n is

the last. When a device-specific block address is retrieved, or if the device supports only one partition, this parameter receives zero.

[out] `lpdwOffsetLow`

Pointer to a variable that receives the low-order bits of the current tape position.

[out] `lpdwOffsetHigh`

Pointer to a variable that receives the high-order bits of the current tape position. This parameter can be **NULL** if the high-order bits are not required.

Return value

If the function succeeds, the return value is `NO_ERROR`.

If the function fails, it can return one of the following error codes.

Error code	Description
<code>ERROR_BEGINNING_OF_MEDIA</code> 1102L	An attempt to access data before the beginning-of-medium marker failed.
<code>ERROR_BUS_RESET</code> 1111L	A reset condition was detected on the bus.
<code>ERROR_DEVICE_NOT_PARTITIONED</code> 1107L	The partition information could not be found when a tape was being loaded.
<code>ERROR_END_OF_MEDIA</code> 1100L	The end-of-tape marker was reached during an operation.
<code>ERROR_FILEMARK_DETECTED</code> 1101L	A filemark was reached during an operation.
<code>ERROR_INVALID_BLOCK_LENGTH</code> 1106L	The block size is incorrect on a new tape in a multivolume partition.
<code>ERROR_MEDIA_CHANGED</code> 1110L	The tape that was in the drive has been replaced or removed.
<code>ERROR_NO_DATA_DETECTED</code> 1104L	The end-of-data marker was reached during an operation.
<code>ERROR_NO_MEDIA_IN_DRIVE</code> 1112L	There is no media in the drive.
<code>ERROR_NOT_SUPPORTED</code> 50L	The tape driver does not support a requested function.

ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.
ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT 19L	The media is write protected.

Remarks

A logical block address is relative to a partition. The first logical block address on each partition is zero.

Call the [GetTapeParameters](#) function to obtain information about the status, capabilities, and capacities of tape drives and media.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[GetTapeParameters](#)

[SetTapePosition](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetTapeStatus function (winbase.h)

Article10/13/2021

The **GetTapeStatus** function determines whether the tape device is ready to process tape commands.

Syntax

C++

```
DWORD GetTapeStatus(  
    [in] HANDLE hDevice  
);
```

Parameters

[in] hDevice

Handle to the device for which to get the device status. This handle is created by using the [CreateFile](#) function.

Return value

If the tape device is ready to accept appropriate tape-access commands without returning errors, the return value is NO_ERROR.

If the function fails, it can return one of the following error codes.

Error code	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_DEVICE_REQUIRES_CLEANING 1165L	The tape drive is capable of reporting that it requires cleaning, and reports that it does require cleaning.
ERROR_END_OF_MEDIA	The end-of-tape marker was reached during an

1100L	operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.
ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT 19L	The media is write protected.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetTempFileName function (winbase.h)

Article09/22/2022

Creates a name for a temporary file. If a unique file name is generated, an empty file is created and the handle to it is released; otherwise, only a file name is generated.

Syntax

C++

```
UINT GetTempFileName(
    [in] LPCTSTR lpPathName,
    [in] LPCTSTR lpPrefixString,
    [in] UINT     uUnique,
    [out] LPTSTR  lpTempFileName
);
```

Parameters

[in] `lpPathName`

The directory path for the file name. Applications typically specify a period (.) for the current directory or the result of the [GetTempPath](#) function. The string cannot be longer than **MAX_PATH**–14 characters or **GetTempFileName** will fail. If this parameter is **NULL**, the function fails.

[in] `lpPrefixString`

The null-terminated prefix string. The function uses up to the first three characters of this string as the prefix of the file name. This string must consist of characters in the OEM-defined character set.

[in] `uUnique`

An unsigned integer to be used in creating the temporary file name. For more information, see Remarks.

If *uUnique* is zero, the function attempts to form a unique file name using the current system time. If the file already exists, the number is increased by one and the functions tests if this file already exists. This continues until a unique filename is found; the function creates a file by that name and closes it. Note that the function does not attempt to verify the uniqueness of the file name when *uUnique* is nonzero.

[out] *lpTempFileName*

A pointer to the buffer that receives the temporary file name. This buffer should be **MAX_PATH** characters to accommodate the path plus the terminating null character.

Return value

If the function succeeds, the return value specifies the unique numeric value used in the temporary file name. If the *uUnique* parameter is nonzero, the return value specifies that same number.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

The following is a possible return value.

Return value	Description
ERROR_BUFFER_OVERFLOW	The length of the string pointed to by the <i>lpPathName</i> parameter is more than MAX_PATH -14 characters.

Remarks

The **GetTempFileName** function creates a temporary file name of the following form:

<*path*>\<*pre*><*uuuu*>.TMP

The following table describes the file name syntax.

Component	Meaning
< <i>path</i> >	Path specified by the <i>lpPathName</i> parameter
< <i>pre</i> >	First three letters of the <i>lpPrefixString</i> string
< <i>uuuu</i> >	Hexadecimal value of <i>uUnique</i>

If *uUnique* is zero, **GetTempFileName** creates an empty file and closes it. If *uUnique* is not zero, you must create the file yourself. Only a file name is created, because **GetTempFileName** is not able to guarantee that the file name is unique.

Only the lower 16 bits of the *uUnique* parameter are used. This limits **GetTempFileName** to a maximum of 65,535 unique file names if the *lpPathName* and *lpPrefixString*

parameters remain the same.

Due to the algorithm used to generate file names, **GetTempFileName** can perform poorly when creating a large number of files with the same prefix. In such cases, it is recommended that you construct unique file names based on **GUIDs**.

Temporary files whose names have been created by this function are not automatically deleted. To delete these files call [DeleteFile](#).

To avoid problems resulting when converting an ANSI string, an application should call the [CreateFile](#) function to create a temporary file.

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 an example, see [Creating and Using a Temporary 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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[DeleteFile](#)

[File Management Functions](#)

[GetTempPath](#)

[Naming Files, Paths, and Namespaces](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetThreadEnabledXStateFeatures function (winbase.h)

Article11/04/2022

This function returns the set of XState features that are currently enabled for the current thread.

Syntax

C++

```
DWORD64 GetThreadEnabledXStateFeatures();
```

Return value

The return value is a bitmask in which each bit represents an XState feature that is currently enabled for the current thread.

Remarks

This function is related to [GetEnabledXStateFeatures](#), which returns the set of XState features enabled in the system. Prior to the introduction of optional XState features, the set of enabled XState features is the same for every thread in the system because all supported features are always enabled, thus the result returned from GetEnabledXStateFeatures and this function are identical. With optional XState features, it is possible for optional XState features to be disabled by default for newly created threads and enabled on demand later. Optional XState features that are currently disabled for the current thread will not be returned by this function, but will still be returned by GetEnabledXStateFeatures.

Requirements

Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022

See also

[GetEnabledXStateFeatures](#)

[EnableProcessOptionalXStateFeatures](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetThreadSelectorEntry function (winbase.h)

Article 10/13/2021

Retrieves a descriptor table entry for the specified selector and thread.

Syntax

C++

```
BOOL GetThreadSelectorEntry(
    [in] HANDLE     hThread,
    [in] DWORD      dwSelector,
    [out] LPLDT_ENTRY lpSelectorEntry
);
```

Parameters

[in] hThread

A handle to the thread containing the specified selector. The handle must have THREAD_QUERY_INFORMATION access. For more information, see [Thread Security and Access Rights](#).

[in] dwSelector

The global or local selector value to look up in the thread's descriptor tables.

[out] lpSelectorEntry

A pointer to an [LDT_ENTRY](#) structure that receives a copy of the descriptor table entry if the specified selector has an entry in the specified thread's descriptor table. This information can be used to convert a segment-relative address to a linear virtual address.

Return value

If the function succeeds, the return value is nonzero. In that case, the structure pointed to by the *lpSelectorEntry* parameter receives a copy of the specified descriptor table entry.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

`GetThreadSelectorEntry` is only functional on x86-based systems. For systems that are not x86-based, the function returns **FALSE**.

Debuggers use this function to convert segment-relative addresses to linear virtual addresses. The [ReadProcessMemory](#) and [WriteProcessMemory](#) functions use linear virtual addresses.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Debugging Functions](#)

[LDT_ENTRY](#)

[ReadProcessMemory](#)

[WriteProcessMemory](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetUmsCompletionListEvent function (winbase.h)

Article03/18/2022

Retrieves a handle to the event associated with the specified user-mode scheduling (UMS) completion list.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL GetUmsCompletionListEvent(
    [in]      PUMS_COMPLETION_LIST UmsCompletionList,
    [in, out] PHANDLE           UmsCompletionEvent
);
```

Parameters

`[in] UmsCompletionList`

A pointer to a UMS completion list. The [CreateUmsCompletionList](#) function provides this pointer.

`[in, out] UmsCompletionEvent`

A pointer to a HANDLE variable. On output, the *UmsCompletionEvent* parameter is set to a handle to the event associated with the specified completion list.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The system signals a UMS completion list event when the system queues items to an empty completion list. A completion list event handle can be used with any [wait function](#) that takes a handle to an event. When the event is signaled, an application typically calls [DequeueUmsCompletionListItems](#) to retrieve the contents of the completion list.

The event handle remains valid until its completion list is deleted. Do not use the event handle to wait on a completion list that has been deleted or is in the process of being deleted.

When the handle is no longer needed, use the [CloseHandle](#) function to close the handle.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[CreateUmsCompletionList](#)

[DequeueUmsCompletionListItems](#)

[Wait Functions](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetUmsSystemThreadInformation function (winbase.h)

Article03/18/2022

Queries whether the specified thread is a UMS scheduler thread, a UMS worker thread, or a non-UMS thread.

⚠️ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL GetUmsSystemThreadInformation(
    [in]      HANDLE           ThreadHandle,
    [in, out] PUMS_SYSTEM_THREAD_INFORMATION SystemThreadInfo
);
```

Parameters

[in] `ThreadHandle`

A handle to a thread. The thread handle must have the `THREAD_QUERY_INFORMATION` access right. For more information, see [Thread Security and Access Rights](#).

[in, out] `SystemThreadInfo`

A pointer to an initialized `UMS_SYSTEM_THREAD_INFORMATION` structure that specifies the kind of thread for the query.

Return value

Returns `TRUE` if the specified thread matches the kind of thread specified by the `SystemThreadInfo` parameter. Otherwise, the function returns `FALSE`.

Remarks

The **GetUmsSystemThreadInformation** function is intended for use in debuggers, troubleshooting tools, and profiling applications. For example, thread-isolated tracing or single-stepping through instructions might involve suspending all other threads in the process. However, if the thread to be traced is a UMS worker thread, suspending UMS scheduler threads might cause a deadlock because a UMS worker thread requires the intervention of a UMS scheduler thread in order to run. A debugger can call **GetUmsSystemThreadInformation** for each thread that it might suspend to determine the kind of thread, and then suspend it or not as needed for the code being debugged.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps only], Windows 7 (64-bit only) and Windows Server 2008 R2 with KB977165 installed
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetUserNameA function (winbase.h)

Article02/09/2023

Retrieves the name of the user associated with the current thread.

Use the [GetUserNameEx](#) function to retrieve the user name in a specified format.

Additional information is provided by the [IADsADSSystemInfo](#) interface.

Syntax

C++

```
BOOL GetUserNameA(
    [out]     LPSTR   lpBuffer,
    [in, out] LPDWORD pcbBuffer
);
```

Parameters

[out] *lpBuffer*

A pointer to the buffer to receive the user's logon name. If this buffer is not large enough to contain the entire user name, the function fails. A buffer size of (UNLEN + 1) characters will hold the maximum length user name including the terminating null character. UNLEN is defined in Lmcons.h.

[in, out] *pcbBuffer*

On input, this variable specifies the size of the *lpBuffer* buffer, in TCHARs. On output, the variable receives the number of TCHARs copied to the buffer, including the terminating null character.

If *lpBuffer* is too small, the function fails and [GetLastError](#) returns ERROR_INSUFFICIENT_BUFFER. This parameter receives the required buffer size, including the terminating null character.

Return value

If the function succeeds, the return value is a nonzero value, and the variable pointed to by *lpsnSize* contains the number of TCHARs copied to the buffer specified by *lpBuffer*, including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If the current thread is impersonating another client, the **GetUserName** function returns the user name of the client that the thread is impersonating.

If **GetUserName** is called from a process that is running under the "NETWORK SERVICE" account, the string returned in *lpBuffer* may be different depending on the version of Windows. On Windows XP, the "NETWORK SERVICE" string is returned. On Windows Vista, the "<HOSTNAME>\$" string is returned.

Examples

For an example, see [Getting System Information](#).

ⓘ Note

The winbase.h header defines GetUserName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[GetUserNameEx](#)

[LookupAccountName](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetUserNameW function (winbase.h)

Article02/09/2023

Retrieves the name of the user associated with the current thread.

Use the [GetUserNameEx](#) function to retrieve the user name in a specified format.

Additional information is provided by the [IADsADSSystemInfo](#) interface.

Syntax

C++

```
BOOL GetUserNameW(
    [out]     LPWSTR  lpBuffer,
    [in, out] LPDWORD pcbBuffer
);
```

Parameters

[out] *lpBuffer*

A pointer to the buffer to receive the user's logon name. If this buffer is not large enough to contain the entire user name, the function fails. A buffer size of (UNLEN + 1) characters will hold the maximum length user name including the terminating null character. UNLEN is defined in Lmcons.h.

[in, out] *pcbBuffer*

On input, this variable specifies the size of the *lpBuffer* buffer, in TCHARs. On output, the variable receives the number of TCHARs copied to the buffer, including the terminating null character.

If *lpBuffer* is too small, the function fails and [GetLastError](#) returns ERROR_INSUFFICIENT_BUFFER. This parameter receives the required buffer size, including the terminating null character.

Return value

If the function succeeds, the return value is a nonzero value, and the variable pointed to by *lpnSize* contains the number of TCHARs copied to the buffer specified by *lpBuffer*, including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If the current thread is impersonating another client, the **GetUserName** function returns the user name of the client that the thread is impersonating.

If **GetUserName** is called from a process that is running under the "NETWORK SERVICE" account, the string returned in *lpBuffer* may be different depending on the version of Windows. On Windows XP, the "NETWORK SERVICE" string is returned. On Windows Vista, the "<HOSTNAME>\$" string is returned.

Examples

For an example, see [Getting System Information](#).

ⓘ Note

The winbase.h header defines GetUserName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[GetUserNameEx](#)

[LookupAccountName](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetVolumeNameForVolumeMountPoint

A function (winbase.h)

Article 11/22/2022

Retrieves a volume **GUID** path for the volume that is associated with the specified volume mount point (drive letter, volume **GUID** path, or mounted folder).

Syntax

C++

```
BOOL GetVolumeNameForVolumeMountPointA(
    [in]  LPCSTR lpszVolumeMountPoint,
    [out] LPSTR  lpszVolumeName,
    [in]  DWORD   cchBufferLength
);
```

Parameters

[in] lpszVolumeMountPoint

A pointer to a string that contains the path of a mounted folder (for example, "Y:\MountX") or a drive letter (for example, "X:\"). The string must end with a trailing backslash ("").

[out] lpszVolumeName

A pointer to a string that receives the volume **GUID** path. This path is of the form "\? \Volume{GUID}" where *GUID* is a **GUID** that identifies the volume. If there is more than one volume **GUID** path for the volume, only the first one in the mount manager's cache is returned.

[in] cchBufferLength

The length of the output buffer, in TCHARs. A reasonable size for the buffer to accommodate the largest possible volume **GUID** path is 50 characters.

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

Use [GetVolumeNameForVolumeMountPoint](#) to obtain a volume **GUID** path for use with functions such as [SetVolumeMountPoint](#) and [FindFirstVolumeMountPoint](#) that require a volume **GUID** path as an input parameter. For more information about volume **GUID** paths, see [Naming A Volume](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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)	No

SMB does not support volume management functions.

Mount points aren't supported by ReFS volumes.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeleteVolumeMountPoint](#)

[GetVolumePathName](#)

[Mounted Folders](#)

[SetVolumeMountPoint](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetVolumePathNameA function (winbase.h)

Article 07/27/2022

Retrieves the volume mount point where the specified path is mounted.

Syntax

C++

```
BOOL GetVolumePathNameA(
    [in]  LPCSTR lpszFileName,
    [out] LPSTR  lpszVolumePathName,
    [in]  DWORD   cchBufferLength
);
```

Parameters

[in] lpszFileName

A pointer to the input path string. Both absolute and relative file and directory names, for example "..", are acceptable in this path.

If you specify a relative directory or file name without a volume qualifier, **GetVolumePathName** returns the drive letter of the boot volume.

If this parameter is an empty string, "", the function fails but the last error is set to **ERROR_SUCCESS**.

[out] lpszVolumePathName

A pointer to a string that receives the volume mount point for the input path.

[in] cchBufferLength

The length of the output buffer, in TCHARs.

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

If a specified path is passed, **GetVolumePathName** returns the path to the volume mount point, which means that it returns the root of the volume where the end point of the specified path is located.

For example, assume that you have volume D mounted at C:\Mnt\Ddrive and volume E mounted at "C:\Mnt\Ddrive\Mnt\Edrive". Also assume that you have a file with the path "E:\Dir\Subdir\MyFile". If you pass "C:\Mnt\Ddrive\Mnt\Edrive\Dir\Subdir\MyFile" to **GetVolumePathName**, it returns the path "C:\Mnt\Ddrive\Mnt\Edrive".

If either a relative directory or a file is passed without a volume qualifier, the function returns the drive letter of the boot volume. The drive letter of the boot volume is also returned if an invalid file or directory name is specified without a valid volume qualifier. If a valid volume specifier is given, and the volume exists, but an invalid file or directory name is specified, the function will succeed and that volume name will be returned. For examples, see the Examples section of this topic.

You must specify a valid Win32 namespace path. If you specify an NT namespace path, for example, "\DosDevices\H:" or "\Device\HardDiskVolume6", the function returns the drive letter of the boot volume, not the drive letter of that NT namespace path.

For more information about path names and namespaces, see [Naming Files, Paths, and Namespaces](#).

You can specify both local and remote paths. If you specify a local path, **GetVolumePathName** returns a full path whose prefix is the longest prefix that represents a volume.

If a network share is specified, **GetVolumePathName** returns the shortest path for which **GetDriveType** returns **DRIVE_REMOTE**, which means that the path is validated as a remote drive that exists, which the current user can access.

There are certain special cases that do not return a trailing backslash. These occur when the output buffer length is one character too short. For example, if *lpszFileName* is C: and *lpszVolumePathName* is 4 characters long, the value returned is "C:"; however, if *lpszVolumePathName* is 3 characters long, the value returned is "C.". A safer but slower way to set the size of the return buffer is to call the [GetFullPathName](#) function, and then make sure that the buffer size is at least the same size as the full path that

GetFullPathName returns. If the output buffer is more than one character too short, the function will fail and return an error.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

Trailing Path Elements

Trailing path elements that are invalid are ignored. For remote paths, the entire path (not just trailing elements) is considered invalid if one of the following conditions is true:

- The path is not formed correctly.
- The path does not exist.
- The current user does not have access to the path.

Junction Points and Mounted Folders

If the specified path traverses a junction point, **GetVolumePathName** returns the volume to which the junction point refers. For example, if `W:\Adir` is a junction point that points to `C:\Adir`, then **GetVolumePathName** invoked on `W:\Adir\Afile` returns "`c:\`". If the specified path traverses multiple junction points, the entire chain is followed, and **GetVolumePathName** returns the volume to which the last junction point in the chain refers.

If a remote path to a mounted folder or junction point is specified, the path is parsed as a remote path, and the mounted folder or junction point are ignored. For example if `C:\Dir_C` is linked to `D:\Dir_D` and `C:` is mapped to `X:` on a remote computer, calling **GetVolumePathName** and specifying `X:\Dir_C` on the remote computer returns `X:`
`</code>.`

Examples

For the following set of examples, U: is mapped to the remote computer \\YourComputer\C\$, and Q is a local drive.

Specified path	Function returns
\YourComputer\C\$\Windows	\YourComputer\C\$\
\?\UNC\YourComputer\C\$\Windows	\?\UNC\YourComputer\C\$\
Q:\Windows	Q:\
\?\Q:\Windows	\?\Q:\
\.\Q:\Windows	\.\Q:\
\?\UNC\W:\Windows	FALSE with error 123 because a specified remote path was not valid; W\$ share does not exist or no user access granted.
C:\COM2 (which exists)	\.\COM2\
C:\COM3 (non-existent)	FALSE with error 123 because a non-existent COM device was specified.

For the following set of examples, the paths contain invalid trailing path elements.

Specified path	Function returns
G:\invalid (invalid path)	G:\
\.\I:\aaa\invalid (invalid path)	\.\I:\
\YourComputer\C\$\invalid (invalid trailing path element)	\YourComputer\C\$\

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeleteVolumeMountPoint](#)

[GetFullPathName](#)

[GetVolumeNameForVolumeMountPoint](#)

[SetVolumeMountPoint](#)

[Volume Management Functions](#)

[Volume Mount Points](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GetVolumePathNamesForVolumeName

A function (winbase.h)

Article 07/27/2022

Retrieves a list of drive letters and mounted folder paths for the specified volume.

Syntax

C++

```
BOOL GetVolumePathNamesForVolumeNameA(
    [in]  LPCSTR lpszVolumeName,
    [out] LPCH   lpszVolumePathNames,
    [in]  DWORD  cchBufferLength,
    [out] PDWORD lpcchReturnLength
);
```

Parameters

[in] *lpszVolumeName*

A volume **GUID** path for the volume. A volume **GUID** path is of the form "\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}".

[out] *lpszVolumePathNames*

A pointer to a buffer that receives the list of drive letters and mounted folder paths. The list is an array of null-terminated strings terminated by an additional **NULL** character. If the buffer is not large enough to hold the complete list, the buffer holds as much of the list as possible.

[in] *cchBufferLength*

The length of the *lpszVolumePathNames* buffer, in **TCHARs**, including all **NULL** characters.

[out] *lpcchReturnLength*

If the call is successful, this parameter is the number of **TCHARs** copied to the *lpszVolumePathNames* buffer. Otherwise, this parameter is the size of the buffer required to hold the complete list, in **TCHARs**.

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 buffer is not large enough to hold the complete list, the error code is **ERROR_MORE_DATA** and the *lpcchReturnLength* parameter receives the required buffer size.

Remarks

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

Examples

For an example, see [Displaying Volume Paths](#).

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)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

GetXStateFeaturesMask function (winbase.h)

Article 08/23/2022

Returns the mask of XState features set within a [CONTEXT](#) structure.

Syntax

C++

```
BOOL GetXStateFeaturesMask(
    [in] PCONTEXT Context,
    [out] PDWORD64 FeatureMask
);
```

Parameters

[in] Context

A pointer to a [CONTEXT](#) structure that has been initialized with [InitializeContext](#).

[out] FeatureMask

A pointer to a variable that receives the mask of XState features which are present in the specified [CONTEXT](#) structure.

Return value

This function returns **TRUE** if successful, otherwise **FALSE**.

Remarks

The **GetXStateFeaturesMask** function returns the mask of valid features in the specified context. If a [CONTEXT](#) is to be passed to [GetThreadContext](#) or [Wow64GetThreadContext](#), the application must call [SetXStateFeaturesMask](#) to set which features are to be retrieved. **GetXStateFeaturesMask** should then be called on the [CONTEXT](#) returned by [GetThreadContext](#) or [Wow64GetThreadContext](#) to determine which feature areas contain valid data. If a particular feature bit is not set, the

corresponding state is in a processor-specific **INITIALIZED** state and the contents of the feature area retrieved by [LocateXStateFeature](#) are undefined.

The definition of XState features are processor vendor specific. Please refer to the relevant processor reference manuals for additional information on a particular feature.

Note The value returned by [GetXStateFeaturesMask](#) on a **CONTEXT** after a context operation will always be a subset of the mask specified in a call to [SetXStateFeaturesMask](#) prior to the context operation.

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The [AVX API](#) is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1 . Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this documentation and get pointers to them using [GetModuleHandle](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CONTEXT](#)

[GetThreadContext](#)

[Intel AVX](#)

[SetXStateFeaturesMask](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalAddAtomA function (winbase.h)

Article 02/09/2023

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string.

Syntax

C++

```
ATOM GlobalAddAtomA(  
    [in] LPCSTR lpString  
);
```

Parameters

[in] lpString

Type: LPCTSTR

The null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings that differ only in case are considered identical. The case of the first string of this name added to the table is preserved and returned by the [GlobalGetAtomName](#) function.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

Return value

Type: ATOM

If the function succeeds, the return value is the newly created atom.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If the string already exists in the global atom table, the atom for the existing string is returned and the atom's reference count is incremented.

The string associated with the atom is not deleted from memory until its reference count is zero. For more information, see the [GlobalDeleteAtom](#) function.

Global atoms are not deleted automatically when the application terminates. For every call to the **GlobalAddAtom** function, there must be a corresponding call to the [GlobalDeleteAtom](#) function.

If the *lpString* parameter has the form "#1234", **GlobalAddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000 or is greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the [MAKEINTATOM](#) macro, the low-order word must be in the range 0x0001 through 0xFFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **GlobalAddAtom** returns a string atom.

Note

The winbase.h header defines GlobalAddAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[MAKEINTATOM](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalAddAtomExA function (winbase.h)

Article02/09/2023

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string.

Syntax

C++

```
ATOM GlobalAddAtomExA(  
    [in, optional] LPCSTR lpString,  
    [in]           DWORD   Flags  
)
```

Parameters

[in, optional] lpString

The null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings that differ only in case are considered identical. The case of the first string of this name added to the table is preserved and returned by the [GlobalGetAtomName](#) function.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

[in] Flags

Return value

If the function succeeds, the return value is the newly created atom.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Note

The winbase.h header defines GlobalAddAtomEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GlobalAddAtom](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalAddAtomExW function (winbase.h)

Article02/09/2023

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string.

Syntax

C++

```
ATOM GlobalAddAtomExW(
    [in, optional] LPCWSTR lpString,
    [in]           DWORD   Flags
);
```

Parameters

[in, optional] lpString

The null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings that differ only in case are considered identical. The case of the first string of this name added to the table is preserved and returned by the [GlobalGetAtomName](#) function.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

[in] Flags

Return value

If the function succeeds, the return value is the newly created atom.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Note

The winbase.h header defines GlobalAddAtomEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GlobalAddAtom](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalAddAtomW function (winbase.h)

Article 02/09/2023

Adds a character string to the global atom table and returns a unique value (an atom) identifying the string.

Syntax

C++

```
ATOM GlobalAddAtomW(
    [in] LPCWSTR lpString
);
```

Parameters

[in] lpString

Type: LPCTSTR

The null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings that differ only in case are considered identical. The case of the first string of this name added to the table is preserved and returned by the [GlobalGetAtomName](#) function.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

Return value

Type: ATOM

If the function succeeds, the return value is the newly created atom.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If the string already exists in the global atom table, the atom for the existing string is returned and the atom's reference count is incremented.

The string associated with the atom is not deleted from memory until its reference count is zero. For more information, see the [GlobalDeleteAtom](#) function.

Global atoms are not deleted automatically when the application terminates. For every call to the **GlobalAddAtom** function, there must be a corresponding call to the [GlobalDeleteAtom](#) function.

If the *lpString* parameter has the form "#1234", **GlobalAddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000 or is greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the [MAKEINTATOM](#) macro, the low-order word must be in the range 0x0001 through 0xFFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **GlobalAddAtom** returns a string atom.

Note

The winbase.h header defines GlobalAddAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[MAKEINTATOM](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalAlloc function (winbase.h)

Article07/27/2022

Allocates the specified number of bytes from the heap.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
DECLSPEC_ALLOCATOR HGLOBAL GlobalAlloc(
    [in] UINT    uFlags,
    [in] SIZE_T dwBytes
);
```

Parameters

[in] **uFlags**

The memory allocation attributes. If zero is specified, the default is **GMEM_FIXED**. This parameter can be one or more of the following values, except for the incompatible combinations that are specifically noted.

Value	Meaning
GHND 0x0042	Combines GMEM_MOVEABLE and GMEM_ZEROINIT .
GMEM_FIXED 0x0000	Allocates fixed memory. The return value is a pointer.
GMEM_MOVEABLE 0x0002	Allocates movable memory. Memory blocks are never moved in physical memory, but they can be moved within the default heap. The return value is a handle to the memory object. To translate the handle into a pointer, use the GlobalLock function.

	This value cannot be combined with GMEM_FIXED .
GMEM_ZEROINIT 0x0040	Initializes memory contents to zero.
GPTR 0x0040	Combines GMEM_FIXED and GMEM_ZEROINIT .

The following values are obsolete, but are provided for compatibility with 16-bit Windows. They are ignored.

GMEM_DDESHARE
GMEM_DISCARDABLE
GMEM_LOWER
GMEM_NOCOMPACT
GMEM_NODISCARD
GMEM_NOT_BANKED
GMEM_NOTIFY
GMEM_SHARE

[in] dwBytes

The number of bytes to allocate. If this parameter is zero and the *uFlags* parameter specifies **GMEM_MOVEABLE**, the function returns a handle to a memory object that is marked as discarded.

Return value

If the function succeeds, the return value is a handle to the newly allocated memory object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Windows memory management does not provide a separate local heap and global heap. Therefore, the **GlobalAlloc** and [LocalAlloc](#) functions are essentially the same.

The movable-memory flags **GHND** and **GMEM_MOVABLE** add unnecessary overhead and require locking to be used safely. They should be avoided unless documentation specifically states that they should be used.

New applications should use the [heap functions](#) to allocate and manage memory unless the documentation specifically states that a global function should be used. For example, the global functions are still used with Dynamic Data Exchange (DDE), the clipboard functions, and OLE data objects.

If the **GlobalAlloc** function succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual number of bytes allocated, use the [GlobalSize](#) function.

If the heap does not contain sufficient free space to satisfy the request, **GlobalAlloc** returns **NULL**. Because **NULL** is used to indicate an error, virtual address zero is never allocated. It is, therefore, easy to detect the use of a **NULL** pointer.

Memory allocated with this function is guaranteed to be aligned on an 8-byte boundary. To execute dynamically generated code, use the [VirtualAlloc](#) function to allocate memory and the [VirtualProtect](#) function to grant **PAGE_EXECUTE** access.

To free the memory, use the [GlobalFree](#) function. It is not safe to free memory allocated with **GlobalAlloc** using [LocalFree](#).

Examples

The following code shows a simple use of **GlobalAlloc** and **GlobalFree**.

C++

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _cdecl main()
{
    PSECURITY_DESCRIPTOR pSD;

    pSD = (PSECURITY_DESCRIPTOR) GlobalAlloc(
        GMEM_FIXED,
        sizeof(PSECURITY_DESCRIPTOR));

    // Handle error condition
    if( pSD == NULL )
    {
        _tprintf(TEXT("GlobalAlloc failed (%d)\n"), GetLastError());
        return;
    }

    //see how much memory was allocated
    _tprintf(TEXT("GlobalAlloc allocated %d bytes\n"), GlobalSize(pSD));
```

```
// Use the memory allocated  
  
// Free the memory when finished with it  
GlobalFree(pSD);  
}
```

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalDiscard](#)

[GlobalFree](#)

[GlobalLock](#)

[GlobalSize](#)

[Heap Functions](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GlobalDeleteAtom function (winbase.h)

Article10/13/2021

Decrements the reference count of a global string atom. If the atom's reference count reaches zero, **GlobalDeleteAtom** removes the string associated with the atom from the global atom table.

Syntax

C++

```
ATOM GlobalDeleteAtom(  
    [in] ATOM nAtom  
);
```

Parameters

[in] nAtom

Type: **ATOM**

The atom and character string to be deleted.

Return value

Type: **ATOM**

The function always returns (ATOM) 0.

To determine whether the function has failed, call [SetLastError](#) with **ERROR_SUCCESS** before calling **GlobalDeleteAtom**, then call [GetLastError](#). If the last error code is still **ERROR_SUCCESS**, **GlobalDeleteAtom** has succeeded.

Remarks

A string atom's reference count specifies the number of times the string has been added to the atom table. The [GlobalAddAtom](#) function increments the reference count of a string that already exists in the global atom table each time it is called.

Each call to [GlobalAddAtom](#) should have a corresponding call to [GlobalDeleteAtom](#). Do not call [GlobalDeleteAtom](#) more times than you call [GlobalAddAtom](#), or you may delete the atom while other clients are using it. Applications using Dynamic Data Exchange (DDE) should follow the rules on global atom management to prevent leaks and premature deletion.

[GlobalDeleteAtom](#) has no effect on an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF). The function always returns zero for an integer atom.

Examples

For an example, see [Initiating a Conversation](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GlobalAddAtom](#)

[GlobalFindAtom](#)

[MAKEINTATOM](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalDiscard macro (winbase.h)

Article10/13/2021

Discards the specified global memory block. The lock count of the memory object must be zero.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
void GlobalDiscard(  
    [in] h  
) ;
```

Parameters

[in] h

A handle to the global memory object. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function.

Return value

None

Remarks

Although **GlobalDiscard** discards the object's memory block, the handle to the object remains valid. The process can subsequently pass the handle to the [GlobalReAlloc](#) function to allocate another global memory block identified by the same handle.

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)

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalReAlloc](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalFindAtomA function (winbase.h)

Article02/09/2023

Searches the global atom table for the specified character string and retrieves the global atom associated with that string.

Syntax

C++

```
ATOM GlobalFindAtomA(
    [in] LPCSTR lpString
);
```

Parameters

[in] lpString

Type: **LPCTSTR**

The null-terminated character string for which to search.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

Return value

Type: **ATOM**

If the function succeeds, the return value is the global atom associated with the given string.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Even though the system preserves the case of a string in an atom table as it was originally entered, the search performed by **GlobalFindAtom** is not case sensitive.

If *lpString* was created by the [MAKEINTATOM](#) macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Note

The winbase.h header defines GlobalFindAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalGetAtomName](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalFindAtomW function (winbase.h)

Article02/09/2023

Searches the global atom table for the specified character string and retrieves the global atom associated with that string.

Syntax

C++

```
ATOM GlobalFindAtomW(
    [in] LPCWSTR lpString
);
```

Parameters

[in] lpString

Type: LPCTSTR

The null-terminated character string for which to search.

Alternatively, you can use an integer atom that has been converted using the [MAKEINTATOM](#) macro. See the Remarks for more information.

Return value

Type: ATOM

If the function succeeds, the return value is the global atom associated with the given string.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Even though the system preserves the case of a string in an atom table as it was originally entered, the search performed by **GlobalFindAtom** is not case sensitive.

If *lpString* was created by the [MAKEINTATOM](#) macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Note

The winbase.h header defines GlobalFindAtom as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalGetAtomName](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalFlags function (winbase.h)

Article10/13/2021

Retrieves information about the specified global memory object.

Note This function is provided only for compatibility with 16-bit versions of Windows. New applications should use the [heap functions](#). For more information, see Remarks.

Syntax

C++

```
UINT GlobalFlags(  
    [in] HGLOBAL hMem  
)
```

Parameters

[in] hMem

A handle to the global memory object. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function.

Return value

If the function succeeds, the return value specifies the allocation values and the lock count for the memory object.

If the function fails, the return value is [GMEM_INVALID_HANDLE](#), indicating that the global handle is not valid. To get extended error information, call [GetLastError](#).

Remarks

The low-order byte of the low-order word of the return value contains the lock count of the object. To retrieve the lock count from the return value, use the [GMEM_LOCKCOUNT](#)

mask with the bitwise AND (&) operator. The lock count of memory objects allocated with **GMEM_FIXED** is always zero.

The high-order byte of the low-order word of the return value indicates the allocation values of the memory object. It can be zero or **GMEM_DISCARDED**.

The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the [heap functions](#) unless documentation states that a global function should be used. For more information, see [Global and Local Functions](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalDiscard](#)

[GlobalReAlloc](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GlobalFree function (winbase.h)

Article10/13/2021

Frees the specified global memory object and invalidates its handle.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
HGLOBAL GlobalFree(
    [in] _Frees_ptr_opt_ HGLOBAL hMem
);
```

Parameters

[in] hMem

A handle to the global memory object. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function. It is not safe to free memory allocated with [LocalAlloc](#).

Return value

If the function succeeds, the return value is **NULL**.

If the function fails, the return value is equal to a handle to the global memory object. To get extended error information, call [GetLastError](#).

Remarks

If the process examines or modifies the memory after it has been freed, heap corruption may occur or an access violation exception (EXCEPTION_ACCESS_VIOLATION) may be generated.

The **GlobalFree** function will free a locked memory object. A locked memory object has a lock count greater than zero. The **GlobalLock** function locks a global memory object and increments the lock count by one. The **GlobalUnlock** function unlocks it and decrements the lock count by one. To get the lock count of a global memory object, use the **GlobalFlags** function.

If an application is running under a debug version of the system, **GlobalFree** will issue a message that tells you that a locked object is being freed. If you are debugging the application, **GlobalFree** will enter a breakpoint just before freeing a locked object. This allows you to verify the intended behavior, then continue execution.

Examples

For an example, see [GlobalAlloc](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalFlags](#)

[GlobalLock](#)

[GlobalReAlloc](#)

[GlobalUnlock](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalGetAtomNameA function (winbase.h)

Article 02/09/2023

Retrieves a copy of the character string associated with the specified global atom.

Syntax

C++

```
UINT GlobalGetAtomNameA(
    [in] ATOM nAtom,
    [out] LPSTR lpBuffer,
    [in] int nSize
);
```

Parameters

[in] nAtom

Type: **ATOM**

The global atom associated with the character string to be retrieved.

[out] lpBuffer

Type: **LPTSTR**

The buffer for the character string.

[in] nSize

Type: **int**

The size, in characters, of the buffer.

Return value

Type: **UINT**

If the function succeeds, the return value is the length of the string copied to the buffer, in characters, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Security Considerations

Using this function incorrectly might compromise the security of your program. Incorrect use of this function includes not correctly specifying the size of the *lpBuffer* parameter. Also, note that a global atom is accessible by anyone; thus, privacy and the integrity of its contents is not assured.

ⓘ Note

The winbase.h header defines GlobalGetAtomName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[MAKEINTATOM](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalGetAtomNameW function (winbase.h)

Article 02/09/2023

Retrieves a copy of the character string associated with the specified global atom.

Syntax

C++

```
UINT GlobalGetAtomNameW(
    [in] ATOM nAtom,
    [out] LPWSTR lpBuffer,
    [in] int nSize
);
```

Parameters

[in] nAtom

Type: **ATOM**

The global atom associated with the character string to be retrieved.

[out] lpBuffer

Type: **LPTSTR**

The buffer for the character string.

[in] nSize

Type: **int**

The size, in characters, of the buffer.

Return value

Type: **UINT**

If the function succeeds, the return value is the length of the string copied to the buffer, in characters, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Security Considerations

Using this function incorrectly might compromise the security of your program. Incorrect use of this function includes not correctly specifying the size of the *lpBuffer* parameter. Also, note that a global atom is accessible by anyone; thus, privacy and the integrity of its contents is not assured.

ⓘ Note

The winbase.h header defines GlobalGetAtomName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[MAKEINTATOM](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalHandle function (winbase.h)

Article10/13/2021

Retrieves the handle associated with the specified pointer to a global memory block.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
HGLOBAL GlobalHandle(  
    [in] LPCVOID pMem  
)
```

Parameters

[in] pMem

A pointer to the first byte of the global memory block. This pointer is returned by the [GlobalLock](#) function.

Return value

If the function succeeds, the return value is a handle to the specified global memory object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

When the [GlobalAlloc](#) function allocates a memory object with **GMEM_MOVEABLE**, it returns a handle to the object. The [GlobalLock](#) function converts this handle into a

pointer to the memory block, and **GlobalHandle** converts the pointer back into a handle.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalLock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalLock function (winbase.h)

Article10/13/2021

Locks a global memory object and returns a pointer to the first byte of the object's memory block.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
LPVOID GlobalLock(  
    [in] HGLOBAL hMem  
);
```

Parameters

[in] hMem

A handle to the global memory object. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function.

Return value

If the function succeeds, the return value is a pointer to the first byte of the memory block.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, **GlobalLock** increments the count by one, and the **GlobalUnlock** function decrements the count by one. Each successful call that a process makes to **GlobalLock** for an object must be matched by a corresponding call to **GlobalUnlock**. Locked memory will not be moved or discarded, unless the memory object is reallocated by using the **GlobalReAlloc** function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.

Memory objects allocated with **GMEM_FIXED** always have a lock count of zero. For these objects, the value of the returned pointer is equal to the value of the specified handle.

If the specified memory block has been discarded or if the memory block has a zero-byte size, this function returns **NULL**.

Discarded objects always have a lock count of zero.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalReAlloc](#)

[GlobalUnlock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalMemoryStatus function (winbase.h)

Article07/27/2022

[[GlobalMemoryStatus](#) can return incorrect information. Use the [GlobalMemoryStatusEx](#) function instead.]

Retrieves information about the system's current usage of both physical and virtual memory.

Syntax

C++

```
void GlobalMemoryStatus(
    [out] LPMEMORYSTATUS lpBuffer
);
```

Parameters

`[out] lpBuffer`

A pointer to a [MEMORYSTATUS](#) structure. The [GlobalMemoryStatus](#) function stores information about current memory availability into this structure.

Return value

None

Remarks

On computers with more than 4 GB of memory, the [GlobalMemoryStatus](#) function can return incorrect information, reporting a value of –1 to indicate an overflow. For this reason, applications should use the [GlobalMemoryStatusEx](#) function instead.

On Intel x86 computers with more than 2 GB and less than 4 GB of memory, the [GlobalMemoryStatus](#) function will always return 2 GB in the `dwTotalPhys` member of the [MEMORYSTATUS](#) structure. Similarly, if the total available memory is between 2 and 4 GB, the `dwAvailPhys` member of the [MEMORYSTATUS](#) structure will be rounded down

to 2 GB. If the executable is linked using the `/LARGEADDRESSAWARE` linker option, then the **GlobalMemoryStatus** function will return the correct amount of physical memory in both members.

The information returned by the **GlobalMemoryStatus** function is volatile. There is no guarantee that two sequential calls to this function will return the same information.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GlobalMemoryStatusEx](#)

[MEMORYSTATUS](#)

[Memory Management Functions](#)

[Memory Performance Information](#)

[Virtual Address Space and Physical Storage](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalReAlloc function (winbase.h)

Article10/13/2021

Changes the size or attributes of a specified global memory object. The size can increase or decrease.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
DECLSPEC_ALLOCATOR HGLOBAL GlobalReAlloc(
    [in] _Frees_ptr_ HGLOBAL hMem,
    [in] SIZE_T          dwBytes,
    [in] UINT            uFlags
);
```

Parameters

[in] hMem

A handle to the global memory object to be reallocated. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function.

[in] dwBytes

The new size of the memory block, in bytes. If *uFlags* specifies **GMEM_MODIFY**, this parameter is ignored.

[in] uFlags

The reallocation options. If **GMEM_MODIFY** is specified, the function modifies the attributes of the memory object only (the *dwBytes* parameter is ignored.) Otherwise, the function reallocates the memory object.

You can optionally combine **GMEM_MODIFY** with the following value.

Value	Meaning
GMEM_MOVEABLE 0x0002	Allocates movable memory. If the memory is a locked GMEM_MOVEABLE memory block or a GMEM_FIXED memory block and this flag is not specified, the memory can only be reallocated in place.

If this parameter does not specify **GMEM_MODIFY**, you can use the following value.

Value	Meaning
GMEM_ZEROINIT 0x0040	Causes the additional memory contents to be initialized to zero if the memory object is growing in size.

Return value

If the function succeeds, the return value is a handle to the reallocated memory object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

If **GlobalReAlloc** reallocates a movable object, the return value is a handle to the memory object. To convert the handle to a pointer, use the [GlobalLock](#) function.

If **GlobalReAlloc** reallocates a fixed object, the value of the handle returned is the address of the first byte of the memory block. To access the memory, a process can simply cast the return value to a pointer.

If **GlobalReAlloc** fails, the original memory is not freed, and the original handle and pointer are still valid.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalDiscard](#)

[GlobalLock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalSize function (winbase.h)

Article10/13/2021

Retrieves the current size of the specified global memory object, in bytes.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
SIZE_T GlobalSize(  
    [in] HGLOBAL hMem  
)
```

Parameters

[in] hMem

A handle to the global memory object. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function.

Return value

If the function succeeds, the return value is the size of the specified global memory object, in bytes.

If the specified handle is not valid or if the object has been discarded, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The size of a memory block may be larger than the size requested when the memory was allocated.

To verify that the specified object's memory block has not been discarded, use the [GlobalFlags](#) function before calling [GlobalSize](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalFlags](#)

[GlobalReAlloc](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

GlobalUnlock function (winbase.h)

Article10/13/2021

Decrements the lock count associated with a memory object that was allocated with **GMEM_MOVEABLE**. This function has no effect on memory objects allocated with **GMEM_FIXED**.

Note The global functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a global function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
BOOL GlobalUnlock(  
    [in] HGLOBAL hMem  
);
```

Parameters

[in] hMem

A handle to the global memory object. This handle is returned by either the [GlobalAlloc](#) or [GlobalReAlloc](#) function.

Return value

If the memory object is still locked after decrementing the lock count, the return value is a nonzero value. If the memory object is unlocked after decrementing the lock count, the function returns zero and [GetLastError](#) returns **NO_ERROR**.

If the function fails, the return value is zero and [GetLastError](#) returns a value other than **NO_ERROR**.

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, the [GlobalLock](#) function increments the count by one, and [GlobalUnlock](#) decrements the count by one. For each call that a process makes to [GlobalLock](#) for an object, it must eventually call [GlobalUnlock](#). Locked memory will not be moved or discarded, unless the memory object is reallocated by using the [GlobalReAlloc](#) function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.

Memory objects allocated with **GMEM_FIXED** always have a lock count of zero. If the specified memory block is fixed memory, this function returns **TRUE**.

If the memory object is already unlocked, [GlobalUnlock](#) returns **FALSE** and [GetLastError](#) reports **ERROR_NOT_LOCKED**.

A process should not rely on the return value to determine the number of times it must subsequently call [GlobalUnlock](#) for a memory object.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalAlloc](#)

[GlobalLock](#)

[GlobalReAlloc](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

HasOverlappedIoCompleted macro (winbase.h)

Article04/02/2021

Provides a high performance test operation that can be used to poll for the completion of an outstanding I/O operation.

Syntax

C++

```
void HasOverlappedIoCompleted(  
    _lpOverlapped  
) ;
```

Parameters

`_lpOverlapped`

A pointer to an [OVERLAPPED](#) structure that was specified when the overlapped I/O operation was started.

Return value

None

Remarks

Do not call this macro unless the call to [GetLastError](#) returns [ERROR_IO_PENDING](#), indicating that the overlapped I/O has started.

To cancel all pending asynchronous I/O operations, use the [CancelIo](#) function. The [CancelIo](#) function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error [ERROR_OPERATION_ABORTED](#).

To get more details about a completed I/O operation, call the [GetOverlappedResult](#) or [GetQueuedCompletionStatus](#) function.

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)

See also

[CancelIo](#)

[ConnectNamedPipe](#)

[OVERLAPPED](#)

[ReadFile](#)

[TransactNamedPipe](#)

[WaitCommEvent](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

HW_PROFILE_INFOA structure (winbase.h)

Article09/01/2022

Contains information about a hardware profile. The [GetCurrentHwProfile](#) function uses this structure to retrieve the current hardware profile for the local computer.

Syntax

C++

```
typedef struct tagHW_PROFILE_INFOA {
    DWORD dwDockInfo;
    CHAR szHwProfileGuid[HW_PROFILE_GUIDLEN];
    CHAR szHwProfileName[MAX_PROFILE_LEN];
} HW_PROFILE_INFOA, *LPHW_PROFILE_INFOA;
```

Members

`dwDockInfo`

The reported docking state of the computer. This member can be a combination of the following bit values.

Value	Meaning
DOCKINFO_DOCKED 0x2	The computer is docked.
DOCKINFO_UNDOCKED 0x1	The computer is undocked. This flag is always set for desktop systems that cannot be undocked.
DOCKINFO_USER_SUPPLIED 0x4	If this flag is set, GetCurrentHwProfile retrieved the current docking state from information provided by the user in the Hardware Profiles page of the System control panel application. If there is no such value or the value is set to 0, this flag is set.
DOCKINFO_USER.DockED 0x5	The computer is docked, according to information provided by the user. This value is a combination of the DOCKINFO_USER_SUPPLIED and DOCKINFO_DOCKED flags.

DOCKINFO_USER_UNDOCKED	The computer is undocked, according to information provided by the user. This value is a combination of the DOCKINFO_USER_SUPPLIED and DOCKINFO_UNDOCKED flags.
0x6	

`szHwProfileGuid[HW_PROFILE_GUIDLEN]`

The globally unique identifier (GUID) string for the current hardware profile. The string returned by [GetCurrentHwProfile](#) encloses the GUID in curly braces, {}; for example:

```
{12340001-4980-1920-6788-123456789012}
```

You can use this string as a registry subkey under your application's configuration settings key in **HKEY_CURRENT_USER**. This enables you to store settings for each hardware profile.

`szHwProfileName[MAX_PROFILE_LEN]`

The display name for the current hardware profile.

Remarks

ⓘ Note

The winbase.h header defines HW_PROFILE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winbase.h (include Windows.h)

See also

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

HW_PROFILE_INFOW structure (winbase.h)

Article09/01/2022

Contains information about a hardware profile. The [GetCurrentHwProfile](#) function uses this structure to retrieve the current hardware profile for the local computer.

Syntax

C++

```
typedef struct tagHW_PROFILE_INFOW {
    DWORD dwDockInfo;
    WCHAR szHwProfileGuid[HW_PROFILE_GUIDLEN];
    WCHAR szHwProfileName[MAX_PROFILE_LEN];
} HW_PROFILE_INFOW, *LPHW_PROFILE_INFOW;
```

Members

dwDockInfo

The reported docking state of the computer. This member can be a combination of the following bit values.

Value	Meaning
DOCKINFO_DOCKED 0x2	The computer is docked.
DOCKINFO_UNDOCKED 0x1	The computer is undocked. This flag is always set for desktop systems that cannot be undocked.
DOCKINFO_USER_SUPPLIED 0x4	If this flag is set, GetCurrentHwProfile retrieved the current docking state from information provided by the user in the Hardware Profiles page of the System control panel application. If there is no such value or the value is set to 0, this flag is set.
DOCKINFO_USER_DOCKED 0x5	The computer is docked, according to information provided by the user. This value is a combination of the DOCKINFO_USER_SUPPLIED and DOCKINFO_DOCKED flags.

DOCKINFO_USER_UNDOCKED	The computer is undocked, according to information provided by the user. This value is a combination of the DOCKINFO_USER_SUPPLIED and DOCKINFO_UNDOCKED flags.
0x6	

`szHwProfileGuid[HW_PROFILE_GUIDLEN]`

The globally unique identifier (GUID) string for the current hardware profile. The string returned by [GetCurrentHwProfile](#) encloses the GUID in curly braces, {}; for example:

```
{12340001-4980-1920-6788-123456789012}
```

You can use this string as a registry subkey under your application's configuration settings key in **HKEY_CURRENT_USER**. This enables you to store settings for each hardware profile.

`szHwProfileName[MAX_PROFILE_LEN]`

The display name for the current hardware profile.

Remarks

Note

The winbase.h header defines HW_PROFILE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winbase.h (include Windows.h)

See also

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

InitAtomTable function (winbase.h)

Article10/13/2021

Initializes the local atom table and sets the number of hash buckets to the specified size.

Syntax

C++

```
BOOL InitAtomTable(  
    [in] DWORD nSize  
);
```

Parameters

[in] *nSize*

Type: **DWORD**

The number of hash buckets to use for the atom table. If this parameter is zero, the default number of hash buckets are created.

To achieve better performance, specify a prime number in *nSize*.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

An application need not use this function to use a local atom table. The default number of hash buckets used is 37. If an application does use **InitAtomTable**, however, it should call the function before any other atom-management function.

If an application uses a large number of local atoms, it can reduce the time required to add an atom to the local atom table or to find an atom in the table by increasing the

size of the table. However, this increases the amount of memory required to maintain the table.

The number of buckets in the global atom table cannot be changed. If the atom table has already been initialized, either explicitly by a prior call to **InitAtomTable**, or implicitly by the use of any atom-management function, **InitAtomTable** returns success without changing the number of hash buckets.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddAtom](#)

[DeleteAtom](#)

[FindAtom](#)

[GetAtomName](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalFindAtom](#)

[GlobalGetAtomName](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

InitializeContext function (winbase.h)

Article 10/13/2021

Initializes a [CONTEXT](#) structure inside a buffer with the necessary size and alignment.

Syntax

C++

```
BOOL InitializeContext(
    [out, optional] PVOID    Buffer,
    [in]          DWORD     ContextFlags,
    [out, optional] PCONTEXT *Context,
    [in, out]       PDWORD    ContextLength
);
```

Parameters

[out, optional] *Buffer*

A pointer to a buffer within which to initialize a [CONTEXT](#) structure. This parameter can be **NULL** to determine the buffer size required to hold a context record with the specified *ContextFlags*.

[in] *ContextFlags*

A value indicating which portions of the *Context* structure should be initialized. This parameter influences the size of the initialized *Context* structure.

Note `CONTEXT_XSTATE` is not part of `CONTEXT_FULL` or `CONTEXT_ALL`. It must be specified separately if an XState context is desired.

[out, optional] *Context*

A pointer to a variable which receives the address of the initialized [CONTEXT](#) structure within the *Buffer*.

Note Due to alignment requirements of [CONTEXT](#) structures, the value returned in *Context* may not be at the beginning of the supplied buffer.

[in, out] ContextLength

On input, specifies the length of the buffer pointed to by *Buffer*, in bytes. If the buffer is not large enough to contain the specified portions of the [CONTEXT](#), the function fails, [GetLastError](#) returns [ERROR_INSUFFICIENT_BUFFER](#), and *ContextLength* is set to the required size of the buffer. If the function fails with an error other than [ERROR_INSUFFICIENT_BUFFER](#), the contents of *ContextLength* are undefined.

Return value

This function returns **TRUE** if successful, otherwise **FALSE**. To get extended error information, call [GetLastError](#).

Remarks

InitializeContext can be used to initialize a [CONTEXT](#) structure within a buffer with the required size and alignment characteristics. This routine is required if the [CONTEXT_XSTATE](#) *ContextFlag* is specified since the required context size and alignment may change depending on which processor features are enabled on the system.

First, call this function with the *ContextFlags* parameter set to the maximum number of features you will be using and the *Buffer* parameter to **NULL**. The function returns the required buffer size in bytes in the *ContextLength* parameter. Allocate enough space for the data in the *Buffer* and call the function again to initialize the *Context*. Upon successful completion of this routine, the *ContextFlags* member of the *Context* structure is initialized, but the remaining contents of the structure are undefined. Some bits specified in the *ContextFlags* parameter may not be set in *Context->ContextFlags* if they are not supported by the system. Applications may subsequently remove, but must never add, bits from the *ContextFlags* member of [CONTEXT](#).

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The [AVX API](#) is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1. Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this documentation and get pointers to them using [GetProcAddress](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CONTEXT](#)

[CopyContext](#)

[Intel AVX](#)

[Working with XState Context](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

InitializeContext2 function (winbase.h)

Article 10/20/2021

Initializes a [CONTEXT](#) structure inside a buffer with the necessary size and alignment, with the option to specify an XSTATE compaction mask.

Syntax

C++

```
BOOL InitializeContext2(
    [out, optional] PVOID     Buffer,
                           DWORD     ContextFlags,
    [out, optional] PCONTEXT *Context,
    [in, out]        PDWORD    ContextLength,
                           ULONG64   XStateCompactionMask
);
```

Parameters

[out, optional] *Buffer*

A pointer to a buffer within which to initialize a [CONTEXT](#) structure. This parameter can be **NULL** to determine the buffer size required to hold a context record with the specified *ContextFlags*.

ContextFlags

A value indicating which portions of the *Context* structure should be initialized. This parameter influences the size of the initialized *Context* structure.

Note **CONTEXT_XSTATE** is not part of **CONTEXT_FULL** or **CONTEXT_ALL**. It must be specified separately if an XState context is desired.

[out, optional] *Context*

A pointer to a variable which receives the address of the initialized [CONTEXT](#) structure within the *Buffer*.

Note Due to alignment requirements of **CONTEXT** structures, the value returned in *Context* may not be at the beginning of the supplied buffer.

[in, out] ContextLength

On input, specifies the length of the buffer pointed to by *Buffer*, in bytes. If the buffer is not large enough to contain the specified portions of the **CONTEXT**, the function fails, [GetLastError](#) returns **ERROR_INSUFFICIENT_BUFFER**, and *ContextLength* is set to the required size of the buffer. If the function fails with an error other than **ERROR_INSUFFICIENT_BUFFER**, the contents of *ContextLength* are undefined.

XStateCompactionMask

Supplies the XState compaction mask to use when allocating the *Context* structure. This parameter is only used when **CONTEXT_XSTATE** is supplied to *ContextFlags* and the system has XState enabled in compaction mode.

Return value

This function returns **TRUE** if successful, otherwise **FALSE**. To get extended error information, call [GetLastError](#).

Remarks

InitializeContext can be used to initialize a **CONTEXT** structure within a buffer with the required size and alignment characteristics. This routine is required if the **CONTEXT_XSTATE** *ContextFlag* is specified since the required context size and alignment may change depending on which processor features are enabled on the system.

First, call this function with the *ContextFlags* parameter set to the maximum number of features you will be using and the *Buffer* parameter to **NULL**. The function returns the required buffer size in bytes in the *ContextLength* parameter. Allocate enough space for the data in the *Buffer* and call the function again to initialize the *Context*. Upon successful completion of this routine, the *ContextFlags* member of the *Context* structure is initialized, but the remaining contents of the structure are undefined. Some bits specified in the *ContextFlags* parameter may not be set in *Context->ContextFlags* if they are not supported by the system. Applications may subsequently remove, but must never add, bits from the *ContextFlags* member of **CONTEXT**.

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The AVX API is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1 . Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this documentation and get pointers to them using [GetModuleHandle](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

When XState is enabled in compaction mode, specifying an *XStateCompactionMask* that contains only a subset of the enabled XState components can decrease the buffer size required to store the *Context*. This is particularly useful if the system has many XState components enabled, but the *Context* will only be used to affect a small number of XState components. The full set of enabled XState components can be obtained by calling [GetEnabledXStateFeatures](#). This function copies the specified XState compaction mask into the relevant location in the XState header.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winbase.h

See also

[CONTEXT](#)

[CopyContext](#)

[Intel AVX](#)

[Working with XState Context](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

InitializeThreadpoolEnvironment function (winbase.h)

Article 10/13/2021

Initializes a callback environment.

Syntax

C++

```
void InitializeThreadpoolEnvironment(
    [out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

[out] pcbe

A TP_CALLBACK_ENVIRON structure that defines a callback environment.

Return value

None

Remarks

By default, a callback executes in the default thread pool for the process. No cleanup group is associated with the callback environment, the caller is responsible for keeping the callback's DLL loaded while there are outstanding callbacks, and the callback is expected to run in a reasonable amount of time for the application.

Create a callback environment if you plan to call one of the following functions to modify the environment:

- [SetThreadpoolCallbackCleanupGroup](#)
- [SetThreadpoolCallbackLibrary](#)
- [SetThreadpoolCallbackPool](#)
- [SetThreadpoolCallbackPriority](#)
- [SetThreadpoolCallbackRunsLong](#)

To use the default callback environment, set the optional callback environment parameter to NULL when calling one of the following functions:

- [CreateThreadpoolIo](#)
- [CreateThreadpoolTimer](#)
- [CreateThreadpoolWait](#)
- [CreateThreadpoolWork](#)
- [TrySubmitThreadpoolCallback](#)

The **InitializeThreadpoolEnvironment** function is implemented as an inline function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or higher.

Examples

For an example, see [Using the Thread Pool Functions](#).

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[DestroyThreadpoolEnvironment](#)

[SetThreadpoolCallbackCleanupGroup](#)

[SetThreadpoolCallbackLibrary](#)

[SetThreadpoolCallbackPool](#)

[SetThreadpoolCallbackRunsLong](#)

[Thread Pools](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

InterlockedExchangeSubtract function (winbase.h)

Article 10/13/2021

Performs an atomic subtraction of two values.

Syntax

C++

```
unsigned InterlockedExchangeSubtract(
    [in, out] unsigned volatile *Addend,
    [in]      unsigned           Value
);
```

Parameters

[in, out] Addend

A pointer to a variable. The value of this variable is replaced with the result of the operation.

[in] Value

The value to be subtracted from the variable pointed to by the *Addend* parameter.

Return value

The function returns the initial value of the *Addend* parameter.

Remarks

This function generates a full memory barrier (or fence) to ensure that memory operations are completed in order.

Requirements

Minimum supported client	Windows 7 [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[Interlocked Variable Access](#)

[InterlockedCompareExchange](#)

[InterlockedExchange](#)

[InterlockedExchangeAdd](#)

[InterlockedExchangePointer](#)

[Synchronization Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsBadCodePtr function (winbase.h)

Article10/13/2021

Determines whether the calling process has read access to the memory at the specified address.

Important This function is obsolete and should not be used. Despite its name, it does not guarantee that the pointer is valid or that the memory pointed to is safe to use. For more information, see Remarks on this page.

Syntax

C++

```
BOOL IsBadCodePtr(  
    [in] FARPROC lpfn  
>;
```

Parameters

[in] lpfn

A pointer to a memory address.

Return value

If the calling process has read access to the specified memory, the return value is zero.

If the calling process does not have read access to the specified memory, the return value is nonzero. To get extended error information, call [GetLastError](#).

If the application is compiled as a debugging version, and the process does not have read access to the specified memory location, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[IsBadReadPtr](#)

[IsBadStringPtr](#)

[IsBadWritePtr](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsBadReadPtr function (winbase.h)

Article07/27/2022

Verifies that the calling process has read access to the specified range of memory.

Important This function is obsolete and should not be used. Despite its name, it does not guarantee that the pointer is valid or that the memory pointed to is safe to use. For more information, see Remarks on this page.

Syntax

C++

```
BOOL IsBadReadPtr(  
    [in] const VOID *lp,  
    [in] UINT_PTR    ucb  
)
```

Parameters

[in] lp

A pointer to the first byte of the memory block.

[in] ucb

The size of the memory block, in bytes. If this parameter is zero, the return value is zero.

Return value

If the calling process has read access to all bytes in the specified memory range, the return value is zero.

If the calling process does not have read access to all bytes in the specified memory range, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to all bytes in the specified memory range, the function causes an assertion

and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has read access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[IsBadCodePtr](#)

[IsBadStringPtr](#)

[IsBadWritePtr](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsBadStringPtrA function (winbase.h)

Article02/09/2023

Verifies that the calling process has read access to the specified range of memory.

Important This function is obsolete and should not be used. Despite its name, it does not guarantee that the pointer is valid or that the memory pointed to is safe to use. For more information, see Remarks on this page.

Syntax

C++

```
BOOL IsBadStringPtrA(
    [in] LPCSTR lpsz,
    [in] UINT_PTR ucchMax
);
```

Parameters

[in] `lpsz`

A pointer to a null-terminated string, either Unicode or ASCII.

[in] `ucchMax`

The maximum size of the string, in TCHARs. The function checks for read access in all characters up to the string's terminating null character or up to the number of characters specified by this parameter, whichever is smaller. If this parameter is zero, the return value is zero.

Return value

If the calling process has read access to all characters up to the string's terminating null character or up to the number of characters specified by `ucchMax`, the return value is zero.

If the calling process does not have read access to all characters up to the string's terminating null character or up to the number of characters specified by *ucchMax*, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to the entire memory range specified, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has read access to some, but not all, of the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use [structured exception handling](#) when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

Note

The winbase.h header defines `IsBadStringPtr` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[IsBadCodePtr](#)

[IsBadReadPtr](#)

[IsBadWritePtr](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsBadStringPtrW function (winbase.h)

Article02/09/2023

Verifies that the calling process has read access to the specified range of memory.

Important This function is obsolete and should not be used. Despite its name, it does not guarantee that the pointer is valid or that the memory pointed to is safe to use. For more information, see Remarks on this page.

Syntax

C++

```
BOOL IsBadStringPtrW(
    [in] LPCWSTR lpsz,
    [in] UINT_PTR ucchMax
);
```

Parameters

[in] `lpsz`

A pointer to a null-terminated string, either Unicode or ASCII.

[in] `ucchMax`

The maximum size of the string, in TCHARs. The function checks for read access in all characters up to the string's terminating null character or up to the number of characters specified by this parameter, whichever is smaller. If this parameter is zero, the return value is zero.

Return value

If the calling process has read access to all characters up to the string's terminating null character or up to the number of characters specified by `ucchMax`, the return value is zero.

If the calling process does not have read access to all characters up to the string's terminating null character or up to the number of characters specified by *ucchMax*, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to the entire memory range specified, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has read access to some, but not all, of the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use [structured exception handling](#) when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

Note

The winbase.h header defines `IsBadStringPtr` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[IsBadCodePtr](#)

[IsBadReadPtr](#)

[IsBadWritePtr](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsBadWritePtr function (winbase.h)

Article10/13/2021

Verifies that the calling process has write access to the specified range of memory.

Important This function is obsolete and should not be used. Despite its name, it does not guarantee that the pointer is valid or that the memory pointed to is safe to use. For more information, see Remarks on this page.

Syntax

C++

```
BOOL IsBadWritePtr(
    [in] LPVOID  lp,
    [in] UINT_PTR ucb
);
```

Parameters

[in] lp

A pointer to the first byte of the memory block.

[in] ucb

The size of the memory block, in bytes. If this parameter is zero, the return value is zero.

Return value

If the calling process has write access to all bytes in the specified memory range, the return value is zero.

If the calling process does not have write access to all bytes in the specified memory range, the return value is nonzero.

If the application is run under a debugger and the process does not have write access to all bytes in the specified memory range, the function causes a first chance STATUS_ACCESS_VIOLATION exception. The debugger can be configured to break for

this condition. After resuming process execution in the debugger, the function continues as usual and returns a nonzero value. This behavior is by design and serves as a debugging aid.

Remarks

This function is typically used when working with pointers returned from third-party libraries, where you cannot determine the memory management behavior in the third-party DLL.

Threads in a process are expected to cooperate in such a way that one will not free memory that the other needs. Use of this function does not negate the need to do this. If this is not done, the application may fail in an unpredictable manner.

Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

If the calling process has write access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has write access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

IsBadWritePtr is not multithread safe. To use it properly on a pointer shared by multiple threads, call it inside a critical region of code that allows only one thread to access the memory being checked. Use operating system-level objects such as critical sections or mutexes or the interlocked functions to create the critical region of code.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[IsBadCodePtr](#)

[IsBadReadPtr](#)

[IsBadStringPtr](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsNativeVhdBoot function (winbase.h)

Article10/13/2021

Indicates if the OS was booted from a VHD container.

Syntax

C++

```
BOOL IsNativeVhdBoot(
    [out] PBOOL NativeVhdBoot
);
```

Parameters

[out] NativeVhdBoot

Pointer to a variable that receives a boolean indicating if the OS was booted from a VHD.

Return value

TRUE if the OS was a native VHD boot; otherwise, FALSE.

Call [GetLastError](#) to get extended error information.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProcessHandleCount](#)

[GetProcessMemoryInfo](#)

[GetSystemInfo](#)

[GetSystemRegistryQuota](#)

[GetSystemTimes](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsSystemResumeAutomatic function (winbase.h)

Article 06/29/2021

Determines the current state of the computer.

Syntax

C++

```
BOOL IsSystemResumeAutomatic();
```

Return value

If the system was restored to the working state automatically and the user is not active, the function returns **TRUE**. Otherwise, the function returns **FALSE**.

Remarks

The [PBT_APMRESUMEAUTOMATIC](#) event is broadcast when the system wakes automatically to handle an event. The user is generally not present. If the system detects any user activity after broadcasting the PBT_APMRESUMEAUTOMATIC event, it will broadcast the [PBT_APMRESUMESUSPEND](#) event to let applications know they can resume full interaction with the user.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[PBT_APMRESUMEAUTOMATIC](#)

[PBT_APMRESUMESUSPEND](#)

[Power Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IsTextUnicode function (winbase.h)

Article 10/13/2021

Determines if a buffer is likely to contain a form of Unicode text.

Syntax

C++

```
BOOL IsTextUnicode(
    [in]             const VOID *lpv,
    [in]             int      iSize,
    [in, out, optional] LPINT    lpiResult
);
```

Parameters

[in] *lpv*

Pointer to the input buffer to examine.

[in] *iSize*

Size, in bytes, of the input buffer indicated by *lpv*.

[in, out, optional] *lpiResult*

On input, pointer to the tests to apply to the input buffer text. On output, this parameter receives the results of the specified tests: 1 if the contents of the buffer pass a test, 0 for failure. Only flags that are set upon input to the function are significant upon output.

If *lpiResult* is **NULL**, the function uses all available tests to determine if the data in the buffer is likely to be Unicode text.

This parameter can be one or more of the following values. Values can be combined with binary "OR".

Value	Meaning
IS_TEXT_UNICODE_ASCII16	The text is Unicode, and contains only zero-extended ASCII values/characters.
IS_TEXT_UNICODE_REVERSE_ASCII16	Same as the preceding, except that the Unicode text

	is byte-reversed.
IS_TEXT_UNICODE_STATISTICS	The text is probably Unicode, with the determination made by applying statistical analysis. Absolute certainty is not guaranteed. See the Remarks section.
IS_TEXT_UNICODE_REVERSE_STATISTICS	Same as the preceding, except that the text that is probably Unicode is byte-reversed.
IS_TEXT_UNICODE_CONTROLS	The text contains Unicode representations of one or more of these nonprinting characters: RETURN, LINEFEED, SPACE, CJK_SPACE, TAB.
IS_TEXT_UNICODE_REVERSE_CONTROLS	Same as the preceding, except that the Unicode characters are byte-reversed.
IS_TEXT_UNICODE_BUFFER_TOO_SMALL	There are too few characters in the buffer for meaningful analysis (fewer than two bytes).
IS_TEXT_UNICODE_SIGNATURE	The text contains the Unicode byte-order mark (BOM) 0xFEFF as its first character.
IS_TEXT_UNICODE_REVERSE_SIGNATURE	The text contains the Unicode byte-reversed byte-order mark (Reverse BOM) 0xFFFF as its first character.
IS_TEXT_UNICODE_ILLEGAL_CHARS	The text contains one of these Unicode-illegal characters: embedded Reverse BOM, UNICODE_NUL, CRLF (packed into one word), or 0xFFFF.
IS_TEXT_UNICODE_ODD_LENGTH	The number of characters in the string is odd. A string of odd length cannot (by definition) be Unicode text.
IS_TEXT_UNICODE_NULL_BYTES	The text contains null bytes, which indicate non-ASCII text.
IS_TEXT_UNICODE_UNICODE_MASK	The value is a combination of IS_TEXT_UNICODE_ASCII16, IS_TEXT_UNICODE_STATISTICS, IS_TEXT_UNICODE_CONTROLS, IS_TEXT_UNICODE_SIGNATURE.
IS_TEXT_UNICODE_REVERSE_MASK	The value is a combination of IS_TEXT_UNICODE_REVERSE_ASCII16, IS_TEXT_UNICODE_REVERSE_STATISTICS, IS_TEXT_UNICODE_REVERSE_CONTROLS, IS_TEXT_UNICODE_REVERSE_SIGNATURE.
IS_TEXT_UNICODE_NOT_UNICODE_MASK	The value is a combination of IS_TEXT_UNICODE_ILLEGAL_CHARS,

	IS_TEXT_UNICODE_ODD_LENGTH, and two currently unused bit flags.
IS_TEXT_UNICODE_NOT_ASCII_MASK	The value is a combination of IS_TEXT_UNICODE_NULL_BYTEx and three currently unused bit flags.

Return value

Returns a nonzero value if the data in the buffer passes the specified tests. The function returns 0 if the data in the buffer does not pass the specified tests.

Remarks

This function uses various statistical and deterministic methods to make its determination, under the control of flags passed in the *lpiResult* parameter. When the function returns, the results of such tests are reported using the same parameter.

The IS_TEXT_UNICODE_STATISTICS and IS_TEXT_UNICODE_REVERSE_STATISTICS tests use statistical analysis. These tests are not foolproof. The statistical tests assume certain amounts of variation between low and high bytes in a string, and some ASCII strings can slip through. For example, if *lpv* indicates the ASCII string 0x41, 0x0A, 0x0D, 0x1D (A\n\r^Z), the string passes the IS_TEXT_UNICODE_STATISTICS test, although failure would be preferable.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LoadModule function (winbase.h)

Article10/13/2021

Loads and executes an application or creates a new instance of an existing application.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should use the [CreateProcess](#) function.

Syntax

C++

```
DWORD LoadModule(
    [in] LPCSTR lpModuleName,
    [in] LPVOID lpParameterBlock
);
```

Parameters

[in] lpModuleName

The file name of the application to be run. When specifying a path, be sure to use backslashes (\), not forward slashes (/). If the *lpModuleName* parameter does not contain a directory path, the system searches for the executable file in this order:

1. The directory from which the application loaded.
2. The current directory.
3. The system directory. Use the [GetSystemDirectory](#) function to get the path of this directory.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

[in] lpParameterBlock

A pointer to an application-defined **LOADPARMS32** structure that defines the new application's parameter block.

Set all unused members to NULL, except for **lpCmdLine**, which must point to a null-terminated string if it is not used. For more information, see Remarks.

Return value

If the function succeeds, the return value is greater than 31.

If the function fails, the return value is an error value, which may be one of the following values.

Return code/value	Description
0	The system is out of memory or resources.
ERROR_BAD_FORMAT 11L	The .exe file is invalid.
ERROR_FILE_NOT_FOUND 2L	The specified file was not found.
ERROR_PATH_NOT_FOUND 3L	The specified path was not found.

Remarks

The **LOADPARMS32** structure has the following form:

syntax

```
typedef struct tagLOADPARMS32 {  
    LPSTR lpEnvAddress; // address of environment strings  
    LPSTR lpCmdLine; // address of command line  
    LPSTR lpCmdShow; // how to show new program  
    DWORD dwReserved; // must be zero  
} LOADPARMS32;
```

Member	Meaning
lpEnvAddress	Pointer to an array of null-terminated strings that supply the environment strings for the new process. The array has a value of NULL as its last entry. A value of NULL for this parameter causes the new process to start with the same environment as the calling process.
lpCmdLine	Pointer to a Pascal-style string that contains a correctly formed command line. The first byte of the string contains the number of bytes in the string. The remainder of the string contains the command line arguments, excluding the

name of the child process. If there are no command line arguments, this parameter must point to a zero length string; it cannot be NULL.

lpCmdShow	Pointer to a structure containing two WORD values. The first value must always be set to two. The second value specifies how the application window is to be shown and is used to supply the wShowWindow member of the STARTUPINFO structure to the CreateProcess function. See the description of the nCmdShow parameter of the ShowWindow function for a list of acceptable values.
dwReserved	This parameter is reserved; it must be zero.

Applications should use the [CreateProcess](#) function instead of [LoadModule](#). The [LoadModule](#) function calls [CreateProcess](#) by forming the parameters as follows.

CreateProcess parameter	Argument used
<i>lpszApplicationName</i>	<i>lpModuleName</i>
<i>lpszCommandLine</i>	<i>lpParameterBlock.lpCmdLine</i>
<i>lpProcessAttributes</i>	NULL
<i>lpThreadAttributes</i>	NULL
<i>bInheritHandles</i>	FALSE
<i>dwCreationFlags</i>	0
<i>lpEnvironment</i>	<i>lpParameterBlock.lpEnvAddress</i>
<i>lpCurrentDirectory</i>	NULL
<i>lpStartupInfo</i>	The structure is initialized to zero. The cb member is set to the size of the structure. The wShowWindow member is set to the value of the second word of <i>lpParameterBlock.lpCmdShow</i> .
<i>lpProcessInformation.hProcess</i>	The handle is immediately closed.
<i>lpProcessInformation.hThread</i>	The handle is immediately closed.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateProcess](#)

[Dynamic-Link Library Functions](#)

[GetSystemDirectory](#)

[GetWindowsDirectory](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LoadPackagedLibrary function (winbase.h)

Article03/11/2023

Loads the specified packaged module and its dependencies into the address space of the calling process.

Syntax

C++

```
HMODULE LoadPackagedLibrary(
    [in] LPCWSTR lpwLibFileName,
    DWORD     Reserved
);
```

Parameters

[in] `lpwLibFileName`

The file name of the packaged module to load. The module can be a library module (a .dll file) or an executable module (an .exe file).

If this parameter specifies a module name without a path and the file name extension is omitted, the function appends the default library extension .dll to the module name. To prevent the function from appending .dll to the module name, include a trailing point character (.) in the module name string.

If this parameter specifies a path, the function searches that path for the module. The path cannot be an absolute path or a relative path that contains ".." in the path. When specifying a path, be sure to use backslashes (\), not forward slashes (/). For more information about paths, see [Naming Files, Paths, and Namespaces](#).

If the specified module is already loaded in the process, the function returns a handle to the loaded module. The module must have been originally loaded from the package dependency graph of the process.

If loading the specified module causes the system to load other associated modules, the function first searches loaded modules, then it searches the package dependency graph of the process. For more information, see Remarks.

Reserved

This parameter is reserved. It must be 0.

Return value

If the function succeeds, the return value is a handle to the loaded module.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The **LoadPackagedLibrary** function is a simplified version of [LoadLibraryEx](#). Packaged apps can use **LoadPackagedLibrary** to load packaged modules. Unpackaged applications cannot use **LoadPackagedLibrary**; if an unpackaged application calls this function it fails with **APPMODEL_ERROR_NO_PACKAGE**.

LoadPackagedLibrary returns a handle to the specified module and increments its reference count. If the module is already loaded, the function returns a handle to the loaded module. The calling process can use the handle returned by **LoadPackagedLibrary** to identify the module in calls to the [GetProcAddress](#) function. Use the [FreeLibrary](#) function to free a loaded module and decrement its reference count.

If the function must search for the specified module or its dependencies, it searches only the package dependency graph of the process. This is the application's package plus any dependencies specified as `<PackageDependency>` in the `<Dependencies>` section of the application's package manifest. Dependencies are searched in the order they appear in the manifest. The package dependency graph is specified in the `<Dependencies>` section of the application's package manifest. Dependencies are searched in the order they appear in the manifest. The search proceeds as follows:

1. The function first searches modules that are already loaded. If the specified module was originally loaded from the package dependency graph of the process, the function returns a handle to the loaded module. If the specified module was not loaded from the package dependency graph of the process, the function returns **NULL**.
2. If the module is not already loaded, the function searches the package dependency graph of the process.
3. If the function cannot find the specified module or one of its dependencies, the function fails.

It is not safe to call `LoadPackagedLibrary` from `DllMain`. For more information, see the Remarks section in `DllMain`.

 **Note**

On Windows Phone, `LoadPackagedLibrary` must be called from `PhoneAppModelHost.dll`. Using `Kernel32.dll` is not supported.

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Kernel32.lib</code>
DLL	<code>Kernel32.dll</code>

See also

[Dynamic-Link Library Search Order](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalAlloc function (winbase.h)

Article07/27/2022

Allocates the specified number of bytes from the heap.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
DECLSPEC_ALLOCATOR HLOCAL LocalAlloc(
    [in] UINT    uFlags,
    [in] SIZE_T  uBytes
);
```

Parameters

[in] `uFlags`

The memory allocation attributes. The default is the **LMEM_FIXED** value. This parameter can be one or more of the following values, except for the incompatible combinations that are specifically noted.

Value	Meaning
LHND 0x0042	Combines LMEM_MOVEABLE and LMEM_ZEROINIT .
LMEM_FIXED 0x0000	Allocates fixed memory. The return value is a pointer to the memory object.
LMEM_MOVEABLE 0x0002	Allocates movable memory. Memory blocks are never moved in physical memory, but they can be moved within the default heap. The return value is a handle to the memory object. To translate the handle to a pointer, use the LocalLock function.

	This value cannot be combined with LMEM_FIXED .
LMEM_ZEROINIT 0x0040	Initializes memory contents to zero.
L PTR 0x0040	Combines LMEM_FIXED and LMEM_ZEROINIT .
NONZEROLHND	Same as LMEM_MOVEABLE .
NONZEROLOPTR	Same as LMEM_FIXED .

The following values are obsolete, but are provided for compatibility with 16-bit Windows. They are ignored.

LMEM_DISCARDABLE

LMEM_NOCOMPACT

LMEM_NODISCARD

[in] uBytes

The number of bytes to allocate. If this parameter is zero and the *uFlags* parameter specifies **LMEM_MOVEABLE**, the function returns a handle to a memory object that is marked as discarded.

Return value

If the function succeeds, the return value is a handle to the newly allocated memory object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Windows memory management does not provide a separate local heap and global heap. Therefore, the **LocalAlloc** and [GlobalAlloc](#) functions are essentially the same.

The movable-memory flags **LHND**, **LMEM_MOVEABLE**, and **NONZEROLHND** add unnecessary overhead and require locking to be used safely. They should be avoided unless documentation specifically states that they should be used.

New applications should use the [heap functions](#) unless the documentation specifically states that a local function should be used. For example, some Windows functions

allocate memory that must be freed with [LocalFree](#).

If the heap does not contain sufficient free space to satisfy the request, [LocalAlloc](#) returns **NULL**. Because **NULL** is used to indicate an error, virtual address zero is never allocated. It is, therefore, easy to detect the use of a **NULL** pointer.

If the [LocalAlloc](#) function succeeds, it allocates at least the amount requested. If the amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual number of bytes allocated, use the [LocalSize](#) function.

To free the memory, use the [LocalFree](#) function. It is not safe to free memory allocated with [LocalAlloc](#) using [GlobalFree](#).

Examples

The following code shows a simple use of [LocalAlloc](#) and [LocalFree](#).

C++

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _cdecl _tmain()
{
    LPTSTR pszBuf=NULL;

    pszBuf = (LPTSTR)LocalAlloc(
        LPTR,
        MAX_PATH*sizeof(TCHAR));

    // Handle error condition
    if( pszBuf == NULL )
    {
        _tprintf(TEXT("LocalAlloc failed (%d)\n"), GetLastError());
        return;
    }

    //see how much memory was allocated
    _tprintf(TEXT("LocalAlloc allocated %d bytes\n"), LocalSize(pszBuf));

    // Use the memory allocated

    // Free the memory when finished with it
    LocalFree(pszBuf);
}
```

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[Heap Functions](#)

[LocalFree](#)

[LocalLock](#)

[LocalReAlloc](#)

[LocalSize](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalFlags function (winbase.h)

Article10/13/2021

Retrieves information about the specified local memory object.

Note This function is provided only for compatibility with 16-bit versions of Windows. New applications should use the [heap functions](#). For more information, see Remarks.

Syntax

C++

```
UINT LocalFlags(  
    [in] HLOCAL hMem  
)
```

Parameters

[in] hMem

A handle to the local memory object. This handle is returned by either the [LocalAlloc](#) or [LocalReAlloc](#) function.

Return value

If the function succeeds, the return value specifies the allocation values and the lock count for the memory object.

If the function fails, the return value is [LMEM_INVALID_HANDLE](#), indicating that the local handle is not valid. To get extended error information, call [GetLastError](#).

Remarks

The low-order byte of the low-order word of the return value contains the lock count of the object. To retrieve the lock count from the return value, use the [LMEM_LOCKCOUNT](#)

mask with the bitwise AND (&) operator. The lock count of memory objects allocated with **LMEM_FIXED** is always zero.

The high-order byte of the low-order word of the return value indicates the allocation values of the memory object. It can be zero or **LMEM_DISCARDABLE**.

The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the [heap functions](#) unless documentation states that a local function should be used. For more information, see [Global and Local Functions](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalFlags](#)

[LocalAlloc](#)

[LocalDiscard](#)

[LocalLock](#)

[LocalReAlloc](#)

[LocalUnlock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalFree function (winbase.h)

Article10/13/2021

Frees the specified local memory object and invalidates its handle.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
HLOCAL LocalFree(  
    [in] _Frees_ptr_opt_ HLOCAL hMem  
)
```

Parameters

[in] hMem

A handle to the local memory object. This handle is returned by either the [LocalAlloc](#) or [LocalReAlloc](#) function. It is not safe to free memory allocated with [GlobalAlloc](#).

Return value

If the function succeeds, the return value is **NULL**.

If the function fails, the return value is equal to a handle to the local memory object. To get extended error information, call [GetLastError](#).

Remarks

If the process tries to examine or modify the memory after it has been freed, heap corruption may occur or an access violation exception (EXCEPTION_ACCESS_VIOLATION) may be generated.

If the *hMem* parameter is **NULL**, **LocalFree** ignores the parameter and returns **NULL**.

The **LocalFree** function will free a locked memory object. A locked memory object has a lock count greater than zero. The **LocalLock** function locks a local memory object and increments the lock count by one. The **LocalUnlock** function unlocks it and decrements the lock count by one. To get the lock count of a local memory object, use the **LocalFlags** function.

If an application is running under a debug version of the system, **LocalFree** will issue a message that tells you that a locked object is being freed. If you are debugging the application, **LocalFree** will enter a breakpoint just before freeing a locked object. This allows you to verify the intended behavior, then continue execution.

Examples

For an example, see [LocalAlloc](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[GlobalFree](#)

[LocalAlloc](#)

[LocalFlags](#)

[LocalLock](#)

[LocalReAlloc](#)

[LocalUnlock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalHandle function (winbase.h)

Article10/13/2021

Retrieves the handle associated with the specified pointer to a local memory object.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
HLOCAL LocalHandle(  
    [in] LPCVOID pMem  
)
```

Parameters

[in] pMem

A pointer to the first byte of the local memory object. This pointer is returned by the [LocalLock](#) function.

Return value

If the function succeeds, the return value is a handle to the specified local memory object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

When the [LocalAlloc](#) function allocates a local memory object with **LMEM_MOVEABLE**, it returns a handle to the object. The [LocalLock](#) function converts this handle into a

pointer to the object's memory block, and **LocalHandle** converts the pointer back into a handle.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[LocalAlloc](#)

[LocalLock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalLock function (winbase.h)

Article10/13/2021

Locks a local memory object and returns a pointer to the first byte of the object's memory block.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
LPVOID LocalLock(  
    [in] HLOCAL hMem  
);
```

Parameters

[in] hMem

A handle to the local memory object. This handle is returned by either the [LocalAlloc](#) or [LocalReAlloc](#) function.

Return value

If the function succeeds, the return value is a pointer to the first byte of the memory block.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, **LocalLock** increments the count by one, and the **LocalUnlock** function decrements the count by one. Each successful call that a process makes to **LocalLock** for an object must be matched by a corresponding call to **LocalUnlock**. Locked memory will not be moved or discarded unless the memory object is reallocated by using the **LocalReAlloc** function. The memory block of a locked memory object remains locked in memory until its lock count is decremented to zero, at which time it can be moved or discarded.

Memory objects allocated with **LMEM_FIXED** always have a lock count of zero. For these objects, the value of the returned pointer is equal to the value of the specified handle.

If the specified memory block has been discarded or if the memory block has a zero-byte size, this function returns **NULL**.

Discarded objects always have a lock count of zero.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[LocalAlloc](#)

[LocalFlags](#)

[LocalReAlloc](#)

[LocalUnlock](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalReAlloc function (winbase.h)

Article10/13/2021

Changes the size or the attributes of a specified local memory object. The size can increase or decrease.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
DECLSPEC_ALLOCATOR HLOCAL LocalReAlloc(
    [in] _Frees_ptr_opt_ HLOCAL hMem,
    [in] SIZE_T             uBytes,
    [in] UINT               uFlags
);
```

Parameters

[in] hMem

A handle to the local memory object to be reallocated. This handle is returned by either the [LocalAlloc](#) or [LocalReAlloc](#) function.

[in] uBytes

The new size of the memory block, in bytes. If *uFlags* specifies **LMEM MODIFY**, this parameter is ignored.

[in] uFlags

The reallocation options. If **LMEM MODIFY** is specified, the function modifies the attributes of the memory object only (the *uBytes* parameter is ignored.) Otherwise, the function reallocates the memory object.

You can optionally combine **LMEM MODIFY** with the following value.

Value	Meaning
LMEM_MOVEABLE 0x0002	Allocates fixed or movable memory. If the memory is a locked LMEM_MOVEABLE memory block or a LMEM_FIXED memory block and this flag is not specified, the memory can only be reallocated in place.

If this parameter does not specify LMEM_MODIFY, you can use the following value.

Value	Meaning
LMEM_ZEROINIT 0x0040	Causes the additional memory contents to be initialized to zero if the memory object is growing in size.

Return value

If the function succeeds, the return value is a handle to the reallocated memory object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

If **LocalReAlloc** fails, the original memory is not freed, and the original handle and pointer are still valid.

If **LocalReAlloc** reallocates a fixed object, the value of the handle returned is the address of the first byte of the memory block. To access the memory, a process can simply cast the return value to a pointer.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[LocalAlloc](#)

[LocalFree](#)

[LocalLock](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalSize function (winbase.h)

Article10/13/2021

Retrieves the current size of the specified local memory object, in bytes.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see [Global and Local Functions](#).

Syntax

C++

```
SIZE_T LocalSize(  
    [in] HLOCAL hMem  
)
```

Parameters

[in] hMem

A handle to the local memory object. This handle is returned by the [LocalAlloc](#), [LocalReAlloc](#), or [LocalHandle](#) function.

Return value

If the function succeeds, the return value is the size of the specified local memory object, in bytes. If the specified handle is not valid or if the object has been discarded, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The size of a memory block may be larger than the size requested when the memory was allocated.

To verify that the specified object's memory block has not been discarded, call the [LocalFlags](#) function before calling [LocalSize](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[LocalAlloc](#)

[LocalFlags](#)

[LocalHandle](#)

[LocalReAlloc](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocalUnlock function (winbase.h)

Article10/13/2021

Decrements the lock count associated with a memory object that was allocated with **LMEM_MOVEABLE**. This function has no effect on memory objects allocated with **LMEM_FIXED**.

Note The local functions have greater overhead and provide fewer features than other memory management functions. New applications should use the **heap functions** unless documentation states that a local function should be used. For more information, see **Global and Local Functions**.

Syntax

C++

```
BOOL LocalUnlock(  
    [in] HLOCAL hMem  
);
```

Parameters

[in] hMem

A handle to the local memory object. This handle is returned by either the [LocalAlloc](#) or [LocalReAlloc](#) function.

Return value

If the memory object is still locked after decrementing the lock count, the return value is nonzero. If the memory object is unlocked after decrementing the lock count, the function returns zero and [GetLastError](#) returns **NO_ERROR**.

If the function fails, the return value is zero and [GetLastError](#) returns a value other than **NO_ERROR**.

Remarks

The internal data structures for each memory object include a lock count that is initially zero. For movable memory objects, the [LocalLock](#) function increments the count by one, and [LocalUnlock](#) decrements the count by one. For each call that a process makes to [LocalLock](#) for an object, it must eventually call [LocalUnlock](#). Locked memory will not be moved or discarded unless the memory object is reallocated by using the [LocalReAlloc](#) function. The memory block of a locked memory object remains locked until its lock count is decremented to zero, at which time it can be moved or discarded.

If the memory object is already unlocked, [LocalUnlock](#) returns **FALSE** and [GetLastError](#) reports **ERROR_NOT_LOCKED**. Memory objects allocated with **LMEM_FIXED** always have a lock count of zero and cause the **ERROR_NOT_LOCKED** error.

A process should not rely on the return value to determine the number of times it must subsequently call [LocalUnlock](#) for the memory block.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Global and Local Functions](#)

[LocalAlloc](#)

[LocalFlags](#)

[LocalLock](#)

[LocalReAlloc](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LocateXStateFeature function (winbase.h)

Article03/03/2023

Retrieves a pointer to the processor state for an XState feature within a [CONTEXT](#) structure.

The definition of XState feature bits are processor vendor specific. Please refer to the relevant processor reference manuals for additional information on a particular feature.

Syntax

C++

```
PVOID LocateXStateFeature(
    [in]          PCONTEXT Context,
    [in]          DWORD    FeatureId,
    [out, optional] PDWORD   Length
);
```

Parameters

[in] Context

A pointer to a [CONTEXT](#) structure containing the state to retrieve or set. This [CONTEXT](#) should have been initialized with [InitializeContext](#) with the [CONTEXT_XSTATE](#) flag set in the *ContextFlags* parameter.

[in] FeatureId

The number of the feature to locate within the [CONTEXT](#) structure.

[out, optional] Length

A pointer to a variable which receives the length of the feature area in bytes. The contents of this variable are undefined if this function returns **NULL**.

Return value

If the specified feature is supported by the system and the specified [CONTEXT](#) structure has been initialized with the [CONTEXT_XSTATE](#) flag, this function returns a pointer to the feature area for the specified feature. The contents and layout of this area is processor-specific.

If the [CONTEXT_XSTATE](#) flag is not set in the [CONTEXT](#) structure or the *FeatureID* is not supported by the system, the return value is [NULL](#). No additional error information is available.

Remarks

The [LocateXStateFeature](#) function must be used to find an individual XState feature within an extensible [CONTEXT](#) structure. Features are not necessarily contiguous in memory and applications should not assume the offset between two consecutive features will remain constant in the future.

The *FeatureID* parameter of the function corresponds to a bit within the feature mask. For example, *FeatureId* 2 corresponds to a *FeatureMask* of 4 in [SetXStateFeaturesMask](#). *FeatureID* values of 0 and 1 correspond to X87 FPU state and SSE state, respectively.

If you are setting XState on a thread via the [SetThreadContext](#) or [Wow64SetThreadContext](#) APIs, you must also call [SetXStateFeaturesMask](#) on the [CONTEXT](#) structure with the mask value of the filled-in feature to mark the feature as active.

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The [AVX API](#) is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1. Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this documentation and get pointers to them using [GetModuleHandle](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CONTEXT](#)

[Intel AVX](#)

[SetThreadContext](#)

[SetXStateFeaturesMask](#)

[Working with XState Context](#)

[Wow64SetThreadContext](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LogonUserA function (winbase.h)

Article03/11/2023

The **LogonUser** function attempts to log a user on to the local computer. The local computer is the computer from which **LogonUser** was called. You cannot use **LogonUser** to log on to a remote computer. You specify the user with a user name and domain and **authenticate** the user with a **plaintext** password. If the function succeeds, you receive a handle to a token that represents the logged-on user. You can then use this token handle to impersonate the specified user or, in most cases, to create a **process** that runs in the context of the specified user.

Syntax

C++

```
BOOL LogonUserA(
    [in]          LPCSTR lpszUsername,
    [in, optional] LPCSTR lpszDomain,
    [in, optional] LPCSTR lpszPassword,
    [in]          DWORD   dwLogonType,
    [in]          DWORD   dwLogonProvider,
    [out]         PHANDLE phToken
);
```

Parameters

[in] *lpszUsername*

A pointer to a null-terminated string that specifies the name of the user. This is the name of the user account to log on to. If you use the **user principal name** (UPN) format, *User@DNSDomainName*, the *lpszDomain* parameter must be **NULL**.

[in, optional] *lpszDomain*

A pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *lpszUsername* account. If this parameter is **NULL**, the user name must be specified in UPN format. If this parameter is ".", the function validates the account by using only the local account database.

[in, optional] *lpszPassword*

A pointer to a null-terminated string that specifies the plaintext password for the user account specified by *lpszUsername*. When you have finished using the password, clear the password from memory by calling the [SecureZeroMemory](#) function. For more information about protecting passwords, see [Handling Passwords](#).

[in] dwLogonType

The type of logon operation to perform. This parameter can be one of the following values, defined in Winbase.h.

Value	Meaning
LOGON32_LOGON_BATCH	This logon type is intended for batch servers, where processes may be executing on behalf of a user without their direct intervention. This type is also for higher performance servers that process many plaintext authentication attempts at a time, such as mail or web servers.
LOGON32_LOGON_INTERACTIVE	This logon type is intended for users who will be interactively using the computer, such as a user being logged on by a terminal server, remote shell, or similar process. This logon type has the additional expense of caching logon information for disconnected operations; therefore, it is inappropriate for some client/server applications, such as a mail server.
LOGON32_LOGON_NETWORK	This logon type is intended for high performance servers to authenticate plaintext passwords. The LogonUser function does not cache credentials for this logon type.
LOGON32_LOGON_NETWORK_CLEARTEXT	This logon type preserves the name and password in the authentication package , which allows the server to make connections to other network servers while impersonating the client. A server can accept plaintext credentials from a client, call LogonUser , verify that the user can access the system across the network, and still communicate with other servers.
LOGON32_LOGON_NEW_CREDENTIALS	This logon type allows the caller to clone its current token and specify new credentials for outbound connections. The new logon session has the same local identifier but uses different credentials for other network connections. This logon type is supported only by the LOGON32_PROVIDER_WINNT50 logon provider.

Note: As of January 2023, it is not possible to use the LOGON32_LOGON_NEW_CREDENTIALS logon type with a Group Managed Service Account (gMSA).

LOGON32_LOGON_SERVICE	Indicates a service-type logon. The account provided must have the service privilege enabled.
LOGON32_LOGON_UNLOCK	GINAs are no longer supported. Windows Server 2003 and Windows XP: This logon type is for GINA DLLs that log on users who will be interactively using the computer. This logon type can generate a unique audit record that shows when the workstation was unlocked.

[in] dwLogonProvider

Specifies the logon provider. This parameter can be one of the following values.

Value	Meaning
LOGON32_PROVIDER_DEFAULT	Use the standard logon provider for the system. The default security provider is negotiate, unless you pass NULL for the domain name and the user name is not in UPN format. In this case, the default provider is NTLM.
LOGON32_PROVIDER_WINNT50	Use the negotiate logon provider.
LOGON32_PROVIDER_WINNT40	Use the NTLM logon provider.

[out] phToken

A pointer to a handle variable that receives a handle to a token that represents the specified user.

You can use the returned handle in calls to the [ImpersonateLoggedOnUser](#) function.

In most cases, the returned handle is a [primary token](#) that you can use in calls to the [CreateProcessAsUser](#) function. However, if you specify the LOGON32_LOGON_NETWORK flag, **LogonUser** returns an [impersonation token](#) that you cannot use in [CreateProcessAsUser](#) unless you call [DuplicateTokenEx](#) to convert it to a primary token.

When you no longer need this handle, close it by calling the [CloseHandle](#) function.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The LOGON32_LOGON_NETWORK logon type is fastest, but it has the following limitations:

- The function returns an [impersonation token](#), not a primary token. You cannot use this token directly in the [CreateProcessAsUser](#) function. However, you can call the [DuplicateTokenEx](#) function to convert the token to a primary token, and then use it in [CreateProcessAsUser](#).
- If you convert the token to a primary token and use it in [CreateProcessAsUser](#) to start a process, the new process cannot access other network resources, such as remote servers or printers, through the redirector. An exception is that if the network resource is not access controlled, then the new process will be able to access it.

The SE_TCB_NAME privilege is not required for this function unless you are logging onto a Passport account.

The account specified by *lpszUsername*, must have the necessary account rights. For example, to log on a user with the LOGON32_LOGON_INTERACTIVE flag, the user (or a group to which the user belongs) must have the SE_INTERACTIVE_LOGON_NAME account right. For a list of the account rights that affect the various logon operations, see [Account Rights Constants](#).

A user is considered logged on if at least one token exists. If you call [CreateProcessAsUser](#) and then close the token, the system considers the user as still logged on until the process (and all child processes) have ended.

If the [LogonUser](#) call is successful, the system notifies network providers that the logon occurred by calling the provider's [NPLogonNotify](#) entry-point function.

Examples

You can generate a LocalService token by using the following code.

C++

```
LogonUser(L"LocalService", L"NT AUTHORITY", NULL, LOGON32_LOGON_SERVICE,  
LOGON32_PROVIDER_DEFAULT, &hToken)
```

ⓘ Note

The winbase.h header defines LogonUser as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[CloseHandle](#)

[CreateProcessAsUser](#)

[DuplicateTokenEx](#)

[ImpersonateLoggedOnUser](#)

Feedback

Was this page helpful?

Get help at Microsoft Q&A

LogonUserExA function (winbase.h)

Article02/09/2023

The **LogonUserEx** function attempts to log a user on to the local computer. The local computer is the computer from which **LogonUserEx** was called. You cannot use **LogonUserEx** to log on to a remote computer. You specify the user with a user name and domain and [authenticate](#) the user with a plaintext password. If the function succeeds, you receive a handle to a token that represents the logged-on user. You can then use this token handle to impersonate the specified user or, in most cases, to create a [process](#) that runs in the context of the specified user.

Syntax

C++

```
BOOL LogonUserExA(
    [in]          LPCSTR      lpszUsername,
    [in, optional] LPCSTR      lpszDomain,
    [in, optional] LPCSTR      lpszPassword,
    [in]          DWORD       dwLogonType,
    [in]          DWORD       dwLogonProvider,
    [out, optional] PHANDLE    phToken,
    [out, optional] PSID        *ppLogonSid,
    [out, optional] PVOID       *ppProfileBuffer,
    [out, optional] LPDWORD     pdwProfileLength,
    [out, optional] PQUOTA_LIMITS pQuotaLimits
);
```

Parameters

[in] *lpszUsername*

A pointer to a null-terminated string that specifies the name of the user. This is the name of the user account to log on to. If you use the [user principal name](#) (UPN) format, *user@DNS_domain_name*, the *lpszDomain* parameter must be **NULL**.

[in, optional] *lpszDomain*

A pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *lpszUsername* account. If this parameter is **NULL**, the user name must be specified in UPN format. If this parameter is ".", the function validates the account by using only the local account database.

[in, optional] *lpszPassword*

A pointer to a null-terminated string that specifies the plaintext password for the user account specified by *lpszUsername*. When you have finished using the password, clear the password from memory by calling the [SecureZeroMemory](#) function. For more information about protecting passwords, see [Handling Passwords](#).

[in] *dwLogonType*

The type of logon operation to perform. This parameter can be one of the following values.

Value	Meaning
LOGON32_LOGON_BATCH	This logon type is intended for batch servers, where processes may be executing on behalf of a user without their direct intervention. This type is also for higher performance servers that process many plaintext authentication attempts at a time, such as mail or web servers. The LogonUserEx function does not cache credentials for this logon type.
LOGON32_LOGON_INTERACTIVE	This logon type is intended for users who will be interactively using the computer, such as a user being logged on by a terminal server, remote shell, or similar process. This logon type has the additional expense of caching logon information for disconnected operations; therefore, it is inappropriate for some client/server applications, such as a mail server.
LOGON32_LOGON_NETWORK	This logon type is intended for high performance servers to authenticate plaintext passwords. The LogonUserEx function does not cache credentials for this logon type.
LOGON32_LOGON_NETWORK_CLEARTEXT	This logon type preserves the name and password in the authentication package , which allows the server to make connections to other network servers while impersonating the client. A server can accept plaintext credentials from a client, call LogonUserEx , verify that the user can access the system across the network, and still communicate with other servers.
LOGON32_LOGON_NEW_CREDENTIALS	This logon type allows the caller to clone its current token and specify new credentials for outbound connections. The new logon session has the same

	<p>local identifier but uses different credentials for other network connections.</p> <p>This logon type is supported only by the LOGON32_PROVIDER_WINNT50 logon provider.</p>
LOGON32_LOGON_SERVICE	Indicates a service-type logon. The account provided must have the service privilege enabled.
LOGON32_LOGON_UNLOCK	This logon type is for GINA DLLs that log on users who will be interactively using the computer. This logon type can generate a unique audit record that shows when the workstation was unlocked.

[in] dwLogonProvider

The logon provider. This parameter can be one of the following values.

Value	Meaning
LOGON32_PROVIDER_DEFAULT	Use the standard logon provider for the system. The default security provider is NTLM.
LOGON32_PROVIDER_WINNT50	Use the negotiate logon provider.
LOGON32_PROVIDER_WINNT40	Use the NTLM logon provider.

[out, optional] phToken

A pointer to a handle variable that receives a handle to a token that represents the specified user.

You can use the returned handle in calls to the [ImpersonateLoggedOnUser](#) function.

In most cases, the returned handle is a [primary token](#) that you can use in calls to the [CreateProcessAsUser](#) function. However, if you specify the LOGON32_LOGON_NETWORK flag, [LogonUserEx](#) returns an [impersonation token](#) that you cannot use in [CreateProcessAsUser](#) unless you call [DuplicateTokenEx](#) to convert the impersonation token to a primary token.

When you no longer need this handle, close it by calling the [CloseHandle](#) function.

[out, optional] ppLogonSid

A pointer to a pointer to a [security identifier](#) (SID) that receives the SID of the user logged on.

When you have finished using the SID, free it by calling the [LocalFree](#) function.

[out, optional] `ppProfileBuffer`

A pointer to a pointer that receives the address of a buffer that contains the logged on user's profile.

[out, optional] `pdwProfileLength`

A pointer to a **DWORD** that receives the length of the profile buffer.

[out, optional] `pQuotaLimits`

A pointer to a [QUOTA_LIMITS](#) structure that receives information about the quotas for the logged on user.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The LOGON32_LOGON_NETWORK logon type is fastest, but it has the following limitations:

- The function returns an [impersonation token](#), not a primary token. You cannot use this token directly in the [CreateProcessAsUser](#) function. However, you can call the [DuplicateTokenEx](#) function to convert the token to a primary token, and then use it in [CreateProcessAsUser](#).
- If you convert the token to a primary token and use it in [CreateProcessAsUser](#) to start a process, the new process cannot access other network resources, such as remote servers or printers, through the redirector. An exception is that if the network resource is not access controlled, then the new process will be able to access it.

The SE_TCB_NAME privilege is not required for this function unless you are logging onto a Passport account.

The account specified by *lpszUsername* must have the necessary account rights. For example, to log on a user with the LOGON32_LOGON_INTERACTIVE flag, the user (or a group to which the user belongs) must have the SE_INTERACTIVE_LOGON_NAME account right. For a list of the account rights that affect the various logon operations, see [Account Object Access Rights](#).

A user is considered logged on if at least one token exists. If you call [CreateProcessAsUser](#) and then close the token, the user is still logged on until the process (and all child processes) have ended.

If the [LogonUserEx](#) call is successful, the system notifies network providers that the logon occurred by calling the provider's [NPLogonNotify](#) entry-point function.

 **Note**

The winbase.h header defines LogonUserEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[CloseHandle](#)

[CreateProcessAsUser](#)

[DuplicateTokenEx](#)

[ImpersonateLoggedOnUser](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LogonUserExW function (winbase.h)

Article02/09/2023

The **LogonUserEx** function attempts to log a user on to the local computer. The local computer is the computer from which **LogonUserEx** was called. You cannot use **LogonUserEx** to log on to a remote computer. You specify the user with a user name and domain and [authenticate](#) the user with a plaintext password. If the function succeeds, you receive a handle to a token that represents the logged-on user. You can then use this token handle to impersonate the specified user or, in most cases, to create a [process](#) that runs in the context of the specified user.

Syntax

C++

```
BOOL LogonUserExW(
    [in]           LPCWSTR      lpszUsername,
    [in, optional] LPCWSTR      lpszDomain,
    [in, optional] LPCWSTR      lpszPassword,
    [in]           DWORD        dwLogonType,
    [in]           DWORD        dwLogonProvider,
    [out, optional] PHANDLE     phToken,
    [out, optional] PSID        *ppLogonSid,
    [out, optional] PVOID       *ppProfileBuffer,
    [out, optional] LPDWORD     pdwProfileLength,
    [out, optional] PQUOTA_LIMITS pQuotaLimits
);
```

Parameters

[in] *lpszUsername*

A pointer to a null-terminated string that specifies the name of the user. This is the name of the user account to log on to. If you use the [user principal name](#) (UPN) format, *user@DNS_domain_name*, the *lpszDomain* parameter must be **NULL**.

[in, optional] *lpszDomain*

A pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *lpszUsername* account. If this parameter is **NULL**, the user name must be specified in UPN format. If this parameter is ".", the function validates the account by using only the local account database.

[in, optional] *lpszPassword*

A pointer to a null-terminated string that specifies the plaintext password for the user account specified by *lpszUsername*. When you have finished using the password, clear the password from memory by calling the [SecureZeroMemory](#) function. For more information about protecting passwords, see [Handling Passwords](#).

[in] *dwLogonType*

The type of logon operation to perform. This parameter can be one of the following values.

Value	Meaning
LOGON32_LOGON_BATCH	This logon type is intended for batch servers, where processes may be executing on behalf of a user without their direct intervention. This type is also for higher performance servers that process many plaintext authentication attempts at a time, such as mail or web servers. The LogonUserEx function does not cache credentials for this logon type.
LOGON32_LOGON_INTERACTIVE	This logon type is intended for users who will be interactively using the computer, such as a user being logged on by a terminal server, remote shell, or similar process. This logon type has the additional expense of caching logon information for disconnected operations; therefore, it is inappropriate for some client/server applications, such as a mail server.
LOGON32_LOGON_NETWORK	This logon type is intended for high performance servers to authenticate plaintext passwords. The LogonUserEx function does not cache credentials for this logon type.
LOGON32_LOGON_NETWORK_CLEARTEXT	This logon type preserves the name and password in the authentication package , which allows the server to make connections to other network servers while impersonating the client. A server can accept plaintext credentials from a client, call LogonUserEx , verify that the user can access the system across the network, and still communicate with other servers.
LOGON32_LOGON_NEW_CREDENTIALS	This logon type allows the caller to clone its current token and specify new credentials for outbound connections. The new logon session has the same

	<p>local identifier but uses different credentials for other network connections.</p> <p>This logon type is supported only by the LOGON32_PROVIDER_WINNT50 logon provider.</p>
LOGON32_LOGON_SERVICE	Indicates a service-type logon. The account provided must have the service privilege enabled.
LOGON32_LOGON_UNLOCK	This logon type is for GINA DLLs that log on users who will be interactively using the computer. This logon type can generate a unique audit record that shows when the workstation was unlocked.

[in] dwLogonProvider

The logon provider. This parameter can be one of the following values.

Value	Meaning
LOGON32_PROVIDER_DEFAULT	Use the standard logon provider for the system. The default security provider is NTLM.
LOGON32_PROVIDER_WINNT50	Use the negotiate logon provider.
LOGON32_PROVIDER_WINNT40	Use the NTLM logon provider.

[out, optional] phToken

A pointer to a handle variable that receives a handle to a token that represents the specified user.

You can use the returned handle in calls to the [ImpersonateLoggedOnUser](#) function.

In most cases, the returned handle is a [primary token](#) that you can use in calls to the [CreateProcessAsUser](#) function. However, if you specify the LOGON32_LOGON_NETWORK flag, [LogonUserEx](#) returns an [impersonation token](#) that you cannot use in [CreateProcessAsUser](#) unless you call [DuplicateTokenEx](#) to convert the impersonation token to a primary token.

When you no longer need this handle, close it by calling the [CloseHandle](#) function.

[out, optional] ppLogonSid

A pointer to a pointer to a [security identifier](#) (SID) that receives the SID of the user logged on.

When you have finished using the SID, free it by calling the [LocalFree](#) function.

[out, optional] ppProfileBuffer

A pointer to a pointer that receives the address of a buffer that contains the logged on user's profile.

[out, optional] pdwProfileLength

A pointer to a **DWORD** that receives the length of the profile buffer.

[out, optional] pQuotaLimits

A pointer to a [QUOTA_LIMITS](#) structure that receives information about the quotas for the logged on user.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The LOGON32_LOGON_NETWORK logon type is fastest, but it has the following limitations:

- The function returns an [impersonation token](#), not a primary token. You cannot use this token directly in the [CreateProcessAsUser](#) function. However, you can call the [DuplicateTokenEx](#) function to convert the token to a primary token, and then use it in [CreateProcessAsUser](#).
- If you convert the token to a primary token and use it in [CreateProcessAsUser](#) to start a process, the new process cannot access other network resources, such as remote servers or printers, through the redirector. An exception is that if the network resource is not access controlled, then the new process will be able to access it.

The SE_TCB_NAME privilege is not required for this function unless you are logging onto a Passport account.

The account specified by *lpszUsername* must have the necessary account rights. For example, to log on a user with the LOGON32_LOGON_INTERACTIVE flag, the user (or a group to which the user belongs) must have the SE_INTERACTIVE_LOGON_NAME account right. For a list of the account rights that affect the various logon operations, see [Account Object Access Rights](#).

A user is considered logged on if at least one token exists. If you call [CreateProcessAsUser](#) and then close the token, the user is still logged on until the process (and all child processes) have ended.

If the [LogonUserEx](#) call is successful, the system notifies network providers that the logon occurred by calling the provider's [NPLogonNotify](#) entry-point function.

 **Note**

The winbase.h header defines LogonUserEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[CloseHandle](#)

[CreateProcessAsUser](#)

[DuplicateTokenEx](#)

[ImpersonateLoggedOnUser](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LogonUserW function (winbase.h)

Article03/11/2023

The **LogonUser** function attempts to log a user on to the local computer. The local computer is the computer from which **LogonUser** was called. You cannot use **LogonUser** to log on to a remote computer. You specify the user with a user name and domain and **authenticate** the user with a **plaintext** password. If the function succeeds, you receive a handle to a token that represents the logged-on user. You can then use this token handle to impersonate the specified user or, in most cases, to create a **process** that runs in the context of the specified user.

Syntax

C++

```
BOOL LogonUserW(
    [in]          LPCWSTR lpszUsername,
    [in, optional] LPCWSTR lpszDomain,
    [in, optional] LPCWSTR lpszPassword,
    [in]          DWORD   dwLogonType,
    [in]          DWORD   dwLogonProvider,
    [out]         PHANDLE phToken
);
```

Parameters

[in] lpszUsername

A pointer to a null-terminated string that specifies the name of the user. This is the name of the user account to log on to. If you use the **user principal name** (UPN) format, *User@DNSDomainName*, the *lpszDomain* parameter must be **NULL**.

[in, optional] lpszDomain

A pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *lpszUsername* account. If this parameter is **NULL**, the user name must be specified in UPN format. If this parameter is ".", the function validates the account by using only the local account database.

[in, optional] lpszPassword

A pointer to a null-terminated string that specifies the plaintext password for the user account specified by *lpszUsername*. When you have finished using the password, clear the password from memory by calling the [SecureZeroMemory](#) function. For more information about protecting passwords, see [Handling Passwords](#).

[in] dwLogonType

The type of logon operation to perform. This parameter can be one of the following values, defined in Winbase.h.

Value	Meaning
LOGON32_LOGON_BATCH	This logon type is intended for batch servers, where processes may be executing on behalf of a user without their direct intervention. This type is also for higher performance servers that process many plaintext authentication attempts at a time, such as mail or web servers.
LOGON32_LOGON_INTERACTIVE	This logon type is intended for users who will be interactively using the computer, such as a user being logged on by a terminal server, remote shell, or similar process. This logon type has the additional expense of caching logon information for disconnected operations; therefore, it is inappropriate for some client/server applications, such as a mail server.
LOGON32_LOGON_NETWORK	This logon type is intended for high performance servers to authenticate plaintext passwords. The LogonUser function does not cache credentials for this logon type.
LOGON32_LOGON_NETWORK_CLEARTEXT	This logon type preserves the name and password in the authentication package , which allows the server to make connections to other network servers while impersonating the client. A server can accept plaintext credentials from a client, call LogonUser , verify that the user can access the system across the network, and still communicate with other servers.
LOGON32_LOGON_NEW_CREDENTIALS	This logon type allows the caller to clone its current token and specify new credentials for outbound connections. The new logon session has the same local identifier but uses different credentials for other network connections. This logon type is supported only by the LOGON32_PROVIDER_WINNT50 logon provider.

Note: As of January 2023, it is not possible to use the LOGON32_LOGON_NEW_CREDENTIALS logon type with a Group Managed Service Account (gMSA).

LOGON32_LOGON_SERVICE	Indicates a service-type logon. The account provided must have the service privilege enabled.
LOGON32_LOGON_UNLOCK	GINAs are no longer supported. Windows Server 2003 and Windows XP: This logon type is for GINA DLLs that log on users who will be interactively using the computer. This logon type can generate a unique audit record that shows when the workstation was unlocked.

[in] dwLogonProvider

Specifies the logon provider. This parameter can be one of the following values.

Value	Meaning
LOGON32_PROVIDER_DEFAULT	Use the standard logon provider for the system. The default security provider is negotiate, unless you pass NULL for the domain name and the user name is not in UPN format. In this case, the default provider is NTLM.
LOGON32_PROVIDER_WINNT50	Use the negotiate logon provider.
LOGON32_PROVIDER_WINNT40	Use the NTLM logon provider.

[out] phToken

A pointer to a handle variable that receives a handle to a token that represents the specified user.

You can use the returned handle in calls to the [ImpersonateLoggedOnUser](#) function.

In most cases, the returned handle is a [primary token](#) that you can use in calls to the [CreateProcessAsUser](#) function. However, if you specify the LOGON32_LOGON_NETWORK flag, **LogonUser** returns an [impersonation token](#) that you cannot use in [CreateProcessAsUser](#) unless you call [DuplicateTokenEx](#) to convert it to a primary token.

When you no longer need this handle, close it by calling the [CloseHandle](#) function.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The LOGON32_LOGON_NETWORK logon type is fastest, but it has the following limitations:

- The function returns an [impersonation token](#), not a primary token. You cannot use this token directly in the [CreateProcessAsUser](#) function. However, you can call the [DuplicateTokenEx](#) function to convert the token to a primary token, and then use it in [CreateProcessAsUser](#).
- If you convert the token to a primary token and use it in [CreateProcessAsUser](#) to start a process, the new process cannot access other network resources, such as remote servers or printers, through the redirector. An exception is that if the network resource is not access controlled, then the new process will be able to access it.

The SE_TCB_NAME privilege is not required for this function unless you are logging onto a Passport account.

The account specified by *lpszUsername*, must have the necessary account rights. For example, to log on a user with the LOGON32_LOGON_INTERACTIVE flag, the user (or a group to which the user belongs) must have the SE_INTERACTIVE_LOGON_NAME account right. For a list of the account rights that affect the various logon operations, see [Account Rights Constants](#).

A user is considered logged on if at least one token exists. If you call [CreateProcessAsUser](#) and then close the token, the system considers the user as still logged on until the process (and all child processes) have ended.

If the [LogonUser](#) call is successful, the system notifies network providers that the logon occurred by calling the provider's [NPLogonNotify](#) entry-point function.

Examples

You can generate a LocalService token by using the following code.

C++

```
LogonUser(L"LocalService", L"NT AUTHORITY", NULL, LOGON32_LOGON_SERVICE,  
LOGON32_PROVIDER_DEFAULT, &hToken)
```

ⓘ Note

The winbase.h header defines LogonUser as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[CloseHandle](#)

[CreateProcessAsUser](#)

[DuplicateTokenEx](#)

[ImpersonateLoggedOnUser](#)

Feedback



Was this page helpful?

Get help at Microsoft Q&A

LookupAccountNameA function (winbase.h)

Article02/09/2023

The **LookupAccountName** function accepts the name of a system and an account as input. It retrieves a [security identifier](#) (SID) for the account and the name of the domain on which the account was found.

The [LsaLookupNames](#) function can also retrieve computer accounts.

Syntax

C++

```
BOOL LookupAccountNameA(
    [in, optional] LPCSTR     lpSystemName,
    [in]          LPCSTR     lpAccountName,
    [out, optional] PSID      Sid,
    [in, out]       LPDWORD   cbSid,
    [out, optional] LPSTR      ReferencedDomainName,
    [in, out]       LPDWORD   cchReferencedDomainName,
    [out]          PSID_NAME_USE peUse
);
```

Parameters

[in, optional] *lpSystemName*

A pointer to a **null**-terminated character string that specifies the name of the system. This string can be the name of a remote computer. If this string is **NULL**, the account name translation begins on the local system. If the name cannot be resolved on the local system, this function will try to resolve the name using domain controllers trusted by the local system. Generally, specify a value for *lpSystemName* only when the account is in an untrusted domain and the name of a computer in that domain is known.

[in] *lpAccountName*

A pointer to a **null**-terminated string that specifies the account name.

Use a fully qualified string in the `domain_name\user_name` format to ensure that **LookupAccountName** finds the account in the desired domain.

[out, optional] `Sid`

A pointer to a buffer that receives the [SID](#) structure that corresponds to the account name pointed to by the *lpAccountName* parameter. If this parameter is **NULL**, *cbSid* must be zero.

[in, out] `cbSid`

A pointer to a variable. On input, this value specifies the size, in bytes, of the *Sid* buffer. If the function fails because the buffer is too small or if *cbSid* is zero, this variable receives the required buffer size.

[out, optional] `ReferencedDomainName`

A pointer to a buffer that receives the name of the domain where the account name is found. For computers that are not joined to a domain, this buffer receives the computer name. If this parameter is **NULL**, the function returns the required buffer size.

[in, out] `cchReferencedDomainName`

A pointer to a variable. On input, this value specifies the size, in **TCHARs**, of the *ReferencedDomainName* buffer. If the function fails because the buffer is too small, this variable receives the required buffer size, including the terminating **null** character. If the *ReferencedDomainName* parameter is **NULL**, this parameter must be zero.

[out] `peUse`

A pointer to a [SID_NAME_USE](#) enumerated type that indicates the type of the account when the function returns.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. For extended error information, call [GetLastError](#).

Remarks

The **LookupAccountName** function attempts to find a SID for the specified name by first checking a list of well-known SIDs. If the name does not correspond to a well-known SID, the function checks built-in and administratively defined local accounts. Next, the function checks the primary domain. If the name is not found there, trusted domains are checked.

Use fully qualified account names (for example, domain_name\user_name) instead of isolated names (for example, user_name). Fully qualified names are unambiguous and provide better performance when the lookup is performed. This function also supports fully qualified DNS names (for example, example.example.com\user_name) and [user principal names](#) (UPN) (for example, someone@example.com).

In addition to looking up local accounts, local domain accounts, and explicitly trusted domain accounts, **LookupAccountName** can look up the name for any account in any domain in the forest.

 **Note**

The winbase.h header defines **LookupAccountName** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[EqualPrefixSid](#)

[GetUserName](#)

[LookupAccountSid](#)

[LsaLookupNames2](#)

[SID](#)

[SID_NAME_USE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupAccountNameW function (winbase.h)

Article02/09/2023

The **LookupAccountName** function accepts the name of a system and an account as input. It retrieves a [security identifier](#) (SID) for the account and the name of the domain on which the account was found.

The [LsaLookupNames](#) function can also retrieve computer accounts.

Syntax

C++

```
BOOL LookupAccountName(
    [in, optional] LPCWSTR     lpSystemName,
    [in]           LPCWSTR     lpAccountName,
    [out, optional] PSID        Sid,
    [in, out]       LPDWORD     cbSid,
    [out, optional] LPWSTR      ReferencedDomainName,
    [in, out]       LPDWORD     cchReferencedDomainName,
    [out]          PSID_NAME_USE peUse
);
```

Parameters

[in, optional] *lpSystemName*

A pointer to a **null**-terminated character string that specifies the name of the system. This string can be the name of a remote computer. If this string is **NULL**, the account name translation begins on the local system. If the name cannot be resolved on the local system, this function will try to resolve the name using domain controllers trusted by the local system. Generally, specify a value for *lpSystemName* only when the account is in an untrusted domain and the name of a computer in that domain is known.

[in] *lpAccountName*

A pointer to a **null**-terminated string that specifies the account name.

Use a fully qualified string in the `domain_name\user_name` format to ensure that **LookupAccountName** finds the account in the desired domain.

[out, optional] `Sid`

A pointer to a buffer that receives the [SID](#) structure that corresponds to the account name pointed to by the *lpAccountName* parameter. If this parameter is **NULL**, *cbSid* must be zero.

[in, out] `cbSid`

A pointer to a variable. On input, this value specifies the size, in bytes, of the *Sid* buffer. If the function fails because the buffer is too small or if *cbSid* is zero, this variable receives the required buffer size.

[out, optional] `ReferencedDomainName`

A pointer to a buffer that receives the name of the domain where the account name is found. For computers that are not joined to a domain, this buffer receives the computer name. If this parameter is **NULL**, the function returns the required buffer size.

[in, out] `cchReferencedDomainName`

A pointer to a variable. On input, this value specifies the size, in **TCHARs**, of the *ReferencedDomainName* buffer. If the function fails because the buffer is too small, this variable receives the required buffer size, including the terminating **null** character. If the *ReferencedDomainName* parameter is **NULL**, this parameter must be zero.

[out] `peUse`

A pointer to a [SID_NAME_USE](#) enumerated type that indicates the type of the account when the function returns.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. For extended error information, call [GetLastError](#).

Remarks

The **LookupAccountName** function attempts to find a SID for the specified name by first checking a list of well-known SIDs. If the name does not correspond to a well-known SID, the function checks built-in and administratively defined local accounts. Next, the function checks the primary domain. If the name is not found there, trusted domains are checked.

Use fully qualified account names (for example, domain_name\user_name) instead of isolated names (for example, user_name). Fully qualified names are unambiguous and provide better performance when the lookup is performed. This function also supports fully qualified DNS names (for example, example.example.com\user_name) and [user principal names](#) (UPN) (for example, someone@example.com).

In addition to looking up local accounts, local domain accounts, and explicitly trusted domain accounts, **LookupAccountName** can look up the name for any account in any domain in the forest.

 **Note**

The winbase.h header defines **LookupAccountName** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[EqualPrefixSid](#)

[GetUserName](#)

[LookupAccountSid](#)

[LsaLookupNames2](#)

[SID](#)

[SID_NAME_USE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupAccountSidA function (winbase.h)

Article02/09/2023

The **LookupAccountSid** function accepts a [security identifier](#) (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found.

Syntax

C++

```
BOOL LookupAccountSidA(  
    [in, optional] LPCSTR     lpSystemName,  
    [in]          PSID        Sid,  
    [out, optional] LPSTR      Name,  
    [in, out]       LPDWORD    cchName,  
    [out, optional] LPSTR      ReferencedDomainName,  
    [in, out]       LPDWORD    cchReferencedDomainName,  
    [out]          PSID_NAME_USE peUse  
) ;
```

Parameters

[in, optional] *lpSystemName*

A pointer to a **null**-terminated character string that specifies the target computer. This string can be the name of a remote computer. If this parameter is **NULL**, the account name translation begins on the local system. If the name cannot be resolved on the local system, this function will try to resolve the name using domain controllers trusted by the local system. Generally, specify a value for *lpSystemName* only when the account is in an untrusted domain and the name of a computer in that domain is known.

[in] *Sid*

A pointer to the [SID](#) to look up.

[out, optional] *Name*

A pointer to a buffer that receives a **null**-terminated string that contains the account name that corresponds to the *lpSid* parameter.

[in, out] cchName

On input, specifies the size, in TCHARs, of the *lpName* buffer. If the function fails because the buffer is too small or if *cchName* is zero, *cchName* receives the required buffer size, including the terminating **null** character.

[out, optional] ReferencedDomainName

A pointer to a buffer that receives a **null**-terminated string that contains the name of the domain where the account name was found.

On a server, the domain name returned for most accounts in the security database of the local computer is the name of the domain for which the server is a domain controller.

On a workstation, the domain name returned for most accounts in the security database of the local computer is the name of the computer as of the last start of the system (backslashes are excluded). If the name of the computer changes, the old name continues to be returned as the domain name until the system is restarted.

Some accounts are predefined by the system. The domain name returned for these accounts is BUILTIN.

[in, out] cchReferencedDomainName

On input, specifies the size, in TCHARs, of the *lpReferencedDomainName* buffer. If the function fails because the buffer is too small or if *cchReferencedDomainName* is zero, *cchReferencedDomainName* receives the required buffer size, including the terminating **null** character.

[out] peUse

A pointer to a variable that receives a [SID_NAME_USE](#) value that indicates the type of the account.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The **LookupAccountSid** function attempts to find a name for the specified SID by first checking a list of well-known SIDs. If the supplied SID does not correspond to a well-known SID, the function checks built-in and administratively defined local accounts. Next, the function checks the primary domain. [Security identifiers](#) not recognized by the primary domain are checked against the trusted domains that correspond to their SID prefixes.

If the function cannot find an account name for the SID, [GetLastError](#) returns **ERROR_NONE_MAPPED**. This can occur if a network time-out prevents the function from finding the name. It also occurs for SIDs that have no corresponding account name, such as a [logon SID](#) that identifies a [logon session](#).

In addition to looking up SIDs for local accounts, local domain accounts, and explicitly trusted domain accounts, **LookupAccountSid** can look up SIDs for any account in any domain in the forest, including SIDs that appear only in the **SIDHistory** field of an account in the forest. The **SIDHistory** field stores former SIDs of an account that has been moved from another domain. To look up a SID, **LookupAccountSid** queries the global catalog of the forest.

Examples

For an example that uses this function, see [Searching for a SID in an Access Token](#).

Note

The `winbase.h` header defines `LookupAccountSid` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)

Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[EqualPrefixSid](#)

[LookupAccountName](#)

[SID](#)

[SID_NAME_USE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupAccountSidLocalA function (winbase.h)

Article02/09/2023

`LookupAccountSidLocalA` is defined as a macro that calls [LookupAccountSidA](#) with `NULL` as the first parameter. Retrieves the name of the account for the specified SID on the local machine.

Syntax

C++

```
BOOL LookupAccountSidLocalA(
    [in]             PSID      Sid,
    [out, optional] LPSTR     Name,
    [in, out]        LPDWORD   cchName,
    [out, optional] LPSTR     ReferencedDomainName,
    [in, out]        LPDWORD   cchReferencedDomainName,
    [out]           PSID_NAME_USE peUse
);
```

Parameters

[in] `Sid`

A pointer to the [SID](#) to look up.

[out, optional] `Name`

A pointer to a buffer that receives a **null**-terminated string that contains the account name that corresponds to the *lpSid* parameter.

[in, out] `cchName`

On input, specifies the size, in **TCHARs**, of the *lpName* buffer. If the function fails because the buffer is too small or if *cchName* is zero, *cchName* receives the required buffer size, including the terminating **null** character.

[out, optional] `ReferencedDomainName`

A pointer to a buffer that receives a **null**-terminated string that contains the name of the domain where the account name was found.

On a server, the domain name returned for most accounts in the security database of the local computer is the name of the domain for which the server is a domain controller.

On a workstation, the domain name returned for most accounts in the security database of the local computer is the name of the computer as of the last start of the system (backslashes are excluded). If the name of the computer changes, the old name continues to be returned as the domain name until the system is restarted.

Some accounts are predefined by the system. The domain name returned for these accounts is BUILTIN.

[in, out] *cchReferencedDomainName*

On input, specifies the size, in TCHARs, of the *lpReferencedDomainName* buffer. If the function fails because the buffer is too small or if *cchReferencedDomainName* is zero, *cchReferencedDomainName* receives the required buffer size, including the terminating null character.

[out] *peUse*

A pointer to a variable that receives a [SID_NAME_USE](#) value that indicates the type of the account.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

This function is similar to [LookupAccountSid](#), but restricts the search to the local machine.

Note

The winbase.h header defines `LookupAccountSidLocal` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[EqualPrefixSid](#)

[LookupAccountName](#)

[SID](#)

[SID_NAME_USE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupAccountSidLocalW function (winbase.h)

Article02/09/2023

`LookupAccountSidLocalW` is defined as a macro that calls [LookupAccountSidW](#) with `NULL` as the first parameter. Retrieves the name of the account for the specified SID on the local machine.

Syntax

C++

```
BOOL LookupAccountSidLocalW(  
    [in]             PSID      Sid,  
    [out, optional] LPWSTR    Name,  
    [in, out]        LPDWORD   cchName,  
    [out, optional] LPWSTR    ReferencedDomainName,  
    [in, out]        LPDWORD   cchReferencedDomainName,  
    [out]           PSID_NAME_USE peUse  
);
```

Parameters

[in] `Sid`

A pointer to the [SID](#) to look up.

[out, optional] `Name`

A pointer to a buffer that receives a **null**-terminated string that contains the account name that corresponds to the *lpSid* parameter.

[in, out] `cchName`

On input, specifies the size, in **TCHARs**, of the *lpName* buffer. If the function fails because the buffer is too small or if *cchName* is zero, *cchName* receives the required buffer size, including the terminating **null** character.

[out, optional] `ReferencedDomainName`

A pointer to a buffer that receives a **null**-terminated string that contains the name of the domain where the account name was found.

On a server, the domain name returned for most accounts in the security database of the local computer is the name of the domain for which the server is a domain controller.

On a workstation, the domain name returned for most accounts in the security database of the local computer is the name of the computer as of the last start of the system (backslashes are excluded). If the name of the computer changes, the old name continues to be returned as the domain name until the system is restarted.

Some accounts are predefined by the system. The domain name returned for these accounts is BUILTIN.

[in, out] *cchReferencedDomainName*

On input, specifies the size, in TCHARs, of the *lpReferencedDomainName* buffer. If the function fails because the buffer is too small or if *cchReferencedDomainName* is zero, *cchReferencedDomainName* receives the required buffer size, including the terminating null character.

[out] *peUse*

A pointer to a variable that receives a [SID_NAME_USE](#) value that indicates the type of the account.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

This function is similar to [LookupAccountSid](#), but restricts the search to the local machine.

Note

The winbase.h header defines `LookupAccountSidLocal` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[EqualPrefixSid](#)

[LookupAccountName](#)

[SID](#)

[SID_NAME_USE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupAccountSidW function (winbase.h)

Article02/09/2023

The **LookupAccountSid** function accepts a [security identifier](#) (SID) as input. It retrieves the name of the account for this SID and the name of the first domain on which this SID is found.

Syntax

C++

```
BOOL LookupAccountSidW(  
    [in, optional] LPCWSTR     lpSystemName,  
    [in]          PSID          Sid,  
    [out, optional] LPWSTR      Name,  
    [in, out]       LPDWORD     cchName,  
    [out, optional] LPWSTR      ReferencedDomainName,  
    [in, out]       LPDWORD     cchReferencedDomainName,  
    [out]          PSID_NAME_USE peUse  
) ;
```

Parameters

[in, optional] *lpSystemName*

A pointer to a **null**-terminated character string that specifies the target computer. This string can be the name of a remote computer. If this parameter is **NULL**, the account name translation begins on the local system. If the name cannot be resolved on the local system, this function will try to resolve the name using domain controllers trusted by the local system. Generally, specify a value for *lpSystemName* only when the account is in an untrusted domain and the name of a computer in that domain is known.

[in] *Sid*

A pointer to the [SID](#) to look up.

[out, optional] *Name*

A pointer to a buffer that receives a **null**-terminated string that contains the account name that corresponds to the *lpSid* parameter.

[in, out] cchName

On input, specifies the size, in TCHARs, of the *lpName* buffer. If the function fails because the buffer is too small or if *cchName* is zero, *cchName* receives the required buffer size, including the terminating **null** character.

[out, optional] ReferencedDomainName

A pointer to a buffer that receives a **null**-terminated string that contains the name of the domain where the account name was found.

On a server, the domain name returned for most accounts in the security database of the local computer is the name of the domain for which the server is a domain controller.

On a workstation, the domain name returned for most accounts in the security database of the local computer is the name of the computer as of the last start of the system (backslashes are excluded). If the name of the computer changes, the old name continues to be returned as the domain name until the system is restarted.

Some accounts are predefined by the system. The domain name returned for these accounts is BUILTIN.

[in, out] cchReferencedDomainName

On input, specifies the size, in TCHARs, of the *lpReferencedDomainName* buffer. If the function fails because the buffer is too small or if *cchReferencedDomainName* is zero, *cchReferencedDomainName* receives the required buffer size, including the terminating **null** character.

[out] peUse

A pointer to a variable that receives a [SID_NAME_USE](#) value that indicates the type of the account.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The **LookupAccountSid** function attempts to find a name for the specified SID by first checking a list of well-known SIDs. If the supplied SID does not correspond to a well-known SID, the function checks built-in and administratively defined local accounts. Next, the function checks the primary domain. [Security identifiers](#) not recognized by the primary domain are checked against the trusted domains that correspond to their SID prefixes.

If the function cannot find an account name for the SID, [GetLastError](#) returns **ERROR_NONE_MAPPED**. This can occur if a network time-out prevents the function from finding the name. It also occurs for SIDs that have no corresponding account name, such as a [logon SID](#) that identifies a [logon session](#).

In addition to looking up SIDs for local accounts, local domain accounts, and explicitly trusted domain accounts, **LookupAccountSid** can look up SIDs for any account in any domain in the forest, including SIDs that appear only in the **SIDHistory** field of an account in the forest. The **SIDHistory** field stores former SIDs of an account that has been moved from another domain. To look up a SID, **LookupAccountSid** queries the global catalog of the forest.

Examples

For an example that uses this function, see [Searching for a SID in an Access Token](#).

Note

The `winbase.h` header defines `LookupAccountSid` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)

Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[EqualPrefixSid](#)

[LookupAccountName](#)

[SID](#)

[SID_NAME_USE](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupPrivilegeDisplayNameA function (winbase.h)

Article02/09/2023

The `LookupPrivilegeDisplayName` function retrieves the display name that represents a specified privilege.

Syntax

C++

```
BOOL LookupPrivilegeDisplayNameA(
    [in, optional] LPCSTR lpSystemName,
    [in]           LPCSTR lpName,
    [out, optional] LPSTR  lpDisplayName,
    [in, out]       LPDWORD cchDisplayName,
    [out]          LPDWORD lpLanguageId
);
```

Parameters

[in, optional] `lpSystemName`

A pointer to a null-terminated string that specifies the name of the system on which the privilege name is retrieved. If a null string is specified, the function attempts to find the display name on the local system.

[in] `lpName`

A pointer to a null-terminated string that specifies the name of the privilege, as defined in Winnt.h. For example, this parameter could specify the constant, `SE_REMOTE_SHUTDOWN_NAME`, or its corresponding string, "SeRemoteShutdownPrivilege". For a list of values, see [Privilege Constants](#).

[out, optional] `lpDisplayName`

A pointer to a buffer that receives a null-terminated string that specifies the privilege display name. For example, if the `lpName` parameter is `SE_REMOTE_SHUTDOWN_NAME`, the privilege display name is "Force shutdown from a remote system."

[in, out] `cchDisplayName`

A pointer to a variable that specifies the size, in **TCHARs**, of the *lpDisplayName* buffer. When the function returns, this parameter contains the length of the privilege display name, not including the terminating null character. If the buffer pointed to by the *lpDisplayName* parameter is too small, this variable contains the required size.

[out] lpLanguageId

A pointer to a variable that receives the language identifier for the returned display name.

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

The **LookupPrivilegeDisplayName** function retrieves display names only for the privileges specified in the Defined Privileges section of Winnt.h.

 **Note**

The winbase.h header defines **LookupPrivilegeDisplayName** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[LookupPrivilegeName](#)

[LookupPrivilegeValue](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupPrivilegeDisplayNameW function (winbase.h)

Article02/09/2023

The `LookupPrivilegeDisplayName` function retrieves the display name that represents a specified privilege.

Syntax

C++

```
BOOL LookupPrivilegeDisplayNameW(  
    [in, optional]  LPCWSTR lpSystemName,  
    [in]           LPCWSTR lpName,  
    [out, optional] LPWSTR  lpDisplayName,  
    [in, out]        LPDWORD cchDisplayName,  
    [out]           LPDWORD lpLanguageId  
) ;
```

Parameters

[in, optional] `lpSystemName`

A pointer to a null-terminated string that specifies the name of the system on which the privilege name is retrieved. If a null string is specified, the function attempts to find the display name on the local system.

[in] `lpName`

A pointer to a null-terminated string that specifies the name of the privilege, as defined in Winnt.h. For example, this parameter could specify the constant, `SE_REMOTE_SHUTDOWN_NAME`, or its corresponding string, "SeRemoteShutdownPrivilege". For a list of values, see [Privilege Constants](#).

[out, optional] `lpDisplayName`

A pointer to a buffer that receives a null-terminated string that specifies the privilege display name. For example, if the `lpName` parameter is `SE_REMOTE_SHUTDOWN_NAME`, the privilege display name is "Force shutdown from a remote system."

[in, out] `cchDisplayName`

A pointer to a variable that specifies the size, in **TCHARs**, of the *lpDisplayName* buffer. When the function returns, this parameter contains the length of the privilege display name, not including the terminating null character. If the buffer pointed to by the *lpDisplayName* parameter is too small, this variable contains the required size.

[out] lpLanguageId

A pointer to a variable that receives the language identifier for the returned display name.

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

The **LookupPrivilegeDisplayName** function retrieves display names only for the privileges specified in the Defined Privileges section of Winnt.h.

 **Note**

The winbase.h header defines **LookupPrivilegeDisplayName** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[LookupPrivilegeName](#)

[LookupPrivilegeValue](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

LookupPrivilegeNameA function (winbase.h)

Article 02/09/2023

The **LookupPrivilegeName** function retrieves the name that corresponds to the privilege represented on a specific system by a specified [locally unique identifier](#) (LUID).

Syntax

C++

```
BOOL LookupPrivilegeNameA(
    [in, optional] LPCSTR lpSystemName,
    [in]           PLUID lpLuid,
    [out, optional] LPSTR lpName,
    [in, out]       LPDWORD cchName
);
```

Parameters

[in, optional] *lpSystemName*

A pointer to a null-terminated string that specifies the name of the system on which the privilege name is retrieved. If a null string is specified, the function attempts to find the privilege name on the local system.

[in] *lpLuid*

A pointer to the LUID by which the privilege is known on the target system.

[out, optional] *lpName*

A pointer to a buffer that receives a null-terminated string that represents the privilege name. For example, this string could be "SeSecurityPrivilege".

[in, out] *cchName*

A pointer to a variable that specifies the size, in a **TCHAR** value, of the *lpName* buffer. When the function returns, this parameter contains the length of the privilege name, not including the terminating null character. If the buffer pointed to by the *lpName* parameter is too small, this variable contains the required size.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The `LookupPrivilegeName` function supports only the privileges specified in the Defined Privileges section of Winnt.h. For a list of values, see [Privilege Constants](#).

Note

The winbase.h header defines `LookupPrivilegeName` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control](#)

[Basic Access Control Functions](#)

[LookupPrivilegeDisplayName](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LookupPrivilegeNameW function (winbase.h)

Article 02/09/2023

The **LookupPrivilegeName** function retrieves the name that corresponds to the privilege represented on a specific system by a specified [locally unique identifier](#) (LUID).

Syntax

C++

```
BOOL LookupPrivilegeNameW(
    [in, optional] LPCWSTR lpSystemName,
    [in]           PLUID   lpLuid,
    [out, optional] LPWSTR  lpName,
    [in, out]       LPDWORD cchName
);
```

Parameters

[in, optional] *lpSystemName*

A pointer to a null-terminated string that specifies the name of the system on which the privilege name is retrieved. If a null string is specified, the function attempts to find the privilege name on the local system.

[in] *lpLuid*

A pointer to the LUID by which the privilege is known on the target system.

[out, optional] *lpName*

A pointer to a buffer that receives a null-terminated string that represents the privilege name. For example, this string could be "SeSecurityPrivilege".

[in, out] *cchName*

A pointer to a variable that specifies the size, in a **TCHAR** value, of the *lpName* buffer. When the function returns, this parameter contains the length of the privilege name, not including the terminating null character. If the buffer pointed to by the *lpName* parameter is too small, this variable contains the required size.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The `LookupPrivilegeName` function supports only the privileges specified in the Defined Privileges section of Winnt.h. For a list of values, see [Privilege Constants](#).

Note

The winbase.h header defines `LookupPrivilegeName` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control](#)

[Basic Access Control Functions](#)

[LookupPrivilegeDisplayName](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LookupPrivilegeValueA function (winbase.h)

Article02/09/2023

The **LookupPrivilegeValue** function retrieves the [locally unique identifier](#) (LUID) used on a specified system to locally represent the specified privilege name.

Syntax

C++

```
BOOL LookupPrivilegeValueA(
    [in, optional] LPCSTR lpSystemName,
    [in]           LPCSTR lpName,
    [out]          PLUID  lpLuid
);
```

Parameters

[in, optional] *lpSystemName*

A pointer to a null-terminated string that specifies the name of the system on which the privilege name is retrieved. If a null string is specified, the function attempts to find the privilege name on the local system.

[in] *lpName*

A pointer to a null-terminated string that specifies the name of the privilege, as defined in the Winnt.h header file. For example, this parameter could specify the constant, SE_SECURITY_NAME, or its corresponding string, "SeSecurityPrivilege".

[out] *lpLuid*

A pointer to a variable that receives the LUID by which the privilege is known on the system specified by the *lpSystemName* parameter.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The `LookupPrivilegeValue` function supports only the privileges specified in the Defined Privileges section of Winnt.h. For a list of values, see [Privilege Constants](#).

Examples

For an example that uses this function, see [Enabling and Disabling Privileges](#).

Note

The winbase.h header defines `LookupPrivilegeValue` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control](#)

[Basic Access Control Functions](#)

[LookupPrivilegeDisplayName](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LookupPrivilegeValueW function (winbase.h)

Article02/09/2023

The **LookupPrivilegeValue** function retrieves the [locally unique identifier](#) (LUID) used on a specified system to locally represent the specified privilege name.

Syntax

C++

```
BOOL LookupPrivilegeValueW(
    [in, optional] LPCWSTR lpSystemName,
    [in]           LPCWSTR lpName,
    [out]          PLUID   lpLuid
);
```

Parameters

[in, optional] *lpSystemName*

A pointer to a null-terminated string that specifies the name of the system on which the privilege name is retrieved. If a null string is specified, the function attempts to find the privilege name on the local system.

[in] *lpName*

A pointer to a null-terminated string that specifies the name of the privilege, as defined in the Winnt.h header file. For example, this parameter could specify the constant, SE_SECURITY_NAME, or its corresponding string, "SeSecurityPrivilege".

[out] *lpLuid*

A pointer to a variable that receives the LUID by which the privilege is known on the system specified by the *lpSystemName* parameter.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The `LookupPrivilegeValue` function supports only the privileges specified in the Defined Privileges section of Winnt.h. For a list of values, see [Privilege Constants](#).

Examples

For an example that uses this function, see [Enabling and Disabling Privileges](#).

Note

The winbase.h header defines `LookupPrivilegeValue` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control](#)

[Basic Access Control Functions](#)

[LookupPrivilegeDisplayName](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LPPROGRESS_ROUTINE callback function (winbase.h)

Article 10/13/2021

An application-defined callback function used with the [CopyFileEx](#), [MoveFileTransacted](#), and [MoveFileWithProgress](#) functions. It is called when a portion of a copy or move operation is completed. The **LPPROGRESS_ROUTINE** type defines a pointer to this callback function. **CopyProgressRoutine** is a placeholder for the application-defined function name.

Syntax

C++

```
LPPROGRESS_ROUTINE LpprogressRoutine;

DWORD LpprogressRoutine(
    [in]          LARGE_INTEGER TotalFileSize,
    [in]          LARGE_INTEGER TotalBytesTransferred,
    [in]          LARGE_INTEGER StreamSize,
    [in]          LARGE_INTEGER StreamBytesTransferred,
    [in]          DWORD dwStreamNumber,
    [in]          DWORD dwCallbackReason,
    [in]          HANDLE hSourceFile,
    [in]          HANDLE hDestinationFile,
    [in, optional] LPVOID lpData
)
{...}
```

Parameters

[in] **TotalFileSize**

The total size of the file, in bytes.

[in] **TotalBytesTransferred**

The total number of bytes transferred from the source file to the destination file since the copy operation began.

[in] **StreamSize**

The total size of the current file stream, in bytes.

[in] StreamBytesTransferred

The total number of bytes in the current stream that have been transferred from the source file to the destination file since the copy operation began.

[in] dwStreamNumber

A handle to the current stream. The first time **CopyProgressRoutine** is called, the stream number is 1.

[in] dwCallbackReason

The reason that **CopyProgressRoutine** was called. This parameter can be one of the following values.

Value	Meaning
CALLBACK_CHUNK_FINISHED 0x00000000	Another part of the data file was copied.
CALLBACK_STREAM_SWITCH 0x00000001	Another stream was created and is about to be copied. This is the callback reason given when the callback routine is first invoked.

[in] hSourceFile

A handle to the source file.

[in] hDestinationFile

A handle to the destination file

[in, optional] lpData

Argument passed to **CopyProgressRoutine** by [CopyFileEx](#), [MoveFileTransacted](#), or [MoveFileWithProgress](#).

Return value

The **CopyProgressRoutine** function should return one of the following values.

Return code/value	Description
PROGRESS_CANCEL 1	Cancel the copy operation and delete the destination file.
PROGRESS_CONTINUE	Continue the copy operation.

0

PROGRESS QUIET 3	Continue the copy operation, but stop invoking CopyProgressRoutine to report progress.
PROGRESS_STOP 2	Stop the copy operation. It can be restarted at a later time.

Remarks

An application can use this information to display a progress bar that shows the total number of bytes copied as a percent of the total file size.

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)

See also

[CopyFileEx](#)

[File Management Functions](#)

[MoveFileTransacted](#)

[MoveFileWithProgress](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcatA function (winbase.h)

Article 02/09/2023

Appends one string to another.

Warning Do not use. Consider using **StringCchCat** instead. See Security Considerations.

Syntax

C++

```
LPSTR IstrcatA(  
    [in, out] LPSTR lpString1,  
    [in]      LPCSTR lpString2  
)
```

Parameters

[in, out] lpString1

Type: **LPTSTR**

The first null-terminated string. This buffer must be large enough to contain both strings.

[in] lpString2

Type: **LPTSTR**

The null-terminated string to be appended to the string specified in the *lpString1* parameter.

Return value

Type: **LPTSTR**

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is **NULL** and *lpString1* may not be null-terminated.

Remarks

Note

The winbase.h header defines `Istrcat` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[Reference](#)

[`StringCbCat`](#)

[`StringCbCatEx`](#)

[`StringCbCatN`](#)

[`StringCbCatNEx`](#)

[`StringCchCat`](#)

[StringCchCatEx](#)

[StringCchCatN](#)

[StringCchCatNEx](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcatW function (winbase.h)

Article02/09/2023

Appends one string to another.

Warning Do not use. Consider using [StringCchCat](#) instead. See Security Considerations.

Syntax

C++

```
LPWSTR IstrcatW(
    [in, out] LPWSTR lpString1,
    [in]      LPCWSTR lpString2
);
```

Parameters

[in, out] lpString1

Type: [LPTSTR](#)

The first null-terminated string. This buffer must be large enough to contain both strings.

[in] lpString2

Type: [LPTSTR](#)

The null-terminated string to be appended to the string specified in the *lpString1* parameter.

Return value

Type: [LPTSTR](#)

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is **NULL** and *lpString1* may not be null-terminated.

Remarks

Note

The winbase.h header defines `Istrcat` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[Reference](#)

[`StringCbCat`](#)

[`StringCbCatEx`](#)

[`StringCbCatN`](#)

[`StringCbCatNEx`](#)

[`StringCchCat`](#)

[StringCchCatEx](#)

[StringCchCatN](#)

[StringCchCatNEx](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcmpA function (winbase.h)

Article 02/09/2023

Compares two character strings. The comparison is case-sensitive.

To perform a comparison that is not case-sensitive, use the [Istrcmpi](#) function.

Syntax

C++

```
int IstrcmpA(
    [in] LPCSTR lpString1,
    [in] LPCSTR lpString2
);
```

Parameters

[in] lpString1

Type: **LPCTSTR**

The first null-terminated string to be compared.

[in] lpString2

Type: **LPCTSTR**

The second null-terminated string to be compared.

Return value

Type: **int**

If the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative. If the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, the return value is positive. If the strings are equal, the return value is zero.

Remarks

The **Istrcmp** function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

Note that the *lpString1* and *lpString2* parameters must be null-terminated, otherwise the string comparison can be incorrect.

The function calls [CompareStringEx](#), using the current thread locale, and subtracts 2 from the result, to maintain the C run-time conventions for comparing strings.

The language (user locale) selected by the user at setup time, or through Control Panel, determines which string is greater (or whether the strings are the same). If no language (user locale) is selected, the system performs the comparison by using default values.

With a double-byte character set (DBCS) version of the system, this function can compare two DBCS strings.

The **Istrcmp** function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. For a detailed discussion of word sorts and string sorts, see [Handling Sorting in Your Applications](#).

Security Remarks

See [Security Considerations: International Features](#) for security considerations regarding choice of comparison functions.

Note

The winbase.h header defines **Istrcmp** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client

Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CompareString](#)

[CompareStringEx](#)

[CompareStringOrdinal](#)

[Conceptual](#)

[Other Resources](#)

[Reference](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrcpy](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcmpiA function (winbase.h)

Article02/09/2023

Compares two character strings. The comparison is not case-sensitive.

To perform a comparison that is case-sensitive, use the [Istrcmp](#) function.

Syntax

C++

```
int IstrcmpiA(
    [in] LPCSTR lpString1,
    [in] LPCSTR lpString2
);
```

Parameters

[in] lpString1

Type: LPCTSTR

The first null-terminated string to be compared.

[in] lpString2

Type: LPCTSTR

The second null-terminated string to be compared.

Return value

Type: int

If the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative. If the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, the return value is positive. If the strings are equal, the return value is zero.

Remarks

The **Istrcmpi** function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

Note that the *lpString1* and *lpString2* parameters must be null-terminated, otherwise the string comparison can be incorrect.

The function calls [CompareStringEx](#), using the current thread locale, and subtracts 2 from the result, to maintain the C run-time conventions for comparing strings.

For some locales, the **Istrcmpi** function may be insufficient. If this occurs, use [CompareStringEx](#) to ensure proper comparison. For example, in Japan call with the **NORM_IGNORECASE**, **NORM_IGNOREKANATYPE**, and **NORM_IGNOREWIDTH** values to achieve the most appropriate non-exact string comparison. The **NORM_IGNOREKANATYPE** and **NORM_IGNOREWIDTH** values are ignored in non-Asian locales, so you can set these values for all locales and be guaranteed to have a culturally correct "insensitive" sorting regardless of the locale. Note that specifying these values slows performance, so use them only when necessary.

With a double-byte character set (DBCS) version of the system, this function can compare two DBCS strings.

The **Istrcmpi** function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. For a detailed discussion of word sorts and string sorts, see [Handling Sorting in Your Applications](#).

Security Remarks

See [Security Considerations: International Features](#) for security considerations regarding choice of comparison functions.

Note

The winbase.h header defines **Istrcmpi** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CompareString](#)

[CompareStringEx](#)

[CompareStringOrdinal](#)

[Conceptual](#)

Other Resources

Reference

[Strings](#)

[lstrcat](#)

[lstrcmp](#)

[lstrcpy](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcmpiW function (winbase.h)

Article 02/09/2023

Compares two character strings. The comparison is not case-sensitive.

To perform a comparison that is case-sensitive, use the [Istrcmp](#) function.

Syntax

C++

```
int IstrcmpiW(
    [in] LPCWSTR lpString1,
    [in] LPCWSTR lpString2
);
```

Parameters

[in] lpString1

Type: LPCTSTR

The first null-terminated string to be compared.

[in] lpString2

Type: LPCTSTR

The second null-terminated string to be compared.

Return value

Type: int

If the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative. If the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, the return value is positive. If the strings are equal, the return value is zero.

Remarks

The **Istrcmpi** function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

Note that the *lpString1* and *lpString2* parameters must be null-terminated, otherwise the string comparison can be incorrect.

The function calls [CompareStringEx](#), using the current thread locale, and subtracts 2 from the result, to maintain the C run-time conventions for comparing strings.

For some locales, the **Istrcmpi** function may be insufficient. If this occurs, use [CompareStringEx](#) to ensure proper comparison. For example, in Japan call with the **NORM_IGNORECASE**, **NORM_IGNOREKANATYPE**, and **NORM_IGNOREWIDTH** values to achieve the most appropriate non-exact string comparison. The **NORM_IGNOREKANATYPE** and **NORM_IGNOREWIDTH** values are ignored in non-Asian locales, so you can set these values for all locales and be guaranteed to have a culturally correct "insensitive" sorting regardless of the locale. Note that specifying these values slows performance, so use them only when necessary.

With a double-byte character set (DBCS) version of the system, this function can compare two DBCS strings.

The **Istrcmpi** function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. For a detailed discussion of word sorts and string sorts, see [Handling Sorting in Your Applications](#).

Security Remarks

See [Security Considerations: International Features](#) for security considerations regarding choice of comparison functions.

Note

The winbase.h header defines **Istrcmpi** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CompareString](#)

[CompareStringEx](#)

[CompareStringOrdinal](#)

[Conceptual](#)

Other Resources

Reference

[Strings](#)

[lstrcat](#)

[lstrcmp](#)

[lstrcpy](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcmpW function (winbase.h)

Article 02/09/2023

Compares two character strings. The comparison is case-sensitive.

To perform a comparison that is not case-sensitive, use the [Istrcmpi](#) function.

Syntax

C++

```
int IstrcmpW(
    [in] LPCWSTR lpString1,
    [in] LPCWSTR lpString2
);
```

Parameters

[in] lpString1

Type: LPCTSTR

The first null-terminated string to be compared.

[in] lpString2

Type: LPCTSTR

The second null-terminated string to be compared.

Return value

Type: int

If the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative. If the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, the return value is positive. If the strings are equal, the return value is zero.

Remarks

The **Istrcmp** function compares two strings by checking the first characters against each other, the second characters against each other, and so on until it finds an inequality or reaches the ends of the strings.

Note that the *lpString1* and *lpString2* parameters must be null-terminated, otherwise the string comparison can be incorrect.

The function calls [CompareStringEx](#), using the current thread locale, and subtracts 2 from the result, to maintain the C run-time conventions for comparing strings.

The language (user locale) selected by the user at setup time, or through Control Panel, determines which string is greater (or whether the strings are the same). If no language (user locale) is selected, the system performs the comparison by using default values.

With a double-byte character set (DBCS) version of the system, this function can compare two DBCS strings.

The **Istrcmp** function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as "coop" and "co-op" stay together within a sorted list. For a detailed discussion of word sorts and string sorts, see [Handling Sorting in Your Applications](#).

Security Remarks

See [Security Considerations: International Features](#) for security considerations regarding choice of comparison functions.

Note

The winbase.h header defines **Istrcmp** as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client

Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CompareString](#)

[CompareStringEx](#)

[CompareStringOrdinal](#)

[Conceptual](#)

[Other Resources](#)

[Reference](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrcpy](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcpyA function (winbase.h)

Article 02/09/2023

Copies a string to a buffer.

Warning Do not use. Consider using [StringCchCopy](#) instead. See Remarks.

Syntax

C++

```
LPSTR IstrcpyA(  
    [out] LPSTR lpString1,  
    [in]  LPCSTR lpString2  
)
```

Parameters

[out] lpString1

Type: [LPTSTR](#)

A buffer to receive the contents of the string pointed to by the *lpString2* parameter. The buffer must be large enough to contain the string, including the terminating null character.

[in] lpString2

Type: [LPTSTR](#)

The null-terminated string to be copied.

Return value

Type: [LPTSTR](#)

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is [NULL](#) and *lpString1* may not be null-terminated.

Remarks

With a double-byte character set (DBCS) version of the system, this function can be used to copy a DBCS string.

The `Istrcpy` function has an undefined behavior if source and destination buffers overlap.

Security Remarks

Using this function incorrectly can compromise the security of your application. This function uses structured exception handling (SEH) to catch access violations and other errors. When this function catches SEH errors, it returns `NULL` without null-terminating the string and without notifying the caller of the error. The caller is not safe to assume that insufficient space is the error condition.

lpString1 must be large enough to hold *lpString2* and the closing '\0', otherwise a buffer overrun may occur.

Buffer overflow situations are the cause of many security problems in applications and can cause a denial of service attack against the application if an access violation occurs. In the worst case, a buffer overrun may allow an attacker to inject executable code into your process, especially if *lpString1* is a stack-based buffer.

Consider using [StringCchCopy](#) instead; use either `StringCchCopy(buffer, sizeof(buffer)/sizeof(buffer[0]), src);`, being aware that `buffer` must not be a pointer or use `StringCchCopy(buffer, ARYSIZE(buffer), src);`, being aware that, when copying to a pointer, the caller is responsible for passing in the size of the pointed-to memory in characters.

Note

The `winbase.h` header defines `Istrcpy` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Conceptual

Reference

[StringCbCopy](#)

[StringCbCopyEx](#)

[StringCbCopyN](#)

[StringCbCopyNEx](#)

[StringCchCopy](#)

[StringCchCopyEx](#)

[StringCchCopyN](#)

[StringCchCopyNEx](#)

[Strings](#)

[Istrcmp](#)

[Istrcmpi](#)

[Istrlen](#)

Feedback



Was this page helpful? [!\[\]\(c47677d40dd8f5c4ff9bcb53c7abbcfd_img.jpg\) Yes](#) [!\[\]\(da6072f27b9bed06ad07ffd15ba092d3_img.jpg\) No](#)

Get help at Microsoft Q&A

IstrcpyN function (winbase.h)

Article02/09/2023

Copies a specified number of characters from a source string into a buffer.

Warning Do not use. Consider using [StringCchCopy](#) instead. See Remarks.

Syntax

C++

```
LPSTR strcpyN(
    [out] LPSTR lpString1,
    [in]  LPCSTR lpString2,
    [in]  int    iMaxLength
);
```

Parameters

[out] lpString1

Type: [LPTSTR](#)

The destination buffer, which receives the copied characters. The buffer must be large enough to contain the number of **TCHAR** values specified by *iMaxLength*, including room for a terminating null character.

[in] lpString2

Type: [LPCTSTR](#)

The source string from which the function is to copy characters.

[in] iMaxLength

Type: [int](#)

The number of **TCHAR** values to be copied from the string pointed to by *lpString2* into the buffer pointed to by *lpString1*, including a terminating null character.

Return value

Type: LPTSTR

If the function succeeds, the return value is a pointer to the buffer. The function can succeed even if the source string is greater than *iMaxLength* characters.

If the function fails, the return value is **NULL** and *lpString1* may not be null-terminated.

Remarks

The buffer pointed to by *lpString1* must be large enough to include a terminating null character, and the string length value specified by *iMaxLength* includes room for a terminating null character.

The **Istrcpyn** function has an undefined behavior if source and destination buffers overlap.

Security Warning

Using this function incorrectly can compromise the security of your application. This function uses structured exception handling (SEH) to catch access violations and other errors. When this function catches SEH errors, it returns **NULL** without null-terminating the string and without notifying the caller of the error. The caller is not safe to assume that insufficient space is the error condition.

If the buffer pointed to by *lpString1* is not large enough to contain the copied string, a buffer overrun can occur. When copying an entire string, note that **sizeof** returns the number of bytes. For example, if *lpString1* points to a buffer *szString1* which is declared as `TCHAR szString[100]`, then `sizeof(szString1)` gives the size of the buffer in bytes rather than **WCHAR**, which could lead to a buffer overflow for the Unicode version of the function.

Buffer overflow situations are the cause of many security problems in applications and can cause a denial of service attack against the application if an access violation occurs. In the worst case, a buffer overrun may allow an attacker to inject executable code into your process, especially if *lpString1* is a stack-based buffer.

Using `sizeof(szString1)/sizeof(szString1[0])` gives the proper size of the buffer.

Consider using [StringCchCopy](#) instead; use either `StringCchCopy(buffer, sizeof(buffer)/sizeof(buffer[0]), src);`, being aware that `buffer` must not be a

pointer or use `StringCchCopy(buffer, ARRAYSIZE(buffer), src);`, being aware that, when copying to a pointer, the caller is responsible for passing in the size of the pointed-to memory in characters.

Review [Security Considerations: Windows User Interface](#) before continuing.

 **Note**

The winbase.h header defines `Istrcpyn` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[Reference](#)

[StringCbCopy](#)

[StringCbCopyEx](#)

[StringCbCopyN](#)

[StringCbCopyNEx](#)

[StringCbLength](#)

[StringCchCopy](#)

[StringCchCopyEx](#)

[StringCchCopyN](#)

[StringCchCopyNEx](#)

[StringCchLength](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcpyNW function (winbase.h)

Article02/09/2023

Copies a specified number of characters from a source string into a buffer.

Warning Do not use. Consider using [StringCchCopy](#) instead. See Remarks.

Syntax

C++

```
LPWSTR strcpyNW(
    [out] LPWSTR  lpString1,
    [in]  LPCWSTR lpString2,
    [in]  int      iMaxLength
);
```

Parameters

[out] `lpString1`

Type: [LPTSTR](#)

The destination buffer, which receives the copied characters. The buffer must be large enough to contain the number of **TCHAR** values specified by *iMaxLength*, including room for a terminating null character.

[in] `lpString2`

Type: [LPCTSTR](#)

The source string from which the function is to copy characters.

[in] `iMaxLength`

Type: [int](#)

The number of **TCHAR** values to be copied from the string pointed to by *lpString2* into the buffer pointed to by *lpString1*, including a terminating null character.

Return value

Type: LPTSTR

If the function succeeds, the return value is a pointer to the buffer. The function can succeed even if the source string is greater than *iMaxLength* characters.

If the function fails, the return value is **NULL** and *lpString1* may not be null-terminated.

Remarks

The buffer pointed to by *lpString1* must be large enough to include a terminating null character, and the string length value specified by *iMaxLength* includes room for a terminating null character.

The **Istrcpyn** function has an undefined behavior if source and destination buffers overlap.

Security Warning

Using this function incorrectly can compromise the security of your application. This function uses structured exception handling (SEH) to catch access violations and other errors. When this function catches SEH errors, it returns **NULL** without null-terminating the string and without notifying the caller of the error. The caller is not safe to assume that insufficient space is the error condition.

If the buffer pointed to by *lpString1* is not large enough to contain the copied string, a buffer overrun can occur. When copying an entire string, note that **sizeof** returns the number of bytes. For example, if *lpString1* points to a buffer *szString1* which is declared as `TCHAR szString[100]`, then `sizeof(szString1)` gives the size of the buffer in bytes rather than **WCHAR**, which could lead to a buffer overflow for the Unicode version of the function.

Buffer overflow situations are the cause of many security problems in applications and can cause a denial of service attack against the application if an access violation occurs. In the worst case, a buffer overrun may allow an attacker to inject executable code into your process, especially if *lpString1* is a stack-based buffer.

Using `sizeof(szString1)/sizeof(szString1[0])` gives the proper size of the buffer.

Consider using [StringCchCopy](#) instead; use either `StringCchCopy(buffer, sizeof(buffer)/sizeof(buffer[0]), src);`, being aware that `buffer` must not be a

pointer or use `StringCchCopy(buffer, ARRAYSIZE(buffer), src);`, being aware that, when copying to a pointer, the caller is responsible for passing in the size of the pointed-to memory in characters.

Review [Security Considerations: Windows User Interface](#) before continuing.

 **Note**

The winbase.h header defines `Istrcpyn` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[Reference](#)

[StringCbCopy](#)

[StringCbCopyEx](#)

[StringCbCopyN](#)

[StringCbCopyNEx](#)

[StringCbLength](#)

[StringCchCopy](#)

[StringCchCopyEx](#)

[StringCchCopyN](#)

[StringCchCopyNEx](#)

[StringCchLength](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrlen](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrcpyW function (winbase.h)

Article 02/09/2023

Copies a string to a buffer.

Warning Do not use. Consider using [StringCchCopy](#) instead. See Remarks.

Syntax

C++

```
LPWSTR IstrcpyW(  
    [out] LPWSTR  lpString1,  
    [in]  LPCWSTR lpString2  
)
```

Parameters

[out] lpString1

Type: [LPTSTR](#)

A buffer to receive the contents of the string pointed to by the *lpString2* parameter. The buffer must be large enough to contain the string, including the terminating null character.

[in] lpString2

Type: [LPTSTR](#)

The null-terminated string to be copied.

Return value

Type: [LPTSTR](#)

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is **NULL** and *lpString1* may not be null-terminated.

Remarks

With a double-byte character set (DBCS) version of the system, this function can be used to copy a DBCS string.

The `Istrcpy` function has an undefined behavior if source and destination buffers overlap.

Security Remarks

Using this function incorrectly can compromise the security of your application. This function uses structured exception handling (SEH) to catch access violations and other errors. When this function catches SEH errors, it returns **NULL** without null-terminating the string and without notifying the caller of the error. The caller is not safe to assume that insufficient space is the error condition.

lpString1 must be large enough to hold *lpString2* and the closing '\0', otherwise a buffer overrun may occur.

Buffer overflow situations are the cause of many security problems in applications and can cause a denial of service attack against the application if an access violation occurs. In the worst case, a buffer overrun may allow an attacker to inject executable code into your process, especially if *lpString1* is a stack-based buffer.

Consider using [StringCchCopy](#) instead; use either `StringCchCopy(buffer, sizeof(buffer)/sizeof(buffer[0]), src);`, being aware that `buffer` must not be a pointer or use `StringCchCopy(buffer, ARYSIZE(buffer), src);`, being aware that, when copying to a pointer, the caller is responsible for passing in the size of the pointed-to memory in characters.

Note

The `winbase.h` header defines `Istrcpy` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Conceptual

Reference

[StringCbCopy](#)

[StringCbCopyEx](#)

[StringCbCopyN](#)

[StringCbCopyNEx](#)

[StringCchCopy](#)

[StringCchCopyEx](#)

[StringCchCopyN](#)

[StringCchCopyNEx](#)

[Strings](#)

[Istrcmp](#)

[Istrcmpi](#)

[Istrlen](#)

Feedback



Was this page helpful? [!\[\]\(aff6d5e37156edf9d28d615e31b21feb_img.jpg\) Yes](#) [!\[\]\(1f722ce9b58216da0f05a135c98801f2_img.jpg\) No](#)

Get help at Microsoft Q&A

IstrlenA function (winbase.h)

Article 02/09/2023

Determines the length of the specified string (not including the terminating null character).

Syntax

C++

```
int lstrlenA(
    [in] LPCSTR lpString
);
```

Parameters

[in] *lpString*

Type: **LPCTSTR**

The null-terminated string to be checked.

Return value

Type: **int**

The function returns the length of the string, in characters. If *lpString* is **NULL**, the function returns 0.

Remarks

ⓘ Note

The winbase.h header defines `Istrlen` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[Reference](#)

[StringCbLength](#)

[StringCchLength](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrcpy](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

IstrlenW function (winbase.h)

Article 02/09/2023

Determines the length of the specified string (not including the terminating null character).

Syntax

C++

```
int lstrlenW(
    [in] LPCWSTR lpString
);
```

Parameters

[in] *lpString*

Type: **LPCTSTR**

The null-terminated string to be checked.

Return value

Type: **int**

The function returns the length of the string, in characters. If *lpString* is **NULL**, the function returns 0.

Remarks

ⓘ Note

The winbase.h header defines `Istrlen` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Conceptual](#)

[Reference](#)

[StringCbLength](#)

[StringCchLength](#)

[Strings](#)

[lstrcmp](#)

[lstrcmpi](#)

[lstrcpy](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MAKEINTATOM macro (winbase.h)

Article04/02/2021

Converts the specified atom into a string, so it can be passed to functions which accept either atoms or strings.

Syntax

C++

```
void MAKEINTATOM(  
    i  
);
```

Parameters

i

The numeric value to be made into an integer atom. This parameter can be either an integer atom or a string atom.

Return value

None

Remarks

Although the return value of the **MAKEINTATOM** macro is cast as an **LPTSTR** value, it cannot be used as a string pointer except when it is passed to atom-management functions that require an **LPTSTR** argument.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[AddAtom](#)

[DeleteAtom](#)

[GetAtomName](#)

[GlobalAddAtom](#)

[GlobalDeleteAtom](#)

[GlobalGetAtomName](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MapUserPhysicalPagesScatter function (winbase.h)

Article07/27/2022

Maps previously allocated physical memory pages at a specified address in an [Address Windowing Extensions](#) (AWE) region.

64-bit Windows on Itanium-based systems: Due to the difference in page sizes, **MapUserPhysicalPagesScatter** is not supported for 32-bit applications.

Syntax

C++

```
BOOL MapUserPhysicalPagesScatter(
    [in] PVOID      *VirtualAddresses,
    [in] ULONG_PTR  NumberOfPages,
    [in] PULONG_PTR PageArray
);
```

Parameters

[in] *VirtualAddresses*

A pointer to an array of starting addresses of the regions of memory to remap.

Each entry in *VirtualAddresses* must be within the address range that the [VirtualAlloc](#) function returns when the [Address Windowing Extensions](#) (AWE) region is allocated. The value in *NumberOfPages* indicates the size of the array. Entries can be from multiple Address Windowing Extensions (AWE) regions.

[in] *NumberOfPages*

The size of the physical memory and virtual address space for which to establish translations, in pages.

The array at *VirtualAddresses* specifies the virtual address range.

[in] *PageArray*

A pointer to an array of values that indicates how each corresponding page in *VirtualAddresses* should be treated.

A 0 (zero) indicates that the corresponding entry in *VirtualAddresses* should be unmapped, and any nonzero value that it has should be mapped.

If this parameter is **NULL**, then every address in the *VirtualAddresses* array is unmapped.

The value in *NumberOfPages* indicates the size of the array.

Return value

If the function succeeds, the return value is **TRUE**.

If the function fails, the return value is **FALSE**, and the function does not map or unmap —partial or otherwise. To get extended error information, call [GetLastError](#).

Remarks

The physical pages may be unmapped, but they are not freed. You must call [FreeUserPhysicalPages](#) to free the physical pages.

You can specify any number of physical memory pages, but the memory cannot extend outside the virtual address space that is allocated by [VirtualAlloc](#). Any existing address maps are automatically overwritten with the new translations, and the old translations are unmapped.

You cannot map physical memory pages outside the range that is specified in [AllocateUserPhysicalPages](#). You can map multiple regions simultaneously, but they cannot overlap.

Physical pages can be located at any physical address, but do not make assumptions about the contiguity of the physical pages.

In a multiprocessor environment, this function maintains hardware translation buffer coherence. On return from this function, all threads on all processors are guaranteed to see the correct mapping.

To compile an application that uses this function, define the `_WIN32_WINNT` macro as 0x0500 or later. For more information, see [Using the Windows Headers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Address Windowing Extensions](#)

[AllocateUserPhysicalPages](#)

[FreeUserPhysicalPages](#)

[MapUserPhysicalPages](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MapViewOfFileExNuma function (winbase.h)

Article 10/13/2021

Maps a view of a file mapping into the address space of a calling process and specifies the NUMA node for the physical memory.

Syntax

C++

```
LPVOID MapViewOfFileExNuma(
    [in]          HANDLE hFileMappingObject,
    [in]          DWORD  dwDesiredAccess,
    [in]          DWORD  dwFileOffsetHigh,
    [in]          DWORD  dwFileOffsetLow,
    [in]          SIZE_T dwNumberOfBytesToMap,
    [in, optional] LPVOID lpBaseAddress,
    [in]          DWORD  nndPreferred
);
```

Parameters

[in] `hFileMappingObject`

A handle to a file mapping object. The [CreateFileMappingNuma](#) and [OpenFileMapping](#) functions return this handle.

[in] `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
<code>FILE_MAP_ALL_ACCESS</code>	A read/write view of the file is mapped. The file mapping object must have been created with <code>PAGE_READWRITE</code> or <code>PAGE_EXECUTE_READWRITE</code> protection. When used with <code>MapViewOfFileExNuma</code> , <code>FILE_MAP_ALL_ACCESS</code> is equivalent to <code>FILE_MAP_WRITE</code> .

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. When used with MapViewOfFileExNuma , <code>(FILE_MAP_WRITE FILE_MAP_READ)</code> is 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. 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.
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](#).

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](#).

[in] dwFileOffsetHigh

The high-order **DWORD** of the file offset where the view is to begin.

[in] dwFileOffsetLow

The 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.

[in] dwNumberOfBytesToMap

The number of bytes of a file mapping to map to a 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.

[in, optional] lpBaseAddress

A pointer to the memory address in the calling process address space where mapping begins. This must be a multiple of the system's memory allocation granularity, or the function fails. To determine the memory allocation granularity of the system, use the

[GetSystemInfo](#) function. If there is not enough address space at the specified address, the function fails.

If the *lpBaseAddress* parameter is **NULL**, the operating system chooses the mapping address.

While it is possible to specify an address that is safe now (not used by the operating system), there is no guarantee that the address will remain safe over time. Therefore, it is better to let the operating system choose the address. In this case, you would not store pointers in the memory mapped file; you would store offsets from the base of the file mapping so that the mapping can be used at any address.

[in] *nndPreferred*

The NUMA node where the physical memory should reside.

Value	Meaning
NUMA_NO_PREFERRED_NODE 0xffffffff	No NUMA node is preferred. This is the same as calling the MapViewOfFileEx function.

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 the [GetLastError](#) function.

Remarks

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

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

To obtain the size of a view, use the [VirtualQueryEx](#) function.

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

If a suggested mapping address is supplied, the file is mapped at the specified address (rounded down to the nearest 64-KB boundary) if there is enough address space at the

specified address. If there is not enough address space, the function fails.

Typically, the suggested address is used to specify that a file should be mapped at the same address in multiple processes. This requires the region of address space to be available in all involved processes. No other memory allocation can take place in the region that is used for mapping, including the use of the [VirtualAllocExNuma](#) function to reserve memory.

If the *lpBaseAddress* parameter specifies a base offset, the function succeeds if the specified memory region is not already in use by the calling process. The system does not ensure that the same memory region is available for the memory mapped file in other 32-bit processes.

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 the same file mapping object. 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 [MapViewOfFileExNuma](#) 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 being accessed by the [ReadFile](#) or [WriteFile](#) function.

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.

To have a file with executable permissions, an application must call the [CreateFileMappingNuma](#) function with either **PAGE_EXECUTE_READWRITE** or

PAGE_EXECUTE_READ and then call the **MapViewOfFileExNuma** function 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

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFileMappingNuma](#)

[DuplicateHandle](#)

[File Mapping Functions](#)

[GetSystemInfo](#)

[MapViewOfFileEx](#)

[NUMA Support](#)

[OpenFileMapping](#)

[ReadFile](#)

[SYSTEM_INFO](#)

[UnmapViewOfFile](#)

[VirtualAlloc](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MEMORYSTATUS structure (winbase.h)

Article 04/02/2021

Contains information about the current state of both physical and virtual memory. The [GlobalMemoryStatus](#) function stores information in a **MEMORYSTATUS** structure.

Syntax

C++

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    SIZE_T dwTotalPhys;
    SIZE_T dwAvailPhys;
    SIZE_T dwTotalPageFile;
    SIZE_T dwAvailPageFile;
    SIZE_T dwTotalVirtual;
    SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Members

`dwLength`

The size of the **MEMORYSTATUS** data structure, in bytes. You do not need to set this member before calling the [GlobalMemoryStatus](#) function; the function sets it.

`dwMemoryLoad`

A number between 0 and 100 that specifies the approximate percentage of physical memory that is in use (0 indicates no memory use and 100 indicates full memory use).

`dwTotalPhys`

The amount of actual physical memory, in bytes.

`dwAvailPhys`

The amount of physical memory currently available, in bytes. This is the amount of physical memory that can be immediately reused without having to write its contents to disk first. It is the sum of the size of the standby, free, and zero lists.

`dwTotalPageFile`

The current size of the committed memory limit, in bytes. This is physical memory plus the size of the page file, minus a small overhead.

`dwAvailPageFile`

The maximum amount of memory the current process can commit, in bytes. This value should be smaller than the system-wide available commit. To calculate this value, call [GetPerformanceInfo](#) and subtract the value of **CommitTotal** from **CommitLimit**.

`dwTotalVirtual`

The size of the user-mode portion of the virtual address space of the calling process, in bytes. This value depends on the type of process, the type of processor, and the configuration of the operating system. For example, this value is approximately 2 GB for most 32-bit processes on an x86 processor and approximately 3 GB for 32-bit processes that are large address aware running on a system with 4 GT RAM Tuning enabled.

`dwAvailVirtual`

The amount of unreserved and uncommitted memory currently in the user-mode portion of the virtual address space of the calling process, in bytes.

Remarks

MEMORYSTATUS reflects the state of memory at the time of the call. It also reflects the size of the paging file at that time. The operating system can enlarge the paging file up to the maximum size set by the administrator.

On computers with more than 4 GB of memory, the **MEMORYSTATUS** structure can return incorrect information, reporting a value of -1 to indicate an overflow. If your application is at risk for this behavior, use the [GlobalMemoryStatusEx](#) function instead of the [GlobalMemoryStatus](#) function.

Examples

For an example, see the [GlobalMemoryStatus](#) function.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[GlobalMemoryStatus](#)

[GlobalMemoryStatusEx](#)

[Memory Performance Information](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFile function (winbase.h)

Article06/01/2023

Moves an existing file or a directory, including its children.

To specify how to move the file, use the [MoveFileEx](#) or [MoveFileWithProgress](#) function.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFile(
    [in] LPCTSTR lpExistingFileName,
    [in] LPCTSTR lpNewFileName
);
```

Parameters

[in] lpExistingFileName

The current name of the file or directory on the local computer.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpNewFileName

The new name for the file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

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

The **MoveFile** function will move (rename) either a file or a directory (including its children) either in the same directory or across directories. The one caveat is that the **MoveFile** function will fail on directory moves when the destination is on a different volume.

If a file is moved across volumes, **MoveFile** does not move the security descriptor with the file. The file will be assigned the default security descriptor in the destination directory.

The **MoveFile** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

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

SMB 3.0 does not support rename of alternate data streams on file shares with continuous availability capability.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[File Management Functions](#)

[MoveFileEx](#)

[MoveFileTransacted](#)

[MoveFileWithProgress](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFileA function (winbase.h)

Article06/01/2023

Moves an existing file or a directory, including its children.

To specify how to move the file, use the [MoveFileEx](#) or [MoveFileWithProgress](#) function.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFileA(
    [in] LPCSTR lpExistingFileName,
    [in] LPCSTR lpNewFileName
);
```

Parameters

[in] lpExistingFileName

The current name of the file or directory on the local computer.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpNewFileName

The new name for the file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

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

The **MoveFile** function will move (rename) either a file or a directory (including its children) either in the same directory or across directories. The one caveat is that the **MoveFile** function will fail on directory moves when the destination is on a different volume.

If a file is moved across volumes, **MoveFile** does not move the security descriptor with the file. The file will be assigned the default security descriptor in the destination directory.

The **MoveFile** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

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

SMB 3.0 does not support rename of alternate data streams on file shares with continuous availability capability.

 **Note**

The winbase.h header defines MoveFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[File Management Functions](#)

[MoveFileEx](#)

[MoveFileTransacted](#)

[MoveFileWithProgress](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFileExA function (winbase.h)

Article06/01/2023

Moves an existing file or directory, including its children, with various move options.

The [MoveFileWithProgress](#) function is equivalent to the **MoveFileEx** function, except that **MoveFileWithProgress** allows you to provide a callback function that receives progress notifications.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFileExA(
    [in]          LPCSTR lpExistingFileName,
    [in, optional] LPCSTR lpNewFileName,
    [in]          DWORD  dwFlags
);
```

Parameters

`[in] lpExistingFileName`

The current name of the file or directory on the local computer.

If *dwFlags* specifies **MOVEFILE_DELAY_UNTIL_REBOOT**, the file cannot exist on a remote share, because delayed operations are performed before the network is available.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpNewFileName*

The new name of the file or directory on the local computer.

When moving a file, the destination can be on a different file system or volume. If the destination is on another drive, you must set the **MOVEFILE_COPY_ALLOWED** flag in *dwFlags*.

When moving a directory, the destination must be on the same drive.

If *dwFlags* specifies **MOVEFILE_DELAY_UNTIL_REBOOT** and *lpNewFileName* is **NULL**, **MoveFileEx** registers the *lpExistingFileName* file to be deleted when the system restarts. If *lpExistingFileName* refers to a directory, the system removes the directory at restart only if the directory is empty.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] *dwFlags*

This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED 2 (0x2)	If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions. If the file is successfully copied to a different volume and the original file is unable to be deleted, the function succeeds leaving the source file intact.
	This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT .
MOVEFILE_CREATE_HARDLINK 16 (0x10)	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating

4 (0x4)	<p>system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups.</p> <p>This value can be used only if the process is in the context of a user who belongs to the administrators group or the LocalSystem account.</p> <p>This value cannot be used with MOVEFILE_COPY_ALLOWED.</p>
MOVEFILE_FAIL_IF_NOT_TRACKABLE 32 (0x20)	The function fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING 1 (0x1)	<p>If a file named <i>lpNewFileName</i> exists, the function replaces its contents with the contents of the <i>lpExistingFileName</i> file, provided that security requirements regarding access control lists (ACLs) are met. For more information, see the Remarks section of this topic.</p> <p>If <i>lpNewFileName</i> names an existing directory, an error is reported.</p>
MOVEFILE_WRITE_THROUGH 8 (0x8)	<p>The function does not return until the file is actually moved on the disk.</p> <p>Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation.</p> <p>This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.</p>

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

Remarks

If the *dwFlags* parameter specifies **MOVEFILE_DELAY_UNTIL_REBOOT**, **MoveFileEx** fails if it cannot access the registry. The function stores the locations of the files to be renamed at restart in the following registry value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

This registry value is of type **REG_MULTI_SZ**. Each rename operation stores one of the following NULL-terminated strings, depending on whether the rename is a delete or not:

- *szDstFile\0\0*
- *szSrcFile\0szDstFile\0*

The string *szDstFile\0\0* indicates that the file *szDstFile* is to be deleted on reboot. The string *szSrcFile\0szDstFile\0* indicates that *szSrcFile* is to be renamed *szDstFile* on reboot.

Note Although \0\0 is technically not allowed in a **REG_MULTI_SZ** node, it can because the file is considered to be renamed to a null name.

The system uses these registry entries to complete the operations at restart in the same order that they were issued. For example, the following code fragment creates registry entries that delete *szDstFile* and rename *szSrcFile* to be *szDstFile* at restart:

C++

```
MoveFileEx(szDstFile, NULL, MOVEFILE_DELAY_UNTIL_REBOOT);
MoveFileEx(szSrcFile, szDstFile, MOVEFILE_DELAY_UNTIL_REBOOT);
```

Because the actual move and deletion operations specified with the **MOVEFILE_DELAY_UNTIL_REBOOT** flag take place after the calling application has ceased running, the return value cannot reflect success or failure in moving or deleting the file. Rather, it reflects success or failure in placing the appropriate entries into the registry.

The system deletes a directory that is tagged for deletion with the **MOVEFILE_DELAY_UNTIL_REBOOT** flag only if it is empty. To ensure deletion of directories, move or delete all files from the directory before attempting to delete it. Files may be in the directory at boot time, but they must be deleted or moved before the system can delete the directory.

The move and deletion operations are carried out at boot time in the same order that they are specified in the calling application. To delete a directory that has files in it at

boot time, first delete the files.

If a file is moved across volumes, **MoveFileEx** does not move the security descriptor with the file. The file is assigned the default security descriptor in the destination directory.

The **MoveFileEx** function coordinates its operation with the [link tracking](#) service, so link sources can be tracked as they are moved.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and get all the access you request on the handle that is returned to you at the time that you create the file. If you request delete permission at the time you create the file, you can delete or rename the file with that handle but not with any other handle. For more information, see [File Security and Access Rights](#).

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 an example, see [Creating and Using a Temporary 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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[DeleteFile](#)

[File Management Functions](#)

[File Security and Access Rights](#)

[GetWindowsDirectory](#)

[MoveFileTransacted](#)

[MoveFileWithProgress](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFileExW function (winbase.h)

Article06/01/2023

Moves an existing file or directory, including its children, with various move options.

The [MoveFileWithProgress](#) function is equivalent to the **MoveFileEx** function, except that **MoveFileWithProgress** allows you to provide a callback function that receives progress notifications.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFileExW(
    [in]          LPCWSTR lpExistingFileName,
    [in, optional] LPCWSTR lpNewFileName,
    [in]          DWORD   dwFlags
);
```

Parameters

[in] `lpExistingFileName`

The current name of the file or directory on the local computer.

If `dwFlags` specifies **MOVEFILE_DELAY_UNTIL_REBOOT**, the file cannot exist on a remote share, because delayed operations are performed before the network is available.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpNewFileName*

The new name of the file or directory on the local computer.

When moving a file, the destination can be on a different file system or volume. If the destination is on another drive, you must set the **MOVEFILE_COPY_ALLOWED** flag in *dwFlags*.

When moving a directory, the destination must be on the same drive.

If *dwFlags* specifies **MOVEFILE_DELAY_UNTIL_REBOOT** and *lpNewFileName* is **NULL**, **MoveFileEx** registers the *lpExistingFileName* file to be deleted when the system restarts. If *lpExistingFileName* refers to a directory, the system removes the directory at restart only if the directory is empty.

In the ANSI version of this function, the name is limited to **MAX_PATH** characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\?" to the path. For more information, see [Naming a File](#)

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the **MAX_PATH** limitation without prepending "\?\\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] *dwFlags*

This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED 2 (0x2)	If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions. If the file is successfully copied to a different volume and the original file is unable to be deleted, the function succeeds leaving the source file intact.
	This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT .
MOVEFILE_CREATE_HARDLINK 16 (0x10)	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating

4 (0x4)	<p>system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups.</p> <p>This value can be used only if the process is in the context of a user who belongs to the administrators group or the LocalSystem account.</p> <p>This value cannot be used with MOVEFILE_COPY_ALLOWED.</p>
MOVEFILE_FAIL_IF_NOT_TRACKABLE 32 (0x20)	The function fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING 1 (0x1)	<p>If a file named <i>lpNewFileName</i> exists, the function replaces its contents with the contents of the <i>lpExistingFileName</i> file, provided that security requirements regarding access control lists (ACLs) are met. For more information, see the Remarks section of this topic.</p> <p>If <i>lpNewFileName</i> names an existing directory, an error is reported.</p>
MOVEFILE_WRITE_THROUGH 8 (0x8)	<p>The function does not return until the file is actually moved on the disk.</p> <p>Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation.</p> <p>This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.</p>

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero (0). To get extended error information, call [GetLastError](#).

Remarks

If the *dwFlags* parameter specifies **MOVEFILE_DELAY_UNTIL_REBOOT**, **MoveFileEx** fails if it cannot access the registry. The function stores the locations of the files to be renamed at restart in the following registry value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

This registry value is of type **REG_MULTI_SZ**. Each rename operation stores one of the following NULL-terminated strings, depending on whether the rename is a delete or not:

- *szDstFile\0\0*
- *szSrcFile\0szDstFile\0*

The string *szDstFile\0\0* indicates that the file *szDstFile* is to be deleted on reboot. The string *szSrcFile\0szDstFile\0* indicates that *szSrcFile* is to be renamed *szDstFile* on reboot.

Note Although \0\0 is technically not allowed in a **REG_MULTI_SZ** node, it can because the file is considered to be renamed to a null name.

The system uses these registry entries to complete the operations at restart in the same order that they were issued. For example, the following code fragment creates registry entries that delete *szDstFile* and rename *szSrcFile* to be *szDstFile* at restart:

C++

```
MoveFileEx(szDstFile, NULL, MOVEFILE_DELAY_UNTIL_REBOOT);
MoveFileEx(szSrcFile, szDstFile, MOVEFILE_DELAY_UNTIL_REBOOT);
```

Because the actual move and deletion operations specified with the **MOVEFILE_DELAY_UNTIL_REBOOT** flag take place after the calling application has ceased running, the return value cannot reflect success or failure in moving or deleting the file. Rather, it reflects success or failure in placing the appropriate entries into the registry.

The system deletes a directory that is tagged for deletion with the **MOVEFILE_DELAY_UNTIL_REBOOT** flag only if it is empty. To ensure deletion of directories, move or delete all files from the directory before attempting to delete it. Files may be in the directory at boot time, but they must be deleted or moved before the system can delete the directory.

The move and deletion operations are carried out at boot time in the same order that they are specified in the calling application. To delete a directory that has files in it at

boot time, first delete the files.

If a file is moved across volumes, **MoveFileEx** does not move the security descriptor with the file. The file is assigned the default security descriptor in the destination directory.

The **MoveFileEx** function coordinates its operation with the [link tracking](#) service, so link sources can be tracked as they are moved.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and get all the access you request on the handle that is returned to you at the time that you create the file. If you request delete permission at the time you create the file, you can delete or rename the file with that handle but not with any other handle. For more information, see [File Security and Access Rights](#).

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 an example, see [Creating and Using a Temporary 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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[DeleteFile](#)

[File Management Functions](#)

[File Security and Access Rights](#)

[GetWindowsDirectory](#)

[MoveFileTransacted](#)

[MoveFileWithProgress](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFileTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Moves an existing file or a directory, including its children, as a transacted operation.

Syntax

C++

```
BOOL MoveFileTransactedA(
    [in]           LPCSTR          lpExistingFileName,
    [in, optional] LPCSTR          lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID          lpData,
    [in]           DWORD           dwFlags,
    [in]           HANDLE          hTransaction
);
```

Parameters

`[in] lpExistingFileName`

The current name of the existing file or directory on the local computer.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] lpNewFileName

The new name for the file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] lpProgressRoutine

A pointer to a [CopyProgressRoutine](#) callback function that is called each time another portion of the file has been moved. The callback function can be useful if you provide a user interface that displays the progress of the operation. This parameter can be NULL.

[in, optional] lpData

An argument to be passed to the [CopyProgressRoutine](#) callback function. This parameter can be NULL.

[in] dwFlags

The move options. This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED 2 (0x2)	If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions. If the file is successfully copied to a different volume and the original file is unable to be deleted, the function succeeds leaving the source file intact.
	This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT .
MOVEFILE_CREATE_HARDLINK 16 (0x10)	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating

4 (0x4)	<p>system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups.</p> <p>This value can only be used if the process is in the context of a user who belongs to the administrators group or the LocalSystem account.</p> <p>This value cannot be used with MOVEFILE_COPY_ALLOWED.</p> <p>The write operation to the registry value as detailed in the Remarks section is what is transacted. The file move is finished when the computer restarts, after the transaction is complete.</p>
MOVEFILE_REPLACE_EXISTING 1 (0x1)	<p>If a file named <i>lpNewFileName</i> exists, the function replaces its contents with the contents of the <i>lpExistingFileName</i> file.</p> <p>This value cannot be used if <i>lpNewFileName</i> or <i>lpExistingFileName</i> names a directory.</p>
MOVEFILE_WRITE_THROUGH 8 (0x8)	<p>A call to MoveFileTransacted means that the move file operation is complete when the commit operation is completed. This flag is unnecessary; there are no negative affects if this flag is specified, other than an operation slowdown. The function does not return until the file has actually been moved on the disk.</p> <p>Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation.</p> <p>This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.</p>

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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](#).

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **MoveFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **MoveFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

Remarks

If the *dwFlags* parameter specifies **MOVEFILE_DELAY_UNTIL_REBOOT**, **MoveFileTransacted** fails if it cannot access the registry. The function transactionally stores the locations of the files to be renamed at restart in the following registry value:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

This registry value is of type **REG_MULTI_SZ**. Each rename operation stores one of the following NULL-terminated strings, depending on whether the rename is a delete or not:

szDstFile\0\0

szSrcFile\0szDstFile\0

The string *szDstFile\0\0* indicates that the file *szDstFile* is to be deleted on reboot.

The string *szSrcFile\0szDstFile\0* indicates that *szSrcFile* is to be renamed *szDstFile* on reboot.

Note Although \0\0 is technically not allowed in a **REG_MULTI_SZ** node, it can because the file is considered to be renamed to a null name.

The system uses these registry entries to complete the operations at restart in the same order that they were issued. For more information about using the **MOVEFILE_DELAY_UNTIL_REBOOT** flag, see [MoveFileWithProgress](#).

If a file is moved across volumes, **MoveFileTransacted** does not move the security descriptor with the file. The file is assigned the default security descriptor in the destination directory.

This function always fails if you specify the **MOVEFILE_FAIL_IF_NOT_TRACKABLE** flag; tracking is not supported by TxF.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileTransacted](#)

[File Management Functions](#)

[MoveFileWithProgress](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

MoveFileTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS.](#)]

Moves an existing file or a directory, including its children, as a transacted operation.

Syntax

C++

```
BOOL MoveFileTransactedW(
    [in]           LPCWSTR          lpExistingFileName,
    [in, optional] LPCWSTR          lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID            lpData,
    [in]           DWORD             dwFlags,
    [in]           HANDLE            hTransaction
);
```

Parameters

`[in] lpExistingFileName`

The current name of the existing file or directory on the local computer.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] lpNewFileName

The new name for the file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] lpProgressRoutine

A pointer to a [CopyProgressRoutine](#) callback function that is called each time another portion of the file has been moved. The callback function can be useful if you provide a user interface that displays the progress of the operation. This parameter can be NULL.

[in, optional] lpData

An argument to be passed to the [CopyProgressRoutine](#) callback function. This parameter can be NULL.

[in] dwFlags

The move options. This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED 2 (0x2)	If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions. If the file is successfully copied to a different volume and the original file is unable to be deleted, the function succeeds leaving the source file intact.
	This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT .
MOVEFILE_CREATE_HARDLINK 16 (0x10)	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating

4 (0x4)	<p>system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups.</p> <p>This value can only be used if the process is in the context of a user who belongs to the administrators group or the LocalSystem account.</p> <p>This value cannot be used with MOVEFILE_COPY_ALLOWED.</p> <p>The write operation to the registry value as detailed in the Remarks section is what is transacted. The file move is finished when the computer restarts, after the transaction is complete.</p>
MOVEFILE_REPLACE_EXISTING 1 (0x1)	<p>If a file named <i>lpNewFileName</i> exists, the function replaces its contents with the contents of the <i>lpExistingFileName</i> file.</p> <p>This value cannot be used if <i>lpNewFileName</i> or <i>lpExistingFileName</i> names a directory.</p>
MOVEFILE_WRITE_THROUGH 8 (0x8)	<p>A call to MoveFileTransacted means that the move file operation is complete when the commit operation is completed. This flag is unnecessary; there are no negative affects if this flag is specified, other than an operation slowdown. The function does not return until the file has actually been moved on the disk.</p> <p>Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation.</p> <p>This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.</p>

[in] `hTransaction`

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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](#).

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **MoveFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **MoveFileTransacted** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

Remarks

If the *dwFlags* parameter specifies **MOVEFILE_DELAY_UNTIL_REBOOT**, **MoveFileTransacted** fails if it cannot access the registry. The function transactionally stores the locations of the files to be renamed at restart in the following registry value:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

This registry value is of type **REG_MULTI_SZ**. Each rename operation stores one of the following NULL-terminated strings, depending on whether the rename is a delete or not:

szDstFile\0\0

szSrcFile\0szDstFile\0

The string *szDstFile\0\0* indicates that the file *szDstFile* is to be deleted on reboot.

The string *szSrcFile\0szDstFile\0* indicates that *szSrcFile* is to be renamed *szDstFile* on reboot.

Note Although \0\0 is technically not allowed in a **REG_MULTI_SZ** node, it can because the file is considered to be renamed to a null name.

The system uses these registry entries to complete the operations at restart in the same order that they were issued. For more information about using the **MOVEFILE_DELAY_UNTIL_REBOOT** flag, see [MoveFileWithProgress](#).

If a file is moved across volumes, **MoveFileTransacted** does not move the security descriptor with the file. The file is assigned the default security descriptor in the destination directory.

This function always fails if you specify the **MOVEFILE_FAIL_IF_NOT_TRACKABLE** flag; tracking is not supported by TxF.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileTransacted](#)

[File Management Functions](#)

[MoveFileWithProgress](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

MoveFileW function (winbase.h)

Article06/01/2023

Moves an existing file or a directory, including its children.

To specify how to move the file, use the [MoveFileEx](#) or [MoveFileWithProgress](#) function.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFileW(
    [in] LPCWSTR lpExistingFileName,
    [in] LPCWSTR lpNewFileName
);
```

Parameters

[in] lpExistingFileName

The current name of the file or directory on the local computer.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] lpNewFileName

The new name for the file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

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

The **MoveFile** function will move (rename) either a file or a directory (including its children) either in the same directory or across directories. The one caveat is that the **MoveFile** function will fail on directory moves when the destination is on a different volume.

If a file is moved across volumes, **MoveFile** does not move the security descriptor with the file. The file will be assigned the default security descriptor in the destination directory.

The **MoveFile** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

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

SMB 3.0 does not support rename of alternate data streams on file shares with continuous availability capability.

 **Note**

The winbase.h header defines MoveFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFile](#)

[File Management Functions](#)

[MoveFileEx](#)

[MoveFileTransacted](#)

[MoveFileWithProgress](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFileWithProgressA function (winbase.h)

Article06/01/2023

Moves a file or directory, including its children. You can provide a callback function that receives progress notifications.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFileWithProgressA(
    [in]           LPCSTR      lpExistingFileName,
    [in, optional] LPCSTR      lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID       lpData,
    [in]           DWORD        dwFlags
);
```

Parameters

[in] `lpExistingFileName`

The name of the existing file or directory on the local computer.

If `dwFlags` specifies `MOVEFILE_DELAY_UNTIL_REBOOT`, the file cannot exist on a remote share because delayed operations are performed before the network is available.

By default, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, prepend "`\?\`" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the `MAX_PATH` limitation without prepending "`\?\`". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpNewFileName*

The new name of the file or directory on the local computer.

When moving a file, *lpNewFileName* can be on a different file system or volume. If *lpNewFileName* is on another drive, you must set the **MOVEFILE_COPY_ALLOWED** flag in *dwFlags*.

When moving a directory, *lpExistingFileName* and *lpNewFileName* must be on the same drive.

If *dwFlags* specifies **MOVEFILE_DELAY_UNTIL_REBOOT** and *lpNewFileName* is **NULL**, **MoveFileWithProgress** registers *lpExistingFileName* to be deleted when the system restarts. The function fails if it cannot access the registry to store the information about the delete operation. If *lpExistingFileName* refers to a directory, the system removes the directory at restart only if the directory is empty.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpProgressRoutine*

A pointer to a [CopyProgressRoutine](#) callback function that is called each time another portion of the file has been moved. The callback function can be useful if you provide a user interface that displays the progress of the operation. This parameter can be **NULL**.

[in, optional] *lpData*

An argument to be passed to the [CopyProgressRoutine](#) callback function. This parameter can be **NULL**.

[in] *dwFlags*

The move options. This parameter can be one or more of the following values.

Value	Meaning

MOVEFILE_COPY_ALLOWED 2 (0x2)	If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions.
	If the file is successfully copied to a different volume and the original file is unable to be deleted, the function succeeds leaving the source file intact.
	This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT .
MOVEFILE_CREATE_HARDLINK 16 (0x10)	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT 4 (0x4)	<p>The system does not move the file until the operating system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups.</p> <p>This value can only be used if the process is in the context of a user who belongs to the administrators group or the LocalSystem account.</p>
	This value cannot be used with MOVEFILE_COPY_ALLOWED .
MOVEFILE_FAIL_IF_NOT_TRACKABLE 32 (0x20)	The function fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING 1 (0x1)	<p>If a file named <i>lpNewFileName</i> exists, the function replaces its contents with the contents of the <i>lpExistingFileName</i> file.</p> <p>This value cannot be used if <i>lpNewFileName</i> or <i>lpExistingFileName</i> names a directory.</p>
MOVEFILE_WRITE_THROUGH 8 (0x8)	<p>The function does not return until the file has actually been moved on the disk.</p> <p>Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation.</p>
	This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.

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](#).

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **MoveFileWithProgress** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **MoveFileWithProgress** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

Remarks

The **MoveFileWithProgress** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

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

CsvFs will do redirected IO for compressed files.

 **Note**

The winbase.h header defines MoveFileWithProgress as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileEx](#)

[CopyProgressRoutine](#)

[File Management Functions](#)

[MoveFileEx](#)

[MoveFileTransacted](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MoveFileWithProgressW function (winbase.h)

Article06/01/2023

Moves a file or directory, including its children. You can provide a callback function that receives progress notifications.

To perform this operation as a transacted operation, use the [MoveFileTransacted](#) function.

Syntax

C++

```
BOOL MoveFileWithProgressW(
    [in]           LPCWSTR      lpExistingFileName,
    [in, optional] LPCWSTR      lpNewFileName,
    [in, optional] LPPROGRESS_ROUTINE lpProgressRoutine,
    [in, optional] LPVOID       lpData,
    [in]           DWORD        dwFlags
);
```

Parameters

[in] `lpExistingFileName`

The name of the existing file or directory on the local computer.

If `dwFlags` specifies `MOVEFILE_DELAY_UNTIL_REBOOT`, the file cannot exist on a remote share because delayed operations are performed before the network is available.

By default, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, prepend "`\?\`" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the `MAX_PATH` limitation without prepending "`\?\`". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpNewFileName*

The new name of the file or directory on the local computer.

When moving a file, *lpNewFileName* can be on a different file system or volume. If *lpNewFileName* is on another drive, you must set the **MOVEFILE_COPY_ALLOWED** flag in *dwFlags*.

When moving a directory, *lpExistingFileName* and *lpNewFileName* must be on the same drive.

If *dwFlags* specifies **MOVEFILE_DELAY_UNTIL_REBOOT** and *lpNewFileName* is **NULL**, **MoveFileWithProgress** registers *lpExistingFileName* to be deleted when the system restarts. The function fails if it cannot access the registry to store the information about the delete operation. If *lpExistingFileName* refers to a directory, the system removes the directory at restart only if the directory is empty.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in, optional] *lpProgressRoutine*

A pointer to a [CopyProgressRoutine](#) callback function that is called each time another portion of the file has been moved. The callback function can be useful if you provide a user interface that displays the progress of the operation. This parameter can be **NULL**.

[in, optional] *lpData*

An argument to be passed to the [CopyProgressRoutine](#) callback function. This parameter can be **NULL**.

[in] *dwFlags*

The move options. This parameter can be one or more of the following values.

Value	Meaning

MOVEFILE_COPY_ALLOWED 2 (0x2)	If the file is to be moved to a different volume, the function simulates the move by using the CopyFile and DeleteFile functions.
	If the file is successfully copied to a different volume and the original file is unable to be deleted, the function succeeds leaving the source file intact.
	This value cannot be used with MOVEFILE_DELAY_UNTIL_REBOOT .
MOVEFILE_CREATE_HARDLINK 16 (0x10)	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT 4 (0x4)	<p>The system does not move the file until the operating system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups.</p> <p>This value can only be used if the process is in the context of a user who belongs to the administrators group or the LocalSystem account.</p>
	This value cannot be used with MOVEFILE_COPY_ALLOWED .
MOVEFILE_FAIL_IF_NOT_TRACKABLE 32 (0x20)	The function fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING 1 (0x1)	<p>If a file named <i>lpNewFileName</i> exists, the function replaces its contents with the contents of the <i>lpExistingFileName</i> file.</p> <p>This value cannot be used if <i>lpNewFileName</i> or <i>lpExistingFileName</i> names a directory.</p>
MOVEFILE_WRITE_THROUGH 8 (0x8)	<p>The function does not return until the file has actually been moved on the disk.</p> <p>Setting this value guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation.</p>
	This value has no effect if MOVEFILE_DELAY_UNTIL_REBOOT is set.

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](#).

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_CANCEL** due to the user canceling the operation, **MoveFileWithProgress** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

When moving a file across volumes, if *lpProgressRoutine* returns **PROGRESS_STOP** due to the user stopping the operation, **MoveFileWithProgress** will return zero and [GetLastError](#) will return **ERROR_REQUEST_ABORTED**. The existing file is left intact.

Remarks

The **MoveFileWithProgress** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

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

CsvFs will do redirected IO for compressed files.

 **Note**

The winbase.h header defines MoveFileWithProgress as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CopyFileEx](#)

[CopyProgressRoutine](#)

[File Management Functions](#)

[MoveFileEx](#)

[MoveFileTransacted](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

MulDiv function (winbase.h)

Article 10/13/2021

Multiplies two 32-bit values and then divides the 64-bit result by a third 32-bit value. The final result is rounded to the nearest integer.

Syntax

C++

```
int MulDiv(
    [in] int nNumber,
    [in] int nNumerator,
    [in] int nDenominator
);
```

Parameters

[in] `nNumber`

The multiplicand.

[in] `nNumerator`

The multiplier.

[in] `nDenominator`

The number by which the result of the multiplication operation is to be divided.

Return value

If the function succeeds, the return value is the result of the multiplication and division, rounded to the nearest integer. If the result is a positive half integer (ends in .5), it is rounded up. If the result is a negative half integer, it is rounded down.

If either an overflow occurred or `nDenominator` was 0, the return value is -1.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Int32x32To64](#)

[Large Integers](#)

[UInt32x32To64](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

NotifyChangeEventLog function (winbase.h)

Article10/13/2021

Enables an application to receive notification when an event is written to the specified event log. When the event is written to the log, the specified event object is set to the signaled state.

Syntax

C++

```
BOOL NotifyChangeEventLog(  
    [in] HANDLE hEventLog,  
    [in] HANDLE hEvent  
);
```

Parameters

[in] *hEventLog*

A handle to an event log. The [OpenEventLog](#) function returns this handle.

[in] *hEvent*

A handle to a manual-reset or auto-reset event object. Use the [CreateEvent](#) function to create the event object.

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

The **NotifyChangeEventLog** function does not work with remote handles. If the *hEventLog* parameter is the handle to an event log on a remote computer,

`NotifyChangeEventLog` returns zero, and `GetLastError` returns `ERROR_INVALID_HANDLE`.

If the thread is not waiting on the event when the system calls `PulseEvent`, the thread will not receive the notification. Therefore, you should create a separate thread to wait for notifications.

The system will continue to notify you of changes until you close the handle to the event log. To close the event log, use the [CloseEventLog](#) or [DeregisterEventSource](#) function.

Examples

For an example, see [Receiving Event Notification](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-l1-1-1 (introduced in Windows 10, version 10.0.10240)

See also

[CloseEventLog](#)

[CreateEvent](#)

[DeregisterEventSource](#)

[Event Logging Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ObjectCloseAuditAlarmA function (winbase.h)

Article07/27/2022

The **ObjectCloseAuditAlarm** function generates an audit message in the security event log when a handle to a private object is deleted. Alarms are not currently supported.

Syntax

C++

```
BOOL ObjectCloseAuditAlarmA(
    [in] LPCSTR SubsystemName,
    [in] LPVOID HandleId,
    [in] BOOL    GenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string specifying the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in] HandleId

A unique value representing the client's handle to the object. This should be the same value that was passed to the [AccessCheckAndAuditAlarm](#) or [ObjectOpenAuditAlarm](#) function.

[in] GenerateOnClose

Specifies a flag set by a call to the [AccessCheckAndAuditAlarm](#) or [ObjectCloseAuditAlarm](#) function when the object handle is created. If this flag is **TRUE**, the function generates an audit message. If it is **FALSE**, the function does not generate an audit message.

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

The **ObjectCloseAuditAlarm** function requires the calling application to have the SE_AUDIT_NAME privilege enabled. The test for this privilege is always performed against the [primary token](#) of the calling [process](#), allowing the calling process to impersonate a client.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AccessCheckAndAuditAlarm](#)

[Client/Server Access Control Functions](#)

[Client/Server Access Control Overview](#)

[ObjectDeleteAuditAlarm](#)

[ObjectOpenAuditAlarm](#)

[ObjectPrivilegeAuditAlarm](#)

Feedback



Was this page helpful? [Yes](#) | [No](#)

Get help at Microsoft Q&A

ObjectDeleteAuditAlarmA function (winbase.h)

Article09/22/2022

The **ObjectDeleteAuditAlarm** function generates audit messages when an object is deleted. Alarms are not currently supported.

Syntax

C++

```
BOOL ObjectDeleteAuditAlarmA(
    [in] LPCSTR SubsystemName,
    [in] LPVOID HandleId,
    [in] BOOL    GenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string specifying the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in] HandleId

Specifies a unique value representing the client's handle to the object. This must be the same value that was passed to the [AccessCheckAndAuditAlarm](#) or [ObjectOpenAuditAlarm](#) function.

[in] GenerateOnClose

Specifies a flag set by a call to the [AccessCheckAndAuditAlarm](#) or [ObjectOpenAuditAlarm](#) function when the object handle is created.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The **ObjectDeleteAuditAlarm** function requires the calling application to have the SE_AUDIT_NAME privilege enabled. The test for this privilege is always performed against the [primary token](#) of the calling [process](#), allowing the calling process to impersonate a client.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AccessCheck](#)

[AccessCheckAndAuditAlarm](#)

[AreAllAccessesGranted](#)

[AreAnyAccessesGranted](#)

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[MapGenericMask](#)

[ObjectCloseAuditAlarm](#)

[ObjectOpenAuditAlarm](#)

[ObjectPrivilegeAuditAlarm](#)

[PrivilegeCheck](#)

[PrivilegedServiceAuditAlarm](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ObjectOpenAuditAlarmA function (winbase.h)

Article07/27/2022

The **ObjectOpenAuditAlarm** function generates audit messages when a client application attempts to gain access to an object or to create a new one. Alarms are not currently supported.

Syntax

C++

```
BOOL ObjectOpenAuditAlarmA(
    [in]          LPCSTR             SubsystemName,
    [in]          LPVOID              HandleId,
    [in]          LPSTR               ObjectTypeName,
    [in, optional] LPSTR               ObjectName,
    [in]          PSECURITY_DESCRIPTOR pSecurityDescriptor,
    [in]          HANDLE              ClientToken,
    [in]          DWORD               DesiredAccess,
    [in]          DWORD               GrantedAccess,
    [in, optional] PPRIVILEGE_SET     Privileges,
    [in]          BOOL                ObjectCreation,
    [in]          BOOL                AccessGranted,
    [out]         LPBOOL              GenerateOnClose
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string specifying the name of the subsystem calling the function. This string appears in any audit message that the function generates.

[in] HandleId

A pointer to a unique value representing the client's handle to the object. If the access is denied, this parameter is ignored.

For cross-platform compatibility, the value addressed by this pointer must be sizeof(LPVOID) bytes long.

[in] ObjectTypeName

A pointer to a **null**-terminated string specifying the type of object to which the client is requesting access. This string appears in any audit message that the function generates.

[in, optional] ObjectName

A pointer to a **null**-terminated string specifying the name of the object to which the client is requesting access. This string appears in any audit message that the function generates.

[in] pSecurityDescriptor

A pointer to the [SECURITY_DESCRIPTOR](#) structure for the object being accessed.

[in] ClientToken

Identifies an [access token](#) representing the client requesting the operation. This handle must be obtained by opening the token of a thread impersonating the client. The token must be open for TOKEN_QUERY access.

[in] DesiredAccess

Specifies the desired [access mask](#). This mask must have been previously mapped by the [MapGenericMask](#) function to contain no generic access rights.

[in] GrantedAccess

Specifies an [access mask](#) indicating which access rights are granted. This access mask is intended to be the same value set by one of the access-checking functions in its GrantedAccess parameter. Examples of access-checking functions include [AccessCheckAndAuditAlarm](#) and [AccessCheck](#).

[in, optional] Privileges

A pointer to a [PRIVILEGE_SET](#) structure that specifies the set of [privileges](#) required for the access attempt. This parameter can be **NULL**.

[in] ObjectCreation

Specifies a flag that determines whether the application creates a new object when access is granted. When this value is **TRUE**, the application creates a new object; when it is **FALSE**, the application opens an existing object.

[in] AccessGranted

Specifies a flag indicating whether access was granted or denied in a previous call to an access-checking function, such as [AccessCheck](#). If access was granted, this value is **TRUE**.

If not, it is FALSE.

[out] GenerateOnClose

A pointer to a flag set by the audit-generation routine when the function returns. This value must be passed to the [ObjectCloseAuditAlarm](#) function when the object handle is closed.

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

The [ObjectOpenAuditAlarm](#) function requires the calling application to have the SE_AUDIT_NAME privilege enabled. The test for this privilege is always performed against the [primary token](#) of the calling [process](#), not the [impersonation token](#) of the thread. This allows the calling process to impersonate a client during the call.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AccessCheck](#)

[AccessCheckAndAuditAlarm](#)

[AreAllAccessesGranted](#)

[AreAnyAccessesGranted](#)

[Client/Server Access Control](#)

[Client/Server Access Control Functions](#)

[MapGenericMask](#)

[ObjectCloseAuditAlarm](#)

[ObjectDeleteAuditAlarm](#)

[ObjectPrivilegeAuditAlarm](#)

[PRIVILEGE_SET](#)

[PrivilegeCheck](#)

[PrivilegedServiceAuditAlarm](#)

[SECURITY_DESCRIPTOR](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ObjectPrivilegeAuditAlarmA function (winbase.h)

Article07/27/2022

The **ObjectPrivilegeAuditAlarm** function generates an audit message in the security event log. A protected server can use this function to log attempts by a client to use a specified set of [privileges](#) with an open handle to a private object. Alarms are not currently supported.

Syntax

C++

```
BOOL ObjectPrivilegeAuditAlarmA(
    [in] LPCSTR          SubsystemName,
    [in] LPVOID           HandleId,
    [in] HANDLE           ClientToken,
    [in] DWORD            DesiredAccess,
    [in] PPRIVILEGE_SET  Privileges,
    [in] BOOL             AccessGranted
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string specifying the name of the subsystem calling the function. This string appears in the audit message.

[in] HandleId

A pointer to a unique value representing the client's handle to the object.

[in] ClientToken

Identifies an [access token](#) representing the client that requested the operation. This handle must have been obtained by opening the token of a thread impersonating the client. The token must be open for TOKEN_QUERY access. The function uses this token to get the identity of the client for the audit message.

[in] DesiredAccess

Specifies an [access mask](#) indicating the privileged access types being used or whose use is being attempted. The access mask can be mapped by the [MapGenericMask](#) function so it does not contain any generic access types.

[in] **Privileges**

A pointer to a [PRIVILEGE_SET](#) structure containing the [privileges](#) that the client attempted to use. The names of the privileges appear in the audit message.

[in] **AccessGranted**

Indicates whether the client's attempt to use the privileges was successful. If this value is **TRUE**, the audit message indicates success. If this value is **FALSE**, the audit message indicates failure.

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

The [ObjectPrivilegeAuditAlarm](#) function does not check the client's access to the object or check the client's access token to determine whether the privileges are held or enabled. Typically, you call the [PrivilegeCheck](#) function to determine whether the specified privileges are enabled in the access token, call the [AccessCheck](#) function to check the client's access to the object, and then call [ObjectPrivilegeAuditAlarm](#) to log the results.

The [ObjectPrivilegeAuditAlarm](#) function requires the calling [process](#) to have **SE_AUDIT_NAME** privilege enabled. The test for this privilege is always performed against the [primary token](#) of the calling process, not the [impersonation token](#) of the thread. This allows the calling process to impersonate a client during the call.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[AccessCheck](#)

[AccessCheckAndAuditAlarm](#)

[Client/Server Access Control Functions](#)

[Client/Server Access Control Overview](#)

[MapGenericMask](#)

[ObjectCloseAuditAlarm](#)

[ObjectOpenAuditAlarm](#)

[PRIVILEGE_SET](#)

[PrivilegeCheck](#)

[PrivilegedServiceAuditAlarm](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OFSTRUCT structure (winbase.h)

Article09/01/2022

Contains information about a file that the [OpenFile](#) function opened or attempted to open.

Syntax

C++

```
typedef struct _OFSSTRUCT {
    BYTE cBytes;
    BYTE fFixedDisk;
    WORD nErrCode;
    WORD Reserved1;
    WORD Reserved2;
    CHAR szPathName[OFS_MAXPATHNAME];
} OFSTRUCT, *LPOFSTRUCT, *POFSTRUCT;
```

Members

cBytes

The size of the structure, in bytes.

fFixedDisk

If this member is nonzero, the file is on a hard (fixed) disk. Otherwise, it is not.

nErrCode

The MS-DOS error code if the [OpenFile](#) function failed.

Reserved1

Reserved; do not use.

Reserved2

Reserved; do not use.

szPathName[OFS_MAXPATHNAME]

The path and file name of the file.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[OpenFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenBackupEventLogA function (winbase.h)

Article02/09/2023

Opens a handle to a backup event log created by the [BackupEventLog](#) function.

Syntax

C++

```
HANDLE OpenBackupEventLogA(
    [in] LPCSTR lpUNCServerName,
    [in] LPCSTR lpFileName
);
```

Parameters

[in] `lpUNCServerName`

The Universal Naming Convention (UNC) name of the remote server on which this operation is to be performed. If this parameter is **NULL**, the local computer is used.

[in] `lpFileName`

The full path of the backup file.

Return value

If the function succeeds, the return value is a handle to the backup event log.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

If the backup filename specifies a remote server, the `lpUNCServerName` parameter must be **NULL**.

When this function is used on Windows Vista and later computers, only backup event logs that were saved with the [BackupEventLog](#) function on Windows Vista and later

computers can be opened.

ⓘ Note

The winbase.h header defines OpenBackupEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[BackupEventLog](#)

[Event Logging Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenBackupEventLogW function (winbase.h)

Article02/09/2023

Opens a handle to a backup event log created by the [BackupEventLog](#) function.

Syntax

C++

```
HANDLE OpenBackupEventLog(
    [in] LPCWSTR lpUNCServerName,
    [in] LPCWSTR lpFileName
);
```

Parameters

[in] `lpUNCServerName`

The Universal Naming Convention (UNC) name of the remote server on which this operation is to be performed. If this parameter is **NULL**, the local computer is used.

[in] `lpFileName`

The full path of the backup file.

Return value

If the function succeeds, the return value is a handle to the backup event log.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

If the backup filename specifies a remote server, the `lpUNCServerName` parameter must be **NULL**.

When this function is used on Windows Vista and later computers, only backup event logs that were saved with the [BackupEventLog](#) function on Windows Vista and later

computers can be opened.

ⓘ Note

The winbase.h header defines OpenBackupEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[BackupEventLog](#)

[Event Logging Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenCommPort function (winbase.h)

Article10/13/2021

Attempts to open a communication device.

Syntax

C++

```
HANDLE OpenCommPort(
    [in] ULONG uPortNumber,
    [in] DWORD dwDesiredAccess,
    [in] DWORD dwFlagsAndAttributes
);
```

Parameters

[in] uPortNumber

A one-based port number for the communication device to open.

[in] dwDesiredAccess

The requested access to the device.

For more information about requested access, see [CreateFile](#) and [Creating and Opening Files](#).

[in] dwFlagsAndAttributes

The requested flags and attributes to the device.

Note

For this function, only values of `FILE_FLAG_OVERLAPPED` or `0x0` are expected for this parameter.

Value	Meaning
<code>FILE_FLAG_OVERLAPPED</code> <code>0x40000000</code>	The file or device is being opened or created for asynchronous I/O.

Return value

If the function succeeds, the function returns a valid **HANDLE**. Use [CloseHandle](#) to close that handle.

If an error occurs, the function returns **INVALID_HANDLE_VALUE**.

Remarks

The *uPortNumber* parameter accepts one-based values. A value of 1 for *uPortNumber* causes this function to attempt to open COM1.

To support UWP, link against WindowsApp.lib.

Requirements

Minimum supported client	Windows 10, version 1709 [desktop apps UWP apps]
Minimum supported server	Windows Server, version 1709 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	OneCore.lib
DLL	KernelBase.dll

See also

[CloseHandle](#)

[CreateFile](#)

[Creating and Opening Files](#)

Feedback



Was this page helpful? Yes No

Get help at Microsoft Q&A

OpenEncryptedFileRawA function (winbase.h)

Article02/09/2023

Opens an encrypted file in order to backup (export) or restore (import) the file. This is one of a group of Encrypted File System (EFS) functions that is intended to implement backup and restore functionality, while maintaining files in their encrypted state.

Syntax

C++

```
DWORD OpenEncryptedFileRawA(
    [in]  LPCSTR lpFileName,
    [in]  ULONG   ulFlags,
    [out] PVOID   *pvContext
);
```

Parameters

[in] lpFileName

The name of the file to be opened. The string must consist of characters from the Windows character set.

[in] ulFlags

The operation to be performed. This parameter may be one of the following values.

Value	Meaning
0	Open the file for export (backup).
CREATE_FOR_IMPORT	The file is being opened for import (restore).
1	
CREATE_FOR_DIR	Import (restore) a directory containing encrypted files. This must be combined with one of the previous two flags to indicate the operation.
2	
OVERWRITE_HIDDEN	Overwrite a hidden file on import.
4	

[out] *pvContext*

The address of a context block that must be presented in subsequent calls to [ReadEncryptedFileRaw](#), [WriteEncryptedFileRaw](#), or [CloseEncryptedFileRaw](#). Do not modify it.

Return value

If the function succeeds, it returns **ERROR_SUCCESS**.

If the function fails, it returns a nonzero error code defined in *WinError.h*. You can use [FormatMessage](#) with the **FORMAT_MESSAGE_FROM_SYSTEM** flag to get a generic text description of the error.

Remarks

The caller must either have read or write access to the file, or it must have backup privilege [SeBackupPrivilege](#) on the machine on which the files reside in order for the call to succeed.

To back up an encrypted file, call [OpenEncryptedFileRaw](#) to open the file and then call [ReadEncryptedFileRaw](#). When the backup is complete, call [CloseEncryptedFileRaw](#).

To restore an encrypted file, call [OpenEncryptedFileRaw](#), specifying **CREATE_FOR_IMPORT** in the *ulFlags* parameter, and then call [WriteEncryptedFileRaw](#) once. When the operation is completed, call [CloseEncryptedFileRaw](#).

[OpenEncryptedFileRaw](#) fails if *lpFileName* exceeds **MAX_PATH** characters when opening an encrypted file on a remote machine.

If the caller does not have access to the key for the file, the caller needs [SeBackupPrivilege](#) to export encrypted files or [SeRestorePrivilege](#) to import encrypted files.

The [BackupRead](#) and [BackupWrite](#) functions handle backup and restore of unencrypted files.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

ⓘ Note

The `winbase.h` header defines `OpenEncryptedFileRaw` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Advapi32.lib</code>
DLL	<code>Advapi32.dll</code>
API set	<code>ext-ms-win-advapi32-encryptedfile-l1-1-0</code> (introduced in Windows 8)

See also

[BackupRead](#)

[BackupWrite](#)

[CloseEncryptedFileRaw](#)

[File Encryption](#)

[File Management Functions](#)

[ReadEncryptedFileRaw](#)

[WriteEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenEncryptedFileRawW function (winbase.h)

Article02/09/2023

Opens an encrypted file in order to backup (export) or restore (import) the file. This is one of a group of Encrypted File System (EFS) functions that is intended to implement backup and restore functionality, while maintaining files in their encrypted state.

Syntax

C++

```
DWORD OpenEncryptedFileRawW(
    [in]  LPCWSTR lpFileName,
    [in]  ULONG    ulFlags,
    [out] PVOID   *pvContext
);
```

Parameters

[in] lpFileName

The name of the file to be opened. The string must consist of characters from the Windows character set.

[in] ulFlags

The operation to be performed. This parameter may be one of the following values.

Value	Meaning
0	Open the file for export (backup).
CREATE_FOR_IMPORT	The file is being opened for import (restore).
1	
CREATE_FOR_DIR	Import (restore) a directory containing encrypted files. This must be combined with one of the previous two flags to indicate the operation.
2	
OVERWRITE_HIDDEN	Overwrite a hidden file on import.
4	

[out] *pvContext*

The address of a context block that must be presented in subsequent calls to [ReadEncryptedFileRaw](#), [WriteEncryptedFileRaw](#), or [CloseEncryptedFileRaw](#). Do not modify it.

Return value

If the function succeeds, it returns **ERROR_SUCCESS**.

If the function fails, it returns a nonzero error code defined in **WinError.h**. You can use [FormatMessage](#) with the **FORMAT_MESSAGE_FROM_SYSTEM** flag to get a generic text description of the error.

Remarks

The caller must either have read or write access to the file, or it must have backup privilege [SeBackupPrivilege](#) on the machine on which the files reside in order for the call to succeed.

To back up an encrypted file, call [OpenEncryptedFileRaw](#) to open the file and then call [ReadEncryptedFileRaw](#). When the backup is complete, call [CloseEncryptedFileRaw](#).

To restore an encrypted file, call [OpenEncryptedFileRaw](#), specifying **CREATE_FOR_IMPORT** in the *ulFlags* parameter, and then call [WriteEncryptedFileRaw](#) once. When the operation is completed, call [CloseEncryptedFileRaw](#).

[OpenEncryptedFileRaw](#) fails if *lpFileName* exceeds **MAX_PATH** characters when opening an encrypted file on a remote machine.

If the caller does not have access to the key for the file, the caller needs [SeBackupPrivilege](#) to export encrypted files or [SeRestorePrivilege](#) to import encrypted files.

The [BackupRead](#) and [BackupWrite](#) functions handle backup and restore of unencrypted files.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

ⓘ Note

The `winbase.h` header defines `OpenEncryptedFileRaw` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)
Library	<code>Advapi32.lib</code>
DLL	<code>Advapi32.dll</code>
API set	<code>ext-ms-win-advapi32-encryptedfile-l1-1-0</code> (introduced in Windows 8)

See also

[BackupRead](#)

[BackupWrite](#)

[CloseEncryptedFileRaw](#)

[File Encryption](#)

[File Management Functions](#)

[ReadEncryptedFileRaw](#)

[WriteEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenEventLogA function (winbase.h)

Article 02/09/2023

Opens a handle to the specified event log.

Syntax

C++

```
HANDLE OpenEventLogA(
    [in] LPCSTR lpUNCServerName,
    [in] LPCSTR lpSourceName
);
```

Parameters

[in] lpUNCServerName

The Universal Naming Convention (UNC) name of the remote server on which the event log is to be opened. If this parameter is **NULL**, the local computer is used.

[in] lpSourceName

The name of the log.

If you specify a custom log and it cannot be found, the event logging service opens the **Application** log; however, there will be no associated message or category string file.

Return value

If the function succeeds, the return value is the handle to an event log.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

To close the handle to the event log, use the [CloseEventLog](#) function.

Examples

For an example, see [Querying for Event Information](#).

 **Note**

The winbase.h header defines OpenEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[ClearEventLog](#)

[CloseEventLog](#)

[Event Logging Functions](#)

[Eventlog Key](#)

[ReadEventLog](#)

[ReportEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenEventLogW function (winbase.h)

Article02/09/2023

Opens a handle to the specified event log.

Syntax

C++

```
HANDLE OpenEventLogW(
    [in] LPCWSTR lpUNCServerName,
    [in] LPCWSTR lpSourceName
);
```

Parameters

[in] lpUNCServerName

The Universal Naming Convention (UNC) name of the remote server on which the event log is to be opened. If this parameter is **NULL**, the local computer is used.

[in] lpSourceName

The name of the log.

If you specify a custom log and it cannot be found, the event logging service opens the **Application** log; however, there will be no associated message or category string file.

Return value

If the function succeeds, the return value is the handle to an event log.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

To close the handle to the event log, use the [CloseEventLog](#) function.

Examples

For an example, see [Querying for Event Information](#).

 **Note**

The winbase.h header defines OpenEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[ClearEventLog](#)

[CloseEventLog](#)

[Event Logging Functions](#)

[Eventlog Key](#)

[ReadEventLog](#)

[ReportEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenFile function (winbase.h)

Article10/13/2021

Creates, opens, reopens, or deletes a file.

Note This function has limited capabilities and is not recommended. For new application development, use the [CreateFile](#) function.

Syntax

C++

```
HFILE OpenFile(
    [in]  LPCSTR      lpFileName,
    [out] LPOFSTRUCT lpReOpenBuff,
    [in]  UINT        uStyle
);
```

Parameters

[in] lpFileName

The name of the file.

The string must consist of characters from the 8-bit Windows character set. The [OpenFile](#) function does not support Unicode file names or opening named pipes.

[out] lpReOpenBuff

A pointer to the [OFSTRUCT](#) structure that receives information about a file when it is first opened.

The structure can be used in subsequent calls to the [OpenFile](#) function to see an open file.

The [OFSTRUCT](#) structure contains a path string member with a length that is limited to [OFS_MAXPATHNAME](#) characters, which is 128 characters. Because of this, you cannot use the [OpenFile](#) function to open a file with a path length that exceeds 128 characters. The [CreateFile](#) function does not have this path length limitation.

[in] uStyle

The action to be taken.

This parameter can be one or more of the following values.

Value	Meaning
OF_CANCEL 0x00000800	Ignored. To produce a dialog box containing a Cancel button, use OF_PROMPT .
OF_CREATE 0x00001000	Creates a new file. If the file exists, it is truncated to zero (0) length.
OF_DELETE 0x00000200	Deletes a file.
OF_EXIST 0x00004000	Opens a file and then closes it. Use this to test for the existence of a file.
OF_PARSE 0x00000100	Fills the OFSTRUCT structure, but does not do anything else.
OF_PROMPT 0x00002000	Displays a dialog box if a requested file does not exist. A dialog box informs a user that the system cannot find a file, and it contains Retry and Cancel buttons. The Cancel button directs OpenFile to return a file-not-found error message.
OF_READ 0x00000000	Opens a file for reading only.
OF_READWRITE 0x00000002	Opens a file with read/write permissions.
OF_REOPEN 0x00008000	Opens a file by using information in the reopen buffer.
OF_SHARE_COMPAT 0x00000000	For MS-DOS-based file systems, opens a file with compatibility mode, allows any process on a specified computer to open the file any number of times. Other efforts to open a file with other sharing modes fail. This flag is mapped to the FILE_SHARE_READ FILE_SHARE_WRITE flags of the CreateFile function.
OF_SHARE_DENY_NONE 0x00000040	Opens a file without denying read or write access to other processes. On MS-DOS-based file systems, if the file has been opened in compatibility mode by any other process, the

	<p>function fails.</p> <p>This flag is mapped to the FILE_SHARE_READ FILE_SHARE_WRITE flags of the CreateFile function.</p>
OF_SHARE_DENY_READ 0x00000030	<p>Opens a file and denies read access to other processes. On MS-DOS-based file systems, if the file has been opened in compatibility mode, or for read access by any other process, the function fails.</p> <p>This flag is mapped to the FILE_SHARE_WRITE flag of the CreateFile function.</p>
OF_SHARE_DENY_WRITE 0x00000020	<p>Opens a file and denies write access to other processes. On MS-DOS-based file systems, if a file has been opened in compatibility mode, or for write access by any other process, the function fails.</p> <p>This flag is mapped to the FILE_SHARE_READ flag of the CreateFile function.</p>
OF_SHARE_EXCLUSIVE 0x00000010	<p>Opens a file with exclusive mode, and denies both read/write access to other processes. If a file has been opened in any other mode for read/write access, even by the current process, the function fails.</p>
OF_VERIFY	<p>Verifies that the date and time of a file are the same as when it was opened previously. This is useful as an extra check for read-only files.</p>
OF_WRITE 0x00000001	<p>Opens a file for write access only.</p>

Return value

If the function succeeds, the return value specifies a file handle to use when performing file I/O. To close the file, call the [CloseHandle](#) function using this handle.

If the function fails, the return value is **HFILE_ERROR**. To get extended error information, call [GetLastError](#).

Remarks

If the *lpFileName* parameter specifies a file name and extension only, this function searches for a matching file in the following directories and the order shown:

1. The directory where an application is loaded.

2. The current directory.
3. The Windows system directory.

Use the [GetSystemDirectory](#) function to get the path of this directory.

4. The 16-bit Windows system directory.

There is not a function that retrieves the path of this directory, but it is searched.

5. The Windows directory.

Use the [GetWindowsDirectory](#) function to get the path of this directory.

6. The directories that are listed in the PATH environment variable.

The *lpFileName* parameter cannot contain wildcard characters.

The [OpenFile](#) function does not support the **OF_SEARCH** flag that the 16-bit Windows [OpenFile](#) function supports. The **OF_SEARCH** flag directs the system to search for a matching file even when a file name includes a full path. Use the [SearchPath](#) function to search for a file.

A sharing violation occurs if an attempt is made to open a file or directory for deletion on a remote machine when the value of the *uStyle* parameter is the **OF_DELETE** access flag OR'ed with any other access flag, and the remote file or directory has not been opened with **FILE_SHARE_DELETE** share access. To avoid the sharing violation in this scenario, open the remote file or directory with **OF_DELETE** access only, or call [DeleteFile](#) without first opening the file or directory for deletion.

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

CsvFs will do redirected IO for compressed files.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[File Management Functions](#)

[GetSystemDirectory](#)

[GetWindowsDirectory](#)

[OFSTRUCT](#)

[SearchPath](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenFileById function (winbase.h)

Article10/13/2021

Opens the file that matches the specified identifier.

Syntax

C++

```
HANDLE OpenFileById(
    [in]          HANDLE           hVolumeHint,
    [in]          LPFILE_ID_DESCRIPTOR lpFileId,
    [in]          DWORD            dwDesiredAccess,
    [in]          DWORD            dwShareMode,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    [in]          DWORD            dwFlagsAndAttributes
);
```

Parameters

[in] hVolumeHint

A handle to any file on a volume or share on which the file to be opened is stored.

[in] lpFileId

A pointer to a [FILE_ID_DESCRIPTOR](#) that identifies the file to open.

[in] dwDesiredAccess

The access to the object. Access can be read, write, or both.

For more information, see [File Security and Access Rights](#). You cannot request an access mode that conflicts with the sharing mode that is specified in an open request that has an open handle.

If this parameter is zero (0), the application can query file and 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 in a drive. It can also be used to test for the existence of a file or directory without opening them for read or write access.

[in] dwShareMode

The sharing mode of an object, which can be read, write, both, or none.

You cannot request a sharing mode that conflicts with the access mode that is specified in an open request that has an open handle, because that would result in the following sharing violation: ([ERROR_SHARING_VIOLATION](#)). For more information, see [Creating and Opening Files](#).

If this parameter is zero (0) and [OpenFileById](#) succeeds, the object cannot be shared and cannot be opened again until the handle is closed. For more information, see the Remarks section of this topic.

The sharing options remain in effect until you close the handle to an object.

To enable a processes to share an object while another process has the object open, use a combination of one or more of the following values to specify the access mode they can request to open the object.

Value	Meaning
FILE_SHARE_DELETE 0x00000004	Enables subsequent open operations on an object to request delete access. Otherwise, other processes cannot open the object if they request delete access. If this flag is not specified, but the object has been opened for delete access, the function fails.
FILE_SHARE_READ 0x00000001	Enables subsequent open operations on an object to request read access. Otherwise, other processes cannot open the object if they request read access. If this flag is not specified, but the object has been opened for read access, the function fails.
FILE_SHARE_WRITE 0x00000002	Enables subsequent open operations on an object to request write access. Otherwise, other processes cannot open the object if they request write access. If this flag is not specified, but the object has been opened for write access or has a file mapping with write access, the function fails.

[in, optional] `lpSecurityAttributes`

Reserved.

[in] dwFlagsAndAttributes

The file flags.

When **OpenFileById** opens a file, it combines the file flags with existing file attributes, and ignores any supplied file attributes. This parameter can include any combination of the following flags.

Value	Meaning
FILE_FLAG_BACKUP_SEMANTICS 0x02000000	A file is being opened for a backup or restore operation. The system ensures that the calling process overrides file security checks when the process has SE_BACKUP_NAME and SE_RESTORE_NAME privileges. For more information, see Changing Privileges in a Token . You must set this flag to obtain a handle to a directory. A directory handle can be passed to some functions instead of a file handle. For more information, see Directory Handles .
FILE_FLAG_NO_BUFFERING 0x20000000	The system opens a file with no system caching. This flag does not affect hard disk caching. When combined with FILE_FLAG_OVERLAPPED , the flag gives maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations take more time, because data is not being held in the cache. Also, the file metadata may still be cached. To flush the metadata to disk, use the FlushFileBuffers function. An application must meet certain requirements when working with files that are opened with FILE_FLAG_NO_BUFFERING : <ul style="list-style-type: none">• File access must begin at byte offsets within a file that are integer multiples of the volume sector size.• File access must be for numbers of bytes that are integer multiples of the volume sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, 1536, or 2048 bytes, but not of 335, 981, or 7171 bytes.• Buffer addresses for read and write operations should be sector aligned, which means aligned on addresses in memory that are integer multiples of the volume sector size. Depending on the disk, this requirement may not be enforced.

One way to align buffers on integer multiples of the volume sector size is to use [VirtualAlloc](#) to allocate the buffers. It allocates memory that is aligned on addresses that are integer multiples of the operating system's memory page size. Because both memory page and volume sector sizes are powers of 2, this memory is also aligned on addresses that are integer multiples of a volume sector size. Memory pages are 4-8 KB in size; sectors are 512 bytes (hard disks) or 2048 bytes (CD), and therefore, volume sectors can never be larger than memory pages.

An application can determine a volume sector size by calling the [GetDiskFreeSpace](#) function.

FILE_FLAG_OPEN_NO_RECALL 0x00100000	The file data is requested, but it should continue to be located in remote storage. It should not be transported back to local storage. This flag is for use by remote storage systems.
FILE_FLAG_OPEN_REPARSE_POINT 0x00200000	When this flag is used, normal reparse point processing does not occur, and OpenFileById attempts to open the reparse point. When a file is opened, a file handle is returned, whether or not the filter that controls the reparse point is operational. This flag cannot be used with the CREATE_ALWAYS flag. If the file is not a reparse point, then this flag is ignored.
FILE_FLAG_OVERLAPPED 0x40000000	<p>The file is being opened or created for asynchronous I/O. When the operation is complete, the event specified to the call in the OVERLAPPED structure is set to the signaled state. Operations that take a significant amount of time to process return ERROR_IO_PENDING.</p> <p>If this flag is specified, the file can be used for simultaneous read and write operations. The system does not maintain the file pointer, therefore you must pass the file position to the read and write functions in the OVERLAPPED structure or update the file pointer.</p> <p>If this flag is not specified, then I/O operations are serialized, even if the calls to the read and write functions specify an OVERLAPPED structure.</p>
FILE_FLAG_RANDOM_ACCESS 0x10000000	A file is accessed randomly. The system can use this as a hint to optimize file caching.
FILE_FLAG_SEQUENTIAL_SCAN 0x08000000	A file is accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access,

optimum caching may not occur. However, correct operation is still guaranteed.

Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.

FILE_FLAG_WRITE_THROUGH
0x80000000

The system writes through any intermediate cache and goes directly to disk.

If **FILE_FLAG_NO_BUFFERING** is not also specified, so that system caching is in effect, then the data is written to the system cache, but is flushed to disk without delay.

If **FILE_FLAG_NO_BUFFERING** is also specified, so that system caching is not in effect, then the data is immediately flushed to disk without going through the system cache. The operating system also requests a write-through the hard disk cache to persistent media. However, not all hardware supports this write-through capability.

Return value

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

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call [GetLastError](#).

Remarks

Use the [CloseHandle](#) function to close an object handle that [OpenFileById](#) returns.

If you call [OpenFileById](#) 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**.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib; FileExtd.lib on Windows Server 2003 and Windows XP
DLL	Kernel32.dll
Redistributable	Windows SDK on Windows Server 2003 and Windows XP.

See also

[ACCESS_MASK](#)

[CloseHandle](#)

[CreateFile](#)

[DeleteFile](#)

[FILE_ID_DESCRIPTOR](#)

[File Management Functions](#)

[GetFileInformationByHandleEx](#)

[GetOverlappedResult](#)

[OVERLAPPED](#)

[OpenFile](#)

[ReadFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenFileMappingA function (winbase.h)

Article 07/27/2022

Opens a named file mapping object.

Syntax

C++

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

Parameters

[in] 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](#).

[in] bInheritHandle

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

[in] 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 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](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OpenJobObjectA function (winbase.h)

Article 04/26/2023

Opens an existing job object.

Syntax

C++

```
HANDLE OpenJobObjectA(
    [in] DWORD  dwDesiredAccess,
    [in] BOOL   bInheritHandle,
    [in] LPCSTR lpName
);
```

Parameters

[in] dwDesiredAccess

The access to the job object. This parameter can be one or more of the [job object access rights](#). This access right is checked against any security descriptor for the object.

[in] bInheritHandle

If this value is TRUE, processes created by this process will inherit the handle. Otherwise, the processes do not inherit this handle.

[in] lpName

The name of the job to be opened. Name comparisons are case sensitive.

This function can open objects in a private namespace. For more information, see [Object Namespaces](#).

Terminal Services: The name can have a "Global" or "Local" prefix to explicitly open the 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](#).

Return value

If the function succeeds, the return value is a handle to the job. The handle provides the requested access to the job.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

An error of **ERROR_FILE_NOT_FOUND** indicates that the job specified in *lpName* does not exist.

Remarks

To associate a process with a job, use the [AssignProcessToJobObject](#) function.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see [Using the Windows Headers](#).

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, Jobapi2.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AssignProcessToJobObject](#)

[Job Objects](#)

[Process and Thread Functions](#)

Feedback



Was this page helpful? [Yes](#) | [No](#)

Get help at Microsoft Q&A

OpenPrivateNamespaceA function (winbase.h)

Article 09/22/2022

Opens a private namespace.

Syntax

C++

```
HANDLE OpenPrivateNamespaceA(
    [in] LPVOID lpBoundaryDescriptor,
    [in] LPCSTR lpAliasPrefix
);
```

Parameters

[in] lpBoundaryDescriptor

A descriptor that defines how the namespace is to be isolated. The [CreateBoundaryDescriptor](#) function creates a boundary descriptor.

[in] lpAliasPrefix

The prefix for the namespace. To create an object in this namespace, specify the object name as *prefix\objectname*.

Return value

The function returns the handle to the existing namespace.

Remarks

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ClosePrivateNamespace](#)

[CreateBoundaryDescriptor](#)

[CreatePrivateNamespace](#)

[Object Namespaces](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OPERATION_END_PARAMETERS structure (winbase.h)

Article04/02/2021

This structure is used by the [OperationEnd](#) function.

Syntax

C++

```
typedef struct _OPERATION_END_PARAMETERS {
    ULONG      Version;
    OPERATION_ID OperationId;
    ULONG      Flags;
} OPERATION_END_PARAMETERS, *POPERATION_END_PARAMETERS;
```

Members

Version

This parameter should be initialized to the **OPERATION_API_VERSION** defined in the Windows SDK.

Value	Meaning
OPERATION_API_VERSION 1	This API was introduced in Windows 8 and Windows Server 2012 as version 1.

OperationId

Each operation has an **OPERATION_ID** namespace that is unique for each process. If two applications both use the same **OPERATION_ID** value to identify two operations, the system maintains separate contexts for each operation.

Flags

The value of this parameter can include any combination of the following values.

Value	Meaning
OPERATION_END_DISCARD 1	Specifies that the system should discard the information it has been tracking for this operation. Specify this flag

when the operation either fails or does not follow the expected sequence of steps.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[OPERATION_ID](#)

[OPERATION_START_PARAMETERS](#)

[Operation Recorder](#)

[OperationEnd](#)

[OperationStart](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OPERATION_START_PARAMETERS structure (winbase.h)

Article04/02/2021

This structure is used by the [OperationStart](#) function.

Syntax

C++

```
typedef struct _OPERATION_START_PARAMETERS {
    ULONG      Version;
    OPERATION_ID OperationId;
    ULONG      Flags;
} OPERATION_START_PARAMETERS, *POPERATION_START_PARAMETERS;
```

Members

Version

This parameter should be initialized to the **OPERATION_API_VERSION** value defined in the Windows SDK.

Value	Meaning
OPERATION_API_VERSION 1	This API was introduced in Windows 8 and Windows Server 2012 as version 1.

OperationId

Each operation has an **OPERATION_ID** namespace that is unique for each process. If two applications both use the same **OPERATION_ID** value to identify two operations, the system maintains separate contexts for each operation.

Flags

The value of this parameter can include any combination of the following values.

Value	Meaning
OPERATION_START_TRACE_CURRENT_THREAD 1	Specifies that the system should only track the activities of the calling thread in a multi-

threaded application. Specify this flag when the operation is performed on a single thread to isolate its activity from other threads in the process.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[OPERATION_END_PARAMETERS](#)

[OPERATION_ID](#)

[Operation Recorder](#)

[OperationEnd](#)

[OperationStart](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OperationEnd function (winbase.h)

Article10/13/2021

Notifies the system that the application is about to end an operation

Every call to [OperationStart](#) must be followed by a call to [OperationEnd](#), otherwise the operation's record of file access patterns is discarded after 10 seconds.

Syntax

C++

```
BOOL OperationEnd(
    [in] OPERATION_END_PARAMETERS *OperationEndParams
);
```

Parameters

[in] OperationEndParams

An [_OPERATION_END_PARAMETERS](#) structure that specifies **VERSION**, **OPERATION_ID** and **FLAGS**.

Return value

TRUE for all valid parameters and FALSE otherwise. To get extended error information, call [GetLastError](#).

Remarks

The version of the [_OPERATION_END_PARAMETERS](#) structure is defined as **OPERATION_API_VERSION** in the Windows SDK.

The [OperationEnd](#) function is safe to call on any thread.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[OPERATION_ID](#)

[Operation Recorder](#)

[OperationStart](#)

[_OPERATION_END_PARAMETERS](#)

[_OPERATION_START_PARAMETERS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

OperationStart function (winbase.h)

Article10/13/2021

Notifies the system that the application is about to start an operation.

If an application calls **OperationStart** with a valid **OPERATION_ID** value, the system records the specified operation's file access patterns until **OperationEnd** is called for the same operation ID. This record is stored in a *filename.pf* prefetch file. Every call to **OperationStart** must be followed by a call to **OperationEnd**, otherwise the operation's record is discarded after 10 seconds.

If an application calls **OperationStart** for an operation ID for which a prefetch file exists, the system loads the operation's files into memory prior to running the operation. The recording process remains the same and the system updates the appropriate *filename.pf* prefetch file.

Syntax

C++

```
BOOL OperationStart(
    [in] OPERATION_START_PARAMETERS *OperationStartParams
);
```

Parameters

[in] OperationStartParams

An **_OPERATION_START_PARAMETERS** structure that specifies **VERSION**, **OPERATION_ID** and **FLAGS**.

Return value

TRUE for all valid parameters and **FALSE** otherwise. To get extended error information, call **GetLastError**.

Remarks

The version of the [_OPERATION_START_PARAMETERS](#) structure is defined as **OPERATION_API_VERSION** in the Windows SDK.

Because the **OperationStart** function is synchronous, it can take several seconds to return. This should be avoided in UI threads for the best responsiveness.

There is a single instance of the operation recorder in a process. Although the operation recorder APIs can be called from multiple threads within the process, all calls act on the single instance.

Application launch tracing lasts for the first 10 second of the process lifetime.

OperationStart should be called after the end of application launch tracing by the system.

Every call to **OperationStart** must be followed by a call to [OperationEnd](#). Otherwise, the operation trace will be discarded after about 10s.

The maximum number of operations that can be recorded on a given system is configurable. If this maximum is exceeded, the least recently used prefetch files are replaced.

On Windows 8, this functionality requires the Superfetch service to be enabled. Windows 8 will have the service enabled by default. For Windows Server 2012, this prefetching functionality needs to be enabled and disabled as required. This can be done using CIM based PowerShell cmdlets. The prefetcher functionality can be exposed using the [CIM class](#) of the [CIM_PrefetcherService](#).

Examples

syntax

```
BOOL Success;
DWORD ErrorCode;
OPERATION_START_PARAMETERS OpStart;
OPERATION_END_PARAMETERS OpEnd;

// We want to notify Windows that we are going to be performing some
// disk-bound work that repeatedly access the same file data. The system
will
    // try to record data about our activity to make future operations
faster.

ZeroMemory(&OpStart, sizeof(OpStart));
OpStart.Version = OPERATION_API_VERSION;
OpStart.OperationId = MY_OPERATION_ID_1;

ZeroMemory(&OpEnd, sizeof(OpEnd));
```

```

OpEnd.Version = OPERATION_API_VERSION;
OpEnd.OperationId = MY_OPERATION_ID_1;

// We want the system to only record activity in this thread.

OpStart.Flags = OPERATION_START_TRACE_CURRENT_THREAD;
OpEnd.Flags = 0;

Success = OperationStart(&OpStart);

if (!Success) {
    ErrorCode = GetLastError();
    fprintf(stderr, "OperationStart failed: %d\n", ErrorCode);

    // We could not notify the system about our operation. That's OK.

}

// Perform the disk-bound work that should be recorded here.
// This may involve opening/reading many files or loading
// and running many DLLs.

Success = OperationEnd(&OpEnd);

if (!Success) {
    fprintf(stderr, "OperationEnd failed: %d\n", GetLastError());
}

```

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[OPERATION_ID](#)

Operation Recorder

OperationEnd

_OPERATION_END_PARAMETERS

_OPERATION_START_PARAMETERS

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

PCOPYFILE2_PROGRESS_ROUTINE callback function (winbase.h)

Article 10/13/2021

An application-defined callback function used with the [CopyFile2](#) function. It is called when a portion of a copy or move operation is completed. The **PCOPYFILE2_PROGRESS_ROUTINE** type defines a pointer to this callback function. **CopyFile2ProgressRoutine** is a placeholder for the application-defined function name.

Syntax

C++

```
PCOPYFILE2_PROGRESS_ROUTINE Pcopyfile2ProgressRoutine;

COPYFILE2_MESSAGE_ACTION Pcopyfile2ProgressRoutine(
    [in]           const COPYFILE2_MESSAGE *pMessage,
    [in, optional] PVOID pvCallbackContext
)
{...}
```

Parameters

[in] **pMessage**

Pointer to a [COPYFILE2_MESSAGE](#) structure.

[in, optional] **pvCallbackContext**

Copy of value passed in the **pvCallbackContext** member of the [COPYFILE2_EXTENDED_PARAMETERS](#) structure passed to [CopyFile2](#).

Return value

Value from the [COPYFILE2_MESSAGE_ACTION](#) enumeration indicating what action should be taken.

Return code/value	Description
COPYFILE2_PROGRESS_CONTINUE	Continue the copy operation.
0	

COPYFILE2_PROGRESS_CANCEL	Cancel the copy operation. The CopyFile2 function will fail, return <code>HRESULT_FROM_WIN32(ERROR_REQUEST_ABORTED)</code> and any partially copied fragments will be deleted.
COPYFILE2_PROGRESS_STOP	Stop the copy operation. The CopyFile2 function will fail, return <code>HRESULT_FROM_WIN32(ERROR_REQUEST_ABORTED)</code> and any partially copied fragments will be left intact. The operation can be restarted using the <code>COPY_FILE_RESUME_FROM_PAUSE</code> flag only if <code>COPY_FILE_RESTARTABLE</code> was set in the <code>dwCopyFlags</code> member of the COPYFILE2_EXTENDED_PARAMETERS structure passed to the CopyFile2 function.
COPYFILE2_PROGRESS QUIET	Continue the copy operation but do not call the CopyFile2ProgressRoutine callback function again for this operation.
COPYFILE2_PROGRESS_PAUSE	Pause the copy operation. In most cases the CopyFile2 function will fail and return <code>HRESULT_FROM_WIN32(ERROR_REQUEST_PAUSED)</code> and any partially copied fragments will be left intact (except for the header written that is used to resume the copy operation later.) In case the copy operation was complete at the time the pause request is processed the CopyFile2 call will complete successfully and no resume header will be written.

Remarks

The `COPYFILE2_CALLBACK_STREAM_FINISHED` message is the last message for a paused copy. If `COPYFILE2_PROGRESS_PAUSE` is returned in response to a `COPYFILE2_CALLBACK_STREAM_FINISHED` message then no further callbacks will be sent.

To compile an application that uses the `PCOPYFILE2_PROGRESS_ROUTINE` function pointer type, define the `_WIN32_WINNT` macro as 0x0601 or later. For more information, see [Using the Windows Headers](#).

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

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PFE_EXPORT_FUNC callback function (winbase.h)

Article10/13/2021

An application-defined callback function used with [ReadEncryptedFileRaw](#). The system calls **ExportCallback** one or more times, each time with a block of the encrypted file's data, until it has received all of the file data. **ExportCallback** writes the encrypted file's data to another storage media, usually for purposes of backing up the file.

The **PFE_EXPORT_FUNC** type defines a pointer to the callback function. **ExportCallback** is a placeholder for the application-defined function name.

Syntax

C++

```
PFE_EXPORT_FUNC PfeExportFunc;

DWORD PfeExportFunc(
    [in]          PBYTE pbData,
    [in, optional] PVOID pvCallbackContext,
    [in]          ULONG ulLength
)
{...}
```

Parameters

[in] **pbData**

A pointer to a block of the encrypted file's data to be backed up. This block of data is allocated by the system.

[in, optional] **pvCallbackContext**

A pointer to an application-defined and allocated context block. The application passes this pointer to [ReadEncryptedFileRaw](#), and [ReadEncryptedFileRaw](#) passes this pointer to the callback function so that it can have access to application-specific data. This data can be a structure and can contain any data the application needs, such as the handle to the file that contains the backup copy of the encrypted file.

[in] **ulLength**

The size of the data pointed to by the *pbData* parameter, in bytes.

Return value

If the function succeeds, it must set the return value to **ERROR_SUCCESS**.

If the function fails, set the return value to a nonzero error code defined in WinError.h. For example, if this function fails because an API that it calls fails, you can set the return value to the value returned by [GetLastError](#) for the failed API.

Remarks

You can use the application-defined context block for internal tracking of information such as the file handle and the current offset in the file.

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)

See also

[CloseEncryptedFileRaw](#)

[File Encryption](#)

[File Management Functions](#)

[ImportCallback](#)

[OpenEncryptedFileRaw](#)

[ReadEncryptedFileRaw](#)

[WriteEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PFE_IMPORT_FUNC callback function (winbase.h)

Article10/13/2021

An application-defined callback function used with [WriteEncryptedFileRaw](#). The system calls **ImportCallback** one or more times, each time to retrieve a portion of a backup file's data. **ImportCallback** reads the data from a backup file sequentially and restores the data, and the system continues calling it until it has read all of the backup file data.

The **PFE_IMPORT_FUNC** type defines a pointer to this callback function. **ImportCallback** is a placeholder for the application-defined function name.

Syntax

C++

```
PFE_IMPORT_FUNC PfeImportFunc;

DWORD PfeImportFunc(
    [in]          PBYTE pbData,
    [in, optional] PVOID pvCallbackContext,
    [in, out]      PULONG ulLength
)
{...}
```

Parameters

[in] **pbData**

A pointer to a system-supplied buffer that will receive a block of data to be restored.

[in, optional] **pvCallbackContext**

A pointer to an application-defined and allocated context block. The application passes this pointer to [WriteEncryptedFileRaw](#), and it passes this pointer to the callback function so that the callback function can have access to application-specific data. This data can be a structure and can contain any data the application needs, such as the handle to the file that contains the backup copy of the encrypted file.

[in, out] **ulLength**

On function entry, this parameter specifies the length of the buffer the system has supplied. The callback function must write no more than this many bytes to the buffer pointed to by the *pbData* parameter.

On exit, the function must set this to the number of bytes of data written into the *pbData*.

Return value

If the function succeeds, it must set the return value to **ERROR_SUCCESS**, and set the value pointed to by the *ullLength* parameter to the number of bytes copied into *pbData*.

When the end of the backup file is reached, set *ullLength* to zero to tell the system that the entire file has been processed.

If the function fails, set the return value to a nonzero error code defined in WinError.h. For example, if this function fails because an API that it calls fails, you can set the return value to the value returned by [GetLastError](#) for the failed API.

Remarks

The system calls the **ImportCallback** function until the callback function indicates there is no more data to restore. To indicate that there is no more data to be restored, set **ullLength* to 0 and use a return code of **ERROR_SUCCESS**. You can use the application-defined context block for internal tracking of information such as the file handle and the current offset in the file.

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)

See also

[CloseEncryptedFileRaw](#)

[ExportCallback](#)

[File Encryption](#)

[File Management Functions](#)

[OpenEncryptedFileRaw](#)

[ReadEncryptedFileRaw](#)

[WriteEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PFIBER_START_ROUTINE callback function (winbase.h)

Article04/02/2021

An application-defined function used with the [CreateFiber](#) function. It serves as the starting address for a fiber. The **LPFIBER_START_ROUTINE** type defines a pointer to this callback function. **FiberProc** is a placeholder for the application-defined function name.

Syntax

```
C++  
  
PFIBER_START_ROUTINE PfiberStartRoutine;  
  
void PfiberStartRoutine(  
    LPVOID lpFiberParameter  
)  
{...}
```

Parameters

lpFiberParameter

Return value

None

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)

See also

[CreateFiber](#)

[Fibers](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PowerClearRequest function (winbase.h)

Article10/13/2021

Decrements the count of power requests of the specified type for a power request object.

Syntax

C++

```
BOOL PowerClearRequest(
    [in] HANDLE             PowerRequest,
    [in] POWER_REQUEST_TYPE RequestType
);
```

Parameters

[in] PowerRequest

A handle to a power request object.

[in] RequestType

The power request type to be decremented. This parameter can be one of the following values.

Value	Meaning
PowerRequestDisplayRequired	The display remains on even if there is no user input for an extended period of time.
PowerRequestSystemRequired	The system continues to run instead of entering sleep after a period of user inactivity.
PowerRequestAwayModeRequired	The system enters away mode instead of sleep. In away mode, the system continues to run but turns off audio and video to give the appearance of sleep.
PowerRequestExecutionRequired	The calling process continues to run instead of being suspended or terminated by process lifetime management mechanisms. When and how long the process is allowed to run depends on the operating system and power policy settings.

When a **PowerRequestExecutionRequired** request is active, it implies **PowerRequestSystemRequired**.

The **PowerRequestExecutionRequired** request type can be used only by applications. Services cannot use this request type.

Windows 7 and Windows Server 2008 R2: This request type is supported starting with Windows 8 and Windows Server 2012.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[PowerCreateRequest](#)

[PowerSetRequest](#)

Feedback

Was this page helpful?

Yes

No

Get help at Microsoft Q&A

PowerCreateRequest function (winbase.h)

Article 10/13/2021

Creates a new power request object.

Syntax

C++

```
HANDLE PowerCreateRequest(
    [in] PREASON_CONTEXT Context
);
```

Parameters

[in] Context

Points to a [REASON_CONTEXT](#) structure that contains information about the power request.

Return value

If the function succeeds, the return value is a handle to the power request object.

If the function fails, the return value is INVALID_HANDLE_VALUE. To get extended error information, call [GetLastError](#).

Remarks

When the power request object is no longer needed, use the [CloseHandle](#) function to free the handle and clean up the object.

Requirements

Minimum supported client

Windows 7 [desktop apps only]

Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[PowerClearRequest](#)

[PowerSetRequest](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PowerSetRequest function (winbase.h)

Article07/27/2022

Increments the count of power requests of the specified type for a power request object.

Syntax

C++

```
BOOL PowerSetRequest(  
    [in] HANDLE             PowerRequest,  
    [in] POWER_REQUEST_TYPE RequestType  
) ;
```

Parameters

[in] PowerRequest

A handle to a power request object.

[in] RequestType

The power request type to be incremented. This parameter can be one of the following values.

Value	Description
PowerRequestDisplayRequired	The display remains on even if there is no user input for an extended period of time. <div style="background-color: #f0f0f0; padding: 10px; border-radius: 10px;"><p>Note: A PowerRequestSystemRequired must be taken in addition to a PowerRequestDisplayRequired to ensure the display stays on and the system does not enter sleep for the duration of the request.</p></div>
PowerRequestSystemRequired	The system continues to run instead of entering sleep after a period of user inactivity.

Value	Description
PowerRequestAwayModeRequired	The system enters away mode instead of sleep in response to explicit action by the user. In away mode, the system continues to run but turns off audio and video to give the appearance of sleep. PowerRequestAwayModeRequired is only applicable on Traditional Sleep (S3) systems.
PowerRequestExecutionRequired	<p>The calling process continues to run instead of being suspended or terminated by process lifetime management mechanisms. When and how long the process is allowed to run depends on the operating system and power policy settings.</p> <p>On Traditional Sleep (S3) systems, an active PowerRequestExecutionRequired request implies PowerRequestSystemRequired.</p>

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

On Modern Standby systems on DC power, system and execution required power requests are terminated 5 minutes after the system sleep timeout has expired.

Except for **PowerRequestAwayModeRequired** on Traditional Sleep (S3) systems, power requests are terminated upon user-initiated system sleep entry (power button, lid close or selecting **Sleep** from the **Start** menu).

To conserve power and provide the best user experience, applications that use power requests should follow these best practices:

- When creating a power request, provide a localized text string that describes the reason for the request in the [REASON_CONTEXT](#) structure.
- Call **PowerSetRequest** immediately before the scenario that requires the request.
- Call **PowerClearRequest** to decrement the reference count for the request as soon as the scenario is finished.
- Clean up all request objects and associated handles before the process exits or the service stops.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[PowerClearRequest](#)

[PowerCreateRequest](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PrepareTape function (winbase.h)

Article10/13/2021

The **PrepareTape** function prepares the tape to be accessed or removed.

Syntax

C++

```
DWORD PrepareTape(
    [in] HANDLE hDevice,
    [in] DWORD dwOperation,
    [in] BOOL bImmediate
);
```

Parameters

[in] `hDevice`

Handle to the device preparing the tape. This handle is created by using the [CreateFile](#) function.

[in] `dwOperation`

Tape device preparation. This parameter can be one of the following values.

Value	Meaning
TAPE_FORMAT 5L	Performs a low-level format of the tape. Currently, only the QIC117 device supports this feature.
TAPE_LOAD 0L	Loads the tape and moves the tape to the beginning.
TAPE_LOCK 3L	Locks the tape ejection mechanism so that the tape is not ejected accidentally.
TAPE_TENSION 2L	Adjusts the tension by moving the tape to the end of the tape and back to the beginning. This option is not supported by all devices. This value is ignored if it is not supported.
TAPE_UNLOAD 1L	Moves the tape to the beginning for removal from the device. After a successful unload operation, the device

	returns errors to applications that attempt to access the tape, until the tape is loaded again.
TAPE_UNLOCK 4L	Unlocks the tape ejection mechanism.

[in] bImmediate

If this parameter is **TRUE**, the function returns immediately. If it is **FALSE**, the function does not return until the operation has been completed.

Return value

If the function succeeds, the return value is NO_ERROR.

If the function fails, it can return one of the following error codes.

Error	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE	The tape could not be partitioned.

1105L	
1103L	ERROR_SETMARK_DETECTED A setmark was reached during an operation.
1108L	ERROR_UNABLE_TO_LOCK_MEDIA An attempt to lock the ejection mechanism failed.
1109L	ERROR_UNABLE_TO_UNLOAD_MEDIA An attempt to unload the tape failed.
19L	ERROR_WRITE_PROTECT The media is write protected.

Remarks

Some tape devices do not support certain tape operations. See your tape device documentation and use the [GetTapeParameters](#) function to determine your tape device's capabilities.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[GetTapeParameters](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

PRIORITY_HINT enumeration (winbase.h)

Article01/31/2022

Defines values that are used with the [FILE_IO_PRIORITY_HINT_INFO](#) structure to specify the priority hint for a file I/O operation.

Syntax

C++

```
typedef enum _PRIORITY_HINT {
    IoPriorityHintVeryLow = 0,
    IoPriorityHintLow,
    IoPriorityHintNormal,
    MaximumIoPriorityHintType
} PRIORITY_HINT;
```

Constants

`IoPriorityHintVeryLow`

Value: 0

The lowest possible priority hint level. The system uses this value for background I/O operations.

`IoPriorityHintLow`

A low-priority hint level.

`IoPriorityHintNormal`

A normal-priority hint level. This value is the default setting for an I/O operation.

`MaximumIoPriorityHintType`

This value is used for validation. Supported values are less than this value.

Requirements

Minimum supported client

Windows Vista [desktop apps only]

Minimum supported server

Windows Server 2008 [desktop apps only]

Header

winbase.h (include Windows.h)

See also

[FILE_IO_PRIORITY_HINT_INFO](#)

[SetFileInformationByHandle](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PrivilegedServiceAuditAlarmA function (winbase.h)

Article07/27/2022

The **PrivilegedServiceAuditAlarm** function generates an audit message in the security event log. A protected server can use this function to log attempts by a client to use a specified set of [privileges](#).

Alarms are not currently supported.

Syntax

C++

```
BOOL PrivilegedServiceAuditAlarmA(
    [in] LPCSTR SubsystemName,
    [in] LPCSTR ServiceName,
    [in] HANDLE ClientToken,
    [in] PPRIVILEGE_SET Privileges,
    [in] BOOL AccessGranted
);
```

Parameters

[in] SubsystemName

A pointer to a null-terminated string specifying the name of the subsystem calling the function. This information appears in the security event log record.

[in] ServiceName

A pointer to a null-terminated string specifying the name of the privileged subsystem service. This information appears in the security event log record.

[in] ClientToken

Identifies an [access token](#) representing the client that requested the operation. This handle must have been obtained by opening the token of a thread impersonating the client. The token must be open for TOKEN_QUERY access. The function uses this token to get the identity of the client for the security event log record.

[in] Privileges

A pointer to a [PRIVILEGE_SET](#) structure containing the privileges that the client attempted to use. The names of the privileges appear in the security event log record.

[in] AccessGranted

Indicates whether the client's attempt to use the privileges was successful. If this value is **TRUE**, the security event log record indicates success. If this value is **FALSE**, the security event log record indicates failure.

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

The **PrivilegedServiceAuditAlarm** function does not check the client's access token to determine whether the privileges are held or enabled. Typically, you first call the [PrivilegeCheck](#) function to determine whether the specified privileges are enabled in the access token, and then call **PrivilegedServiceAuditAlarm** to log the results.

The **PrivilegedServiceAuditAlarm** function requires the calling process to have **SE_AUDIT_NAME** privilege enabled. The test for this privilege is always performed against the [primary token](#) of the calling process. This allows the calling process to impersonate a client during the call.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Client/Server Access Control Functions](#)

[Client/Server Access Control Overview](#)

[ObjectPrivilegeAuditAlarm](#)

[PRIVILEGE_SET](#)

[PrivilegeCheck](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PulseEvent function (winbase.h)

Article10/13/2021

Sets the specified event object to the signaled state and then resets it to the nonsignaled state after releasing the appropriate number of waiting threads.

Note This function is unreliable and should not be used. It exists mainly for backward compatibility. For more information, see Remarks.

Syntax

C++

```
BOOL PulseEvent(  
    [in] HANDLE hEvent  
);
```

Parameters

[in] hEvent

A handle to the event object. The [CreateEvent](#) or [OpenEvent](#) function returns this handle.

The handle must have the EVENT_MODIFY_STATE access right. For more information, see [Synchronization Object Security and Access Rights](#).

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

A thread waiting on a synchronization object can be momentarily removed from the wait state by a kernel-mode APC, and then returned to the wait state after the APC is complete. If the call to **PulseEvent** occurs during the time when the thread has been removed from the wait state, the thread will not be released because **PulseEvent** releases only those threads that are waiting at the moment it is called. Therefore, **PulseEvent** is unreliable and should not be used by new applications. Instead, use [condition variables](#).

For a manual-reset event object, all waiting threads that can be released immediately are released. The function then resets the event object's state to nonsignaled and returns.

For an auto-reset event object, the function resets the state to nonsignaled and returns after releasing a single waiting thread, even if multiple threads are waiting.

If no threads are waiting, or if no thread can be released immediately, **PulseEvent** simply sets the event object's state to nonsignaled and returns.

Note that for a thread using the multiple-object [wait functions](#) to wait for all specified objects to be signaled, **PulseEvent** can set the event object's state to signaled and reset it to nonsignaled without causing the wait function to return. This happens if not all of the specified objects are simultaneously signaled.

Use extreme caution when using [SignalObjectAndWait](#) and **PulseEvent** with Windows 7, since using these APIs among multiple threads can cause an application to deadlock. Threads that are signaled by [SignalObjectAndWait](#) call **PulseEvent** to signal the waiting object of the [SignalObjectAndWait](#) call. In some circumstances, the caller of [SignalObjectAndWait](#) can't receive signal state of the waiting object in time, causing a deadlock.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateEvent](#)

[Event Objects](#)

[OpenEvent](#)

[ResetEvent](#)

[SetEvent](#)

[Synchronization Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PurgeComm function (winbase.h)

Article10/13/2021

Discards all characters from the output or input buffer of a specified communications resource. It can also terminate pending read or write operations on the resource.

Syntax

C++

```
BOOL PurgeComm(  
    [in] HANDLE hFile,  
    [in] DWORD dwFlags  
);
```

Parameters

[in] hFile

A handle to the communications resource. The [CreateFile](#) function returns this handle.

[in] dwFlags

This parameter can be one or more of the following values.

Value	Meaning
PURGE_RXABORT 0x0002	Terminates all outstanding overlapped read operations and returns immediately, even if the read operations have not been completed.
PURGE_RXCLEAR 0x0008	Clears the input buffer (if the device driver has one).
PURGE_TXABORT 0x0001	Terminates all outstanding overlapped write operations and returns immediately, even if the write operations have not been completed.
PURGE_TXCLEAR 0x0004	Clears the output buffer (if the device driver has one).

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

If a thread uses **PurgeComm** to flush an output buffer, the deleted characters are not transmitted. To empty the output buffer while ensuring that the contents are transmitted, call the [FlushFileBuffers](#) function (a synchronous operation). Note, however, that **FlushFileBuffers** is subject to flow control but not to write time-outs, and it will not return until all pending write operations have been transmitted.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

QueryActCtxSettingsW function (winbase.h)

Article10/13/2021

The **QueryActCtxSettingsW** function specifies the activation context, and the namespace and name of the attribute that is to be queried.

Syntax

C++

```
BOOL QueryActCtxSettingsW(
    [in, optional] DWORD dwFlags,
    [in, optional] HANDLE hActCtx,
    [in, optional] PCWSTR settingsNameSpace,
    [in]           PCWSTR settingName,
    [out]          PWSTR pvBuffer,
    [in]           SIZE_T dwBuffer,
    [out, optional] SIZE_T *pdwWrittenOrRequired
);
```

Parameters

[in, optional] dwFlags

This value must be 0.

[in, optional] hActCtx

A handle to the activation context that is being queried.

[in, optional] settingsNameSpace

A pointer to a string that contains the value

"<http://schemas.microsoft.com/SMI/2005/WindowsSettings>" or NULL. These values are equivalent.

Windows 8 and Windows Server 2012: A pointer to a string that contains the value "<http://schemas.microsoft.com/SMI/2011/WindowsSettings>" is also a valid parameter. A NULL is still equivalent to the previous value.

[in] settingName

The name of the attribute to be queried.

[out] *pvBuffer*

A pointer to the buffer that receives the query result.

[in] *dwBuffer*

The size of the buffer in characters that receives the query result.

[out, optional] *pdwWrittenOrRequired*

A pointer to a value which is the number of characters written to the buffer specified by *pvBuffer* or that is required to hold the query result.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

QueryActCtxW function (winbase.h)

Article 10/13/2021

The `QueryActCtxW` function queries the activation context.

Syntax

C++

```
BOOL QueryActCtxW(
    [in]          DWORD dwFlags,
    [in]          HANDLE hActCtx,
    [in, optional] PVOID pvSubInstance,
    [in]          ULONG ulInfoClass,
    [out]         PVOID  pvBuffer,
    [in, optional] SIZE_T cbBuffer,
    [out, optional] SIZE_T *pcbWrittenOrRequired
);
```

Parameters

[in] `dwFlags`

This parameter should be set to one of the following flag bits.

Flag	Meaning
<code>QUERY_ACTCTX_FLAG_USE_ACTIVE_ACTCTX</code>	<code>QueryActCtxW</code> queries the activation context active on the thread instead of the context specified by <code>hActCtx</code> . This is usually the last activation context passed to ActivateActCtx . If ActivateActCtx has not been called, the active activation context can be the activation context used by the executable of the current process. In other cases, the operating system determines the active activation context. For example, when the callback function to a new thread is called, the active activation context may be the context that was active when you created the thread by calling CreateThread .
<code>QUERY_ACTCTX_FLAG_ACTCTX_IS_HMODULE</code>	<code>QueryActCtxW</code> interprets <code>hActCtx</code> as an <code>HMODULE</code> data type and queries an activation context that is associated with a DLL or EXE. When a DLL or EXE is loaded, the loader checks for a manifest stored in a resource. If the loader finds an <code>RT_MANIFEST</code> resource with a resource identifier set to <code>ISOLATIONAWARE_MANIFEST_RESOURCE_ID</code> , the loader associates the resulting activation context with the DLL or EXE. This is the activation context that <code>QueryActCtxW</code> queries when the <code>QUERY_ACTCTX_FLAG_ACTCTX_IS_HMODULE</code> flag has been set.
<code>QUERY_ACTCTX_FLAG_ACTCTX_IS_ADDRESS</code>	<code>QueryActCtxW</code> interprets <code>hActCtx</code> as an address within a DLL or EXE and queries an activation context that has been associated with the DLL or EXE. This can be any address within the DLL or EXE. For example, the address of any function within a DLL or EXE or the address of any static data, such as a constant string.

When a DLL or EXE is loaded, the loader checks for a manifest stored in a resource in the same way as `QUERY_ACTCTX_FLAG_ACTCTX_IS_HMODULE`.

[in] `hActCtx`

Handle to the activation context that is being queried.

[in, optional] `pvSubInstance`

Index of the assembly, or assembly and file combination, in the activation context. The meaning of the `pvSubInstance` depends on the option specified by the value of the `ulInfoClass` parameter.

This parameter may be null.

<code>ulInfoClass</code> Option	Meaning
<code>AssemblyDetailedInformationInActivationContext</code>	Pointer to a DWORD that specifies the index of the assembly within the activation context. This is the activation context that <code>QueryActCtxW</code> queries.
<code>FileInformationInAssemblyOfAssemblyInActivationContext</code>	Pointer to an ACTIVATION_CONTEXT_QUERY_INDEX structure. If <code>QueryActCtxW</code> is called with this option and the function succeeds, the returned buffer contains information for a file in the assembly. This information is in the form of the ASSEMBLY_FILE_DETAILED_INFORMATION structure.

[in] `ulInfoClass`

This parameter can have only the values shown in the following table.

Option	Meaning
<code>ActivationContextBasicInformation</code> 1	Not available.
<code>ActivationContextDetailedInformation</code> 2	If <code>QueryActCtxW</code> is called with this option and the function succeeds, the returned buffer contains detailed information about the activation context. This information is in the form of the ACTIVATION_CONTEXT_DETAILED_INFORMATION structure.
<code>AssemblyDetailedInformationInActivationContext</code> 3	If <code>QueryActCtxW</code> is called with this option and the function succeeds, the buffer contains information about the assembly that has the index specified in <code>pvSubInstance</code> . This information is in the form of the ACTIVATION_CONTEXT_ASSEMBLY_DETAILED_INFORMATION structure.
<code>FileInformationInAssemblyOfAssemblyInActivationContext</code> 4	Information about a file in one of the assemblies in Activation Context. The <code>pvSubInstance</code> parameter must point to an ACTIVATION_CONTEXT_QUERY_INDEX structure. If <code>QueryActCtxW</code> is called with this option and the function succeeds, the returned buffer contains information for a file in the assembly. This information is in the form of the ASSEMBLY_FILE_DETAILED_INFORMATION structure.

RunlevelInformationInActivationContext	5	If QueryActCtxW is called with this option and the function succeeds, the buffer contains information about requested run level of the activation context. This information is in the form of the ACTIVATION_CONTEXT_RUN_LEVEL_INFORMATION structure.
CompatibilityInformationInActivationContext	6	Windows Server 2003 and Windows XP: This value is not available.

[out] *pvBuffer*

Pointer to a buffer that holds the returned information. This parameter is optional. If *pvBuffer* is **NULL**, then *cbBuffer* must be zero. If the size of the buffer pointed to by *pvBuffer* is too small, **QueryActCtxW** returns **ERROR_INSUFFICIENT_BUFFER** and no data is written into the buffer. See the Remarks section for the method you can use to determine the required size of the buffer.

[in, optional] *cbBuffer*

Size of the buffer in bytes pointed to by *pvBuffer*. This parameter is optional.

[out, optional] *pcbWrittenOrRequired*

Number of bytes written or required. The parameter *pcbWrittenOrRequired* can only be **NULL** when *pvBuffer* is **NULL**. If *pcbWrittenOrRequired* is non-**NULL**, it is filled with the number of bytes required to store the returned buffer.

Return value

If the function succeeds, it returns **TRUE**. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling **GetLastError**. For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

The parameter *cbBuffer* specifies the size in bytes of the buffer pointed to by *pvBuffer*. If *pvBuffer* is **NULL**, then *cbBuffer* must be 0. The parameter *pcbWrittenOrRequired* can only be **NULL** if *pvBuffer* is **NULL**. If *pcbWrittenOrRequired* is non-**NULL** on return, it is filled with the number of bytes required to store the returned information. When the information data returned is larger than the provided buffer, **QueryActCtxW** returns **ERROR_INSUFFICIENT_BUFFER** and no data is written to the buffer pointed to by *pvBuffer*.

The following example shows the method of calling first with a small buffer and then recalling if the buffer is too small.

syntax

```
SIZE_T cbRequired;
PVOID pvData = NULL;
SIZE_T cbAvailable = 0;

if (!QueryActCtxW(..., pvData, cbAvailable, &cbRequired) && (GetLastError() == ERROR_INSUFFICIENT_BUFFER))
{
    // Allocate enough space to store the returned buffer, fail if too small
    if (NULL == (pvData = HeapAlloc(GetProcessHeap(), 0, cbRequired)))
    {
        SetLastError(ERROR_NOT_ENOUGH_MEMORY);
        return FALSE;
    }
    cbAvailable = cbRequired;
    // Try again, this should succeed.
    if (QueryActCtxW(..., pvData, cbAvailable, &cbRequired))
    {
        // Use the returned data in pvData
    }
    HeapFree(GetProcessHeap(), 0, pvData);
    pvData = NULL;
}
```

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)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

QueryDosDeviceA function (winbase.h)

Article07/27/2022

Retrieves information about MS-DOS device names. The function can obtain the current mapping for a particular MS-DOS device name. The function can also obtain a list of all existing MS-DOS device names.

MS-DOS device names are stored as junctions in the object namespace. The code that converts an MS-DOS path into a corresponding path uses these junctions to map MS-DOS devices and drive letters. The **QueryDosDevice** function enables an application to query the names of the junctions used to implement the MS-DOS device namespace as well as the value of each specific junction.

Syntax

C++

```
DWORD QueryDosDeviceA(
    [in, optional] LPCSTR lpDeviceName,
    [out]          LPSTR  lpTargetPath,
    [in]           DWORD   ucchMax
);
```

Parameters

[in, optional] *lpDeviceName*

An MS-DOS device name string specifying the target of the query. The device name cannot have a trailing backslash; for example, use "C:", not "C:\".

This parameter can be **NULL**. In that case, the **QueryDosDevice** function will store a list of all existing MS-DOS device names into the buffer pointed to by *lpTargetPath*.

[out] *lpTargetPath*

A pointer to a buffer that will receive the result of the query. The function fills this buffer with one or more null-terminated strings. The final null-terminated string is followed by an additional **NULL**.

If *lpDeviceName* is non-**NULL**, the function retrieves information about the particular MS-DOS device specified by *lpDeviceName*. The first null-terminated string stored into

the buffer is the current mapping for the device. The other null-terminated strings represent undeleted prior mappings for the device.

If *lpDeviceName* is **NULL**, the function retrieves a list of all existing MS-DOS device names. Each null-terminated string stored into the buffer is the name of an existing MS-DOS device, for example, \Device\HarddiskVolume1 or \Device\Floppy0.

[in] *ucchMax*

The maximum number of TCHARs that can be stored into the buffer pointed to by *lpTargetPath*.

Return value

If the function succeeds, the return value is the number of TCHARs stored into the buffer pointed to by *lpTargetPath*.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

If the buffer is too small, the function fails and the last error code is **ERROR_INSUFFICIENT_BUFFER**.

Remarks

The [DefineDosDevice](#) function enables an application to create and modify the junctions used to implement the MS-DOS device namespace.

Windows Server 2003 and Windows XP: [QueryDosDevice](#) first searches the Local MS-DOS Device namespace for the specified device name. If the device name is not found, the function will then search the Global MS-DOS Device namespace.

When all existing MS-DOS device names are queried, the list of device names that are returned is dependent on whether it is running in the "LocalSystem" context. If so, only the device names included in the Global MS-DOS Device namespace will be returned. If not, a concatenation of the device names in the Global and Local MS-DOS Device namespaces will be returned. If a device name exists in both namespaces, [QueryDosDevice](#) will return the entry in the Local MS-DOS Device namespace.

For more information on the Global and Local MS-DOS Device namespaces and changes to the accessibility of MS-DOS device names, see [Defining an MS DOS Device Name](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

Examples

For an example, see [Obtaining a File Name From a File Handle](#) or [Displaying Volume Paths](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DefineDosDevice](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

QueryFullProcessImageNameA function (winbase.h)

Article 02/09/2023

Retrieves the full name of the executable image for the specified process.

Syntax

C++

```
BOOL QueryFullProcessImageNameA(
    [in]      HANDLE hProcess,
    [in]      DWORD  dwFlags,
    [out]     LPSTR  lpExeName,
    [in, out] PDWORD lpdwSize
);
```

Parameters

[in] hProcess

A handle to the process. This handle must be created with the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see [Process Security and Access Rights](#).

[in] dwFlags

This parameter can be one of the following values.

Value	Meaning
0	The name should use the Win32 path format.
PROCESS_NAME_NATIVE 0x00000001	The name should use the native system path format.

[out] lpExeName

The path to the executable image. If the function succeeds, this string is null-terminated.

[in, out] lpdwSize

On input, specifies the size of the *lpExeName* buffer, in characters. On success, receives the number of characters written to the buffer, not including the null-terminating character.

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

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later.

Note

The winbase.h header defines `QueryFullProcessImageName` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetModuleFileNameEx](#)

[GetProcessImageFileName](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

QueryFullProcessImageNameW function (winbase.h)

Article 02/09/2023

Retrieves the full name of the executable image for the specified process.

Syntax

C++

```
BOOL QueryFullProcessImageNameW(
    [in]      HANDLE hProcess,
    [in]      DWORD  dwFlags,
    [out]     LPWSTR lpExeName,
    [in, out] PDWORD lpdwSize
);
```

Parameters

[in] hProcess

A handle to the process. This handle must be created with the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see [Process Security and Access Rights](#).

[in] dwFlags

This parameter can be one of the following values.

Value	Meaning
0	The name should use the Win32 path format.
PROCESS_NAME_NATIVE 0x00000001	The name should use the native system path format.

[out] lpExeName

The path to the executable image. If the function succeeds, this string is null-terminated.

[in, out] lpdwSize

On input, specifies the size of the *lpExeName* buffer, in characters. On success, receives the number of characters written to the buffer, not including the null-terminating character.

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

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later.

(!) Note

The winbase.h header defines QueryFullProcessImageName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetModuleFileNameEx](#)

[GetProcessImageFileName](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

QueryThreadProfiling function (winbase.h)

Article 10/13/2021

Determines whether thread profiling is enabled for the specified thread.

Syntax

C++

```
DWORD QueryThreadProfiling(
    [in] HANDLE ThreadHandle,
    [out] PBOOLEAN Enabled
);
```

Parameters

[in] ThreadHandle

The handle to the thread of interest.

[out] Enabled

Is TRUE if thread profiling is enabled for the specified thread; otherwise, FALSE.

Return value

Returns ERROR_SUCCESS if the call is successful; otherwise, a system error code (see Winerror.h).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DisableThreadProfiling](#)

[EnableThreadProfiling](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

QueryUmsThreadInformation function (winbase.h)

Article 08/23/2022

Retrieves information about the specified user-mode scheduling (UMS) worker thread.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL QueryUmsThreadInformation(
    [in]          PUMS_CONTEXT           UmsThread,
    [in]          UMS_THREAD_INFO_CLASS UmsThreadInfoClass,
    [out]         PVOID                 UmsThreadInformation,
    [in]          ULONG                UmsThreadInformationLength,
    [out, optional] PULONG               ReturnLength
);
```

Parameters

[in] `UmsThread`

A pointer to a UMS thread context.

[in] `UmsThreadInfoClass`

A `UMS_THREAD_INFO_CLASS` value that specifies the kind of information to retrieve.

[out] `UmsThreadInformation`

A pointer to a buffer to receive the specified information. The required size of this buffer depends on the specified information class.

If the information class is `UmsThreadContext` or `UmsThreadTeb`, the buffer must be `sizeof(PVOID)`.

If the information class is **UmsThreadIsSuspended** or **UmsThreadIsTerminated**, the buffer must be `sizeof(BOOLEAN)`.

[in] `UmsThreadInformationLength`

The size of the *UmsThreadInformation* buffer, in bytes.

[out, optional] `ReturnLength`

A pointer to a ULONG variable. On output, this parameter receives the number of bytes written to the *UmsThreadInformation* buffer.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include the following.

Return code	Description
<code>ERROR_INFO_LENGTH_MISMATCH</code>	The buffer is too small for the requested information.
<code>ERROR_INVALID_INFO_CLASS</code>	The specified information class is not supported.
<code>ERROR_NOT_SUPPORTED</code>	UMS is not supported.

Remarks

The **QueryUmsThreadInformation** function retrieves information about the specified UMS worker thread such as its application-defined context, its thread execution block ([TEB](#)), and whether the thread is suspended or terminated.

The underlying structures for UMS worker threads are managed by the system. Information that is not exposed through **QueryUmsThreadInformation** should be considered reserved.

Requirements

Minimum supported client

Windows 7 (64-bit only) [desktop apps only]

Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[SetUmsThreadInformation](#)

[UMS_THREAD_INFO_CLASS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReadDirectoryChangesExW function (winbase.h)

Article09/15/2022

Retrieves information that describes the changes within the specified directory, which can include extended information if that information type is specified. The function does not report changes to the specified directory itself.

To track changes on a volume, see [change journals](#).

Syntax

C++

```
BOOL ReadDirectoryChangesExW(
    [in]             HANDLE          hDirectory,
    [out]            LPVOID         lpBuffer,
    [in]             DWORD          nBufferLength,
    [in]             BOOL           bWatchSubtree,
    [in]             DWORD          dwNotifyFilter,
    [out, optional]  LPDWORD        lpBytesReturned,
    [in, out, optional] LPOVERLAPPED      lpOverlapped,
    [in, optional]   LPOVERLAPPED_COMPLETION_ROUTINE
    lpCompletionRoutine,
    [in]             READ_DIRECTORY_NOTIFY_INFORMATION_CLASS
    ReadDirectoryNotifyInformationClass
);
```

Parameters

[in] hDirectory

A handle to the directory to be monitored. This directory must be opened with the **FILE_LIST_DIRECTORY** access right, or an access right such as **GENERIC_READ** that includes the **FILE_LIST_DIRECTORY** access right.

[out] lpBuffer

A pointer to the **DWORD**-aligned formatted buffer in which **ReadDirectoryChangesExW** should return the read results. The structure of this buffer is defined by the **FILE_NOTIFY_EXTENDED_INFORMATION** structure if the value of the

ReadDirectoryNotifyInformationClass parameter is **ReadDirectoryNotifyExtendedInformation**, or by the [FILE_NOTIFY_INFORMATION](#) structure if *ReadDirectoryNotifyInformationClass* is **ReadDirectoryNotifyInformation**.

This buffer is filled either synchronously or asynchronously, depending on how the directory is opened and what value is given to the *lpOverlapped* parameter. For more information, see the Remarks section.

[in] nBufferLength

The size of the buffer to which the *lpBuffer* parameter points, in bytes.

[in] bWatchSubtree

If this parameter is **TRUE**, the function monitors the directory tree rooted at the specified directory. If this parameter is **FALSE**, the function monitors only the directory specified by the *hDirectory* parameter.

[in] dwNotifyFilter

The filter criteria that the function checks to determine if the wait operation has completed. This parameter can be one or more of the following values.

Value	Meaning
FILE_NOTIFY_CHANGE_FILE_NAME 0x00000001	Any file name change in the watched directory or subtree causes a change notification wait operation to return. Changes include renaming, creating, or deleting a file.
FILE_NOTIFY_CHANGE_DIR_NAME 0x00000002	Any directory-name change in the watched directory or subtree causes a change notification wait operation to return. Changes include creating or deleting a directory.
FILE_NOTIFY_CHANGE_ATTRIBUTES 0x00000004	Any attribute change in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SIZE 0x00000008	Any file-size change in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change in file size only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_WRITE 0x00000010	Any change to the last write-time of files in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive

	caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_ACCESS 0x00000020	Any change to the last access time of files in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_CREATION 0x00000040	Any change to the creation time of files in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SECURITY 0x00000100	Any security-descriptor change in the watched directory or subtree causes a change notification wait operation to return.

[out, optional] *lpBytesReturned*

For synchronous calls, this parameter receives the number of bytes transferred into the *lpBuffer* parameter. For asynchronous calls, this parameter is undefined. You must use an asynchronous notification technique to retrieve the number of bytes transferred.

[in, out, optional] *lpOverlapped*

A pointer to an [OVERLAPPED](#) structure that supplies data to be used during asynchronous operation. Otherwise, this value is **NULL**. The **Offset** and **OffsetHigh** members of this structure are not used.

[in, optional] *lpCompletionRoutine*

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

[in] *ReadDirectoryNotifyInformationClass*

The type of information that **ReadDirectoryChangesExW** should write to the buffer to which the *lpBuffer* parameter points. Specify **ReadDirectoryNotifyInformation** to indicate that the information should consist of [FILE_NOTIFY_INFORMATION](#) structures, or **ReadDirectoryNotifyExtendedInformation** to indicate that the information should consist of [FILE_NOTIFY_EXTENDED_INFORMATION](#) structures.

Return value

If the function succeeds, the return value is nonzero. For synchronous calls, this means that the operation succeeded. For asynchronous calls, this indicates that the operation

was successfully queued.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

If the network redirector or the target file system does not support this operation, the function fails with **ERROR_INVALID_FUNCTION**.

Remarks

To obtain a handle to a directory, use the [CreateFile](#) function with the **FILE_FLAG_BACKUP_SEMANTICS** flag.

A call to [ReadDirectoryChangesExW](#) can be completed synchronously or asynchronously. To specify asynchronous completion, open the directory with [CreateFile](#) as shown above, but additionally specify the **FILE_FLAG_OVERLAPPED** attribute in the *dwFlagsAndAttributes* parameter. Then specify an [OVERLAPPED](#) structure when you call [ReadDirectoryChangesExW](#).

When you first call [ReadDirectoryChangesExW](#), the system allocates a buffer to store change information. This buffer is associated with the directory handle until it is closed and its size does not change during its lifetime. Directory changes that occur between calls to this function are added to the buffer and then returned with the next call. If the buffer overflows, [ReadDirectoryChangesExW](#) will still return **true**, but the entire contents of the buffer are discarded and the *lpBytesReturned* parameter will be zero, which indicates that your buffer was too small to hold all of the changes that occurred.

Upon successful synchronous completion, the *lpBuffer* parameter is a formatted buffer and the number of bytes written to the buffer is available in *lpBytesReturned*. If the number of bytes transferred is zero, the buffer was either too large for the system to allocate or too small to provide detailed information on all the changes that occurred in the directory or subtree. In this case, you should compute the changes by enumerating the directory or subtree.

For asynchronous completion, you can receive notification in one of three ways:

- Using the [GetOverlappedResult](#) function. To receive notification through [GetOverlappedResult](#), do not specify a completion routine in the *lpCompletionRoutine* parameter. Be sure to set the **hEvent** member of the [OVERLAPPED](#) structure to a unique event.
- Using the [GetQueuedCompletionStatus](#) function. To receive notification through [GetQueuedCompletionStatus](#), do not specify a completion routine in

lpCompletionRoutine. Associate the directory handle *hDirectory* with a completion port by calling the [CreateIoCompletionPort](#) function.

- Using a completion routine. To receive notification through a completion routine, do not associate the directory with a completion port. Specify a completion routine in *lpCompletionRoutine*. This routine is called whenever the operation has been completed or canceled while the thread is in an alertable wait state. The **hEvent** member of the [OVERLAPPED](#) structure is not used by the system, so you can use it yourself.

For more information, see [Synchronous and Asynchronous I/O](#).

ReadDirectoryChangesExW fails with **ERROR_INVALID_PARAMETER** when the buffer length is greater than 64 KB and the application is monitoring a directory over the network. This is due to a packet size limitation with the underlying file sharing protocols.

ReadDirectoryChangesExW fails with **ERROR_NOACCESS** when the buffer is not aligned on a **DWORD** boundary.

ReadDirectoryChangesExW fails with **ERROR_NOTIFY_ENUM_DIR** when the system was unable to record all the changes to the directory. In this case, you should compute the changes by enumerating the directory or subtree.

If you opened the file using the short name, you can receive change notifications for the short name.

ReadDirectoryChangesExW is currently supported only for the NTFS file system.

Transacted Operations

If there is a transaction bound to the directory handle, then the notifications follow the appropriate transaction isolation rules.

Requirements

Minimum supported client	Windows 10, version 1709 [desktop apps only]
Minimum supported server	Windows Server 2019 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[CreateIoCompletionPort](#)

[Directory Management Functions](#)

[FILE_NOTIFY_EXTENDED_INFORMATION](#)

[FILE_NOTIFY_INFORMATION](#)

[FileIOCompletionRoutine](#)

[GetOverlappedResult](#)

[GetQueuedCompletionStatus](#)

[OVERLAPPED](#)

[READ_DIRECTORY_NOTIFY_INFORMATION_CLASS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReadDirectoryChangesW function (winbase.h)

Article09/15/2022

Retrieves information that describes the changes within the specified directory. The function does not report changes to the specified directory itself.

To track changes on a volume, see [change journals](#).

Syntax

C++

```
BOOL ReadDirectoryChangesW(
    [in]             HANDLE          hDirectory,
    [out]            LPVOID         lpBuffer,
    [in]             DWORD           nBufferLength,
    [in]             BOOL            bWatchSubtree,
    [in]             DWORD           dwNotifyFilter,
    [out, optional]  LPDWORD        lpBytesReturned,
    [in, out, optional] LPOVERLAPPED lpOverlapped,
    [in, optional]   LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

[in] hDirectory

A handle to the directory to be monitored. This directory must be opened with the **FILE_LIST_DIRECTORY** access right, or an access right such as **GENERIC_READ** that includes the **FILE_LIST_DIRECTORY** access right.

[out] lpBuffer

A pointer to the **DWORD**-aligned formatted buffer in which the read results are to be returned. The structure of this buffer is defined by the **FILE_NOTIFY_INFORMATION** structure. This buffer is filled either synchronously or asynchronously, depending on how the directory is opened and what value is given to the *lpOverlapped* parameter. For more information, see the Remarks section.

[in] nBufferLength

The size of the buffer that is pointed to by the *lpBuffer* parameter, in bytes.

[in] bWatchSubtree

If this parameter is **TRUE**, the function monitors the directory tree rooted at the specified directory. If this parameter is **FALSE**, the function monitors only the directory specified by the *hDirectory* parameter.

[in] dwNotifyFilter

The filter criteria that the function checks to determine if the wait operation has completed. This parameter can be one or more of the following values.

Value	Meaning
FILE_NOTIFY_CHANGE_FILE_NAME 0x00000001	Any file name change in the watched directory or subtree causes a change notification wait operation to return. Changes include renaming, creating, or deleting a file.
FILE_NOTIFY_CHANGE_DIR_NAME 0x00000002	Any directory-name change in the watched directory or subtree causes a change notification wait operation to return. Changes include creating or deleting a directory.
FILE_NOTIFY_CHANGE_ATTRIBUTES 0x00000004	Any attribute change in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SIZE 0x00000008	Any file-size change in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change in file size only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_WRITE 0x00000010	Any change to the last write-time of files in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_ACCESS 0x00000020	Any change to the last access time of files in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_CREATION 0x00000040	Any change to the creation time of files in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SECURITY	Any security-descriptor change in the watched directory

0x00000100

or subtree causes a change notification wait operation to return.

[out, optional] *lpBytesReturned*

For synchronous calls, this parameter receives the number of bytes transferred into the *lpBuffer* parameter. For asynchronous calls, this parameter is undefined. You must use an asynchronous notification technique to retrieve the number of bytes transferred.

[in, out, optional] *lpOverlapped*

A pointer to an [OVERLAPPED](#) structure that supplies data to be used during asynchronous operation. Otherwise, this value is **NULL**. The **Offset** and **OffsetHigh** members of this structure are not used.

[in, optional] *lpCompletionRoutine*

A pointer to a completion routine to be called when the operation has been completed or canceled 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. For synchronous calls, this means that the operation succeeded. For asynchronous calls, this indicates that the operation was successfully queued.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

If the network redirector or the target file system does not support this operation, the function fails with **ERROR_INVALID_FUNCTION**.

Remarks

To obtain a handle to a directory, use the [CreateFile](#) function with the **FILE_FLAG_BACKUP_SEMANTICS** flag.

A call to [ReadDirectoryChangesW](#) can be completed synchronously or asynchronously. To specify asynchronous completion, open the directory with [CreateFile](#) as shown above, but additionally specify the **FILE_FLAG_OVERLAPPED** attribute in the *dwFlagsAndAttributes* parameter. Then specify an [OVERLAPPED](#) structure when you call [ReadDirectoryChangesW](#).

When you first call **ReadDirectoryChangesW**, the system allocates a buffer to store change information. This buffer is associated with the directory handle until it is closed and its size does not change during its lifetime. Directory changes that occur between calls to this function are added to the buffer and then returned with the next call. If the buffer overflows, **ReadDirectoryChangesW** will still return **true**, but the entire contents of the buffer are discarded and the *lpBytesReturned* parameter will be zero, which indicates that your buffer was too small to hold all of the changes that occurred.

Upon successful synchronous completion, the *lpBuffer* parameter is a formatted buffer and the number of bytes written to the buffer is available in *lpBytesReturned*. If the number of bytes transferred is zero, the buffer was either too large for the system to allocate or too small to provide detailed information on all the changes that occurred in the directory or subtree. In this case, you should compute the changes by enumerating the directory or subtree.

For asynchronous completion, you can receive notification in one of three ways:

- Using the [GetOverlappedResult](#) function. To receive notification through **GetOverlappedResult**, do not specify a completion routine in the *lpCompletionRoutine* parameter. Be sure to set the **hEvent** member of the [OVERLAPPED](#) structure to a unique event.
- Using the [GetQueuedCompletionStatus](#) function. To receive notification through **GetQueuedCompletionStatus**, do not specify a completion routine in *lpCompletionRoutine*. Associate the directory handle *hDirectory* with a completion port by calling the [CreateIoCompletionPort](#) function.
- Using a completion routine. To receive notification through a completion routine, do not associate the directory with a completion port. Specify a completion routine in *lpCompletionRoutine*. This routine is called whenever the operation has been completed or canceled while the thread is in an alertable wait state. The **hEvent** member of the [OVERLAPPED](#) structure is not used by the system, so you can use it yourself.

For more information, see [Synchronous and Asynchronous I/O](#).

ReadDirectoryChangesW fails with **ERROR_INVALID_PARAMETER** when the buffer length is greater than 64 KB and the application is monitoring a directory over the network. This is due to a packet size limitation with the underlying file sharing protocols.

ReadDirectoryChangesW fails with **ERROR_NOACCESS** when the buffer is not aligned on a **DWORD** boundary.

ReadDirectoryChangesW fails with **ERROR_NOTIFY_ENUM_DIR** when the system was unable to record all the changes to the directory. In this case, you should compute the

changes by enumerating the directory or subtree.

If you opened the file using the short name, you can receive change notifications for the short name.

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 directory handle, then the notifications follow the appropriate transaction isolation rules.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[CreateIoCompletionPort](#)

[Directory Management Functions](#)

[FILE_NOTIFY_INFORMATION](#)

[FileIOCompletionRoutine](#)

[GetOverlappedResult](#)

[GetQueuedCompletionStatus](#)

[OVERLAPPED](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReadEncryptedFileRaw function (winbase.h)

Article 10/13/2021

Backs up (export) encrypted files. This is one of a group of Encrypted File System (EFS) functions that is intended to implement backup and restore functionality, while maintaining files in their encrypted state.

Syntax

C++

```
DWORD ReadEncryptedFileRaw(
    [in]          PFE_EXPORT_FUNC pfExportCallback,
    [in, optional] PVOID         pvCallbackContext,
    [in]          PVOID         pvContext
);
```

Parameters

[in] `pfExportCallback`

A pointer to the export callback function. The system calls the callback function multiple times, each time passing a block of the file's data to the callback function until the entire file has been read. For more information, see [ExportCallback](#).

[in, optional] `pvCallbackContext`

A pointer to an application-defined and allocated context block. The system passes this pointer to the callback function as a parameter so that the callback function can have access to application-specific data. This can be a structure and can contain any data the application needs, such as the handle to the file that will contain the backup copy of the encrypted file.

[in] `pvContext`

A pointer to a system-defined context block. The context block is returned by the [OpenEncryptedFileRaw](#) function. Do not modify it.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

If the function fails, it returns a nonzero error code defined in WinError.h. You can use [FormatMessage](#) with the **FORMAT_MESSAGE_FROM_SYSTEM** flag to get a generic text description of the error.

Remarks

The file being backed up is not decrypted; it is backed up in its encrypted state.

To back up an encrypted file, call [OpenEncryptedFileRaw](#) to open the file. Then call [ReadEncryptedFileRaw](#), passing it the address of an application-defined export callback function. The system calls this callback function multiple times until the entire file's contents have been read and backed up. When the backup is complete, call [CloseEncryptedFileRaw](#) to free resources and close the file. See [ExportCallback](#) for details about how to declare the export callback function.

To restore an encrypted file, call [OpenEncryptedFileRaw](#), specifying **CREATE_FOR_IMPORT** in the *ulFlags* parameter. Then call [WriteEncryptedFileRaw](#), passing it the address of an application-defined import callback function. The system calls this callback function multiple times until the entire file's contents have been read and restored. When the restore is complete, call [CloseEncryptedFileRaw](#) to free resources and close the file. See [ImportCallback](#) for details about how to declare the import callback function.

This function is intended for the backup of only encrypted files; see [BackupRead](#) for backup of unencrypted files.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)

See also

[BackupRead](#)

[CloseEncryptedFileRaw](#)

[File Encryption](#)

[File Management Functions](#)

[OpenEncryptedFileRaw](#)

[WriteEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReadEventLogA function (winbase.h)

Article 02/09/2023

Reads the specified number of entries from the specified event log. The function can be used to read log entries in chronological or reverse chronological order.

Syntax

C++

```
BOOL ReadEventLogA(  
    [in]  HANDLE hEventLog,  
    [in]  DWORD  dwReadFlags,  
    [in]  DWORD  dwRecordOffset,  
    [out] LPVOID lpBuffer,  
    [in]  DWORD  nNumberOfBytesToRead,  
    [out] DWORD  *pnBytesRead,  
    [out] DWORD  *pnMinNumberOfBytesNeeded  
) ;
```

Parameters

[in] hEventLog

A handle to the event log to be read. The [OpenEventLog](#) function returns this handle.

[in] dwReadFlags

Use the following flag values to indicate how to read the log file. This parameter must include one of the following values (the flags are mutually exclusive).

Value	Meaning
EVENTLOG_SEEK_READ 0x0002	Begin reading from the record specified in the <i>dwRecordOffset</i> parameter. This option may not work with large log files if the function cannot determine the log file's size. For details, see Knowledge Base article, 177199.
EVENTLOG_SEQUENTIAL_READ 0x0001	Read the records sequentially. If this is the first read operation, the EVENTLOG_FORWARDS_READ EVENTLOG_BACKWARDS_READ flags determines which record is read first.

You must specify one of the following flags to indicate the direction for successive read operations (the flags are mutually exclusive).

Value	Meaning
EVENTLOG_FORWARDS_READ 0x0004	The log is read in chronological order (oldest to newest). The default.
EVENTLOG_BACKWARDS_READ 0x0008	The log is read in reverse chronological order (newest to oldest).

[in] dwRecordOffset

The record number of the log-entry at which the read operation should start. This parameter is ignored unless *dwReadFlags* includes the **EVENTLOG_SEEK_READ** flag.

[out] lpBuffer

An application-allocated buffer that will receive one or more **EVENTLOGRECORD** structures. This parameter cannot be **NULL**, even if the *nNumberOfBytesToRead* parameter is zero.

The maximum size of this buffer is 0x7ffff bytes.

[in] nNumberOfBytesToRead

The size of the *lpBuffer* buffer, in bytes. This function will read as many log entries as will fit in the buffer; the function will not return partial entries.

[out] pnBytesRead

A pointer to a variable that receives the number of bytes read by the function.

[out] pnMinNumberOfBytesNeeded

A pointer to a variable that receives the required size of the *lpBuffer* buffer. This value is valid only if this function returns zero and [GetLastError](#) returns **ERROR_INSUFFICIENT_BUFFER**.

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

When this function returns successfully, the read position in the event log is adjusted by the number of records read.

Note The configured file name for this source may also be the configured file name for other sources (several sources can exist as subkeys under a single log). Therefore, this function may return events that were logged by more than one source.

Examples

For an example, see [Querying for Event Information](#).

Note

The winbase.h header defines ReadEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib

DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[ClearEventLog](#)

[CloseEventLog](#)

[EVENTLOGRECORD](#)

[Event Logging Functions](#)

[OpenEventLog](#)

[ReportEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReadEventLogW function (winbase.h)

Article 02/09/2023

Reads the specified number of entries from the specified event log. The function can be used to read log entries in chronological or reverse chronological order.

Syntax

C++

```
BOOL ReadEventLogW(
    [in]  HANDLE hEventLog,
    [in]  DWORD  dwReadFlags,
    [in]  DWORD  dwRecordOffset,
    [out] LPVOID lpBuffer,
    [in]  DWORD  nNumberOfBytesToRead,
    [out] DWORD  *pnBytesRead,
    [out] DWORD  *pnMinNumberOfBytesNeeded
);
```

Parameters

[in] hEventLog

A handle to the event log to be read. The [OpenEventLog](#) function returns this handle.

[in] dwReadFlags

Use the following flag values to indicate how to read the log file. This parameter must include one of the following values (the flags are mutually exclusive).

Value	Meaning
EVENTLOG_SEEK_READ 0x0002	Begin reading from the record specified in the <i>dwRecordOffset</i> parameter. This option may not work with large log files if the function cannot determine the log file's size. For details, see Knowledge Base article, 177199.
EVENTLOG_SEQUENTIAL_READ 0x0001	Read the records sequentially. If this is the first read operation, the EVENTLOG_FORWARDS_READ EVENTLOG_BACKWARDS_READ flags determines which record is read first.

You must specify one of the following flags to indicate the direction for successive read operations (the flags are mutually exclusive).

Value	Meaning
EVENTLOG_FORWARDS_READ 0x0004	The log is read in chronological order (oldest to newest). The default.
EVENTLOG_BACKWARDS_READ 0x0008	The log is read in reverse chronological order (newest to oldest).

[in] dwRecordOffset

The record number of the log-entry at which the read operation should start. This parameter is ignored unless *dwReadFlags* includes the **EVENTLOG_SEEK_READ** flag.

[out] lpBuffer

An application-allocated buffer that will receive one or more **EVENTLOGRECORD** structures. This parameter cannot be **NULL**, even if the *nNumberOfBytesToRead* parameter is zero.

The maximum size of this buffer is 0x7ffff bytes.

[in] nNumberOfBytesToRead

The size of the *lpBuffer* buffer, in bytes. This function will read as many log entries as will fit in the buffer; the function will not return partial entries.

[out] pnBytesRead

A pointer to a variable that receives the number of bytes read by the function.

[out] pnMinNumberOfBytesNeeded

A pointer to a variable that receives the required size of the *lpBuffer* buffer. This value is valid only this function returns zero and [GetLastError](#) returns **ERROR_INSUFFICIENT_BUFFER**.

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

When this function returns successfully, the read position in the event log is adjusted by the number of records read.

Note The configured file name for this source may also be the configured file name for other sources (several sources can exist as subkeys under a single log). Therefore, this function may return events that were logged by more than one source.

Examples

For an example, see [Querying for Event Information](#).

 **Note**

The winbase.h header defines ReadEventLog as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib

DLL	Advapi32.dll
API set	ext-ms-win-advapi32-eventlog-ansi-l1-1-0 (introduced in Windows 10, version 10.0.10240)

See also

[ClearEventLog](#)

[CloseEventLog](#)

[EVENTLOGRECORD](#)

[Event Logging Functions](#)

[OpenEventLog](#)

[ReportEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReadThreadProfilingData function (winbase.h)

Article10/13/2021

Reads the specified profiling data associated with the thread.

Syntax

C++

```
DWORD ReadThreadProfilingData(
    [in]    HANDLE          PerformanceDataHandle,
    [in]    DWORD           Flags,
    [out]   PPERFORMANCE_DATA PerformanceData
);
```

Parameters

[in] PerformanceDataHandle

The handle that the [EnableThreadProfiling](#) function returned.

[in] Flags

One or more of the following flags that specify the counter data to read. The flags must have been set when you called the [EnableThreadProfiling](#) function.

Value	Meaning
READ_THREAD_PROFILING_FLAG_DISPATCHING 0x00000001	Get the thread profiling data.
READ_THREAD_PROFILING_FLAG_HARDWARE_COUNTERS 0x00000002	Get the hardware performance counters data.

[out] PerformanceData

A [PERFORMANCE_DATA](#) structure that contains the thread profiling and hardware counter data.

Return value

Returns ERROR_SUCCESS if the call is successful; otherwise, a system error code (see Winerror.h).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[EnableThreadProfiling](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RegisterApplicationRecoveryCallback function (winbase.h)

Article10/13/2021

Registers the active instance of an application for recovery.

Syntax

C++

```
HRESULT RegisterApplicationRecoveryCallback(
    [in]          APPLICATION_RECOVERY_CALLBACK pRecoveryCallback,
    [in, optional] PVOID                      pvParameter,
    [in]          DWORD                     dwPingInterval,
    [in]          DWORD                     dwFlags
);
```

Parameters

[in] `pRecoveryCallback`

A pointer to the recovery callback function. For more information, see [ApplicationRecoveryCallback](#).

[in, optional] `pvParameter`

A pointer to a variable to be passed to the callback function. Can be **NULL**.

[in] `dwPingInterval`

The recovery ping interval, in milliseconds. By default, the interval is 5 seconds (RECOVERY_DEFAULT_PING_INTERVAL). The maximum interval is 5 minutes. If you specify zero, the default interval is used.

You must call the [ApplicationRecoveryInProgress](#) function within the specified interval to indicate to ARR that you are still actively recovering; otherwise, WER terminates recovery. Typically, you perform recovery in a loop with each iteration lasting no longer than the ping interval. Each iteration performs a block of recovery work followed by a call to [ApplicationRecoveryInProgress](#). Since you also use [ApplicationRecoveryInProgress](#) to determine if the user wants to cancel recovery, you should consider a smaller interval, so you do not perform a lot of work unnecessarily.

[in] dwFlags

Reserved for future use. Set to zero.

Return value

This function returns **S_OK** on success or one of the following error codes.

Return code	Description
E_FAIL	Internal error; the registration failed.
E_INVALIDARG	The ping interval cannot be more than five minutes.

Remarks

If the application encounters an unhandled exception or becomes unresponsive, Windows Error Reporting (WER) calls the specified recovery callback. You should use the callback to save data and state information. You can use the information if you also call the [RegisterApplicationRestart](#) function to request that WER restart the application.

WER will not call your recovery callback if an installer wants to update a component of your application. To save data and state information in the update case, you should handle the [WM_QUERYENDSESSION](#) and [WM_ENDSESSION](#) messages. For details, see each message. The timeout for responding to these messages is five seconds. Most of the available recovery time is in the [WM_CLOSE](#) message for which you have 30 seconds.

A console application that can be updated uses the [CTRL_C_EVENT](#) notification to initiate recovery (for details, see the [HandlerRoutine](#) callback function). The timeout for the handler to complete is 30 seconds.

Applications should consider saving data and state information on a periodic bases to shorten the amount of time required for recovery.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ApplicationRecoveryCallback](#)

[ApplicationRecoveryInProgress](#)

[RegisterApplicationRestart](#)

[UnregisterApplicationRecoveryCallback](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RegisterApplicationRestart function (winbase.h)

Article 10/13/2021

Registers the active instance of an application for restart.

Syntax

C++

```
HRESULT RegisterApplicationRestart(
    [in, optional] PCWSTR pwzCommandline,
    [in]           DWORD   dwFlags
);
```

Parameters

[in, optional] pwzCommandline

A pointer to a Unicode string that specifies the command-line arguments for the application when it is restarted. The maximum size of the command line that you can specify is RESTART_MAX_CMD_LINE characters. Do not include the name of the executable in the command line; this function adds it for you.

If this parameter is **NULL** or an empty string, the previously registered command line is removed. If the argument contains spaces, use quotes around the argument.

[in] dwFlags

This parameter can be 0 or one or more of the following values.

Value	Meaning
RESTART_NO_CRASH 1	Do not restart the process if it terminates due to an unhandled exception.
RESTART_NO_HANG 2	Do not restart the process if it terminates due to the application not responding.
RESTART_NO_PATCH 4	Do not restart the process if it terminates due to the installation of an update.
RESTART_NO_REBOOT	Do not restart the process if the computer is restarted as

Return value

This function returns `S_OK` on success or one of the following error codes.

Return code	Description
<code>E_FAIL</code>	Internal error.
<code>E_INVALIDARG</code>	The specified command line is too long.

Remarks

Your initial registration for restart must occur before the application encounters an unhandled exception or becomes unresponsive. You could then call this function from inside your recovery callback to update the command line.

For a Windows application that is being updated, the last opportunity to call this function is while processing the [WM_QUERYENDSESSION](#) message. For a console application that is being updated, the registration must occur before the installer tries to shutdown the application (you need to keep the registration current; you cannot call this function when handling the `CTRL_C_EVENT` notification).

If you register for restart and the application encounters an unhandled exception or is not responsive, the user is offered the opportunity to restart the application; the application is not automatically restarted without the user's consent. However, if the application is being updated and requires a restart, the application is restarted automatically.

To prevent cyclical restarts, the system will only restart the application if it has been running for a minimum of 60 seconds.

Note that for an application to be restarted when the update requires a computer restart, the installer must call the [ExitWindowsEx](#) function with the `EWX_RESTARTAPPS` flag set or the [InitiateShutdown](#) function with the `SHUTDOWN_RESTARTAPPS` flag set.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
--------------------------	-----------------------------------

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[UnregisterApplicationRestart](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RegisterEventSourceA function (winbase.h)

Article02/09/2023

Retrieves a registered handle to the specified event log.

Syntax

C++

```
HANDLE RegisterEventSourceA(
    [in] LPCSTR lpUNCServerName,
    [in] LPCSTR lpSourceName
);
```

Parameters

[in] lpUNCServerName

The Universal Naming Convention (UNC) name of the remote server on which this operation is to be performed. If this parameter is **NULL**, the local computer is used.

[in] lpSourceName

The name of the [event source](#) whose handle is to be retrieved. The source name must be a subkey of a log under the **Eventlog** registry key. Note that the **Security** log is for system use only.

Note This string must not contain characters prohibited in XML Attributes, with the exception of XML Escape sequences such as < >.

Return value

If the function succeeds, the return value is a handle to the event log.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

The function returns **ERROR_ACCESS_DENIED** if *lpSourceName* specifies the **Security** event log.

Remarks

If the source name cannot be found, the event logging service uses the **Application** log. Although events will be reported, the events will not include descriptions because there are no message and category message files for looking up descriptions related to the event identifiers.

To close the handle to the event log, use the [DeregisterEventSource](#) function.

Examples

For an example, see [Reporting an Event](#).

ⓘ Note

The winbase.h header defines RegisterEventSource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[DeregisterEventSource](#)

[Event Logging Functions](#)

[Event Sources](#)

[ReportEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RegisterEventSourceW function (winbase.h)

Article02/09/2023

Retrieves a registered handle to the specified event log.

Syntax

C++

```
HANDLE RegisterEventSourceW(
    [in] LPCWSTR lpUNCServerName,
    [in] LPCWSTR lpSourceName
);
```

Parameters

[in] lpUNCServerName

The Universal Naming Convention (UNC) name of the remote server on which this operation is to be performed. If this parameter is **NULL**, the local computer is used.

[in] lpSourceName

The name of the [event source](#) whose handle is to be retrieved. The source name must be a subkey of a log under the **Eventlog** registry key. Note that the **Security** log is for system use only.

Note This string must not contain characters prohibited in XML Attributes, with the exception of XML Escape sequences such as < or >.

Return value

If the function succeeds, the return value is a handle to the event log.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

The function returns **ERROR_ACCESS_DENIED** if *lpSourceName* specifies the **Security** event log.

Remarks

If the source name cannot be found, the event logging service uses the **Application** log. Although events will be reported, the events will not include descriptions because there are no message and category message files for looking up descriptions related to the event identifiers.

To close the handle to the event log, use the [DeregisterEventSource](#) function.

Examples

For an example, see [Reporting an Event](#).

ⓘ Note

The winbase.h header defines RegisterEventSource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[DeregisterEventSource](#)

[Event Logging Functions](#)

[Event Sources](#)

[ReportEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RegisterWaitForSingleObject function (winbase.h)

Article 10/13/2021

Directs a wait thread in the [thread pool](#) to wait on the object. The wait thread queues the specified callback function to the thread pool when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses.

Syntax

C++

```
BOOL RegisterWaitForSingleObject(
    [out]          PHANDLE      phNewWaitObject,
    [in]           HANDLE       hObject,
    [in]           WAITORTIMERCALLBACK Callback,
    [in, optional] PVOID        Context,
    [in]           ULONG        dwMilliseconds,
    [in]           ULONG        dwFlags
);
```

Parameters

[out] phNewWaitObject

A pointer to a variable that receives a wait handle on return. Note that a wait handle cannot be used in functions that require an object handle, such as [CloseHandle](#).

[in] hObject

A handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section.

If this handle is closed while the wait is still pending, the function's behavior is undefined.

The handles must have the **SYNCHRONIZE** access right. For more information, see [Standard Access Rights](#).

[in] Callback

A pointer to the application-defined function of type **WAITORTIMERCALLBACK** to be executed when *hObject* is in the signaled state, or *dwMilliseconds* elapses. For more information, see [WaitOrTimerCallback](#).

[in, optional] *Context*

A single value that is passed to the callback function.

[in] *dwMilliseconds*

The time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. If *dwMilliseconds* is zero, the function tests the object's state and returns immediately. If *dwMilliseconds* is **INFINITE**, the function's time-out interval never elapses.

[in] *dwFlags*

This parameter can be one or more of the following values.

For information about using these values with objects that remain signaled, see the Remarks section.

Value	Meaning
WT_EXECUTEDEFAULT 0x00000000	By default, the callback function is queued to a non-I/O worker thread.
WT_EXECUTEINIOTHREAD 0x00000001	This flag is not used. Windows Server 2003 and Windows XP: The callback function is queued to an I/O worker thread. This flag should be used if the function should be executed in a thread that waits in an alertable state. I/O worker threads were removed starting with Windows Vista and Windows Server 2008.
WT_EXECUTEINPERSISTENTTHREAD 0x00000080	The callback function is queued to a thread that never terminates. It does not guarantee that the same thread is used each time. This flag should be used only for short tasks or it could affect other wait operations. This flag must be set if the thread calls functions that use APCs. For more information, see Asynchronous Procedure Calls . Note that currently no worker thread is truly persistent, although no worker thread will terminate if there are any pending I/O requests.
WT_EXECUTEINWAITTHREAD	The callback function is invoked by the wait thread itself.

0x00000004	This flag should be used only for short tasks or it could affect other wait operations. Deadlocks can occur if some other thread acquires an exclusive lock and calls the UnregisterWait or UnregisterWaitEx function while the callback function is trying to acquire the same lock.
WT_EXECUTEFUNCTION 0x00000010	The callback function can perform a long wait. This flag helps the system to decide if it should create a new thread.
WT_EXECUTEONLYONCE 0x00000008	The thread will no longer wait on the handle after the callback function has been called once. Otherwise, the timer is reset every time the wait operation completes until the wait operation is canceled.
WT_TRANSFER_IMPERSONATION 0x00000100	Callback functions will use the current access token, whether it is a process or impersonation token. If this flag is not specified, callback functions execute only with the process token. Windows XP: This flag is not supported until Windows XP with SP2 and Windows Server 2003.

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

New wait threads are created automatically when required. The wait operation is performed by a wait thread from the thread pool. The callback routine is executed by a worker thread when the object's state becomes signaled or the time-out interval elapses. If *dwFlags* is not **WT_EXECUTEONLYONCE**, the timer is reset every time the event is signaled or the time-out interval elapses.

When the wait is completed, you must call the [UnregisterWait](#) or [UnregisterWaitEx](#) function to cancel the wait operation. (Even wait operations that use **WT_EXECUTEONLYONCE** must be canceled.) Do not make a blocking call to either of these functions from within the callback function.

Note that you should not pulse an event object passed to [RegisterWaitForSingleObject](#), because the wait thread might not detect that the event is signaled before it is reset.

You should not register an object that remains signaled (such as a manual reset event or terminated process) unless you set the **WT_EXECUTEONLYONCE** or **WT_EXECUTEINWAITTHREAD** flag. For other flags, the callback function might be called too many times before the event is reset.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object whose signaled state caused the wait condition to be satisfied. For example, the count of a semaphore object is decreased by one.

The [RegisterWaitForSingleObject](#) function can wait for the following objects:

- Change notification
- Console input
- Event
- Memory resource notification
- Mutex
- Process
- Semaphore
- Thread
- Waitable timer

For more information, see [Synchronization Objects](#).

By default, the thread pool has a maximum of 500 threads. To raise this limit, use the **WT_SET_MAX_THREADPOOL_THREAD** macro defined in WinNT.h.

syntax

```
#define WT_SET_MAX_THREADPOOL_THREADS(Flags,Limit) \  
    ((Flags)|=(Limit)<16)
```

Use this macro when specifying the *dwFlags* parameter. The macro parameters are the desired flags and the new limit (up to $(2^{<16}) - 1$ threads). However, note that your application can improve its performance by keeping the number of worker threads low.

The work item and all functions it calls must be thread-pool safe. Therefore, you cannot call an asynchronous call that requires a persistent thread, such as the [RegNotifyChangeKeyValue](#) function, from the default callback environment. Instead, set the thread pool maximum equal to the thread pool minimum using the [SetThreadpoolThreadMaximum](#) and [SetThreadpoolThreadMinimum](#) functions, or create your own thread using the [CreateThread](#) function. (For the original thread pool API, specify **WT_EXECUTEINPERSISTENTTHREAD** using the [QueueUserWorkItem](#) function.)

To compile an application that uses this function, define `_WIN32_WINNT` as 0x0500 or later. For more information, see [Using the Windows Headers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Synchronization Functions](#)

[Thread Pooling](#)

[UnregisterWait](#)

[UnregisterWaitEx](#)

[Wait Functions](#)

[WaitOrTimerCallback](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReleaseActCtx function (winbase.h)

Article10/13/2021

The **ReleaseActCtx** function decrements the reference count of the specified activation context.

Syntax

C++

```
void ReleaseActCtx(  
    [in] HANDLE hActCtx  
);
```

Parameters

[in] `hActCtx`

Handle to the [ACTCTX](#) structure that contains information on the activation context for which the reference count is to be decremented.

Return value

This function does not return a value. On successful completion, the activation context reference count is decremented. The recipient of the reference-counted object must decrement the reference count when the object is no longer required.

Remarks

When the reference count of an activation context becomes zero, the activation context structure is deallocated. Activation contexts have not been implemented as kernel objects, therefore, kernel handler functions cannot be used for activation contexts.

If the value of the `hActCtx` parameter is a null handle, this function does nothing and no error condition occurs.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

[AddRefActCtx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RemoveDirectoryTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Deletes an existing empty directory as a transacted operation.

Syntax

C++

```
BOOL RemoveDirectoryTransactedA(
    [in] LPCSTR lpPathName,
    [in] HANDLE hTransaction
);
```

Parameters

[in] lpPathName

The path of the directory to be removed. The path must specify an empty directory, and the calling process must have delete access to the directory.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The directory must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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

The **RemoveDirectoryTransacted** function marks a directory for deletion on close. Therefore, the directory is not removed until the last handle to the directory is closed.

[RemoveDirectory](#) removes a directory junction, even if the contents of the target are not empty; the function removes directory junctions regardless of the state of the target object.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF .

 **Note**

The winbase.h header defines RemoveDirectoryTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[.CreateDirectoryTransacted](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RemoveDirectoryTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Deletes an existing empty directory as a transacted operation.

Syntax

C++

```
BOOL RemoveDirectoryTransactedW(
    [in] LPCWSTR lpPathName,
    [in] HANDLE hTransaction
);
```

Parameters

[in] lpPathName

The path of the directory to be removed. The path must specify an empty directory, and the calling process must have delete access to the directory.

The directory must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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

The **RemoveDirectoryTransacted** function marks a directory for deletion on close. Therefore, the directory is not removed until the last handle to the directory is closed.

[RemoveDirectory](#) removes a directory junction, even if the contents of the target are not empty; the function removes directory junctions regardless of the state of the target object.

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF .

Note

The winbase.h header defines RemoveDirectoryTransacted as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-

neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[.CreateDirectoryTransacted](#)

[Creating and Deleting Directories](#)

[Directory Management Functions](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RemoveSecureMemoryCacheCallback function (winbase.h)

Article10/13/2021

Unregisters a callback function that was previously registered with the [AddSecureMemoryCacheCallback](#) function.

Syntax

C++

```
BOOL RemoveSecureMemoryCacheCallback(
    [in] PSECURE_MEMORY_CACHE_CALLBACK pfnCallBack
);
```

Parameters

[in] `pfnCallBack`

A pointer to the application-defined [SecureMemoryCacheCallback](#) function to remove.

Return value

If the function succeeds, it returns TRUE.

If the function fails, it returns FALSE.

Remarks

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client

Windows Vista with SP1 [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddSecureMemoryCacheCallback](#)

[SecureMemoryCacheCallback](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReOpenFile function (winbase.h)

Article 10/13/2021

Reopens the specified file system object with different access rights, sharing mode, and flags.

Syntax

C++

```
HANDLE ReOpenFile(
    [in] HANDLE hOriginalFile,
    [in] DWORD dwDesiredAccess,
    [in] DWORD dwShareMode,
    [in] DWORD dwFlagsAndAttributes
);
```

Parameters

[in] `hOriginalFile`

A handle to the object to be reopened. The object must have been created by the [CreateFile](#) function.

[in] `dwDesiredAccess`

The required access to the object. For a list of values, see [File Security and Access Rights](#). You cannot request an access mode that conflicts with the sharing mode specified in a previous open request whose handle is still open.

If this parameter is zero (0), the application can query device attributes without accessing the device. This is useful if an application wants to determine the size of a floppy disk drive and the formats it supports without requiring a floppy in the drive.

[in] `dwShareMode`

The sharing mode of the object. You cannot request a sharing mode that conflicts with the access mode specified in a previous open request whose handle is still open.

If this parameter is zero (0) and [CreateFile](#) succeeds, the object cannot be shared and cannot be opened again until the handle is closed.

To enable other processes to share the object while your process has it open, use a combination of one or more of the following values to specify the type of access they can request when they open the object. These sharing options remain in effect until you close the handle to the object.

Value	Meaning
FILE_SHARE_DELETE 0x00000004	Enables subsequent open operations on the object to request delete access. Otherwise, other processes cannot open the object if they request delete access. If the object has already been opened with delete access, the sharing mode must include this flag.
FILE_SHARE_READ 0x00000001	Enables subsequent open operations on the object to request read access. Otherwise, other processes cannot open the object if they request read access. If the object has already been opened with read access, the sharing mode must include this flag.
FILE_SHARE_WRITE 0x00000002	Enables subsequent open operations on the object to request write access. Otherwise, other processes cannot open the object if they request write access. If the object has already been opened with write access, the sharing mode must include this flag.

[in] dwFlagsAndAttributes

The file flags. This parameter can be one or more of the following values.

Value	Meaning
FILE_FLAG_BACKUP_SEMANTICS 0x02000000	Indicates that the file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks, provided it has the SE_BACKUP_NAME and SE_RESTORE_NAME privileges. For more information, see Changing Privileges in a Token . You can also set this flag to obtain a handle to a directory. Where indicated, a directory handle can be passed to some functions in place of a file handle.
FILE_FLAG_DELETE_ON_CLOSE 0x04000000	Indicates that the operating system is to delete the file immediately after all of its handles have been closed, not just the specified handle but also any other open or duplicated handles. Subsequent open requests for the file fail, unless FILE_SHARE_DELETE is used.

FILE_FLAG_NO_BUFFERING 0x20000000	<p>Instructs the system to open the file with no intermediate buffering or caching. When combined with FILE_FLAG_OVERLAPPED, the flag gives maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations take longer, because data is not being held in the cache.</p> <p>An application must meet specific requirements when working with files opened with FILE_FLAG_NO_BUFFERING:</p> <ul style="list-style-type: none"> • File access must begin at byte offsets within the file that are integer multiples of the volume sector size. • File access must be for numbers of bytes that are integer multiples of the volume sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, 1536, or 2048 bytes, but not of 335, 981, or 7171 bytes. • Buffer addresses for read and write operations should be sector aligned (aligned on addresses in memory that are integer multiples of the volume sector size). Depending on the disk, this requirement may not be enforced. <p>One way to align buffers on integer multiples of the volume sector size is to use VirtualAlloc to allocate the buffers. It allocates memory that is aligned on addresses that are integer multiples of the operating system memory page size. Because both memory page and volume sector sizes are powers of 2, this memory is also aligned on addresses that are integer multiples of a volume sector size. Memory pages are 4-8 KB in size; sectors are 512 bytes (hard disks) or 2048 bytes (CD), and therefore, volume sectors can never be larger than memory pages.</p> <p>An application can determine a volume sector size by calling the GetDiskFreeSpace function.</p>
FILE_FLAG_OPEN_NO_RECALL 0x00100000	Indicates that the file data is requested, but it should continue to reside in remote storage. It should not be transported back to local storage. This flag is intended for use by remote storage systems.
FILE_FLAG_OPEN_REPARSE_POINT 0x00200000	When this flag is used, normal reparse point processing does not occur, and ReOpenFile attempts to open the reparse point. When a file is opened, a file handle is returned, whether or not the filter that controls the reparse point is operational. This flag cannot be used with

	<p>the CREATE_ALWAYS flag. If the file is not a reparse point, then this flag is ignored.</p>
FILE_FLAG_OVERLAPPED 0x40000000	<p>Instructs the system to initialize the object, so that operations that take a significant amount of time to process return ERROR_IO_PENDING. When the operation is finished, the specified event is set to the signaled state.</p> <p>When you specify FILE_FLAG_OVERLAPPED, the file read and write functions must specify an OVERLAPPED structure. That is, when FILE_FLAG_OVERLAPPED is specified, an application must perform overlapped reading and writing.</p> <p>When FILE_FLAG_OVERLAPPED is specified, the system does not maintain the file pointer. The file position must be passed as part of the <i>lpOverlapped</i> parameter (pointing to an OVERLAPPED structure) to the file read and write functions.</p> <p>This flag also enables more than one operation to be performed simultaneously with the handle (a simultaneous read and write operation, for example).</p>
FILE_FLAG_POSIX_SEMANTICS 0x01000000	Indicates that the file is to be accessed according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support such naming. Use care when using this option because files created with this flag may not be accessible by applications written for MS-DOS or 16-bit Windows.
FILE_FLAG_RANDOM_ACCESS 0x10000000	Indicates that the file is accessed randomly. The system can use this as a hint to optimize file caching.
FILE_FLAG_SEQUENTIAL_SCAN 0x08000000	Indicates that the file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur; however, correct operation is still guaranteed. Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.
FILE_FLAG_WRITE_THROUGH 0x80000000	Instructs the system to write through any intermediate cache and go directly to disk. The system can still cache write operations, but cannot lazily flush them.

If the handle represents the client side of a named pipe, the *dwFlags* parameter can also contain Security Quality of Service information. For more information, see [Impersonation Levels](#). When the calling application specifies the **SECURITY_SQOS_PRESENT** flag, the *dwFlags* parameter can contain one or more of the following values.

Value	Meaning
SECURITY_ANONYMOUS	Impersonate the client at the Anonymous impersonation level.
SECURITY_CONTEXT_TRACKING	The security tracking mode is dynamic. If this flag is not specified, the security tracking mode is static.
SECURITY_DELEGATION	Impersonate the client at the Delegation impersonation level.
SECURITY_EFFECTIVE_ONLY	Only the enabled aspects of the client security context are available to the server. If you do not specify this flag, all aspects of the client security context are available. This allows the client to limit the groups and privileges that a server can use while impersonating the client.
SECURITY_IDENTIFICATION	Impersonate the client at the Identification impersonation level.
SECURITY_IMPERSONATION	Impersonate the client at the Impersonation impersonation level.

Return value

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

If the function fails, the return value is **INVALID_HANDLE_VALUE**. To get extended error information, call [GetLastError](#).

Remarks

The *dwFlags* parameter cannot contain any of the file attribute flags (**FILE_ATTRIBUTE_***). These can only be specified when the file is created.

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

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[File Management Functions](#)

Feedback

Was this page helpful? Yes No

[Get help at Microsoft Q&A](#)

ReplaceFileA function (winbase.h)

Article06/01/2023

Replaces one file with another file, with the option of creating a backup copy of the original file. The replacement file assumes the name of the replaced file and its identity.

Syntax

C++

```
BOOL ReplaceFileA(
    [in]          LPCSTR lpReplacedFileName,
    [in]          LPCSTR lpReplacementFileName,
    [in, optional] LPCSTR lpBackupFileName,
    [in]          DWORD dwReplaceFlags,
    [in]          LPVOID lpExclude,
    [in]          LPVOID lpReserved
);
```

Parameters

[in] lpReplacedFileName

The name of the file to be replaced.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

This file is opened with the **GENERIC_READ**, **DELETE**, and **SYNCHRONIZE** access rights. The sharing mode is **FILE_SHARE_READ** | **FILE_SHARE_WRITE** | **FILE_SHARE_DELETE**.

The caller must have write access to the file to be replaced. For more information, see [File Security and Access Rights](#).

[in] `lpReplacementFileName`

The name of the file that will replace the *lpReplacedFileName* file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 **Tip**

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The function attempts to open this file with the **SYNCHRONIZE**, **GENERIC_READ**, **GENERIC_WRITE**, **DELETE**, and **WRITE_DAC** access rights so that it can preserve all attributes and ACLs. If this fails, the function attempts to open the file with the **SYNCHRONIZE**, **GENERIC_READ**, **DELETE**, and **WRITE_DAC** access rights. No sharing mode is specified.

[in, optional] `lpBackupFileName`

The name of the file that will serve as a backup copy of the *lpReplacedFileName* file. If this parameter is **NULL**, no backup file is created. See the Remarks section for implementation details on the backup file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 **Tip**

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] `dwReplaceFlags`

The replacement options. This parameter can be one or more of the following values.

Value	Meaning
<code>REPLACEFILE_WRITE_THROUGH</code>	This value is not supported.

0x00000001

REPLACEFILE_IGNORE_MERGE_ERRORS 0x00000002	Ignores errors that occur while merging information (such as attributes and ACLs) from the replaced file to the replacement file. Therefore, if you specify this flag and do not have WRITE_DAC access, the function succeeds but the ACLs are not preserved.
REPLACEFILE_IGNORE_ACL_ERRORS 0x00000004	Ignores errors that occur while merging ACL information from the replaced file to the replacement file. Therefore, if you specify this flag and do not have WRITE_DAC access, the function succeeds but the ACLs are not preserved. To compile an application that uses this value, define the _WIN32_WINNT macro as 0x0600 or later. Windows Server 2003 and Windows XP: This value is not supported.

lpExclude

Reserved for future use.

lpReserved

Reserved for future use.

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](#). The following are possible error codes for this function.

Return code/value	Description
ERROR_UNABLE_TO_MOVE_REPLACEMENT 1176 (0x498)	The replacement file could not be renamed. If <i>lpBackupFileName</i> was specified, the replaced and replacement files retain their original file names. Otherwise, the replaced file no longer exists and the replacement file exists under its original name.
ERROR_UNABLE_TO_MOVE_REPLACEMENT_2 1177 (0x499)	The replacement file could not be moved. The replacement file still exists under its original name; however, it has inherited the file streams and attributes from the file it is replacing. The file to be replaced still exists with a different name. If

	<i>lpBackupFileName</i> is specified, it will be the name of the replaced file.
ERROR_UNABLE_TO_REMOVE_REPLACED 1175 (0x497)	The replaced file could not be deleted. The replaced and replacement files retain their original file names.

If any other error is returned, such as **ERROR_INVALID_PARAMETER**, the replaced and replacement files will retain their original file names. In this scenario, a backup file does not exist and it is not guaranteed that the replacement file will have inherited all of the attributes and streams of the replaced file.

Remarks

Tip Starting with Windows 10, version 1607, for the unicode version of this function (**ReplaceFileW**), you can opt-in to remove the **MAX_PATH** limitation. See the "Maximum Path Length Limitation" section of **Naming Files, Paths, and Namespaces** for details.

The **ReplaceFile** function combines several steps within a single function. An application can call **ReplaceFile** instead of calling separate functions to save the data to a new file, rename the original file using a temporary name, rename the new file to have the same name as the original file, and delete the original file. Another advantage is that **ReplaceFile** not only copies the new file data, but also preserves the following attributes of the original file:

- Creation time
- Short file name
- Object identifier
- DACLs
- Security resource attributes
- Encryption
- Compression
- Named streams not already in the replacement file

For example, if the replacement file is encrypted, but the replaced file is not encrypted, the resulting file is not encrypted.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource attributes

(**ATTRIBUTE_SECURITY_INFORMATION**) for the original file are not preserved until Windows 8 and Windows Server 2012.

Note

If the replacement file is protected using **Selective Wipe**, then the replaced file will be protected by the enterprise id of the replacement file.

The resulting file has the same file ID as the replacement file.

The backup file, replaced file, and replacement file must all reside on the same volume.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the DACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

ⓘ Note

The winbase.h header defines ReplaceFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[CopyFile](#)

[CopyFileEx](#)

[File Management Functions](#)

[MoveFile](#)

[MoveFileEx](#)

[MoveFileWithProgress](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReplaceFileW function (winbase.h)

Article06/01/2023

Replaces one file with another file, with the option of creating a backup copy of the original file. The replacement file assumes the name of the replaced file and its identity.

Syntax

C++

```
BOOL ReplaceFileW(
    [in]          LPCWSTR lpReplacedFileName,
    [in]          LPCWSTR lpReplacementFileName,
    [in, optional] LPCWSTR lpBackupFileName,
    [in]          DWORD   dwReplaceFlags,
    [in]          LPVOID  lpExclude,
    [in]          LPVOID  lpReserved
);
```

Parameters

[in] lpReplacedFileName

The name of the file to be replaced.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

This file is opened with the **GENERIC_READ**, **DELETE**, and **SYNCHRONIZE** access rights. The sharing mode is **FILE_SHARE_READ** | **FILE_SHARE_WRITE** | **FILE_SHARE_DELETE**.

The caller must have write access to the file to be replaced. For more information, see [File Security and Access Rights](#).

[in] `lpReplacementFileName`

The name of the file that will replace the *lpReplacedFileName* file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 **Tip**

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The function attempts to open this file with the **SYNCHRONIZE**, **GENERIC_READ**, **GENERIC_WRITE**, **DELETE**, and **WRITE_DAC** access rights so that it can preserve all attributes and ACLs. If this fails, the function attempts to open the file with the **SYNCHRONIZE**, **GENERIC_READ**, **DELETE**, and **WRITE_DAC** access rights. No sharing mode is specified.

[in, optional] `lpBackupFileName`

The name of the file that will serve as a backup copy of the *lpReplacedFileName* file. If this parameter is **NULL**, no backup file is created. See the Remarks section for implementation details on the backup file.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

 **Tip**

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] `dwReplaceFlags`

The replacement options. This parameter can be one or more of the following values.

Value	Meaning
<code>REPLACEFILE_WRITE_THROUGH</code>	This value is not supported.

0x00000001

REPLACEFILE_IGNORE_MERGE_ERRORS 0x00000002	Ignores errors that occur while merging information (such as attributes and ACLs) from the replaced file to the replacement file. Therefore, if you specify this flag and do not have WRITE_DAC access, the function succeeds but the ACLs are not preserved.
REPLACEFILE_IGNORE_ACL_ERRORS 0x00000004	Ignores errors that occur while merging ACL information from the replaced file to the replacement file. Therefore, if you specify this flag and do not have WRITE_DAC access, the function succeeds but the ACLs are not preserved. To compile an application that uses this value, define the _WIN32_WINNT macro as 0x0600 or later. Windows Server 2003 and Windows XP: This value is not supported.

lpExclude

Reserved for future use.

lpReserved

Reserved for future use.

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](#). The following are possible error codes for this function.

Return code/value	Description
ERROR_UNABLE_TO_MOVE_REPLACEMENT 1176 (0x498)	The replacement file could not be renamed. If <i>lpBackupFileName</i> was specified, the replaced and replacement files retain their original file names. Otherwise, the replaced file no longer exists and the replacement file exists under its original name.
ERROR_UNABLE_TO_MOVE_REPLACEMENT_2 1177 (0x499)	The replacement file could not be moved. The replacement file still exists under its original name; however, it has inherited the file streams and attributes from the file it is replacing. The file to be replaced still exists with a different name. If

	<i>lpBackupFileName</i> is specified, it will be the name of the replaced file.
ERROR_UNABLE_TO_REMOVE_REPLACED 1175 (0x497)	The replaced file could not be deleted. The replaced and replacement files retain their original file names.

If any other error is returned, such as **ERROR_INVALID_PARAMETER**, the replaced and replacement files will retain their original file names. In this scenario, a backup file does not exist and it is not guaranteed that the replacement file will have inherited all of the attributes and streams of the replaced file.

Remarks

Tip Starting with Windows 10, version 1607, for the unicode version of this function (**ReplaceFileW**), you can opt-in to remove the **MAX_PATH** limitation. See the "Maximum Path Length Limitation" section of **Naming Files, Paths, and Namespaces** for details.

The **ReplaceFile** function combines several steps within a single function. An application can call **ReplaceFile** instead of calling separate functions to save the data to a new file, rename the original file using a temporary name, rename the new file to have the same name as the original file, and delete the original file. Another advantage is that **ReplaceFile** not only copies the new file data, but also preserves the following attributes of the original file:

- Creation time
- Short file name
- Object identifier
- DACLs
- Security resource attributes
- Encryption
- Compression
- Named streams not already in the replacement file

For example, if the replacement file is encrypted, but the replaced file is not encrypted, the resulting file is not encrypted.

Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Security resource attributes

(**ATTRIBUTE_SECURITY_INFORMATION**) for the original file are not preserved until Windows 8 and Windows Server 2012.

Note

If the replacement file is protected using **Selective Wipe**, then the replaced file will be protected by the enterprise id of the replacement file.

The resulting file has the same file ID as the replacement file.

The backup file, replaced file, and replacement file must all reside on the same volume.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the DACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

ⓘ Note

The winbase.h header defines ReplaceFile as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[CopyFile](#)

[CopyFileEx](#)

[File Management Functions](#)

[MoveFile](#)

[MoveFileEx](#)

[MoveFileWithProgress](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReportEventA function (winbase.h)

Article 02/09/2023

Writes an entry at the end of the specified event log.

Syntax

C++

```
BOOL ReportEventA(
    [in] HANDLE hEventLog,
    [in] WORD    wType,
    [in] WORD    wCategory,
    [in] DWORD   dwEventID,
    [in] PSID    lpUserSid,
    [in] WORD    wNumStrings,
    [in] DWORD   dwDataSize,
    [in] LPCSTR  *lpStrings,
    [in] LPVOID  lpRawData
);
```

Parameters

[in] hEventLog

A handle to the event log. The [RegisterEventSource](#) function returns this handle.

As of Windows XP with SP2, this parameter cannot be a handle to the **Security** log. To write an event to the **Security** log, use the [AuthzReportSecurityEvent](#) function.

[in] wType

The type of event to be logged. This parameter can be one of the following values.

Value	Meaning
EVENTLOG_SUCCESS 0x0000	Information event
EVENTLOG_AUDIT_FAILURE 0x0010	Failure Audit event
EVENTLOG_AUDIT_SUCCESS 0x0008	Success Audit event

EVENTLOG_ERROR_TYPE	Error event
0x0001	
EVENTLOG_INFORMATION_TYPE	Information event
0x0004	
EVENTLOG_WARNING_TYPE	Warning event
0x0002	

For more information about event types, see [Event Types](#).

[in] wCategory

The event category. This is source-specific information; the category can have any value. For more information, see [Event Categories](#).

[in] dwEventID

The event identifier. The event identifier specifies the entry in the message file associated with the event source. For more information, see [Event Identifiers](#).

[in] lpUserSid

A pointer to the current user's security identifier. This parameter can be **NULL** if the security identifier is not required.

[in] wNumStrings

The number of insert strings in the array pointed to by the *lpStrings* parameter. A value of zero indicates that no strings are present.

[in] dwDataSize

The number of bytes of event-specific raw (binary) data to write to the log. If this parameter is zero, no event-specific data is present.

[in] lpStrings

A pointer to a buffer containing an array of null-terminated strings that are merged into the message before Event Viewer displays the string to the user. This parameter must be a valid pointer (or **NULL**), even if *wNumStrings* is zero. Each string is limited to 31,839 characters.

Prior to Windows Vista: Each string is limited to 32K characters.

[in] lpRawData

A pointer to the buffer containing the binary data. This parameter must be a valid pointer (or **NULL**), even if the *dwDataSize* parameter is zero.

Return value

If the function succeeds, the return value is nonzero, indicating that the entry was written to the log.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#), which returns one of the following extended error codes.

Error code	Meaning
ERROR_INVALID_PARAMETER	One of the parameters is not valid. This error is returned on Windows Server 2003 if the message data to be logged is too large. This error is returned by the RPC server on Windows Server 2003 if the <i>dwDataSize</i> parameter is larger than 261,991 (0x3ff67).
ERROR_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
RPC_S_INVALID_BOUND	The array bounds are invalid. This error is returned if the message data to be logged is too large. On Windows Vista and later, this error is returned if the <i>dwDataSize</i> parameter is larger than 61,440 (0xf000).
RPC_X_BAD_STUB_DATA	The stub received bad data. This error is returned on Windows XP if the message data to be logged is too large. This error is returned by the RPC server on Windows XP, if the <i>dwDataSize</i> parameter is larger than 262,143 (0xffff).
Other	Use FormatMessage to obtain the message string for the returned error.

Remarks

This function is used to log an event. The entry is written to the end of the configured log for the source identified by the *hEventLog* parameter. The **ReportEvent** function adds the time, the entry's length, and the offsets before storing the entry in the log. To enable the function to add the user name, you must supply the user's SID in the *lpUserSid* parameter.

There are different size limits on the size of the message data that can be logged depending on the version of Windows used by both the client where the application is run and the server where the message is logged. The server is determined by the *lpUNCServerName* parameter passed to the [RegisterEventSource](#) function. Different errors are returned when the size limit is exceeded that depend on the version of Windows.

If the string that you log contains %*n*, where *n* is an integer value (for example, %1), the event viewer treats it as an insertion string. Because an IPv6 address can contain this character sequence, you must provide a format specifier (!S!) to log an event message that contains an IPv6 address. This specifier tells the formatting code to use the string literally and not perform any further expansions (for example, "my IPv6 address is: %1!S!").

Examples

For an example, see [Reporting an Event](#).

Note

The winbase.h header defines ReportEvent as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[ClearEventLog](#)

[CloseEventLog](#)

[Event Log File Format](#)

[Event Logging Functions](#)

[OpenEventLog](#)

[ReadEventLog](#)

[RegisterEventSource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ReportEventW function (winbase.h)

Article 02/09/2023

Writes an entry at the end of the specified event log.

Syntax

C++

```
BOOL ReportEventW(
    [in] HANDLE hEventLog,
    [in] WORD    wType,
    [in] WORD    wCategory,
    [in] DWORD   dwEventID,
    [in] PSID    lpUserSid,
    [in] WORD    wNumStrings,
    [in] DWORD   dwDataSize,
    [in] LPCWSTR *lpStrings,
    [in] LPVOID  lpRawData
);
```

Parameters

[in] hEventLog

A handle to the event log. The [RegisterEventSource](#) function returns this handle.

As of Windows XP with SP2, this parameter cannot be a handle to the **Security** log. To write an event to the **Security** log, use the [AuthzReportSecurityEvent](#) function.

[in] wType

The type of event to be logged. This parameter can be one of the following values.

Value	Meaning
EVENTLOG_SUCCESS 0x0000	Information event
EVENTLOG_AUDIT_FAILURE 0x0010	Failure Audit event
EVENTLOG_AUDIT_SUCCESS 0x0008	Success Audit event

EVENTLOG_ERROR_TYPE	Error event
0x0001	
EVENTLOG_INFORMATION_TYPE	Information event
0x0004	
EVENTLOG_WARNING_TYPE	Warning event
0x0002	

For more information about event types, see [Event Types](#).

[in] wCategory

The event category. This is source-specific information; the category can have any value. For more information, see [Event Categories](#).

[in] dwEventID

The event identifier. The event identifier specifies the entry in the message file associated with the event source. For more information, see [Event Identifiers](#).

[in] lpUserSid

A pointer to the current user's security identifier. This parameter can be **NULL** if the security identifier is not required.

[in] wNumStrings

The number of insert strings in the array pointed to by the *lpStrings* parameter. A value of zero indicates that no strings are present.

[in] dwDataSize

The number of bytes of event-specific raw (binary) data to write to the log. If this parameter is zero, no event-specific data is present.

[in] lpStrings

A pointer to a buffer containing an array of null-terminated strings that are merged into the message before Event Viewer displays the string to the user. This parameter must be a valid pointer (or **NULL**), even if *wNumStrings* is zero. Each string is limited to 31,839 characters.

Prior to Windows Vista: Each string is limited to 32K characters.

[in] lpRawData

A pointer to the buffer containing the binary data. This parameter must be a valid pointer (or **NULL**), even if the *dwDataSize* parameter is zero.

Return value

If the function succeeds, the return value is nonzero, indicating that the entry was written to the log.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#), which returns one of the following extended error codes.

Error code	Meaning
ERROR_INVALID_PARAMETER	One of the parameters is not valid. This error is returned on Windows Server 2003 if the message data to be logged is too large. This error is returned by the RPC server on Windows Server 2003 if the <i>dwDataSize</i> parameter is larger than 261,991 (0x3ff67).
ERROR_NOT_ENOUGH_MEMORY	Insufficient memory resources are available to complete the operation.
RPC_S_INVALID_BOUND	The array bounds are invalid. This error is returned if the message data to be logged is too large. On Windows Vista and later, this error is returned if the <i>dwDataSize</i> parameter is larger than 61,440 (0xf000).
RPC_X_BAD_STUB_DATA	The stub received bad data. This error is returned on Windows XP if the message data to be logged is too large. This error is returned by the RPC server on Windows XP, if the <i>dwDataSize</i> parameter is larger than 262,143 (0xffff).
Other	Use FormatMessage to obtain the message string for the returned error.

Remarks

This function is used to log an event. The entry is written to the end of the configured log for the source identified by the *hEventLog* parameter. The **ReportEvent** function adds the time, the entry's length, and the offsets before storing the entry in the log. To enable the function to add the user name, you must supply the user's SID in the *lpUserSid* parameter.

There are different size limits on the size of the message data that can be logged depending on the version of Windows used by both the client where the application is run and the server where the message is logged. The server is determined by the *lpUNCServerName* parameter passed to the [RegisterEventSource](#) function. Different errors are returned when the size limit is exceeded that depend on the version of Windows.

If the string that you log contains %*n*, where *n* is an integer value (for example, %1), the event viewer treats it as an insertion string. Because an IPv6 address can contain this character sequence, you must provide a format specifier (!S!) to log an event message that contains an IPv6 address. This specifier tells the formatting code to use the string literally and not perform any further expansions (for example, "my IPv6 address is: %1!S!").

Examples

For an example, see [Reporting an Event](#).

Note

The winbase.h header defines ReportEvent as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[ClearEventLog](#)

[CloseEventLog](#)

[Event Log File Format](#)

[Event Logging Functions](#)

[OpenEventLog](#)

[ReadEventLog](#)

[RegisterEventSource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

RequestWakeupLatency function (winbase.h)

Article10/13/2021

[**RequestWakeupLatency** is available for use in the operating systems specified in the Requirements section. It may be altered or unavailable in subsequent versions.]

Has no effect and returns **STATUS_NOT_SUPPORTED**. This function is provided only for compatibility with earlier versions of Windows.

Windows Server 2008 and Windows Vista: Has no effect and always returns success.

Syntax

C++

```
BOOL RequestWakeupLatency(
    [in] LATENCY_TIME latency
);
```

Parameters

[in] latency

The latency requirement for the time it takes to wake the computer. This parameter can be one of the following values.

Value	Meaning
LT_LOWEST_LATENCY 1	PowerSystemSleeping1 state (equivalent to ACPI state S0 and APM state Working).
LT_DONT_CARE 0	Any latency (default).

Return value

The return value is nonzero.

Remarks

The system uses the wake-up latency requirement when choosing a sleeping state. The latency is not guaranteed because wake-up time is determined by the hardware design of the particular computer.

To cancel a latency request, call [RequestWakeupLatency](#) with `LT_DONT_CARE`.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Power Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetCommBreak function (winbase.h)

Article10/13/2021

Suspends character transmission for a specified communications device and places the transmission line in a break state until the [ClearCommBreak](#) function is called.

Syntax

C++

```
BOOL SetCommBreak(
    [in] HANDLE hFile
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

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](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ClearCommBreak](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetCommConfig function (winbase.h)

Article10/13/2021

Sets the current configuration of a communications device.

Syntax

C++

```
BOOL SetCommConfig(
    [in] HANDLE      hCommDev,
    [in] LPCOMMCONFIG lpCC,
    [in] DWORD       dwSize
);
```

Parameters

[in] `hCommDev`

A handle to the open communications device. The [CreateFile](#) function returns this handle.

[in] `lpCC`

A pointer to a [COMMCONFIG](#) structure.

[in] `dwSize`

The size of the structure pointed to by *lpCC*, in bytes.

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](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[GetCommConfig](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetCommMask function (winbase.h)

Article10/13/2021

Specifies a set of events to be monitored for a communications device.

Syntax

C++

```
BOOL SetCommMask(  
    [in] HANDLE hFile,  
    [in] DWORD dwEvtMask  
)
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in] dwEvtMask

The events to be enabled. A value of zero disables all events. This parameter can be one or more of the following values.

Value	Meaning
EV_BREAK 0x0040	A break was detected on input.
EV_CTS 0x0008	The CTS (clear-to-send) signal changed state.
EV_DSR 0x0010	The DSR (data-set-ready) signal changed state.
EV_ERR 0x0080	A line-status error occurred. Line-status errors are CE_FRAME , CE_OVERRUN , and CE_RXPARITY .
EV_RING 0x0100	A ring indicator was detected.
EV_RLSD 0x0020	The RLSD (receive-line-signal-detect) signal changed state.

EV_RXCHAR 0x0001	A character was received and placed in the input buffer.
EV_RXFLAG 0x0002	The event character was received and placed in the input buffer. The event character is specified in the device's DCB structure, which is applied to a serial port by using the SetCommState function.
EV_TXEMPTY 0x0004	The last character in the output buffer was sent.

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

The [SetCommMask](#) function specifies the set of events that can be monitored for a particular communications resource. A handle to the communications resource can be specified in a call to the [WaitCommEvent](#) function, which waits for one of the events to occur. To get the current event mask of a communications resource, use the [GetCommMask](#) function.

Examples

For an example, see [Monitoring Communications Events](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

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

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[DCB](#)

[GetCommMask](#)

[SetCommState](#)

[WaitCommEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetCommState function (winbase.h)

Article10/13/2021

Configures a communications device according to the specifications in a device-control block (a [DCB](#) structure). The function reinitializes all hardware and control settings, but it does not empty output or input queues.

Syntax

C++

```
BOOL SetCommState(
    [in] HANDLE hFile,
    [in] LPDCB  lpDCB
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in] lpDCB

A pointer to a [DCB](#) structure that contains the configuration information for the specified communications device.

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

The [SetCommState](#) function uses a [DCB](#) structure to specify the desired configuration.

The [GetCommState](#) function returns the current configuration.

To set only a few members of the [DCB](#) structure, you should modify a **DCB** structure that has been filled in by a call to [GetCommState](#). This ensures that the other members of the **DCB** structure have appropriate values.

The [SetCommState](#) function fails if the **XonChar** member of the **DCB** structure is equal to the **XoffChar** member.

When [SetCommState](#) is used to configure the 8250, the following restrictions apply to the values for the **DCB** structure's **ByteSize** and **StopBits** members:

The number of data bits must be 5 to 8 bits.

Examples

For an example, see [Configuring a Communications Resource](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BuildCommDCB](#)

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[DCB](#)

[GetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetCommTimeouts function (winbase.h)

Article10/13/2021

Sets the time-out parameters for all read and write operations on a specified communications device.

Syntax

C++

```
BOOL SetCommTimeouts(
    [in] HANDLE         hFile,
    [in] LPCOMMTIMEOUTS lpCommTimeouts
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in] lpCommTimeouts

A pointer to a [COMMTIMEOUTS](#) structure that contains the new time-out values.

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](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMTIMEOUTS](#)

[Communications Functions](#)

[Communications Resources](#)

[GetCommTimeouts](#)

[ReadFile](#)

[ReadFileEx](#)

[WriteFile](#)

[WriteFileEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetCurrentDirectory function (winbase.h)

Article 10/09/2023

Changes the current directory for the current process.

Syntax

C++

```
BOOL SetCurrentDirectory(  
    [in] LPCTSTR lpPathName  
);
```

Parameters

[in] lpPathName

The path to the new current directory. This parameter may specify a relative path or a full path. In either case, the full path of the specified directory is calculated and stored as the current directory.

For more information, see [File Names, Paths, and Namespaces](#).

By default, the name is limited to MAX_PATH characters.

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation. See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The final character before the null character must be a backslash ('\\'). If you do not specify the backslash, it will be added for you; therefore, specify MAX_PATH-2 characters for the path unless you include the trailing backslash, in which case, specify MAX_PATH-1 characters for the path.

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

Each process has a single current directory made up of two parts:

- A disk designator that is either a drive letter followed by a colon, or a server name and share name (`\servername\sharename`)
- A directory on the disk designator

The current directory is shared by all threads of the process: If one thread changes the current directory, it affects all threads in the process. Multithreaded applications and shared library code should avoid calling the **SetCurrentDirectory** function due to the risk of affecting relative path calculations being performed by other threads. Conversely, multithreaded applications and shared library code should avoid using relative paths so that they are unaffected by changes to the current directory performed by other threads.

Note that the current directory for a process is locked while the process is executing. This will prevent the directory from being deleted, moved, or renamed.

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 an example, see [Changing the Current Directory](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Directory Management Functions](#)

[GetCurrentDirectory](#)

[GetFullPathName](#)

Feedback

Was this page helpful?



SetDefaultCommConfigA function (winbase.h)

Article02/09/2023

Sets the default configuration for a communications device.

Syntax

C++

```
BOOL SetDefaultCommConfigA(
    [in] LPCSTR      lpszName,
    [in] LPCOMMCONFIG lpCC,
    [in] DWORD       dwSize
);
```

Parameters

[in] *lpszName*

The name of the device. For example, COM1 through COM9 are serial ports and LPT1 through LPT9 are parallel ports.

[in] *lpCC*

A pointer to a [COMMCONFIG](#) structure.

[in] *dwSize*

The size of the structure pointed to by *lpCC*, in bytes.

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

ⓘ Note

The winbase.h header defines SetDefaultCommConfig as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

[GetDefaultCommConfig](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetDefaultCommConfigW function (winbase.h)

Article02/09/2023

Sets the default configuration for a communications device.

Syntax

C++

```
BOOL SetDefaultCommConfigW(
    [in] LPCWSTR     lpszName,
    [in] LPCOMMCONFIG lpCC,
    [in] DWORD        dwSize
);
```

Parameters

[in] *lpszName*

The name of the device. For example, COM1 through COM9 are serial ports and LPT1 through LPT9 are parallel ports.

[in] *lpCC*

A pointer to a [COMMCONFIG](#) structure.

[in] *dwSize*

The size of the structure pointed to by *lpCC*, in bytes.

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

ⓘ Note

The winbase.h header defines SetDefaultCommConfig as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP
Minimum supported server	Windows Server 2003
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[COMMCONFIG](#)

[Communications Functions](#)

[Communications Resources](#)

[GetDefaultCommConfig](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetDllDirectoryA function (winbase.h)

Article02/09/2023

Adds a directory to the search path used to locate DLLs for the application.

Syntax

C++

```
BOOL SetDllDirectoryA(
    [in, optional] LPCSTR lpPathName
);
```

Parameters

[in, optional] lpPathName

The directory to be added to the search path. If this parameter is an empty string (""), the call removes the current directory from the default DLL search order. If this parameter is NULL, the function restores the default search order.

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

The **SetDllDirectory** function affects all subsequent calls to the [LoadLibrary](#) and [LoadLibraryEx](#) functions. It also effectively disables safe DLL search mode while the specified directory is in the search path.

Note

For Win32 processes that are **not** running a packaged or protected process, calling this function will also affect the DLL search order of the children processes started from the process that has called the function.

After calling **SetDllDirectory**, the standard DLL search path is:

1. The directory from which the application loaded.
2. The directory specified by the *lpPathName* parameter.
3. The system directory. Use the [GetSystemDirectory](#) function to get the path of this directory. The name of this directory is System32.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

Each time the **SetDllDirectory** function is called, it replaces the directory specified in the previous **SetDllDirectory** call. To specify more than one directory, use the [AddDllDirectory](#) function and call [LoadLibraryEx](#) with LOAD_LIBRARY_SEARCH_USER_DIRS.

To revert to the standard search path used by [LoadLibrary](#) and [LoadLibraryEx](#), call **SetDllDirectory** with NULL. This also restores safe DLL search mode based on the **SafeDllSearchMode** registry value.

To compile an application that uses this function, define _WIN32_WINNT as 0x0502 or later. For more information, see [Using the Windows Headers](#).

 **Note**

The winbase.h header defines SetDllDirectory as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddDllDirectory](#)

[Dynamic-Link Library Search Order](#)

[GetDllDirectory](#)

[GetSystemDirectory](#)

[GetWindowsDirectory](#)

[LoadLibrary](#)

[LoadLibraryEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetDllDirectoryW function (winbase.h)

Article02/09/2023

Adds a directory to the search path used to locate DLLs for the application.

Syntax

C++

```
BOOL SetDllDirectoryW(
    [in, optional] LPCWSTR lpPathName
);
```

Parameters

[in, optional] lpPathName

The directory to be added to the search path. If this parameter is an empty string (""), the call removes the current directory from the default DLL search order. If this parameter is NULL, the function restores the default search order.

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

The **SetDllDirectory** function affects all subsequent calls to the [LoadLibrary](#) and [LoadLibraryEx](#) functions. It also effectively disables safe DLL search mode while the specified directory is in the search path.

Note

For Win32 processes that are **not** running a packaged or protected process, calling this function will also affect the DLL search order of the children processes started from the process that has called the function.

After calling **SetDllDirectory**, the standard DLL search path is:

1. The directory from which the application loaded.
2. The directory specified by the *lpPathName* parameter.
3. The system directory. Use the [GetSystemDirectory](#) function to get the path of this directory. The name of this directory is System32.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the [GetWindowsDirectory](#) function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

Each time the **SetDllDirectory** function is called, it replaces the directory specified in the previous **SetDllDirectory** call. To specify more than one directory, use the [AddDllDirectory](#) function and call [LoadLibraryEx](#) with LOAD_LIBRARY_SEARCH_USER_DIRS.

To revert to the standard search path used by [LoadLibrary](#) and [LoadLibraryEx](#), call **SetDllDirectory** with NULL. This also restores safe DLL search mode based on the **SafeDllSearchMode** registry value.

To compile an application that uses this function, define _WIN32_WINNT as 0x0502 or later. For more information, see [Using the Windows Headers](#).

 **Note**

The winbase.h header defines SetDllDirectory as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[AddDllDirectory](#)

[Dynamic-Link Library Search Order](#)

[GetDllDirectory](#)

[GetSystemDirectory](#)

[GetWindowsDirectory](#)

[LoadLibrary](#)

[LoadLibraryEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetEnvironmentVariable function (winbase.h)

Article 09/23/2022

Sets the contents of the specified environment variable for the current process.

Syntax

C++

```
BOOL SetEnvironmentVariable(
    [in]          LPCTSTR lpName,
    [in, optional] LPCTSTR lpValue
);
```

Parameters

[in] *lpName*

The name of the environment variable. The operating system creates the environment variable if it does not exist and *lpValue* is not NULL.

[in, optional] *lpValue*

The contents of the environment variable. The maximum size of a user-defined environment variable is 32,767 characters. For more information, see [Environment Variables](#).

Windows Server 2003 and Windows XP: The total size of the environment block for a process may not exceed 32,767 characters.

If this parameter is NULL, the variable is deleted from the current process's environment.

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

This function has no effect on the system environment variables or the environment variables of other processes.

Examples

For an example, see [Changing Environment Variables](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Environment Variables](#)

[GetEnvironmentVariable](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileAttributesTransactedA function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Sets the attributes for a file or directory as a transacted operation.

Syntax

C++

```
BOOL SetFileAttributesTransactedA(
    [in] LPCSTR lpFileName,
    [in] DWORD dwFileAttributes,
    [in] HANDLE hTransaction
);
```

Parameters

[in] lpFileName

The name of the file whose attributes are to be set.

By default, the name is limited to MAX_PATH characters. To extend this limit to 32,767 wide characters, prepend "\\?\" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the MAX_PATH limitation without prepending "\\?\". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to **ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE**.

[in] dwFileAttributes

The file attributes to set for the file.

For a list of file attribute value and their descriptions, see [File Attribute Constants](#). This parameter can be one or more values, combined using the bitwise-OR operator. However, all other values override **FILE_ATTRIBUTE_NORMAL**.

Not all attributes are supported by this function. For more information, see the Remarks section.

The following is a list of supported attribute values.

FILE_ATTRIBUTE_ARCHIVE (32 (0x20))

FILE_ATTRIBUTE_HIDDEN (2 (0x2))

FILE_ATTRIBUTE_NORMAL (128 (0x80))

FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (8192 (0x2000))

FILE_ATTRIBUTE_OFFLINE (4096 (0x1000))

FILE_ATTRIBUTE_READONLY (1 (0x1))

FILE_ATTRIBUTE_SYSTEM (4 (0x4))

FILE_ATTRIBUTE_TEMPORARY (256 (0x100))

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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

The following table describes how to set the attributes that cannot be set using [SetFileAttributesTransacted](#). Note that these are not transacted operations.

Attribute	How to Set
FILE_ATTRIBUTE_COMPRESSED 0x800	To set a file's compression state, use the DeviceIoControl function with the FSCTL_SET_COMPRESSION operation.
FILE_ATTRIBUTE_DEVICE 0x40	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY 0x10	Files cannot be converted into directories. To create a directory, use the CreateDirectory or CreateDirectoryEx function.
FILE_ATTRIBUTE_ENCRYPTED 0x4000	To create an encrypted file, use the CreateFile function with the FILE_ATTRIBUTE_ENCRYPTED attribute. To convert an existing file into an encrypted file, use the EncryptFile function.
FILE_ATTRIBUTE_REPARSE_POINT 0x400	To associate a reparse point with a file or directory, use the DeviceIoControl function with the FSCTL_SET_REPARSE_POINT operation.
FILE_ATTRIBUTE_SPARSE_FILE 0x200	To set a file's sparse attribute, use the DeviceIoControl function with the FSCTL_SET_SPARSE operation.

If a file is open for modification in a transaction, no other thread can successfully open the file for modification until the transaction is committed. If a transacted thread opens the file first, any subsequent threads that attempt to open the file for modification before the transaction is committed will receive a sharing violation. If a non-transacted thread opens the file for modification before the transacted thread does, and it is still open when the transacted thread attempts to open it, the transaction will receive the [ERROR_TRANSACTIONAL_CONFLICT](#) error.

For more information on transactions, see [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	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Transacted Operations

If a file is open for modification in a transaction, no other thread can open the file for modification until the transaction is committed. So if a transacted thread opens the file first, any subsequent threads that try modifying the file before the transaction is committed receives a sharing violation. If a non-transacted thread modifies the file before the transacted thread does, and the file is still open when the transaction attempts to open it, the transaction receives the error `ERROR_TRANSACTIONAL_CONFLICT`.

ⓘ Note

The `winbase.h` header defines `SetFileAttributesTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Attribute Constants](#)

[File Management Functions](#)

[GetFileAttributesTransacted](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileAttributesTransactedW function (winbase.h)

Article06/01/2023

[Microsoft strongly recommends developers utilize alternative means to achieve your application's needs. Many scenarios that TxF was developed for can be achieved through simpler and more readily available techniques. Furthermore, TxF may not be available in future versions of Microsoft Windows. For more information, and alternatives to TxF, please see [Alternatives to using Transactional NTFS](#).]

Sets the attributes for a file or directory as a transacted operation.

Syntax

C++

```
BOOL SetFileAttributesTransactedW(
    [in] LPCWSTR lpFileName,
    [in] DWORD    dwFileAttributes,
    [in] HANDLE   hTransaction
);
```

Parameters

`[in] lpFileName`

The name of the file whose attributes are to be set.

The file must reside on the local computer; otherwise, the function fails and the last error code is set to `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE`.

By default, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, prepend "`\?\`" to the path. For more information, see [Naming Files, Paths, and Namespaces](#).

Tip

Starting with Windows 10, Version 1607, you can opt-in to remove the `MAX_PATH` limitation without prepending "`\?\`". See the "Maximum Path Length Limitation" section of [Naming Files, Paths, and Namespaces](#) for details.

[in] dwFileAttributes

The file attributes to set for the file.

For a list of file attribute value and their descriptions, see [File Attribute Constants](#). This parameter can be one or more values, combined using the bitwise-OR operator. However, all other values override **FILE_ATTRIBUTE_NORMAL**.

Not all attributes are supported by this function. For more information, see the Remarks section.

The following is a list of supported attribute values.

FILE_ATTRIBUTE_ARCHIVE (32 (0x20))

FILE_ATTRIBUTE_HIDDEN (2 (0x2))

FILE_ATTRIBUTE_NORMAL (128 (0x80))

FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (8192 (0x2000))

FILE_ATTRIBUTE_OFFLINE (4096 (0x1000))

FILE_ATTRIBUTE_READONLY (1 (0x1))

FILE_ATTRIBUTE_SYSTEM (4 (0x4))

FILE_ATTRIBUTE_TEMPORARY (256 (0x100))

[in] hTransaction

A handle to the transaction. This handle is returned by the [CreateTransaction](#) 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

The following table describes how to set the attributes that cannot be set using [SetFileAttributesTransacted](#). Note that these are not transacted operations.

Attribute	How to Set
FILE_ATTRIBUTE_COMPRESSED 0x800	To set a file's compression state, use the DeviceIoControl function with the FSCTL_SET_COMPRESSION operation.
FILE_ATTRIBUTE_DEVICE 0x40	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY 0x10	Files cannot be converted into directories. To create a directory, use the CreateDirectory or CreateDirectoryEx function.
FILE_ATTRIBUTE_ENCRYPTED 0x4000	To create an encrypted file, use the CreateFile function with the FILE_ATTRIBUTE_ENCRYPTED attribute. To convert an existing file into an encrypted file, use the EncryptFile function.
FILE_ATTRIBUTE_REPARSE_POINT 0x400	To associate a reparse point with a file or directory, use the DeviceIoControl function with the FSCTL_SET_REPARSE_POINT operation.
FILE_ATTRIBUTE_SPARSE_FILE 0x200	To set a file's sparse attribute, use the DeviceIoControl function with the FSCTL_SET_SPARSE operation.

If a file is open for modification in a transaction, no other thread can successfully open the file for modification until the transaction is committed. If a transacted thread opens the file first, any subsequent threads that attempt to open the file for modification before the transaction is committed will receive a sharing violation. If a non-transacted thread opens the file for modification before the transacted thread does, and it is still open when the transacted thread attempts to open it, the transaction will receive the [ERROR_TRANSACTIONAL_CONFLICT](#) error.

For more information on transactions, see [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	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support TxF.

Transacted Operations

If a file is open for modification in a transaction, no other thread can open the file for modification until the transaction is committed. So if a transacted thread opens the file first, any subsequent threads that try modifying the file before the transaction is committed receives a sharing violation. If a non-transacted thread modifies the file before the transacted thread does, and the file is still open when the transaction attempts to open it, the transaction receives the error `ERROR_TRANSACTIONAL_CONFLICT`.

ⓘ Note

The `winbase.h` header defines `SetFileAttributesTransacted` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	<code>winbase.h</code> (include <code>Windows.h</code>)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Attribute Constants](#)

[File Management Functions](#)

[GetFileAttributesTransacted](#)

[Symbolic Links](#)

[Transactional NTFS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileBandwidthReservation function (winbase.h)

Article 10/13/2021

Requests that bandwidth for the specified file stream be reserved. The reservation is specified as a number of bytes in a period of milliseconds for I/O requests on the specified file handle.

Syntax

C++

```
BOOL SetFileBandwidthReservation(
    [in] HANDLE hFile,
    [in] DWORD nPeriodMilliseconds,
    [in] DWORD nBytesPerPeriod,
    [in] BOOL bDiscardable,
    [out] LPDWORD lpTransferSize,
    [out] LPDWORD lpNumOutstandingRequests
);
```

Parameters

[in] `hFile`

A handle to the file.

[in] `nPeriodMilliseconds`

The period of the reservation, in milliseconds. The period is the time from which the I/O is issued to the kernel until the time the I/O should be completed. The minimum supported value for the file stream can be determined by looking at the value returned through the `lpPeriodMilliseconds` parameter to the [GetFileBandwidthReservation](#) function, on a handle that has not had a bandwidth reservation set.

[in] `nBytesPerPeriod`

The bandwidth to reserve, in bytes per period. The maximum supported value for the file stream can be determined by looking at the value returned through the `lpBytesPerPeriod` parameter to the [GetFileBandwidthReservation](#) function, on a handle that has not had a bandwidth reservation set.

[in] `bDiscardable`

Indicates whether I/O should be completed with an error if a driver is unable to satisfy an I/O operation before the period expires. If one of the drivers for the specified file stream does not support this functionality, this function may return success and ignore the flag. To verify whether the setting will be honored, call the [GetFileBandwidthReservation](#) function using the same `hFile` handle and examine the `*pDiscardable` return value.

[out] `lpTransferSize`

A pointer to a variable that receives the minimum size of any individual I/O request that may be issued by the application. All I/O requests should be multiples of `TransferSize`.

[out] `lpNumOutstandingRequests`

A pointer to a variable that receives the number of `TransferSize` chunks the application should allow to be outstanding with the operating system. This allows the storage stack to keep the device busy and allows maximum throughput.

Return value

Returns nonzero if successful or zero otherwise.

A reservation can fail if there is not enough bandwidth available on the volume because of existing reservations; in this case `ERROR_NO_SYSTEM_RESOURCES` is returned.

To get extended error information, call [GetLastError](#).

Remarks

The requested bandwidth reservation must be greater than or equal to one packet per period. The minimum period, in milliseconds, maximum bytes per period, and minimum transfer size, in bytes, for a specific volume are returned through the `lpPeriodMilliseconds`, `lpBytesPerPeriod`, and `lpTransferSize` parameters to [GetFileBandwidthReservation](#) on a handle that has not been used in a call to [SetFileBandwidthReservation](#). In other words:

$$1 \leq (nBytesPerPeriod) \times (lpPeriodMilliseconds) / (lpTransferSize) / (nPeriodMilliseconds)$$

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	Yes

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetFileBandwidthReservation](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileCompletionNotificationModes function (winbase.h)

Article 10/13/2021

Sets the notification modes for a file handle, allowing you to specify how completion notifications work for the specified file.

Syntax

C++

```
BOOL SetFileCompletionNotificationModes(
    [in] HANDLE FileHandle,
    [in] UCHAR Flags
);
```

Parameters

[in] FileHandle

A handle to the file.

[in] Flags

The modes to be set. One or more modes can be set at the same time; however, after a mode has been set for a file handle, it cannot be removed.

Value	Meaning
FILE_SKIP_COMPLETION_PORT_ON_SUCCESS 0x1	If the following three conditions are true, the I/O Manager does not queue a completion entry to the port, when it would ordinarily do so. The conditions are: <ul style="list-style-type: none">• A completion port is associated with the file handle.• The file is opened for asynchronous I/O.• A request returns success immediately without returning ERROR_PENDING. When the <i>FileHandle</i> parameter is a socket, this mode is only compatible with Layered Service Providers (LSP) that return Installable File Systems (IFS) handles. To detect whether a non-IFS LSP is

installed, use the [WSAEnumProtocols](#) function and examine the `dwServiceFlag1` member in each returned [WSAPROTOCOL_INFO](#) structure. If the `XP1_IFS_HANDLES` (0x20000) bit is cleared then the specified LSP is not an IFS LSP. Vendors that have non-IFS LSPs are encouraged to migrate to the [Windows Filtering Platform](#) (WFP).

FILE_SKIP_SET_EVENT_ON_HANDLE
0x2

The I/O Manager does not set the event for the file object if a request returns with a success code, or the error returned is `ERROR_PENDING` and the function that is called is not a synchronous function.

If an explicit event is provided for the request, it is still signaled.

Return value

Returns nonzero if successful or zero otherwise.

To get extended error information, call [GetLastError](#).

Remarks

To compile an application that uses this function, define the `_WIN32_WINNT` macro as 0x0600 or later. For more information, see [Using the Windows Headers](#).

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

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileSecurityA function (winbase.h)

Article09/22/2022

The **SetFileSecurity** function sets the security of a file or directory object.

This function is obsolete. Use the [SetNamedSecurityInfo](#) function instead.

Syntax

C++

```
BOOL SetFileSecurityA(
    [in] LPCSTR             lpFileName,
    [in] SECURITY_INFORMATION SecurityInformation,
    [in] PSECURITY_DESCRIPTOR pSecurityDescriptor
);
```

Parameters

[in] *lpFileName*

A pointer to a null-terminated string that specifies the file or directory for which security is set. Note that security applied to a directory is not inherited by its children.

[in] *SecurityInformation*

Specifies a [SECURITY_INFORMATION](#) structure that identifies the contents of the [security descriptor](#) pointed to by the *pSecurityDescriptor* parameter.

[in] *pSecurityDescriptor*

A pointer to a [SECURITY_DESCRIPTOR](#) structure.

Return value

If the function succeeds, the function returns nonzero.

If the function fails, it returns zero. To get extended error information, call [GetLastError](#).

Remarks

The [SetFileSecurity](#) function is successful only if the following conditions are met:

- If the owner of the object is being set, the calling [process](#) must have either WRITE_OWNER permission or be the owner of the object.
- If the [discretionary access control list](#) (DACL) of the object is being set, the calling process must have either WRITE_DAC permission or be the owner of the object.
- If the [system access control list](#) (SACL) of the object is being set, the SE_SECURITY_NAME privilege must be enabled for the calling process.

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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[GetFileSecurity](#)

[Low-level Access Control](#)

[Low-level Access Control Functions](#)

[SECURITY_DESCRIPTOR](#)

[SECURITY_INFORMATION](#)

[SetKernelObjectSecurity](#)

[SetNamedSecurityInfo](#)

[SetPrivateObjectSecurity](#)

[SetUserObjectSecurity](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileShortNameA function (winbase.h)

Article02/09/2023

Sets the short name for the specified file. The file must be on an NTFS file system volume.

Syntax

C++

```
BOOL SetFileShortNameA(
    [in] HANDLE hFile,
    [in] LPCSTR lpShortName
);
```

Parameters

[in] hFile

A handle to the file. The file must be opened with either the **GENERIC_ALL** access right or **GENERIC_WRITE|DELETE**, and with the **FILE_FLAG_BACKUP_SEMANTICS** file attribute.

[in] lpShortName

A pointer to a string that specifies the short name for the file.

Specifying an empty (zero-length) string will remove the short file name, if it exists for the file specified by the *hFile* parameter. If a short file name does not exist, the function will do nothing and return success.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This behavior is not supported. The parameter must contain a valid string of one or more characters.

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](#). [GetLastError](#) may return one of the following error codes that are specific to this function.

Return code	Description
ERROR_ALREADY_EXISTS	The specified short name is not unique.
ERROR_INVALID_PARAMETER	Either the specified file has been opened in case-sensitive mode or the specified short name is invalid.

Remarks

The caller of this function must have the **SE_RESTORE_NAME** privilege. For more information, see [Running with Special Privileges](#).

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)	No

SMB 3.0 does not support short names on shares with continuous availability capability . Short names are not recommended on CsvFs.

ⓘ Note

The winbase.h header defines SetFileShortName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetShortPathName](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFileShortNameW function (winbase.h)

Article02/09/2023

Sets the short name for the specified file. The file must be on an NTFS file system volume.

Syntax

C++

```
BOOL SetFileShortNameW(
    [in] HANDLE hFile,
    [in] LPCWSTR lpShortName
);
```

Parameters

[in] hFile

A handle to the file. The file must be opened with either the **GENERIC_ALL** access right or **GENERIC_WRITE|DELETE**, and with the **FILE_FLAG_BACKUP_SEMANTICS** file attribute.

[in] lpShortName

A pointer to a string that specifies the short name for the file.

Specifying an empty (zero-length) string will remove the short file name, if it exists for the file specified by the *hFile* parameter. If a short file name does not exist, the function will do nothing and return success.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This behavior is not supported. The parameter must contain a valid string of one or more characters.

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](#). [GetLastError](#) may return one of the following error codes that are specific to this function.

Return code	Description
ERROR_ALREADY_EXISTS	The specified short name is not unique.
ERROR_INVALID_PARAMETER	Either the specified file has been opened in case-sensitive mode or the specified short name is invalid.

Remarks

The caller of this function must have the **SE_RESTORE_NAME** privilege. For more information, see [Running with Special Privileges](#).

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)	No

SMB 3.0 does not support short names on shares with continuous availability capability . Short names are not recommended on CsvFs.

Note

The winbase.h header defines SetFileShortName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[File Management Functions](#)

[GetShortPathName](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFirmwareEnvironmentVariableA function (winbase.h)

Article 02/09/2023

Sets the value of the specified firmware environment variable.

Syntax

C++

```
BOOL SetFirmwareEnvironmentVariableA(
    [in] LPCSTR lpName,
    [in] LPCSTR lpGuid,
    [in] PVOID pValue,
    [in] DWORD nSize
);
```

Parameters

[in] lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

[in] lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}". If the system does not support GUID-based namespaces, this parameter is ignored.

[in] pValue

A pointer to the new value for the firmware environment variable.

[in] nSize

The size of the *pBuffer* buffer, in bytes. If this parameter is zero, the firmware environment variable is deleted.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include `ERROR_INVALID_FUNCTION`.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write UEFI firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

Starting with Windows 10, version 1803, reading UEFI firmware variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI firmware variables from UMDF drivers is not supported.

To write a firmware environment variable, the user account that the app is running under must have the `SE_SYSTEM_ENVIRONMENT_NAME` privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The **SetFirmwareEnvironmentVariable** function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with `ERROR_INVALID_FUNCTION`. On a UEFI-based system, the function will fail with an error specific to the firmware, such as `ERROR_NOACCESS`, to indicate that the dummy GUID namespace does not exist.

SetFirmwareEnvironmentVariable is the user-mode equivalent of the **ExSetFirmwareEnvironmentVariable** kernel-mode routine.

Note

The winbase.h header defines SetFirmwareEnvironmentVariable as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariable](#)

[SetFirmwareEnvironmentVariableEx](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFirmwareEnvironmentVariableExA function (winbase.h)

Article 02/09/2023

Sets the value of the specified firmware environment variable as the attributes that indicate how this variable is stored and maintained.

Syntax

C++

```
BOOL SetFirmwareEnvironmentVariableExA(
    [in] LPCSTR lpName,
    [in] LPCSTR lpGuid,
    [in] PVOID pValue,
    [in] DWORD nSize,
    [in] DWORD dwAttributes
);
```

Parameters

[in] lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

[in] lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}". If the system does not support GUID-based namespaces, this parameter is ignored. The pointer must not be **NULL**.

[in] pValue

A pointer to the new value for the firmware environment variable.

[in] nSize

The size of the *pValue* buffer, in bytes. Unless the VARIABLE_ATTRIBUTE_APPEND_WRITE, VARIABLE_ATTRIBUTE_AUTHENTICATED_WRITE_ACCESS, or VARIABLE_ATTRIBUTE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS variable attribute is set via *dwAttributes*, setting this value to zero will result in the deletion of this variable.

[in] dwAttributes

Bitmask to set UEFI variable attributes associated with the variable. See also [UEFI Spec 2.3.1](#), [Section 7.2](#).

Value	Meaning
VARIABLE_ATTRIBUTE_NON_VOLATILE 0x00000001	The firmware environment variable is stored in non-volatile memory (e.g. NVRAM).
VARIABLE_ATTRIBUTE_BOOTSERVICE_ACCESS 0x00000002	The firmware environment variable can be accessed during boot service.
VARIABLE_ATTRIBUTE_RUNTIME_ACCESS 0x00000004	The firmware environment variable can be accessed at runtime. <div style="border: 1px solid #ccc; padding: 10px; border-radius: 10px;"><p>Note Variables with this attribute set, must also have VARIABLE_ATTRIBUTE_BOTSERVICE_ACCESS set.</p></div>
VARIABLE_ATTRIBUTE_HARDWARE_ERROR_RECORD 0x00000008	Indicates hardware related errors encountered at runtime.
VARIABLE_ATTRIBUTE_AUTHENTICATED_WRITE_ACCESS 0x00000010	Indicates an authentication requirement that must be met before writing to this firmware environment variable. For more information see, UEFI spec 2.3.1 .
VARIABLE_ATTRIBUTE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS 0x00000020	Indicates authentication and time stamp requirements that must be met before writing to this firmware environment variable. When this attribute is set, the buffer, represented by <i>pValue</i> , will begin with an instance of a complete (and serialized) EFI_VARIABLE_AUTHENTICATION_2 descriptor. For more information see, UEFI spec 2.3.1 .
VARIABLE_ATTRIBUTE_APPEND_WRITE 0x00000040	Append an existing environment variable with the value of <i>pValue</i> . If the firmware does not support the operation, then SetFirmwareEnvironmentVariableEx

will return
ERROR_INVALID_FUNCTION.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include ERROR_INVALID_FUNCTION.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write UEFI firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

Starting with Windows 10, version 1803, reading UEFI firmware variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI firmware variables from UMDF drivers is not supported.

To write a firmware environment variable, the user account that the app is running under must have the [SE_SYSTEM_ENVIRONMENT_NAME](#) privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

The correct method of changing the attributes of a variable is to delete the variable and recreate it with different attributes.

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The [SetFirmwareEnvironmentVariableEx](#) function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with ERROR_INVALID_FUNCTION. On a UEFI-based system, the function will fail with an error specific to the firmware, such as ERROR_NOACCESS, to indicate that the dummy GUID namespace does not exist.

Note

The winbase.h header defines SetFirmwareEnvironmentVariableEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariableEx](#)

[SetFirmwareEnvironmentVariable](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFirmwareEnvironmentVariableExW function (winbase.h)

Article 02/09/2023

Sets the value of the specified firmware environment variable and the attributes that indicate how this variable is stored and maintained.

Syntax

C++

```
BOOL SetFirmwareEnvironmentVariableExW(
    [in] LPCWSTR lpName,
    [in] LPCWSTR lpGuid,
    [in] PVOID    pValue,
    [in] DWORD    nSize,
    [in] DWORD    dwAttributes
);
```

Parameters

[in] lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

[in] lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}". If the system does not support GUID-based namespaces, this parameter is ignored. The pointer must not be **NULL**.

[in] pValue

A pointer to the new value for the firmware environment variable.

[in] nSize

The size of the *pValue* buffer, in bytes. Unless the VARIABLE_ATTRIBUTE_APPEND_WRITE, VARIABLE_ATTRIBUTE_AUTHENTICATED_WRITE_ACCESS, or VARIABLE_ATTRIBUTE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS variable attribute is set via *dwAttributes*, setting this value to zero will result in the deletion of this variable.

[in] dwAttributes

Bitmask to set UEFI variable attributes associated with the variable. See also [UEFI Spec 2.3.1](#), [Section 7.2](#).

Value	Meaning
VARIABLE_ATTRIBUTE_NON_VOLATILE 0x00000001	The firmware environment variable is stored in non-volatile memory (e.g. NVRAM).
VARIABLE_ATTRIBUTE_BOOTSERVICE_ACCESS 0x00000002	The firmware environment variable can be accessed during boot service.
VARIABLE_ATTRIBUTE_RUNTIME_ACCESS 0x00000004	The firmware environment variable can be accessed at runtime. <div style="border: 1px solid #ccc; padding: 10px; border-radius: 10px;"><p>Note Variables with this attribute set, must also have VARIABLE_ATTRIBUTE_BOTSERVICE_ACCESS set.</p></div>
VARIABLE_ATTRIBUTE_HARDWARE_ERROR_RECORD 0x00000008	Indicates hardware related errors encountered at runtime.
VARIABLE_ATTRIBUTE_AUTHENTICATED_WRITE_ACCESS 0x00000010	Indicates an authentication requirement that must be met before writing to this firmware environment variable. For more information see, UEFI spec 2.3.1 .
VARIABLE_ATTRIBUTE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS 0x00000020	Indicates authentication and time stamp requirements that must be met before writing to this firmware environment variable. When this attribute is set, the buffer, represented by <i>pValue</i> , will begin with an instance of a complete (and serialized) EFI_VARIABLE_AUTHENTICATION_2 descriptor. For more information see, UEFI spec 2.3.1 .
VARIABLE_ATTRIBUTE_APPEND_WRITE 0x00000040	Append an existing environment variable with the value of <i>pValue</i> . If the firmware does not support the operation, then SetFirmwareEnvironmentVariableEx

will return
ERROR_INVALID_FUNCTION.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include ERROR_INVALID_FUNCTION.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write UEFI firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

Starting with Windows 10, version 1803, reading UEFI firmware variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI firmware variables from UMDF drivers is not supported.

To write a firmware environment variable, the user account that the app is running under must have the [SE_SYSTEM_ENVIRONMENT_NAME](#) privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

The correct method of changing the attributes of a variable is to delete the variable and recreate it with different attributes.

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The [SetFirmwareEnvironmentVariableEx](#) function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with ERROR_INVALID_FUNCTION. On a UEFI-based system, the function will fail with an error specific to the firmware, such as ERROR_NOACCESS, to indicate that the dummy GUID namespace does not exist.

Note

The winbase.h header defines SetFirmwareEnvironmentVariableEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariableEx](#)

[SetFirmwareEnvironmentVariable](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetFirmwareEnvironmentVariableW function (winbase.h)

Article 02/09/2023

Sets the value of the specified firmware environment variable.

Syntax

C++

```
BOOL SetFirmwareEnvironmentVariableW(
    [in] LPCWSTR lpName,
    [in] LPCWSTR lpGuid,
    [in] PVOID     pValue,
    [in] DWORD     nSize
);
```

Parameters

[in] lpName

The name of the firmware environment variable. The pointer must not be **NULL**.

[in] lpGuid

The GUID that represents the namespace of the firmware environment variable. The GUID must be a string in the format "{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}". If the system does not support GUID-based namespaces, this parameter is ignored.

[in] pValue

A pointer to the new value for the firmware environment variable.

[in] nSize

The size of the *pBuffer* buffer, in bytes. If this parameter is zero, the firmware environment variable is deleted.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error codes include `ERROR_INVALID_FUNCTION`.

Remarks

Starting with Windows 10, version 1803, Universal Windows apps can read and write UEFI firmware variables. See [Access UEFI firmware variables from a Universal Windows App](#) for details.

Starting with Windows 10, version 1803, reading UEFI firmware variables is also supported from User-Mode Driver Framework (UMDF) drivers. Writing UEFI firmware variables from UMDF drivers is not supported.

To write a firmware environment variable, the user account that the app is running under must have the `SE_SYSTEM_ENVIRONMENT_NAME` privilege. A Universal Windows app must be run from an administrator account and follow the requirements outlined in [Access UEFI firmware variables from a Universal Windows App](#).

The exact set of firmware environment variables is determined by the boot firmware. The location of these environment variables is also specified by the firmware. For example, on a UEFI-based system, NVRAM contains firmware environment variables that specify system boot settings. For information about specific variables used, see the [UEFI specification](#). For more information about UEFI and Windows, see [UEFI and Windows](#).

Firmware variables are not supported on a legacy BIOS-based system. The **SetFirmwareEnvironmentVariable** function will always fail on a legacy BIOS-based system, or if Windows was installed using legacy BIOS on a system that supports both legacy BIOS and UEFI. To identify these conditions, call the function with a dummy firmware environment name such as an empty string ("") for the *lpName* parameter and a dummy GUID such as "{00000000-0000-0000-0000-000000000000}" for the *lpGuid* parameter. On a legacy BIOS-based system, or on a system that supports both legacy BIOS and UEFI where Windows was installed using legacy BIOS, the function will fail with `ERROR_INVALID_FUNCTION`. On a UEFI-based system, the function will fail with an error specific to the firmware, such as `ERROR_NOACCESS`, to indicate that the dummy GUID namespace does not exist.

SetFirmwareEnvironmentVariable is the user-mode equivalent of the **ExSetFirmwareEnvironmentVariable** kernel-mode routine.

Note

The winbase.h header defines SetFirmwareEnvironmentVariable as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Access UEFI firmware variables from a Universal Windows App](#)

[GetFirmwareEnvironmentVariable](#)

[SetFirmwareEnvironmentVariableEx](#)

[System Information Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetHandleCount function (winbase.h)

Article09/22/2022

Changes the number of file handles available to a process. For DOS-based Win32, the default maximum number of file handles available to a process is 20. For Windows Win32 systems, this API has no effect.

Syntax

C++

```
UINT SetHandleCount(  
    UINT uNumber  
) ;
```

Parameters

uNumber

The requested number of available file handles.

Return value

The number of available file handles.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winbase.h
DLL	kernel32.dll

See also

f1_keywords:

- "winbase/SetFirmwareEnvironmentVariable"
-

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetMailslotInfo function (winbase.h)

Article 10/13/2021

Sets the time-out value used by the specified mailslot for a read operation.

Syntax

C++

```
BOOL SetMailslotInfo(
    [in] HANDLE hMailslot,
    [in] DWORD  lReadTimeout
);
```

Parameters

[in] hMailslot

A handle to a mailslot. The [CreateMailslot](#) function must create this handle.

[in] lReadTimeout

The time a read operation can wait for a message to be written to the mailslot before a time-out occurs, in milliseconds. The following values have special meanings.

Value	Meaning
0	Returns immediately if no message is present. (The system does not treat an immediate return as an error.)
MAILSLOT_WAIT_FOREVER ((DWORD)-1)	Waits forever for a message.

This time-out value applies to all subsequent read operations and to all inherited mailslot handles.

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

The initial time-out value used by a mailslot for a read operation is typically set by [CreateMailslot](#) when the mailslot is created.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateMailslot](#)

[GetMailslotInfo](#)

[Mailslot Functions](#)

[Mailslots Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetProcessAffinityMask function (winbase.h)

Article01/27/2022

Sets a processor affinity mask for the threads of the specified process.

Syntax

C++

```
BOOL SetProcessAffinityMask(
    [in] HANDLE     hProcess,
    [in] DWORD_PTR dwProcessAffinityMask
);
```

Parameters

[in] hProcess

A handle to the process whose affinity mask is to be set. This handle must have the **PROCESS_SET_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

[in] dwProcessAffinityMask

The affinity mask for the threads of the process.

On a system with more than 64 processors, the affinity mask must specify processors in a single [processor group](#).

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 process affinity mask requests a processor that is not configured in the system, the last error code is **ERROR_INVALID_PARAMETER**.

On a system with more than 64 processors, if the calling process contains threads in more than one processor group, the last error code is **ERROR_INVALID_PARAMETER**.

Remarks

A process affinity mask is a bit vector in which each bit represents a logical processor on which the threads of the process are allowed to run. The value of the process affinity mask must be a subset of the system affinity mask values obtained by the [GetProcessAffinityMask](#) function. A process is only allowed to run on the processors configured into a system. Therefore, the process affinity mask cannot specify a 1 bit for a processor when the system affinity mask specifies a 0 bit for that processor.

Process affinity is inherited by any child process or newly instantiated local process.

Do not call [SetProcessAffinityMask](#) in a DLL that may be called by processes other than your own.

On a system with more than 64 processors, the [SetProcessAffinityMask](#) function can be used to set the process affinity mask only for processes with threads in a single [processor group](#). Use the [SetThreadAffinityMask](#) function to set the affinity mask for individual threads in multiple groups. This effectively changes the group assignment of the process.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. Instead of always failing in case the calling process contains threads in more than one processor group, the [SetProcessAffinityMask](#) function fails (returning zero with **ERROR_INVALID_PARAMETER** last error code) if the process had explicitly set the affinity of one or more of its threads outside of the process' [primary group](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateProcess](#)

[GetProcessAffinityMask](#)

[Multiple Processors](#)

[Process and Thread Functions](#)

[Processes](#)

[Processor Groups](#)

[SetThreadAffinityMask](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetProcessDEPPolicy function (winbase.h)

Article10/13/2021

Changes data execution prevention (DEP) and DEP-ATL thunk emulation settings for a 32-bit process.

Syntax

C++

```
BOOL SetProcessDEPPolicy(  
    [in] DWORD dwFlags  
);
```

Parameters

[in] dwFlags

A DWORD that can be one or more of the following values.

Value	Meaning
0	If the DEP system policy is OptIn or OptOut and DEP is enabled for the process, setting <i>dwFlags</i> to 0 disables DEP for the process.
PROCESS_DEP_ENABLE 0x00000001	Enables DEP permanently on the current process. After DEP has been enabled for the process by setting PROCESS_DEP_ENABLE , it cannot be disabled for the life of the process.
PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION 0x00000002	Disables DEP-ATL thunk emulation for the current process, which prevents the system from intercepting NX faults that originate from the Active Template Library (ATL) thunk layer. For more information, see the Remarks section. This flag can be specified only with PROCESS_DEP_ENABLE .

Return value

If the function succeeds, it returns **TRUE**.

If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call [GetLastError](#).

Remarks

The **SetProcessDEPPolicy** function overrides the system DEP policy for the current process unless its DEP policy was specified at process creation. The system DEP policy setting must be OptIn or OptOut. If the system DEP policy is AlwaysOff or AlwaysOn, **SetProcessDEPPolicy** returns an error. After DEP is enabled for a process, subsequent calls to **SetProcessDEPPolicy** are ignored.

DEP policy specified at process creation with the [PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY](#) attribute cannot be changed for the life of the process. In this case, calls to **SetProcessDEPPolicy** fail with [ERROR_ACCESS_DENIED](#).

SetProcessDEPPolicy is supported for 32-bit processes only. If this function is called on a 64-bit process, it fails with [ERROR_NOT_SUPPORTED](#).

Applications written to ATL 7.1 and earlier can attempt to execute code on pages marked as non-executable, which triggers an NX fault and terminates the application. DEP-ATL thunk emulation allows an application that would otherwise trigger an NX fault to run with DEP enabled. For information about ATL versions, see [ATL and MFC Version Numbers](#).

If DEP-ATL thunk emulation is enabled, the system intercepts NX faults, emulates the instructions, and handles the exceptions so the application can continue to run. If DEP-ATL thunk emulation is disabled by setting

[PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION](#) for the process, NX faults are not intercepted, which is useful when testing applications for compatibility with DEP.

The following table summarizes the interactions between system DEP policy, DEP-ATL thunk emulation, and **SetProcessDEPPolicy**. To get the system DEP policy setting, use the [GetSystemDEPPolicy](#) function.

System DEP policy	DEP behavior	DEP_ATL thunk emulation behavior	SetProcessDEPPolicy behavior

AlwaysOff 0	Disabled for the operating system and all processes.	Not applicable.	Returns an error.
AlwaysOn 1	Enabled for the operating system and all processes.	Disabled.	Returns an error.
OptIn 2 Default configuration for Windows client versions.	Enabled for the operating system and disabled for nonsystem processes. Administrators can explicitly enable DEP for selected executable files.	Not applicable.	DEP can be enabled for the current process. If DEP is enabled for the current process, DEP-ATL thunk emulation can be disabled for that process.
OptOut 3 Default configuration for Windows Server versions.	Enabled for the operating system and all processes. Administrators can explicitly disable DEP for selected executable files.	Enabled.	DEP can be disabled for the current process. If DEP is disabled for the current process, DEP-ATL thunk emulation is automatically disabled for that process.

To compile an application that calls this function, define _WIN32_WINNT as 0x0600 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows Vista with SP1, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Data Execution Prevention](#)

[GetProcessDEPPolicy](#)

[GetSystemDEPPolicy](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetSearchPathMode function (winbase.h)

Article10/13/2021

Sets the per-process mode that the [SearchPath](#) function uses when locating files.

Syntax

C++

```
BOOL SetSearchPathMode(  
    [in] DWORD Flags  
) ;
```

Parameters

[in] Flags

The search mode to use.

Value	Meaning
BASE_SEARCH_PATH_ENABLE_SAFE_SEARCHMODE 0x00000001	Enable safe process search mode for the process.
BASE_SEARCH_PATH_DISABLE_SAFE_SEARCHMODE 0x00010000	Disable safe process search mode for the process.
BASE_SEARCH_PATH_PERMANENT 0x00008000	Optional flag to use in combination with BASE_SEARCH_PATH_ENABLE_SAFE_SEARCHMODE to make this mode permanent for this process. This is done by bitwise OR operation: <code>(BASE_SEARCH_PATH_ENABLE_SAFE_SEARCHMODE BASE_SEARCH_PATH_PERMANENT)</code> This flag cannot be combined with the BASE_SEARCH_PATH_DISABLE_SAFE_SEARCHMODE flag.

Return value

If the operation completes successfully, the **SetSearchPathMode** function returns a nonzero value.

If the operation fails, the **SetSearchPathMode** function returns zero. To get extended error information, call the [GetLastError](#) function.

If the **SetSearchPathMode** function fails because a parameter value is not valid, the value returned by the [GetLastError](#) function will be **ERROR_INVALID_PARAMETER**.

If the **SetSearchPathMode** function fails because the combination of current state and parameter value is not valid, the value returned by the [GetLastError](#) function will be **ERROR_ACCESS_DENIED**. For more information, see the Remarks section.

Remarks

If the **SetSearchPathMode** function has not been successfully called for the current process, the search mode used by the [SearchPath](#) function is obtained from the system registry. For more information, see [SearchPath](#).

After the **SetSearchPathMode** function has been successfully called for the current process, the setting in the system registry is ignored in favor of the mode most recently set successfully.

If the **SetSearchPathMode** function has been successfully called for the current process with *Flags* set to `(BASE_SEARCH_PATH_ENABLE_SAFE_SEARCHMODE |
BASE_SEARCH_PATH_PERMANENT)`, safe mode is set permanently for the calling process. Any subsequent calls to the **SetSearchPathMode** function from within that process that attempt to change the search mode will fail with **ERROR_ACCESS_DENIED** from the [GetLastError](#) function.

Note Because setting safe search mode permanently cannot be disabled for the life of the process for which it was set, it should be used with careful consideration. This is particularly true for DLL development, where the user of the DLL will be affected by this process-wide setting.

It is not possible to permanently disable safe search mode.

This function does not modify the system registry.

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

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
Redistributable	KB959426 on Windows XP with SP2 and later and Windows Server 2003 with SP1 and later

See also

[File Management Functions](#)

[SearchPath](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetSystemPowerState function (winbase.h)

Article 10/13/2021

[**SetSystemPowerState** is available for use in the operating systems specified in the Requirements section. It may be altered or unavailable in subsequent versions.
Applications written for Windows Vista and later should use [SetSuspendState](#) instead.]

Suspends the system by shutting power down. Depending on the *ForceFlag* parameter, the function either suspends operation immediately or requests permission from all applications and device drivers before doing so.

Syntax

C++

```
BOOL SetSystemPowerState(  
    [in] BOOL fSuspend,  
    [in] BOOL fForce  
) ;
```

Parameters

[in] *fSuspend*

If this parameter is **TRUE**, the system is suspended. If the parameter is **FALSE**, the system hibernates.

[in] *fForce*

This parameter has no effect.

Return value

If power has been suspended and subsequently restored, the return value is nonzero.

If the system was not suspended, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The calling process must have the **SE_SHUTDOWN_NAME** privilege. To enable the **SE_SHUTDOWN_NAME** privilege, use the [AdjustTokenPrivileges](#) function. For more information, see [Changing Privileges in a Token](#).

If any application or driver denies permission to suspend operation, the function broadcasts a [PBT_APMQUERYSUSPENDFAILED](#) event to each application and driver. If power is suspended, this function returns only after system operation is resumed and related [WM_POWERBROADCAST](#) messages have been broadcast to all applications and drivers.

This function is similar to the [SetSuspendState](#) function.

To compile an application that uses this function, define the _WIN32_WINNT macro as 0x0400 or later. For more information, see [Using the Windows Headers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[PBT_APMQUERYSUSPEND](#)

[PBT_APMQUERYSUSPENDFAILED](#)

[PBT_APMSUSPEND](#)

[Power Management Functions](#)

[SetSuspendState](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

SetTapeParameters function (winbase.h)

Article10/13/2021

The **SetTapeParameters** function either specifies the block size of a tape or configures the tape device.

Syntax

C++

```
DWORD SetTapeParameters(
    [in] HANDLE hDevice,
    [in] DWORD dwOperation,
    [in] LPVOID lpTapeInformation
);
```

Parameters

[in] *hDevice*

Handle to the device for which to set configuration information. This handle is created by using the [CreateFile](#) function.

[in] *dwOperation*

Type of information to set. This parameter must be one of the following values.

Value	Meaning
SET_TAPE_DRIVE_INFORMATION 1L	Sets the device-specific information specified by <i>lpTapeInformation</i> .
SET_TAPE_MEDIA_INFORMATION 0L	Sets the tape-specific information specified by the <i>lpTapeInformation</i> parameter.

[in] *lpTapeInformation*

Pointer to a structure that contains the information to set. If the *dwOperation* parameter is SET_TAPE_MEDIA_INFORMATION, *lpTapeInformation* points to a [TAPE_SET_MEDIA_PARAMETERS](#) structure.

If *dwOperation* is SET_TAPE_DRIVE_INFORMATION, *lpTapeInformation* points to a [TAPE_SET_DRIVE_PARAMETERS](#) structure.

Return value

If the function succeeds, the return value is NO_ERROR.

If the function fails, it can return one of the following error codes.

Error	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.
ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetTapeParameters](#)

[TAPE_SET_DRIVE_PARAMETERS](#)

[TAPE_SET_MEDIA_PARAMETERS](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetTapePosition function (winbase.h)

Article10/13/2021

The **SetTapePosition** function sets the tape position on the specified device.

Syntax

C++

```
DWORD SetTapePosition(
    [in] HANDLE hDevice,
    [in] DWORD dwPositionMethod,
    [in] DWORD dwPartition,
    [in] DWORD dwOffsetLow,
    [in] DWORD dwOffsetHigh,
    [in] BOOL bImmediate
);
```

Parameters

[in] `hDevice`

Handle to the device on which to set the tape position. This handle is created by using the [CreateFile](#) function.

[in] `dwPositionMethod`

Type of positioning to perform. This parameter must be one of the following values.

Value	Meaning
<code>TAPE_ABSOLUTE_BLOCK</code> 1L	Moves the tape to the device-specific block address specified by the <code>dwOffsetLow</code> and <code>dwOffsetHigh</code> parameters. The <code>dwPartition</code> parameter is ignored.
<code>TAPE_LOGICAL_BLOCK</code> 2L	Moves the tape to the block address specified by <code>dwOffsetLow</code> and <code>dwOffsetHigh</code> in the partition specified by <code>dwPartition</code> .
<code>TAPE_REWIND</code> 0L	Moves the tape to the beginning of the current partition. The <code>dwPartition</code> , <code>dwOffsetLow</code> , and <code>dwOffsetHigh</code> parameters are ignored.
<code>TAPE_SPACE_END_OF_DATA</code> 4L	Moves the tape to the end of the data on the partition specified by <code>dwPartition</code> .

TAPE_SPACE_FILEMARKS	Moves the tape forward (or backward) the number of filemarks specified by <i>dwOffsetLow</i> and <i>dwOffsetHigh</i> in the current partition. The <i>dwPartition</i> parameter is ignored.
TAPE_SPACE_RELATIVE_BLOCKS	Moves the tape forward (or backward) the number of blocks specified by <i>dwOffsetLow</i> and <i>dwOffsetHigh</i> in the current partition. The <i>dwPartition</i> parameter is ignored.
TAPE_SPACE_SEQUENTIAL_FMKS	Moves the tape forward (or backward) to the first occurrence of n filemarks in the current partition, where n is the number specified by <i>dwOffsetLow</i> and <i>dwOffsetHigh</i> . The <i>dwPartition</i> parameter is ignored.
TAPE_SPACE_SEQUENTIAL_SMKS	Moves the tape forward (or backward) to the first occurrence of n setmarks in the current partition, where n is the number specified by <i>dwOffsetLow</i> and <i>dwOffsetHigh</i> . The <i>dwPartition</i> parameter is ignored.
TAPE_SPACE_SETMARKS	Moves the tape forward (or backward) the number of setmarks specified by <i>dwOffsetLow</i> and <i>dwOffsetHigh</i> in the current partition. The <i>dwPartition</i> parameter is ignored.

[in] dwPartition

Partition to position within. If *dwPartition* is zero, the current partition is used. Partitions are numbered logically from 1 through n, where 1 is the first partition on the tape and n is the last.

[in] dwOffsetLow

Low-order bits of the block address or count for the position operation specified by the *dwPositionMethod* parameter.

[in] dwOffsetHigh

High-order bits of the block address or count for the position operation specified by the *dwPositionMethod* parameter. If the high-order bits are not required, this parameter should be zero.

[in] bImmediate

Indicates whether to return as soon as the move operation begins. If this parameter is **TRUE**, the function returns immediately; if **FALSE**, the function does not return until the move operation has been completed.

Return value

If the function succeeds, the return value is NO_ERROR.

If the function fails, it can return one of the following error codes.

Error	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.
ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT	The media is write protected.

Remarks

If the offset specified by *dwOffsetLow* and *dwOffsetHigh* specifies the number of blocks, filemarks, or setmarks to move, a positive offset moves the tape forward to the end of the last block, filemark, or setmark. A negative offset moves the tape backward to the beginning of the last block, filemark, or setmark. If the offset is zero, the tape does not move.

To obtain information about the status, capabilities, and capacities of tape drives and media, call the [GetTapeParameters](#) function.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

[GetTapeParameters](#)

[GetTapePosition](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadAffinityMask function (winbase.h)

Article01/27/2022

Sets a processor affinity mask for the specified thread.

Syntax

C++

```
DWORD_PTR SetThreadAffinityMask(  
    [in] HANDLE     hThread,  
    [in] DWORD_PTR dwThreadAffinityMask  
)
```

Parameters

[in] hThread

A handle to the thread whose affinity mask is to be set.

This handle must have the **THREAD_SET_INFORMATION** or **THREAD_SET_LIMITED_INFORMATION** access right and the **THREAD_QUERY_INFORMATION** or **THREAD_QUERY_LIMITED_INFORMATION** access right. For more information, see [Thread Security and Access Rights](#).

Windows Server 2003 and Windows XP: The handle must have the **THREAD_SET_INFORMATION** and **THREAD_QUERY_INFORMATION** access rights.

[in] dwThreadAffinityMask

The affinity mask for the thread.

On a system with more than 64 processors, the affinity mask must specify processors in the thread's current [processor group](#).

Return value

If the function succeeds, the return value is the thread's previous affinity mask.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

If the thread affinity mask requests a processor that is not selected for the process affinity mask, the last error code is [ERROR_INVALID_PARAMETER](#).

Remarks

A thread affinity mask is a bit vector in which each bit represents a logical processor that a thread is allowed to run on. A thread affinity mask must be a subset of the process affinity mask for the containing process of a thread. A thread can only run on the processors its process can run on. Therefore, the thread affinity mask cannot specify a 1 bit for a processor when the process affinity mask specifies a 0 bit for that processor.

Setting an affinity mask for a process or thread can result in threads receiving less processor time, as the system is restricted from running the threads on certain processors. In most cases, it is better to let the system select an available processor.

If the new thread affinity mask does not specify the processor that is currently running the thread, the thread is rescheduled on one of the allowable processors.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all [processor groups](#), by default. The *dwThreadAffinityMask* must specify processors in the thread's current primary group.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProcessAffinityMask](#)

[Multiple Processors](#)

[OpenThread](#)

[Process and Thread Functions](#)

[Processor Groups](#)

[SetProcessAffinityMask](#)

[SetThreadIdealProcessor](#)

[Threads](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadExecutionState function (winbase.h)

Article10/13/2021

Enables an application to inform the system that it is in use, thereby preventing the system from entering sleep or turning off the display while the application is running.

Syntax

C++

```
EXECUTION_STATE SetThreadExecutionState(  
    [in] EXECUTION_STATE esFlags  
);
```

Parameters

[in] esFlags

The thread's execution requirements. This parameter can be one or more of the following values.

Value	Meaning
ES_AWAYMODE_REQUIRED 0x00000040	Enables away mode. This value must be specified with ES_CONTINUOUS . Away mode should be used only by media-recording and media-distribution applications that must perform critical background processing on desktop computers while the computer appears to be sleeping. See Remarks.
ES_CONTINUOUS 0x80000000	Informs the system that the state being set should remain in effect until the next call that uses ES_CONTINUOUS and one of the other state flags is cleared.
ES_DISPLAY_REQUIRED 0x00000002	Forces the display to be on by resetting the display idle timer.
ES_SYSTEM_REQUIRED 0x00000001	Forces the system to be in the working state by resetting the system idle timer.
ES_USER_PRESENT 0x00000004	This value is not supported. If ES_USER_PRESENT is combined with other <i>esFlags</i> values, the call will fail and

none of the specified states will be set.

Return value

If the function succeeds, the return value is the previous thread execution state.

If the function fails, the return value is **NULL**.

Remarks

The system automatically detects activities such as local keyboard or mouse input, server activity, and changing window focus. Activities that are not automatically detected include disk or CPU activity and video display.

Calling **SetThreadExecutionState** without **ES_CONTINUOUS** simply resets the idle timer; to keep the display or system in the working state, the thread must call **SetThreadExecutionState** periodically.

To run properly on a power-managed computer, applications such as fax servers, answering machines, backup agents, and network management applications must use both **ES_SYSTEM_REQUIRED** and **ES_CONTINUOUS** when they process events. Multimedia applications, such as video players and presentation applications, must use **ES_DISPLAY_REQUIRED** when they display video for long periods of time without user input. Applications such as word processors, spreadsheets, browsers, and games do not need to call **SetThreadExecutionState**.

The **ES_AWAYMODE_REQUIRED** value should be used only when absolutely necessary by media applications that require the system to perform background tasks such as recording television content or streaming media to other devices while the system appears to be sleeping. Applications that do not require critical background processing or that run on portable computers should not enable away mode because it prevents the system from conserving power by entering true sleep.

To enable away mode, an application uses both **ES_AWAYMODE_REQUIRED** and **ES_CONTINUOUS**; to disable away mode, an application calls **SetThreadExecutionState** with **ES_CONTINUOUS** and clears **ES_AWAYMODE_REQUIRED**. When away mode is enabled, any operation that would put the computer to sleep puts it in away mode instead. The computer appears to be sleeping while the system continues to perform tasks that do not require user input. Away mode does not affect the sleep idle timer; to prevent the system from entering sleep when the timer expires, an application must also set the **ES_SYSTEM_REQUIRED** value.

The **SetThreadExecutionState** function cannot be used to prevent the user from putting the computer to sleep. Applications should respect that the user expects a certain behavior when they close the lid on their laptop or press the power button.

This function does not stop the screen saver from executing.

Examples

C++

```
// Television recording is beginning. Enable away mode and prevent
// the sleep idle time-out.
//
SetThreadExecutionState(ES_CONTINUOUS | ES_SYSTEM_REQUIRED |
ES_AWAYMODE_REQUIRED);

//
// Wait until recording is complete...
//

//
// Clear EXECUTION_STATE flags to disable away mode and allow the system to
idle to sleep normally.
//
SetThreadExecutionState(ES_CONTINUOUS);
```

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Power Management Functions](#)

[SetSuspendState](#)

[SetSystemPowerState](#)

[WM_POWERBROADCAST](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadpoolCallbackCleanupGroup function (winbase.h)

Article07/27/2022

Associates the specified cleanup group with the specified callback environment.

Syntax

C++

```
void SetThreadpoolCallbackCleanupGroup(
    [in, out]     PTP_CALLBACK_ENVIRON          pcbe,
    [in]          PTP_CLEANUP_GROUP            ptpcg,
    [in, optional] PTP_CLEANUP_GROUP_CANCEL_CALLBACK pfng
);
```

Parameters

[in, out] pcbe

A TP_CALLBACK_ENVIRON structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

[in] ptpcg

A TP_CLEANUP_GROUP structure that defines the cleanup group. The [CreateThreadpoolCleanupGroup](#) function returns this structure.

[in, optional] pfng

The cleanup callback to be called if the cleanup group is canceled before the associated object is released. The function is called when you call [CloseThreadpoolCleanupGroupMembers](#).

Return value

None

Remarks

This function is implemented as an inline function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or higher.

Examples

For an example, see [Using the Thread Pool Functions](#).

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[DestroyThreadpoolEnvironment](#)

[InitializeThreadpoolEnvironment](#)

[SetThreadpoolCallbackLibrary](#)

[SetThreadpoolCallbackPool](#)

[SetThreadpoolCallbackPriority](#)

[SetThreadpoolCallbackRunsLong](#)

[Thread Pools](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadpoolCallbackLibrary function (winbase.h)

Article 07/27/2022

Ensures that the specified DLL remains loaded as long as there are outstanding callbacks.

Syntax

C++

```
void SetThreadpoolCallbackLibrary(
    [in, out] PTP_CALLBACK_ENVIRON pcbe,
    [in]     PVOID             mod
);
```

Parameters

[in, out] pcbe

A TP_CALLBACK_ENVIRON structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

[in] mod

A handle to the DLL.

Return value

None

Remarks

You should call this function if a callback might acquire the loader lock. This prevents a deadlock from occurring when one thread in DllMain is waiting for the callback to end, and another thread that is executing the callback attempts to acquire the loader lock.

If the DLL containing the callback might be unloaded, the cleanup code in DllMain must cancel outstanding callbacks before releasing the object.

Managing callbacks created with a TP_CALLBACK_ENVIRON that specifies a callback library is somewhat processing-intensive. You should consider other options for ensuring that the library is not unloaded while callbacks are executing, or to guarantee that callbacks which may be executing do not acquire the loader lock.

The thread pool assumes ownership of the library reference supplied to this function. The caller should not call [FreeLibrary](#) on a module handle after passing it to this function.

This function is implemented as an inline function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or higher.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[DestroyThreadpoolEnvironment](#)

[FreeLibraryWhenCallbackReturns](#)

[InitializeThreadpoolEnvironment](#)

[SetThreadpoolCallbackCleanupGroup](#)

[SetThreadpoolCallbackPool](#)

[SetThreadpoolCallbackPriority](#)

[SetThreadpoolCallbackRunsLong](#)

[Thread Pools](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadpoolCallbackPersistent function (winbase.h)

Article07/27/2022

Specifies that the callback should run on a persistent thread.

Syntax

C++

```
void SetThreadpoolCallbackPersistent(
    [in, out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

[in, out] pcbe

A TP_CALLBACK_ENVIRON structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

Return value

None

Remarks

This function is implemented as an inline function.

To compile an application that uses this function, set _WIN32_WINNT to _WIN32_WINNT_WIN7. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client

Windows 7 [desktop apps only]

Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadpoolCallbackPool function (winbase.h)

Article 10/13/2021

Sets the thread pool to be used when generating callbacks.

Syntax

C++

```
void SetThreadpoolCallbackPool(
    [in, out] PTP_CALLBACK_ENVIRON pcbe,
    [in]      PTP_POOL          ptpp
);
```

Parameters

[in, out] pcbe

A TP_CALLBACK_ENVIRON structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

[in] ptpp

A TP_POOL structure that defines the thread pool. The [CreateThreadpool](#) function returns this structure.

Return value

None

Remarks

If you do not specify a thread pool, the global thread pool is used.

This function is implemented as an inline function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or higher.

Examples

For an example, see [Using the Thread Pool Functions](#).

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[DestroyThreadpoolEnvironment](#)

[InitializeThreadpoolEnvironment](#)

[SetThreadpoolCallbackCleanupGroup](#)

[SetThreadpoolCallbackLibrary](#)

[SetThreadpoolCallbackPriority](#)

[SetThreadpoolCallbackRunsLong](#)

[Thread Pools](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadpoolCallbackPriority function (winbase.h)

Article07/27/2022

Specifies the priority of a callback function relative to other work items in the same thread pool.

Syntax

C++

```
void SetThreadpoolCallbackPriority(
    [in, out] PTP_CALLBACK_ENVIRON pcbe,
    [in]      TP_CALLBACK_PRIORITY Priority
);
```

Parameters

[in, out] pcbe

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

[in] Priority

The priority for the callback relative to other callbacks in the same thread pool. This parameter can be one of the following **TP_CALLBACK_PRIORITY** enumeration values:

Value	Meaning
TP_CALLBACK_PRIORITY_HIGH	The callback should run at high priority.
TP_CALLBACK_PRIORITY_LOW	The callback should run at low priority.
TP_CALLBACK_PRIORITY_NORMAL	The callback should run at normal priority.

Return value

None

Remarks

Higher priority callbacks are guaranteed to be run first by the first available worker thread, but they are not guaranteed to finish before lower priority callbacks.

This function is implemented as an inline function.

To compile an application that uses this function, set `_WIN32_WINNT >= _WIN32_WINNT_WIN7`. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows 7 [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetThreadpoolCallbackRunsLong function (winbase.h)

Article07/27/2022

Indicates that callbacks associated with this callback environment may not return quickly.

Syntax

C++

```
void SetThreadpoolCallbackRunsLong(
    [in, out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

[in, out] pcbe

A TP_CALLBACK_ENVIRON structure that defines the callback environment. The [InitializeThreadpoolEnvironment](#) function returns this structure.

Return value

None

Remarks

The thread pool may use this information to better determine when a new thread should be created.

This function is implemented as an inline function.

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or higher.

Requirements

Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[CallbackMayRunLong](#)

[DestroyThreadpoolEnvironment](#)

[InitializeThreadpoolEnvironment](#)

[SetThreadpoolCallbackCleanupGroup](#)

[SetThreadpoolCallbackLibrary](#)

[SetThreadpoolCallbackPool](#)

[SetThreadpoolCallbackPriority](#)

[Thread Pools](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetUmsThreadInformation function (winbase.h)

Article03/03/2023

Sets application-specific context information for the specified user-mode scheduling (UMS) worker thread.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL SetUmsThreadInformation(
    [in] PUMS_CONTEXT          UmsThread,
    [in] UMS_THREAD_INFO_CLASS UmsThreadInfoClass,
    [in] PVOID                 UmsThreadInformation,
    [in] ULONG                 UmsThreadInformationLength
);
```

Parameters

[in] `UmsThread`

A pointer to a UMS thread context.

[in] `UmsThreadInfoClass`

A `UMS_THREAD_INFO_CLASS` value that specifies the kind of information to set. This parameter must be `UmsThreadUserContext`.

[in] `UmsThreadInformation`

A pointer to a buffer that contains the information to set.

[in] `UmsThreadInformationLength`

The size of the `UmsThreadInformation` buffer, in bytes.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include the following.

Return code	Description
ERROR_INFO_LENGTH_MISMATCH	The buffer size does not match the required size for the specified information class.
ERROR_INVALID_INFO_CLASS	The <i>UmsThreadInfoClass</i> parameter specifies an information class that is not supported.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

The **SetUmsThreadInformation** function can be used to set an application-defined context for the specified UMS worker thread. The context information can consist of anything the application might find useful to track, such as per-scheduler or per-worker thread state. The underlying structures for UMS worker threads are managed by the system and should not be modified directly.

The [QueryUmsThreadInformation](#) function can be used to retrieve other exposed information about the specified thread, such as its thread execution block ([TEB](#)) and whether the thread is suspended or terminated. Information that is not exposed through [QueryUmsThreadInformation](#) should be considered reserved.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

API set

api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[QueryUmsThreadInformation](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetupComm function (winbase.h)

Article10/13/2021

Initializes the communications parameters for a specified communications device.

Syntax

C++

```
BOOL SetupComm(
    [in] HANDLE hFile,
    [in] DWORD dwInQueue,
    [in] DWORD dwOutQueue
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in] dwInQueue

The recommended size of the device's internal input buffer, in bytes.

[in] dwOutQueue

The recommended size of the device's internal output buffer, in bytes.

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

After a process uses the [CreateFile](#) function to open a handle to a communications device, but before doing any I/O with the device, it can call **SetupComm** to set the

communications parameters for the device. If it does not set them, the device uses the default parameters when the first call to another communications function occurs.

The *dwInQueue* and *dwOutQueue* parameters specify the recommended sizes for the internal buffers used by the driver for the specified device. For example, YMODEM protocol packets are slightly larger than 1024 bytes. Therefore, a recommended buffer size might be 1200 bytes for YMODEM communications. For Ethernet-based communications, a recommended buffer size might be 1600 bytes, which is slightly larger than a single Ethernet frame.

The device driver receives the recommended buffer sizes, but is free to use any input and output (I/O) buffering scheme, as long as it provides reasonable performance and data is not lost due to overrun (except under extreme circumstances). For example, the function can succeed even though the driver does not allocate a buffer, as long as some other portion of the system provides equivalent functionality.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[SetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

SetVolumeLabelA function (winbase.h)

Article02/09/2023

Sets the label of a file system volume.

Syntax

C++

```
BOOL SetVolumeLabelA(
    [in, optional] LPCSTR lpRootPathName,
    [in, optional] LPCSTR lpVolumeName
);
```

Parameters

[in, optional] lpRootPathName

A pointer to a string that contains the volume's drive letter (for example, X:) or the path of a mounted folder that is associated with the volume (for example, Y:\MountX). The string must end with a trailing backslash ('\\'). If this parameter is **NULL**, the root of the current directory is used.

[in, optional] lpVolumeName

A pointer to a string that contains the new label for the volume. If this parameter is **NULL**, the function deletes any existing label from the specified volume and does not assign a new label.

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

The maximum volume label length is 32 characters.

FAT filesystems: The maximum volume label length is 11 characters.

A label is a user-friendly name that a user assigns to a volume to make it easier to recognize. A volume can have a label, a drive letter, both, or neither. For more information, see [Naming a Volume](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

 **Note**

The winbase.h header defines SetVolumeLabel as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[GetVolumeInformation](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetVolumeLabelW function (winbase.h)

Article02/09/2023

Sets the label of a file system volume.

Syntax

C++

```
BOOL SetVolumeLabelW(
    [in, optional] LPCWSTR lpRootPathName,
    [in, optional] LPCWSTR lpVolumeName
);
```

Parameters

[in, optional] lpRootPathName

A pointer to a string that contains the volume's drive letter (for example, X:) or the path of a mounted folder that is associated with the volume (for example, Y:\MountX). The string must end with a trailing backslash ('\\'). If this parameter is **NULL**, the root of the current directory is used.

[in, optional] lpVolumeName

A pointer to a string that contains the new label for the volume. If this parameter is **NULL**, the function deletes any existing label from the specified volume and does not assign a new label.

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

The maximum volume label length is 32 characters.

FAT filesystems: The maximum volume label length is 11 characters.

A label is a user-friendly name that a user assigns to a volume to make it easier to recognize. A volume can have a label, a drive letter, both, or neither. For more information, see [Naming a Volume](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
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

SMB does not support volume management functions.

 **Note**

The winbase.h header defines SetVolumeLabel as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[GetVolumeInformation](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetVolumeMountPointA function (winbase.h)

Article02/09/2023

Associates a volume with a drive letter or a directory on another volume.

Syntax

C++

```
BOOL SetVolumeMountPointA(
    [in] LPCSTR lpszVolumeMountPoint,
    [in] LPCSTR lpszVolumeName
);
```

Parameters

[in] `lpszVolumeMountPoint`

The user-mode path to be associated with the volume. This may be a drive letter (for example, "X:\") or a directory on another volume (for example, "Y:\MountX"). The string must end with a trailing backslash ("").

[in] `lpszVolumeName`

A volume **GUID** path for the volume. This string must be of the form "\\\?\Volume{*GUID*}" where *GUID* is a **GUID** that identifies the volume. The "\\\?" turns off path parsing and is ignored as part of the path, as discussed in [Naming a Volume](#).

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 *lpszVolumeMountPoint* parameter contains a path to a mounted folder, [GetLastError](#) returns **ERROR_DIR_NOT_EMPTY**, even if the directory is empty.

Remarks

When this function is used to associate a volume with a directory on another volume, the associated directory is called a *mounted folder*.

It is an error to associate a volume with a directory that has any files or subdirectories in it. This error occurs for system and hidden directories as well as other directories, and it occurs for system and hidden files.

When mounted folders are created on a volume on a clustered disk, they may be deleted unexpectedly under certain circumstances. For information on how to create and configure mounted folders to ensure that this does not happen, see [Cluster Disk and Drive Connection Problems](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. For CsvFS a new mount point will not be replicated to the other nodes on the cluster.

Examples

For an example, see [Creating a Mounted Folder](#).

Note

The winbase.h header defines SetVolumeMountPoint as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeleteVolumeMountPoint](#)

[GetVolumeNameForVolumeMountPoint](#)

[GetVolumePathName](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetVolumeMountPointW function (winbase.h)

Article02/09/2023

Associates a volume with a drive letter or a directory on another volume.

Syntax

C++

```
BOOL SetVolumeMountPointW(
    [in] LPCWSTR lpszVolumeMountPoint,
    [in] LPCWSTR lpszVolumeName
);
```

Parameters

[in] lpszVolumeMountPoint

The user-mode path to be associated with the volume. This may be a drive letter (for example, "X:\") or a directory on another volume (for example, "Y:\MountX"). The string must end with a trailing backslash ("").

[in] lpszVolumeName

A volume **GUID** path for the volume. This string must be of the form "\\\?
\Volume{GUID}\\" where *GUID* is a **GUID** that identifies the volume. The "\\\?\" turns off path parsing and is ignored as part of the path, as discussed in [Naming a Volume](#).

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 *lpszVolumeMountPoint* parameter contains a path to a mounted folder, [GetLastError](#) returns **ERROR_DIR_NOT_EMPTY**, even if the directory is empty.

Remarks

When this function is used to associate a volume with a directory on another volume, the associated directory is called a *mounted folder*.

It is an error to associate a volume with a directory that has any files or subdirectories in it. This error occurs for system and hidden directories as well as other directories, and it occurs for system and hidden files.

When mounted folders are created on a volume on a clustered disk, they may be deleted unexpectedly under certain circumstances. For information on how to create and configure mounted folders to ensure that this does not happen, see [Cluster Disk and Drive Connection Problems](#).

In Windows 8 and Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	No
SMB 3.0 Transparent Failover (TFO)	No
SMB 3.0 with Scale-out File Shares (SO)	No
Cluster Shared Volume File System (CsvFS)	No
Resilient File System (ReFS)	No

SMB does not support volume management functions. For CsvFS a new mount point will not be replicated to the other nodes on the cluster.

Examples

For an example, see [Creating a Mounted Folder](#).

Note

The winbase.h header defines SetVolumeMountPoint as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[DeleteVolumeMountPoint](#)

[GetVolumeNameForVolumeMountPoint](#)

[GetVolumePathName](#)

[Mounted Folders](#)

[Volume Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SetXStateFeaturesMask function (winbase.h)

Article03/15/2023

Sets the mask of XState features set within a [CONTEXT](#) structure.

Syntax

C++

```
BOOL SetXStateFeaturesMask(
    [in, out] PCONTEXT Context,
    [in]      DWORD64 FeatureMask
);
```

Parameters

[in, out] Context

A pointer to a [CONTEXT](#) structure that has been initialized with [InitializeContext](#).

[in] FeatureMask

A mask of XState features to set in the specified [CONTEXT](#) structure.

Return value

This function returns **TRUE** if successful, otherwise **FALSE**.

Remarks

The **SetXStateFeaturesMask** function sets the mask of valid features in the specified context. Before calling [GetThreadContext](#), [Wow64GetThreadContext](#) , [SetThreadContext](#), or [Wow64SetThreadContext](#) the application must call **SetXStateFeaturesMask** to specify which set of features to retrieve or set. The system silently ignores any feature specified in the *FeatureMask* which is not enabled on the processor.

Windows 7 with SP1 and Windows Server 2008 R2 with SP1: The AVX API is first implemented on Windows 7 with SP1 and Windows Server 2008 R2 with SP1 . Since there is no SDK for SP1, that means there are no available headers and library files to work with. In this situation, a caller must declare the needed functions from this documentation and get pointers to them using [GetModuleHandle](#) on "Kernel32.dll", followed by calls to [GetProcAddress](#). See [Working with XState Context](#) for details.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CONTEXT](#)

[GetThreadContext](#)

[Intel AVX](#)

[SetThreadContext](#)

[Working with XState Context](#)

[Wow64GetThreadContext](#) ↗

[Wow64SetThreadContext](#) ↗

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

STARTUPINFOEXA structure (winbase.h)

Article07/27/2022

Specifies the window station, desktop, standard handles, and attributes for a new process. It is used with the [CreateProcess](#) and [CreateProcessAsUser](#) functions.

Syntax

C++

```
typedef struct _STARTUPINFOEXA {
    STARTUPINFOA                         StartupInfo;
    LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList;
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
```

Members

`StartupInfo`

A [STARTUPINFO](#) structure.

`lpAttributeList`

An attribute list. This list is created by the [InitializeProcThreadAttributeList](#) function.

Remarks

Be sure to set the `cb` member of the [STARTUPINFO](#) structure to `sizeof(STARTUPINFOEX)`.

ⓘ Note

The winbase.h header defines `STARTUPINFOEX` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[CreateProcess](#)

[CreateProcessAsUser](#)

[InitializeProcThreadAttributeList](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

STARTUPINFOEXW structure (winbase.h)

Article07/27/2022

Specifies the window station, desktop, standard handles, and attributes for a new process. It is used with the [CreateProcess](#) and [CreateProcessAsUser](#) functions.

Syntax

C++

```
typedef struct _STARTUPINFOEXW {
    STARTUPINFOW           StartupInfo;
    LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList;
} STARTUPINFOEXW, *LPSTARTUPINFOEXW;
```

Members

`StartupInfo`

A [STARTUPINFO](#) structure.

`lpAttributeList`

An attribute list. This list is created by the [InitializeProcThreadAttributeList](#) function.

Remarks

Be sure to set the `cb` member of the [STARTUPINFO](#) structure to `sizeof(STARTUPINFOEX)`.

ⓘ Note

The winbase.h header defines `STARTUPINFOEX` as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the `UNICODE` preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[CreateProcess](#)

[CreateProcessAsUser](#)

[InitializeProcThreadAttributeList](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SwitchToFiber function (winbase.h)

Article10/13/2021

Schedules a fiber. The function must be called on a fiber.

Syntax

C++

```
void SwitchToFiber(  
    [in] LPVOID lpFiber  
);
```

Parameters

[in] lpFiber

The address of the fiber to be scheduled.

Return value

None

Remarks

You create fibers with the [CreateFiber](#) function. Before you can schedule fibers associated with a thread, you must call [ConvertThreadToFiber](#) to set up an area in which to save the fiber state information. The thread is now the currently executing fiber.

The **SwitchToFiber** function saves the state information of the current fiber and restores the state of the specified fiber. You can call **SwitchToFiber** with the address of a fiber created by a different thread. To do this, you must have the address returned to the other thread when it called [CreateFiber](#) and you must use proper synchronization.

Avoid making the following call:

syntax

```
SwitchToFiber( GetCurrentFiber() );
```

This call can cause unpredictable problems.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see [Using the Windows Headers](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ConvertThreadToFiber](#)

[CreateFiber](#)

[Fibers](#)

[Process and Thread Functions](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

SYSTEM_POWER_STATUS structure (winbase.h)

Article01/31/2022

Contains information about the power status of the system.

Syntax

C++

```
typedef struct _SYSTEM_POWER_STATUS {
    BYTE ACLineStatus;
    BYTE BatteryFlag;
    BYTE BatteryLifePercent;
    BYTE SystemStatusFlag;
    DWORD BatteryLifeTime;
    DWORD BatteryFullLifeTime;
} SYSTEM_POWER_STATUS, *LPSYSTEM_POWER_STATUS;
```

Members

ACLineStatus

The AC power status. This member can be one of the following values.

Value	Meaning
0	Offline
1	Online
255	Unknown status

BatteryFlag

The battery charge status. This member can contain one or more of the following flags.

Value	Meaning
1	High—the battery capacity is at more than 66 percent
2	Low—the battery capacity is at less than 33 percent
4	Critical—the battery capacity is at less than five percent

8	Charging
128	No system battery
255	Unknown status—unable to read the battery flag information

The value is zero if the battery is not being charged and the battery capacity is between low and high.

BatteryLifePercent

The percentage of full battery charge remaining. This member can be a value in the range 0 to 100, or 255 if status is unknown.

SystemStatusFlag

The status of battery saver. To participate in energy conservation, avoid resource intensive tasks when battery saver is on. To be notified when this value changes, call the [RegisterPowerSettingNotification](#) function with the power setting GUID, **GUID_POWER_SAVING_STATUS**.

Value	Meaning
0	Battery saver is off.
1	Battery saver on. Save energy where possible.

Note This flag and the **GUID_POWER_SAVING_STATUS** GUID were introduced in Windows 10. This flag was previously reserved, named **Reserved1**, and had a value of 0.

For general information about battery saver, see [battery saver \(in the hardware component guidelines\)](#).

BatteryLifeTime

The number of seconds of battery life remaining, or –1 if remaining seconds are unknown or if the device is connected to AC power.

BatteryFullLifeTime

The number of seconds of battery life when at full charge, or –1 if full battery lifetime is unknown or if the device is connected to AC power.

Remarks

The system is only capable of estimating **BatteryFullLifeTime** based on calculations on **BatteryLifeTime** and **BatteryLifePercent**. Without smart battery subsystems, this value may not be accurate enough to be useful.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

[GetSystemPowerStatus](#)

[PBT_APMPowerStatusChange](#)

[battery saver \(in the hardware component guidelines\)](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

TransmitCommChar function (winbase.h)

Article 10/13/2021

Transmits a specified character ahead of any pending data in the output buffer of the specified communications device.

Syntax

C++

```
BOOL TransmitCommChar(
    [in] HANDLE hFile,
    [in] char    cChar
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[in] cChar

The character to be transmitted.

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

The **TransmitCommChar** function is useful for sending an interrupt character (such as a CTRL+C) to a host system.

If the device is not transmitting, **TransmitCommChar** cannot be called repeatedly. Once **TransmitCommChar** places a character in the output buffer, the character must be

transmitted before the function can be called again. If the previous character has not yet been sent, **TransmitCommChar** returns an error.

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

[WaitCommEvent](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

UMS_SCHEDULER_STARTUP_INFO structure (winbase.h)

Article03/18/2022

Specifies attributes for a user-mode scheduling (UMS) scheduler thread. The [EnterUmsSchedulingMode](#) function uses this structure.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
typedef struct _UMS_SCHEDULER_STARTUP_INFO {
    ULONG UmsVersion;
    PUMS_COMPLETION_LIST CompletionList;
    PUMS_SCHEDULER_ENTRY_POINT SchedulerProc;
    PVOID SchedulerParam;
} UMS_SCHEDULER_STARTUP_INFO, *PUMS_SCHEDULER_STARTUP_INFO;
```

Members

UmsVersion

The UMS version for which the application was built. This parameter must be `UMS_VERSION`.

CompletionList

A pointer to a UMS completion list to associate with the calling thread.

SchedulerProc

A pointer to an application-defined [UmsSchedulerProc](#) entry point function. The system calls this function when the calling thread has been converted to UMS and is ready to run UMS worker threads. Subsequently, it calls this function when a UMS worker thread running on the calling thread yields or blocks.

SchedulerParam

An application-defined parameter to pass to the specified [UmsSchedulerProc](#) function.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

UMS_SYSTEM_THREAD_INFORMATION structure (winbase.h)

Article03/18/2022

Specifies a UMS scheduler thread, UMS worker thread, or non-UMS thread. The [GetUmsSystemThreadInformation](#) function uses this structure.

⚠ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
typedef struct _UMS_SYSTEM_THREAD_INFORMATION {
    ULONG UmsVersion;
    union {
        struct {
            ULONG IsUmsSchedulerThread : 1;
            ULONG IsUmsWorkerThread : 1;
        } DUMMYSTRUCTNAME;
        ULONG ThreadUmsFlags;
    } DUMMYUNIONNAME;
} UMS_SYSTEM_THREAD_INFORMATION, *PUMS_SYSTEM_THREAD_INFORMATION;
```

Members

`UmsVersion`

The UMS version. This member must be `UMS_VERSION`.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IsUmsSchedulerThread`

A bitfield that specifies a UMS scheduler thread. If `IsUmsSchedulerThread` is set, `IsUmsWorkerThread` must be clear.

DUMMYUNIONNAME.DUMMYSRUCTNAME.IsUmsWorkerThread

A bitfield that specifies a UMS worker thread. If **IsUmsWorkerThread** is set, **IsUmsSchedulerThread** must be clear.

DUMMYUNIONNAME.ThreadUmsFlags

Remarks

If both **IsUmsSchedulerThread** and **IsUmsWorkerThread** are clear, the structure specifies a non-UMS thread.

Requirements

Minimum supported client	Windows 7 with SP1 [desktop apps only], Windows 7 (64-bit only) and Windows Server 2008 R2 (64-bit only) with KB977165 installed
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

UmsThreadYield function (winbase.h)

Article03/18/2022

Yields control to the user-mode scheduling (UMS) scheduler thread on which the calling UMS worker thread is running.

⚠️ Warning

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

Syntax

C++

```
BOOL UmsThreadYield(  
    [in] PVOID SchedulerParam  
);
```

Parameters

[in] `SchedulerParam`

A parameter to pass to the scheduler thread's [UmsSchedulerProc](#) function.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A UMS worker thread calls the **UmsThreadYield** function to cooperatively yield control to the UMS scheduler thread on which the worker thread is running. If a UMS worker thread never calls **UmsThreadYield**, the worker thread runs until it either blocks or is terminated.

When control switches to the UMS scheduler thread, the system calls the associated scheduler entry point function with the reason **UmsSchedulerThreadYield** and the *ScheduleParam* parameter specified by the worker thread in the **UmsThreadYield** call.

The application's scheduler is responsible for rescheduling the worker thread.

Requirements

Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

[UmsSchedulerProc](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

UnregisterApplicationRecoveryCallback function (winbase.h)

Article 06/29/2021

Removes the active instance of an application from the recovery list.

Syntax

C++

```
HRESULT UnregisterApplicationRecoveryCallback();
```

Return value

This function returns `S_OK` on success or one of the following error codes.

Return code	Description
<code>E_FAIL</code>	Internal error.

Remarks

You do not need to call this function before exiting. You need to remove the registration only if you choose to not recover data.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[RegisterApplicationRecoveryCallback](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

UnregisterApplicationRestart function (winbase.h)

Article 06/29/2021

Removes the active instance of an application from the restart list.

Syntax

C++

```
HRESULT UnregisterApplicationRestart();
```

Return value

This function returns `S_OK` on success or one of the following error codes.

Return code	Description
<code>E_FAIL</code>	Internal error.

Remarks

You do not need to call this function before exiting. You need to remove the registration only if you choose to not restart the application. For example, you could remove the registration if your application entered a corrupted state where a future restart would also fail. You must call this function before the application fails abnormally.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

DLL

Kernel32.dll

See also

[RegisterApplicationRestart](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

UnregisterWait function (winbase.h)

Article07/27/2022

Cancels a registered wait operation issued by the [RegisterWaitForSingleObject](#) function.

To use a completion event, call the [UnregisterWaitEx](#) function.

Syntax

C++

```
BOOL UnregisterWait(  
    [in] HANDLE WaitHandle  
) ;
```

Parameters

[in] WaitHandle

The wait handle. This handle is returned by the [RegisterWaitForSingleObject](#) 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

If any callback functions associated with the timer have not completed when **UnregisterWait** is called, **UnregisterWait** unregisters the wait on the callback functions and fails with the **ERROR_IO_PENDING** error code. The error code does not indicate that the function has failed, and the function does not need to be called again. If your code requires an error code to set only when the unregister operation has failed, call [UnregisterWaitEx](#) instead.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see [Using the Windows Headers](#).

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[RegisterWaitForSingleObject](#)

[Synchronization Functions](#)

[Thread Pooling](#)

[UnregisterWaitEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

UpdateResourceA function (winbase.h)

Article 02/09/2023

Adds, deletes, or replaces a resource in a portable executable (PE) file. There are some restrictions on resource updates in files that contain Resource Configuration (RC Config) data: [language-neutral](#) (LN) files and language-specific resource (.mui) files.

Syntax

C++

```
BOOL UpdateResourceA(
    [in]           HANDLE hUpdate,
    [in]           LPCSTR lpType,
    [in]           LPCSTR lpName,
    [in]           WORD   wLanguage,
    [in, optional] LPVOID lpData,
    [in]           DWORD   cb
);
```

Parameters

[in] hUpdate

Type: **HANDLE**

A module handle returned by the [BeginUpdateResource](#) function, referencing the file to be updated.

[in] lpType

Type: **LPCTSTR**

The resource type to be updated. Alternatively, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is an integer value representing a predefined resource type. If the first character of the string is a pound sign (#), then the remaining characters represent a decimal number that specifies the integer identifier of the resource type. For example, the string "#258" represents the identifier 258.

For a list of predefined resource types, see [Resource Types](#).

[in] lpName

Type: **LPCTSTR**

The name of the resource to be updated. Alternatively, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is a resource ID. When creating a new resource do not use a string that begins with a '#' character for this parameter.

[in] **wLanguage**

Type: **WORD**

The [language identifier](#) of the resource to be updated. For a list of the primary language identifiers and sublanguage identifiers that make up a language identifier, see the [MAKELANGID](#) macro.

[in, optional] **lpData**

Type: **LPVOID**

The resource data to be inserted into the file indicated by *hUpdate*. If the resource is one of the predefined types, the data must be valid and properly aligned. Note that this is the raw binary data to be stored in the file indicated by *hUpdate*, not the data provided by [LoadIcon](#), [LoadString](#), or other resource-specific load functions. All data containing strings or text must be in Unicode format. *lpData* must not point to ANSI data.

If *lpData* is **NULL** and *cbData* is 0, the specified resource is deleted from the file indicated by *hUpdate*.

[in] **cb**

Type: **DWORD**

The size, in bytes, of the resource data at *lpData*.

Return value

Type: **BOOL**

Returns **TRUE** if successful or **FALSE** otherwise. To get extended error information, call [GetLastError](#).

Remarks

It is recommended that the resource file is not loaded before this function is called. However, if that file is already loaded, it will not cause an error to be returned.

An application can use **UpdateResource** repeatedly to make changes to the resource data. Each call to **UpdateResource** contributes to an internal list of additions, deletions, and replacements but does not actually write the data to the file indicated by *hUpdate*. The application must use the [EndUpdateResource](#) function to write the accumulated changes to the file.

This function can update resources within modules that contain both code and resources.

Prior to Windows 7: If *lpData* is **NULL** and *cbData* is nonzero, the specified resource is NOT deleted and an exception is thrown.

Starting with Windows Vista: As noted above, there are restrictions on resource updates in files that contain RC Config data: LN files and .mui files. The restrictions are as follows:

Action	LN file	.mui file
1. Add a new type that doesn't exist in the LN or .mui files.	Add type in the LN file and treat as language-neutral (non-localizable) and add new type or item in the RC Config data	The only additions allowed are the following types: file Version, RC Config data, Side-by-side Assembly XML Manifest.
2. Add a new resource item to an existing type.	Uses the RC Config data to check whether the type exists in the .mui files associated with this LN file. If the type doesn't exist in the .mui files, add the item and treat new item as un-localizable. If the type exists in the .mui files, then adding is not allowed.	Only items of the following types may be added: File Version, RC Config data, Side-by-side Assembly XML Manifest.
3. Update a resource item.	Uses the RC Config data to check whether the type exists in the .mui files associated with the LN file. If the type doesn't exist in the .mui files, then this resource item update is allowed in the LN file. Otherwise, if the type exists in the .mui files associated with this LN file, then this update is not allowed.	The only updates allowed are items of the following types: file Version, RC Config data, Side-by-side Assembly XML Manifest.
4. Add a type/item for a new language.	Not allowed.	Not allowed.
5. Remove an existing type/item.	Works similarly to case 3. Uses the RC Config data to check whether the type exists in the .mui files associated with the LN file. If not, then the removal of the type/item from the LN file is allowed. Otherwise, if the type/item exists in	The only types allowed to be removed are: file Version, RC Config data, Side-by-side Assembly XML Manifest; also,

	the .mui files associated with this LN file, then the removal is not allowed.	only items of these types may be removed.
6. Add/delete/update a type not included in the RC Config data (such as Version, Side-by-side Assembly XML Manifest, or RC Config data itself).	Allowed.	Allowed.
7. Other update of non-localizable data, such as TYPELIB, reginst, and so on.	Update type or item in the LN file, treat as non-localizable, and add new type or item in the RC Config data.	Not applicable.
8. Add RC Config data.	Can be done but the integrity of the RC Config data is not checked.	Can be done but the integrity of the RC Config data is not checked.

Examples

For an example, see [Updating Resources](#).

Note

The winbase.h header defines UpdateResource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BeginUpdateResource](#)

[Conceptual](#)

[EndUpdateResource](#)

[LoadIcon](#)

[LoadString](#)

[LockResource](#)

[MAKEINTRESOURCE](#)

[MAKELANGID](#)

[Other Resources](#)

[Reference](#)

[Resources](#)

[SizeofResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

UpdateResourceW function (winbase.h)

Article 02/09/2023

Adds, deletes, or replaces a resource in a portable executable (PE) file. There are some restrictions on resource updates in files that contain Resource Configuration (RC Config) data: [language-neutral](#) (LN) files and language-specific resource (.mui) files.

Syntax

C++

```
BOOL UpdateResourceW(
    [in]           HANDLE hUpdate,
    [in]           LPCWSTR lpType,
    [in]           LPCWSTR lpName,
    [in]           WORD   wLanguage,
    [in, optional] LPVOID lpData,
    [in]           DWORD   cb
);
```

Parameters

[in] hUpdate

Type: **HANDLE**

A module handle returned by the [BeginUpdateResource](#) function, referencing the file to be updated.

[in] lpType

Type: **LPCTSTR**

The resource type to be updated. Alternatively, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is an integer value representing a predefined resource type. If the first character of the string is a pound sign (#), then the remaining characters represent a decimal number that specifies the integer identifier of the resource type. For example, the string "#258" represents the identifier 258.

For a list of predefined resource types, see [Resource Types](#).

[in] lpName

Type: **LPCTSTR**

The name of the resource to be updated. Alternatively, rather than a pointer, this parameter can be [MAKEINTRESOURCE](#)(ID), where ID is a resource ID. When creating a new resource do not use a string that begins with a '#' character for this parameter.

[in] **wLanguage**

Type: **WORD**

The [language identifier](#) of the resource to be updated. For a list of the primary language identifiers and sublanguage identifiers that make up a language identifier, see the [MAKELANGID](#) macro.

[in, optional] **lpData**

Type: **LPVOID**

The resource data to be inserted into the file indicated by *hUpdate*. If the resource is one of the predefined types, the data must be valid and properly aligned. Note that this is the raw binary data to be stored in the file indicated by *hUpdate*, not the data provided by [LoadIcon](#), [LoadString](#), or other resource-specific load functions. All data containing strings or text must be in Unicode format. *lpData* must not point to ANSI data.

If *lpData* is **NULL** and *cbData* is 0, the specified resource is deleted from the file indicated by *hUpdate*.

[in] **cb**

Type: **DWORD**

The size, in bytes, of the resource data at *lpData*.

Return value

Type: **BOOL**

Returns **TRUE** if successful or **FALSE** otherwise. To get extended error information, call [GetLastError](#).

Remarks

It is recommended that the resource file is not loaded before this function is called. However, if that file is already loaded, it will not cause an error to be returned.

An application can use **UpdateResource** repeatedly to make changes to the resource data. Each call to **UpdateResource** contributes to an internal list of additions, deletions, and replacements but does not actually write the data to the file indicated by *hUpdate*. The application must use the [EndUpdateResource](#) function to write the accumulated changes to the file.

This function can update resources within modules that contain both code and resources.

Prior to Windows 7: If *lpData* is **NULL** and *cbData* is nonzero, the specified resource is NOT deleted and an exception is thrown.

Starting with Windows Vista: As noted above, there are restrictions on resource updates in files that contain RC Config data: LN files and .mui files. The restrictions are as follows:

Action	LN file	.mui file
1. Add a new type that doesn't exist in the LN or .mui files.	Add type in the LN file and treat as language-neutral (non-localizable) and add new type or item in the RC Config data	The only additions allowed are the following types: file Version, RC Config data, Side-by-side Assembly XML Manifest.
2. Add a new resource item to an existing type.	Uses the RC Config data to check whether the type exists in the .mui files associated with this LN file. If the type doesn't exist in the .mui files, add the item and treat new item as un-localizable. If the type exists in the .mui files, then adding is not allowed.	Only items of the following types may be added: File Version, RC Config data, Side-by-side Assembly XML Manifest.
3. Update a resource item.	Uses the RC Config data to check whether the type exists in the .mui files associated with the LN file. If the type doesn't exist in the .mui files, then this resource item update is allowed in the LN file. Otherwise, if the type exists in the .mui files associated with this LN file, then this update is not allowed.	The only updates allowed are items of the following types: file Version, RC Config data, Side-by-side Assembly XML Manifest.
4. Add a type/item for a new language.	Not allowed.	Not allowed.
5. Remove an existing type/item.	Works similarly to case 3. Uses the RC Config data to check whether the type exists in the .mui files associated with the LN file. If not, then the removal of the type/item from the LN file is allowed. Otherwise, if the type/item exists in	The only types allowed to be removed are: file Version, RC Config data, Side-by-side Assembly XML Manifest; also,

	the .mui files associated with this LN file, then the removal is not allowed.	only items of these types may be removed.
6. Add/delete/update a type not included in the RC Config data (such as Version, Side-by-side Assembly XML Manifest, or RC Config data itself).	Allowed.	Allowed.
7. Other update of non-localizable data, such as TYPELIB, reginst, and so on.	Update type or item in the LN file, treat as non-localizable, and add new type or item in the RC Config data.	Not applicable.
8. Add RC Config data.	Can be done but the integrity of the RC Config data is not checked.	Can be done but the integrity of the RC Config data is not checked.

Examples

For an example, see [Updating Resources](#).

Note

The winbase.h header defines UpdateResource as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows

Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[BeginUpdateResource](#)

[Conceptual](#)

[EndUpdateResource](#)

[LoadIcon](#)

[LoadString](#)

[LockResource](#)

[MAKEINTRESOURCE](#)

[MAKELANGID](#)

[Other Resources](#)

[Reference](#)

[Resources](#)

[SizeofResource](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

VerifyVersionInfoA function (winbase.h)

Article02/09/2023

Compares a set of operating system version requirements to the corresponding values for the currently running version of the system. This function is subject to manifest-based behavior. For more information, see the Remarks section.

Note: This function has been deprecated for Windows 10. See [targeting your applications for Windows](#) for more information.

Syntax

C++

```
BOOL VerifyVersionInfoA(  
    [in] LPOSVERSIONINFOEXA lpVersionInformation,  
    [in] DWORD             dwTypeMask,  
    [in] DWORDLONG         dwlConditionMask  
) ;
```

Parameters

[in] lpVersionInformation

A pointer to an [OSVERSIONINFOEX](#) structure containing the operating system version requirements to compare. The *dwTypeMask* parameter indicates the members of this structure that contain information to compare.

You must set the **dwOSVersionInfoSize** member of this structure to `sizeof(OSVERSIONINFOEX)`. You must also specify valid data for the members indicated by *dwTypeMask*. The function ignores structure members for which the corresponding *dwTypeMask* bit is not set.

[in] dwTypeMask

A mask that indicates the members of the [OSVERSIONINFOEX](#) structure to be tested. This parameter can be one or more of the following values.

Value	Meaning
VER_BUILDNUMBER	<code>dwBuildNumber</code>
0x00000004	

VER_MAJORVERSION	dwMajorVersion
0x0000002	If you are testing the major version, you must also test the minor version and the service pack major and minor versions.
VER_MINORVERSION	dwMinorVersion
0x0000001	
VER_PLATFORMID	dwPlatformId
0x0000008	
VER_SERVICEPACKMAJOR	wServicePackMajor
0x0000020	
VER_SERVICEPACKMINOR	wServicePackMinor
0x0000010	
VER_SUITENAME	wSuiteMask
0x0000040	
VER_PRODUCT_TYPE	wProductType
0x0000080	

[in] dwlConditionMask

The type of comparison to be used for each **IpVersionInfo** member being compared. To build this value, call the [VerSetConditionMask](#) function or the [VER_SET_CONDITION](#) macro once for each [OSVERSIONINFOEX](#) member being compared.

Return value

If the currently running operating system satisfies the specified requirements, the return value is a nonzero value.

If the current system does not satisfy the requirements, the return value is zero and [GetLastError](#) returns [ERROR_OLD_WIN_VERSION](#).

If the function fails, the return value is zero and [GetLastError](#) returns an error code other than [ERROR_OLD_WIN_VERSION](#).

Remarks

The [VerifyVersionInfo](#) function retrieves version information about the currently running operating system and compares it to the valid members of the **IpVersionInfo** structure. This enables you to easily determine the presence of a required set of operating system

version conditions. It is preferable to use **VerifyVersionInfo** rather than calling the [GetVersionEx](#) function to perform your own comparisons.

Typically, **VerifyVersionInfo** returns a nonzero value only if all specified tests succeed. However, major, minor, and service pack versions are tested in a hierarchical manner because the operating system version is a combination of these values. If a condition exists for the major version, it supersedes the conditions specified for minor version and service pack version. (You cannot test for major version greater than 5 and minor version less than or equal to 1. If you specify such a test, the function will change the request to test for a minor version greater than 1 because it is performing a greater than operation on the major version.)

The function tests these values in this order: major version, minor version, and service pack version. The function continues testing values while they are equal, and stops when one of the values does not meet the specified condition. For example, if you test for a system greater than or equal to version 5.1 service pack 1, the test succeeds if the current version is 6.0. (The major version is greater than the specified version, so the testing stops.) In the same way, if you test for a system greater than or equal to version 5.1 service pack 1, the test succeeds if the current version is 5.2. (The minor version is greater than the specified versions, so the testing stops.) However, if you test for a system greater than or equal to version 5.1 service pack 1, the test fails if the current version is 5.0 service pack 2. (The minor version is not greater than the specified version, so the testing stops.)

To verify a range of system versions, you must call **VerifyVersionInfo** twice. For example, to verify that the system version is greater than 5.0 but less than or equal to 5.1, first call **VerifyVersionInfo** to test that the major version is 5 and the minor version is greater than 0, then call **VerifyVersionInfo** again to test that the major version is 5 and the minor version is less than or equal to 1.

Identifying the current operating system is usually not the best way to determine whether a particular operating system feature is present. This is because the operating system may have had new features added in a redistributable DLL. Rather than using [GetVersionEx](#) to determine the operating system platform or version number, test for the presence of the feature itself. For more information, see [Operating System Version](#).

To verify whether the current operating system is either the Media Center or Tablet PC version of Windows, call [GetSystemMetrics](#).

Windows 10: **VerifyVersionInfo** returns false when called by applications that do not have a compatibility manifest for Windows 8.1 or Windows 10 if the *lpVersionInfo* parameter is set so that it specifies Windows 8.1 or Windows 10, even when the current

operating system version is Windows 8.1 or Windows 10. Specifically, **VerifyVersionInfo** has the following behavior:

- If the application has no manifest, **VerifyVersionInfo** behaves as if the operation system version is Windows 8 (6.2).
- If the application has a manifest that contains the GUID that corresponds to Windows 8.1, **VerifyVersionInfo** behaves as if the operation system version is Windows 8.1 (6.3).
- If the application has a manifest that contains the GUID that corresponds to Windows 10, **VerifyVersionInfo** behaves as if the operation system version is Windows 10 (10.0).

The [Version Helper functions](#) use the **VerifyVersionInfo** function, so the behavior [IsWindows8Point1OrGreater](#) and [IsWindows10OrGreater](#) are similarly affected by the presence and content of the manifest.

To manifest your applications for Windows 8.1 or Windows 10, see [Targeting your application for Windows](#).

Examples

For an example, see [Verifying the System Version](#).

ⓘ Note

The winbase.h header defines VerifyVersionInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetVersionEx](#)

[OSVERSIONINFOEX](#)

[Operating System Version](#)

[System Information Functions](#)

[VER_SET_CONDITION](#)

[VerSetConditionMask](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

VerifyVersionInfoW function (winbase.h)

Article02/09/2023

Compares a set of operating system version requirements to the corresponding values for the currently running version of the system. This function is subject to manifest-based behavior. For more information, see the Remarks section.

Note: This function has been deprecated for Windows 10. See [targeting your applications for Windows](#) for more information.

Syntax

C++

```
BOOL VerifyVersionInfoW(
    [in] LPOSVERSIONINFOEXW lpVersionInformation,
    [in] DWORD             dwTypeMask,
    [in] DWORDLONG         dwlConditionMask
);
```

Parameters

[in] lpVersionInformation

A pointer to an [OSVERSIONINFOEX](#) structure containing the operating system version requirements to compare. The *dwTypeMask* parameter indicates the members of this structure that contain information to compare.

You must set the **dwOSVersionInfoSize** member of this structure to `sizeof(OSVERSIONINFOEX)`. You must also specify valid data for the members indicated by *dwTypeMask*. The function ignores structure members for which the corresponding *dwTypeMask* bit is not set.

[in] dwTypeMask

A mask that indicates the members of the [OSVERSIONINFOEX](#) structure to be tested. This parameter can be one or more of the following values.

Value	Meaning
VER_BUILDNUMBER	<code>dwBuildNumber</code>
0x00000004	

VER_MAJORVERSION	dwMajorVersion
0x0000002	If you are testing the major version, you must also test the minor version and the service pack major and minor versions.
VER_MINORVERSION	dwMinorVersion
0x0000001	
VER_PLATFORMID	dwPlatformId
0x0000008	
VER_SERVICEPACKMAJOR	wServicePackMajor
0x0000020	
VER_SERVICEPACKMINOR	wServicePackMinor
0x0000010	
VER_SUITENAME	wSuiteMask
0x0000040	
VER_PRODUCT_TYPE	wProductType
0x0000080	

[in] dwlConditionMask

The type of comparison to be used for each **IpVersionInfo** member being compared. To build this value, call the [VerSetConditionMask](#) function or the [VER_SET_CONDITION](#) macro once for each [OSVERSIONINFOEX](#) member being compared.

Return value

If the currently running operating system satisfies the specified requirements, the return value is a nonzero value.

If the current system does not satisfy the requirements, the return value is zero and [GetLastError](#) returns [ERROR_OLD_WIN_VERSION](#).

If the function fails, the return value is zero and [GetLastError](#) returns an error code other than [ERROR_OLD_WIN_VERSION](#).

Remarks

The [VerifyVersionInfo](#) function retrieves version information about the currently running operating system and compares it to the valid members of the **IpVersionInfo** structure. This enables you to easily determine the presence of a required set of operating system

version conditions. It is preferable to use **VerifyVersionInfo** rather than calling the [GetVersionEx](#) function to perform your own comparisons.

Typically, **VerifyVersionInfo** returns a nonzero value only if all specified tests succeed. However, major, minor, and service pack versions are tested in a hierarchical manner because the operating system version is a combination of these values. If a condition exists for the major version, it supersedes the conditions specified for minor version and service pack version. (You cannot test for major version greater than 5 and minor version less than or equal to 1. If you specify such a test, the function will change the request to test for a minor version greater than 1 because it is performing a greater than operation on the major version.)

The function tests these values in this order: major version, minor version, and service pack version. The function continues testing values while they are equal, and stops when one of the values does not meet the specified condition. For example, if you test for a system greater than or equal to version 5.1 service pack 1, the test succeeds if the current version is 6.0. (The major version is greater than the specified version, so the testing stops.) In the same way, if you test for a system greater than or equal to version 5.1 service pack 1, the test succeeds if the current version is 5.2. (The minor version is greater than the specified versions, so the testing stops.) However, if you test for a system greater than or equal to version 5.1 service pack 1, the test fails if the current version is 5.0 service pack 2. (The minor version is not greater than the specified version, so the testing stops.)

To verify a range of system versions, you must call **VerifyVersionInfo** twice. For example, to verify that the system version is greater than 5.0 but less than or equal to 5.1, first call **VerifyVersionInfo** to test that the major version is 5 and the minor version is greater than 0, then call **VerifyVersionInfo** again to test that the major version is 5 and the minor version is less than or equal to 1.

Identifying the current operating system is usually not the best way to determine whether a particular operating system feature is present. This is because the operating system may have had new features added in a redistributable DLL. Rather than using [GetVersionEx](#) to determine the operating system platform or version number, test for the presence of the feature itself. For more information, see [Operating System Version](#).

To verify whether the current operating system is either the Media Center or Tablet PC version of Windows, call [GetSystemMetrics](#).

Windows 10: **VerifyVersionInfo** returns false when called by applications that do not have a compatibility manifest for Windows 8.1 or Windows 10 if the *lpVersionInfo* parameter is set so that it specifies Windows 8.1 or Windows 10, even when the current

operating system version is Windows 8.1 or Windows 10. Specifically, **VerifyVersionInfo** has the following behavior:

- If the application has no manifest, **VerifyVersionInfo** behaves as if the operation system version is Windows 8 (6.2).
- If the application has a manifest that contains the GUID that corresponds to Windows 8.1, **VerifyVersionInfo** behaves as if the operation system version is Windows 8.1 (6.3).
- If the application has a manifest that contains the GUID that corresponds to Windows 10, **VerifyVersionInfo** behaves as if the operation system version is Windows 10 (10.0).

The [Version Helper functions](#) use the **VerifyVersionInfo** function, so the behavior [IsWindows8Point1OrGreater](#) and [IsWindows10OrGreater](#) are similarly affected by the presence and content of the manifest.

To manifest your applications for Windows 8.1 or Windows 10, see [Targeting your application for Windows](#).

Examples

For an example, see [Verifying the System Version](#).

ⓘ Note

The winbase.h header defines VerifyVersionInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetVersionEx](#)

[OSVERSIONINFOEX](#)

[Operating System Version](#)

[System Information Functions](#)

[VER_SET_CONDITION](#)

[VerSetConditionMask](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WaitCommEvent function (winbase.h)

Article10/13/2021

Waits for an event to occur for a specified communications device. The set of events that are monitored by this function is contained in the event mask associated with the device handle.

Syntax

C++

```
BOOL WaitCommEvent(
    [in] HANDLE     hFile,
    [out] LPDWORD   lpEvtMask,
    [in] LPOVERLAPPED lpOverlapped
);
```

Parameters

[in] hFile

A handle to the communications device. The [CreateFile](#) function returns this handle.

[out] lpEvtMask

A pointer to a variable that receives a mask indicating the type of event that occurred. If an error occurs, the value is zero; otherwise, it is one of the following values.

Value	Meaning
EV_BREAK 0x0040	A break was detected on input.
EV_CTS 0x0008	The CTS (clear-to-send) signal changed state.
EV_DSR 0x0010	The DSR (data-set-ready) signal changed state.
EV_ERR 0x0080	A line-status error occurred. Line-status errors are CE_FRAME , CE_OVERRUN , and CE_RXPARITY .
EV_RING 0x0100	A ring indicator was detected.

EV_RLSD 0x0020	The RLSD (receive-line-signal-detect) signal changed state.
EV_RXCHAR 0x0001	A character was received and placed in the input buffer.
EV_RXFLAG 0x0002	The event character was received and placed in the input buffer. The event character is specified in the device's DCB structure, which is applied to a serial port by using the SetCommState function.
EV_TXEMPTY 0x0004	The last character in the output buffer was sent.

[in] *lpOverlapped*

A pointer to an [OVERLAPPED](#) structure. This structure is required if *hFile* was opened with [FILE_FLAG_OVERLAPPED](#).

If *hFile* was opened with [FILE_FLAG_OVERLAPPED](#), the *lpOverlapped* parameter must not be **NULL**. It must point to a valid [OVERLAPPED](#) structure. If *hFile* was opened with [FILE_FLAG_OVERLAPPED](#) and *lpOverlapped* is **NULL**, the function can incorrectly report that the operation is complete.

If *hFile* was opened with [FILE_FLAG_OVERLAPPED](#) and *lpOverlapped* is not **NULL**, [WaitCommEvent](#) is performed as an overlapped operation. In this case, the [OVERLAPPED](#) structure must contain a handle to a manual-reset event object (created by using the [CreateEvent](#) function).

If *hFile* was not opened with [FILE_FLAG_OVERLAPPED](#), [WaitCommEvent](#) does not return until one of the specified events or an error occurs.

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

The [WaitCommEvent](#) function monitors a set of events for a specified communications resource. To set and query the current event mask of a communications resource, use the [SetCommMask](#) and [GetCommMask](#) functions.

If the overlapped operation cannot be completed immediately, the function returns **FALSE** and the [GetLastError](#) function returns **ERROR_IO_PENDING**, indicating that the operation is executing in the background. When this happens, the system sets the **hEvent** member of the [OVERLAPPED](#) structure to the not-signaled state before **WaitCommEvent** returns, and then it sets it to the signaled state when one of the specified events or an error occurs. The calling process can use one of the [wait functions](#) to determine the event object's state and then use the [GetOverlappedResult](#) function to determine the results of the **WaitCommEvent** operation. [GetOverlappedResult](#) reports the success or failure of the operation, and the variable pointed to by the *lpEvtMask* parameter is set to indicate the event that occurred.

If a process attempts to change the device handle's event mask by using the [SetCommMask](#) function while an overlapped **WaitCommEvent** operation is in progress, **WaitCommEvent** returns immediately. The variable pointed to by the *lpEvtMask* parameter is set to zero.

Examples

For an example, see [Monitoring Communications Events](#).

Requirements

Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[Communications Functions](#)

[Communications Resources](#)

[CreateFile](#)

DCB

[GetCommMask](#)

[GetOverlappedResult](#)

[OVERLAPPED](#)

[SetCommMask](#)

[SetCommState](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WaitNamedPipeA function (winbase.h)

Article02/02/2023

Waits until either a time-out interval elapses or an instance of the specified named pipe is available for connection (that is, the pipe's server process has a pending [ConnectNamedPipe](#) operation on the pipe).

Syntax

C++

```
BOOL WaitNamedPipeA(
    [in] LPCSTR lpNamedPipeName,
    [in] DWORD   nTimeOut
);
```

Parameters

[in] lpNamedPipeName

The name of the named pipe. The string must include the name of the computer on which the server process is executing. A period may be used for the *servername* if the pipe is local. The following pipe name format is used:

\\\servername\\pipe\\pipename

[in] nTimeOut

The number of milliseconds that the function will wait for an instance of the named pipe to be available. You can used one of the following values instead of specifying a number of milliseconds.

Value	Meaning
NMPWAIT_USE_DEFAULT_WAIT 0x00000000	The time-out interval is the default value specified by the server process in the CreateNamedPipe function.
NMPWAIT_WAIT_FOREVER 0xffffffff	The function does not return until an instance of the named pipe is available.

Return value

If an instance of the pipe is available before the time-out interval elapses, the return value is nonzero.

If an instance of the pipe is not available before the time-out interval elapses, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If no instances of the specified named pipe exist, the **WaitNamedPipe** function returns immediately, regardless of the time-out value.

If the time-out interval expires, the **WaitNamedPipe** function will fail with the error **ERROR_SEM_TIMEOUT**.

If the function succeeds, the process should use the [CreateFile](#) function to open a handle to the named pipe. A return value of **TRUE** indicates that there is at least one instance of the pipe available. A subsequent [CreateFile](#) call to the pipe can fail, because the instance was closed by the server or opened by another client.

Windows 10, version 1709: Pipes are only supported within an app-container; ie, from one UWP process to another UWP process that's part of the same app. Also, named pipes must use the syntax `\.\.\pipe\LOCAL\` for the pipe name.

Examples

For an example, see [Named Pipe Client](#).

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	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CallNamedPipe](#)

[ConnectNamedPipe](#)

[CreateFile](#)

[CreateNamedPipe](#)

[Pipe Functions](#)

[Pipes Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WIN32_STREAM_ID structure (winbase.h)

Article09/01/2022

The WIN32_STREAM_ID structure contains stream data.

Syntax

C++

```
typedef struct _WIN32_STREAM_ID {
    DWORD          dwStreamId;
    DWORD          dwStreamAttributes;
    LARGE_INTEGER  Size;
    DWORD          dwStreamNameSize;
    WCHAR         cStreamName[ANYSIZE_ARRAY];
} WIN32_STREAM_ID, *LPWIN32_STREAM_ID;
```

Members

dwStreamId

Type of data. This member can be one of the following values.

Value	Meaning
BACKUP_ALTERNATE_DATA 0x00000004	Alternative data streams. This corresponds to the NTFS \$DATA stream type on a named data stream.
BACKUP_DATA 0x00000001	Standard data. This corresponds to the NTFS \$DATA stream type on the default (unnamed) data stream.
BACKUP_EA_DATA 0x00000002	Extended attribute data. This corresponds to the NTFS \$EA stream type.
BACKUP_LINK 0x00000005	Hard link information. This corresponds to the NTFS \$FILE_NAME stream type.
BACKUP_OBJECT_ID 0x00000007	Objects identifiers. This corresponds to the NTFS \$OBJECT_ID stream type.
BACKUP_PROPERTY_DATA 0x00000006	Property data.

BACKUP_REPARSE_DATA 0x00000008	Reparse points. This corresponds to the NTFS \$REPARSE_POINT stream type.
BACKUP_SECURITY_DATA 0x00000003	Security descriptor data.
BACKUP_SPARSE_BLOCK 0x00000009	Sparse file. This corresponds to the NTFS \$DATA stream type for a sparse file.
BACKUP_TXFS_DATA 0x0000000A	Transactional NTFS (TxF) data stream. This corresponds to the NTFS \$TXF_DATA stream type. Windows Server 2003 and Windows XP: This value is not supported.

`dwStreamAttributes`

Attributes of data to facilitate cross-operating system transfer. This member can be one or more of the following values.

Value	Meaning
STREAM_MODIFIED_WHEN_READ	Attribute set if the stream contains data that is modified when read. Allows the backup application to know that verification of data will fail.
STREAM_CONTAINS_SECURITY	Stream contains security data (general attributes). Allows the stream to be ignored on cross-operations restore.

`Size`

Size of data, in bytes.

`dwStreamNameSize`

Length of the name of the alternative data stream, in bytes.

`cStreamName[ANYSIZE_ARRAY]`

Unicode string that specifies the name of the alternative data stream.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]

Header

winbase.h (include Windows.h)

See also

[BackupRead](#)

[BackupSeek](#)

[BackupWrite](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WinExec function (winbase.h)

Article10/13/2021

Runs the specified application.

Note This function is provided only for compatibility with 16-bit Windows. Applications should use the [CreateProcess](#) function.

Syntax

C++

```
UINT WinExec(
    [in] LPCSTR lpCmdLine,
    [in] UINT    uCmdShow
);
```

Parameters

[in] *lpCmdLine*

The command line (file name plus optional parameters) for the application to be executed. If the name of the executable file in the *lpCmdLine* parameter does not contain a directory path, the system searches for the executable file in this sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory. The [GetSystemDirectory](#) function retrieves the path of this directory.
4. The Windows directory. The [GetWindowsDirectory](#) function retrieves the path of this directory.
5. The directories listed in the PATH environment variable.

[in] *uCmdShow*

The display options. For a list of the acceptable values, see the description of the *nCmdShow* parameter of the [ShowWindow](#) function.

Return value

If the function succeeds, the return value is greater than 31.

If the function fails, the return value is one of the following error values.

Return code/value	Description
0	The system is out of memory or resources.
ERROR_BAD_FORMAT	The .exe file is invalid.
ERROR_FILE_NOT_FOUND	The specified file was not found.
ERROR_PATH_NOT_FOUND	The specified path was not found.

Remarks

The **WinExec** function returns when the started process calls the [GetMessage](#) function or a time-out limit is reached. To avoid waiting for the time out delay, call the [GetMessage](#) function as soon as possible in any process started by a call to **WinExec**.

Security Remarks

The executable name is treated as the first white space-delimited string in *lpCmdLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
WinExec("C:\\\\Program Files\\\\MyApp", ...)
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **WinExec** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, use [CreateProcess](#) rather than **WinExec**. However, if you must use **WinExec** for legacy reasons, make sure the application name is enclosed in quotation marks as shown in the example below.

syntax

```
WinExec("\"C:\\Program Files\\MyApp.exe\" -L -S", ...)
```

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)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	ext-ms-win-kernel32-process-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

[CreateProcess](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WinMain function (winbase.h)

Article 10/13/2021

The user-provided entry point for a graphical Windows-based application.

WinMain is the conventional name used for the application entry point. For more information, see Remarks.

Syntax

C++

```
int __cdecl WinMain(
    [in]           HINSTANCE hInstance,
    [in, optional] HINSTANCE hPrevInstance,
    [in]           LPSTR     lpCmdLine,
    [in]           int       nShowCmd
);
```

Parameters

[in] `hInstance`

Type: **HINSTANCE**

A handle to the current instance of the application.

[in, optional] `hPrevInstance`

Type: **HINSTANCE**

A handle to the previous instance of the application. This parameter is always **NULL**. If you need to detect whether another instance already exists, create a uniquely named mutex using the [CreateMutex](#) function. **CreateMutex** will succeed even if the mutex already exists, but the function will return **ERROR_ALREADY_EXISTS**. This indicates that another instance of your application exists, because it created the mutex first. However, a malicious user can create this mutex before you do and prevent your application from starting. To prevent this situation, create a randomly named mutex and store the name so that it can only be obtained by an authorized user. Alternatively, you can use a file for this purpose. To limit your application to one instance per user, create a locked file in the user's profile directory.

[in] *lpCmdLine*

Type: LPSTR

The command line for the application, excluding the program name. To retrieve the entire command line, use the [GetCommandLine](#) function.

[in] *nShowCmd*

Type: int

Controls how the window is to be shown. This parameter can be any of the values that can be specified in the *nCmdShow* parameter for the [ShowWindow](#) function.

Return value

Type: int

If the function succeeds, terminating when it receives a [WM_QUIT](#) message, it should return the exit value contained in that message's *wParam* parameter. If the function terminates before entering the message loop, it should return zero.

Remarks

The name **WinMain** is used by convention by many programming frameworks.

Depending on the programming framework, the call to the **WinMain** function can be preceded and followed by additional activities specific to that framework.

Your **WinMain** should initialize the application, display its main window, and enter a message retrieval-and-dispatch loop that is the top-level control structure for the remainder of the application's execution. Terminate the message loop when it receives a [WM_QUIT](#) message. At that point, your **WinMain** should exit the application, returning the value passed in the [WM_QUIT](#) message's *wParam* parameter. If [WM_QUIT](#) was received as a result of calling [PostQuitMessage](#), the value of *wParam* is the value of the [PostQuitMessage](#) function's *nExitCode* parameter. For more information, see [Creating a Message Loop](#).

ANSI applications can use the *lpCmdLine* parameter of the **WinMain** function to access the command-line string, excluding the program name. Note that *lpCmdLine* uses the **LPSTR** data type instead of the **LPTSTR** data type. This means that **WinMain** cannot be used by Unicode programs. The [GetCommandLineW](#) function can be used to obtain the command line as a Unicode string. Some programming frameworks might provide an alternative entry point that provides a Unicode command line. For example, the

Microsoft Visual Studio C++ compiler uses the name `wWinMain` for the Unicode entry point.

Example

The following code example demonstrates the use of `WinMain`

C++

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInst, HINSTANCE hInstPrev, PSTR cmdline, int cmdshow)
{
    return MessageBox(NULL, "hello, world", "caption", 0);
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

[Conceptual](#)

[CreateMutex](#)

[DispatchMessage](#)

[GetCommandLine](#)

[GetMessage](#)

[Other Resources](#)

[PostQuitMessage](#)

Reference

[TranslateMessage](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

Wow64GetThreadSelectorEntry function (winbase.h)

Article 10/13/2021

Retrieves a descriptor table entry for the specified selector and WOW64 thread.

Syntax

C++

```
BOOL Wow64GetThreadSelectorEntry(
    [in]    HANDLE      hThread,
    [in]    DWORD       dwSelector,
    [out]   PWOW64_LDT_ENTRY lpSelectorEntry
);
```

Parameters

[in] hThread

A handle to the thread containing the specified selector. The handle must have been created with THREAD_QUERY_INFORMATION access to the thread. For more information, see [Thread Security and Access Rights](#).

[in] dwSelector

The global or local selector value to look up in the thread's descriptor tables.

[out] lpSelectorEntry

A pointer to a [WOW64_LDT_ENTRY](#) structure that receives a copy of the descriptor table entry if the specified selector has an entry in the specified thread's descriptor table. This information can be used to convert a segment-relative address to a linear virtual address.

Return value

If the function succeeds, the return value is nonzero. In that case, the structure pointed to by the *lpSelectorEntry* parameter receives a copy of the specified descriptor table entry.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The **Wow64GetThreadSelectorEntry** function is functional only on 64-bit systems and can be called only by 64-bit processes. If this function is called by a 32-bit process, the function fails with **ERROR_NOT_SUPPORTED**. A 32-bit process should use the [GetThreadSelectorEntry](#) function instead.

Debuggers use this function to convert segment-relative addresses to linear virtual addresses. The [ReadProcessMemory](#) and [WriteProcessMemory](#) functions use linear virtual addresses.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WriteEncryptedFileRaw function (winbase.h)

Article 10/13/2021

Restores (import) encrypted files. This is one of a group of Encrypted File System (EFS) functions that is intended to implement backup and restore functionality, while maintaining files in their encrypted state.

Syntax

C++

```
DWORD WriteEncryptedFileRaw(
    [in]          PFE_IMPORT_FUNC pfImportCallback,
    [in, optional] PVOID         pvCallbackContext,
    [in]          PVOID         pvContext
);
```

Parameters

[in] `pfImportCallback`

A pointer to the import callback function. The system calls the callback function multiple times, each time passing a buffer that will be filled by the callback function with a portion of backed-up file's data. When the callback function signals that the entire file has been processed, it tells the system that the restore operation is finished. For more information, see [ImportCallback](#).

[in, optional] `pvCallbackContext`

A pointer to an application-defined and allocated context block. The system passes this pointer to the callback function as a parameter so that the callback function can have access to application-specific data. This can be a structure and can contain any data the application needs, such as the handle to the file that will contain the backup copy of the encrypted file.

[in] `pvContext`

A pointer to a system-defined context block. The context block is returned by the [OpenEncryptedFileRaw](#) function. Do not modify it.

Return value

If the function succeeds, the return value is **ERROR_SUCCESS**.

If the function fails, it returns a nonzero error code defined in WinError.h. You can use [FormatMessage](#) with the **FORMAT_MESSAGE_FROM_SYSTEM** flag to get a generic text description of the error.

Remarks

The file being restored is not decrypted; it is restored in its encrypted state.

To back up an encrypted file, call [OpenEncryptedFileRaw](#) to open the file. Then call [ReadEncryptedFileRaw](#), passing it the address of an application-defined export callback function. The system calls this callback function multiple times until the entire file's contents have been read and backed up. When the backup is complete, call [CloseEncryptedFileRaw](#) to free resources and close the file. See [ExportCallback](#) for details about how to declare the export callback function.

To restore an encrypted file, call [OpenEncryptedFileRaw](#), specifying **CREATE_FOR_IMPORT** in the *ulFlags* parameter. Then call [WriteEncryptedFileRaw](#), passing it the address of an application-defined import callback function. The system calls this callback function multiple times until the entire file's contents have been read and restored. When the restore is complete, call [CloseEncryptedFileRaw](#) to free resources and close the file. See [ImportCallback](#) for details about how to declare the export callback function.

If the file is a sparse file that was backed up from a volume with a smaller sparse allocation unit size than the volume it is being restored to, the sparse blocks in the middle of the file may not properly align with the larger blocks and the function call would fail and set an **ERROR_INVALID_PARAMETER** last error code. The sparse allocation unit size is either 16 clusters or 64 KB, whichever is smaller.

This function is intended for restoring only encrypted files; see [BackupWrite](#) for restoring unencrypted files.

In Windows 8, Windows Server 2012, and later, 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)	No
Resilient File System (ReFS)	No

SMB 3.0 does not support EFS on shares with continuous availability capability.

Requirements

Minimum supported client	Windows XP Professional [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll
API set	ext-ms-win-advapi32-encryptedfile-l1-1-0 (introduced in Windows 8)

See also

[BackupWrite](#)

[CloseEncryptedFileRaw](#)

[File Encryption](#)

[File Management Functions](#)

[ImportCallback](#)

[OpenEncryptedFileRaw](#)

[ReadEncryptedFileRaw](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WritePrivateProfileSectionA function (winbase.h)

Article02/09/2023

Replaces the keys and values for the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WritePrivateProfileSectionA(
    [in] LPCSTR lpAppName,
    [in] LPCSTR lpString,
    [in] LPCSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section in which data is written. This section name is typically the name of the calling application.

[in] *lpString*

The new key names and associated values that are to be written to the named section. This string is limited to 65,535 bytes.

[in] *lpFileName*

The name of the initialization file. If this parameter does not contain a full path for the file, the function searches the Windows directory for the file. If the file does not exist and *lpFileName* does not contain a full path, the function creates the file in the Windows directory.

If the file exists and was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function creates a file using ANSI characters.

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

The data in the buffer pointed to by the *lpString* parameter consists of one or more null-terminated strings, followed by a final **null** character. Each string has the following form:

key=string

The **WritePrivateProfileSection** function is not case-sensitive; the string pointed to by the *lpAppName* parameter can be a combination of uppercase and lowercase letters.

If no section name matches the string pointed to by the *lpAppName* parameter, **WritePrivateProfileSection** creates the section at the end of the specified initialization file and initializes the new section with the specified key name and value pairs.

WritePrivateProfileSection deletes the existing keys and values for the named section and inserts the key names and values in the buffer pointed to by the *lpString* parameter. The function does not attempt to correlate old and new key names; if the new names appear in a different order from the old names, any comments associated with preexisting keys and values in the initialization file will probably be associated with incorrect keys and values.

This operation is atomic; no operations that read from or write to the specified initialization file are allowed while the information is being written.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.

- **USR:** - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- **SYS:** - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines WritePrivateProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

[RegCreateKeyEx](#)

[RegSetValueEx](#)

[WriteProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

WritePrivateProfileSectionW function (winbase.h)

Article02/09/2023

Replaces the keys and values for the specified section in an initialization file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WritePrivateProfileSectionW(
    [in] LPCWSTR lpAppName,
    [in] LPCWSTR lpString,
    [in] LPCWSTR lpFileName
);
```

Parameters

[in] *lpAppName*

The name of the section in which data is written. This section name is typically the name of the calling application.

[in] *lpString*

The new key names and associated values that are to be written to the named section. This string is limited to 65,535 bytes.

[in] *lpFileName*

The name of the initialization file. If this parameter does not contain a full path for the file, the function searches the Windows directory for the file. If the file does not exist and *lpFileName* does not contain a full path, the function creates the file in the Windows directory.

If the file exists and was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function creates a file using ANSI characters.

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

The data in the buffer pointed to by the *lpString* parameter consists of one or more null-terminated strings, followed by a final **null** character. Each string has the following form:

key=string

The **WritePrivateProfileSection** function is not case-sensitive; the string pointed to by the *lpAppName* parameter can be a combination of uppercase and lowercase letters.

If no section name matches the string pointed to by the *lpAppName* parameter, **WritePrivateProfileSection** creates the section at the end of the specified initialization file and initializes the new section with the specified key name and value pairs.

WritePrivateProfileSection deletes the existing keys and values for the named section and inserts the key names and values in the buffer pointed to by the *lpString* parameter. The function does not attempt to correlate old and new key names; if the new names appear in a different order from the old names, any comments associated with preexisting keys and values in the initialization file will probably be associated with incorrect keys and values.

This operation is atomic; no operations that read from or write to the specified initialization file are allowed while the information is being written.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.

- **USR:** - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- **SYS:** - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

 **Note**

The winbase.h header defines WritePrivateProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileSection](#)

[RegCreateKeyEx](#)

[RegSetValueEx](#)

[WriteProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

WritePrivateProfileStringA function (winbase.h)

Article02/09/2023

Copies a string into the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WritePrivateProfileStringA(
    [in] LPCSTR lpAppName,
    [in] LPCSTR lpKeyName,
    [in] LPCSTR lpString,
    [in] LPCSTR lpFileName
);
```

Parameters

[in] `lpAppName`

The name of the section to which the string will be copied. If the section does not exist, it is created. The name of the section is case-independent; the string can be any combination of uppercase and lowercase letters.

[in] `lpKeyName`

The name of the key to be associated with a string. If the key does not exist in the specified section, it is created. If this parameter is **NULL**, the entire section, including all entries within the section, is deleted.

[in] `lpString`

A null-terminated string to be written to the file. If this parameter is **NULL**, the key pointed to by the `lpKeyName` parameter is deleted.

[in] *lpFileName*

The name of the initialization file.

If the file was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Return value

If the function successfully copies the string to the initialization file, the return value is nonzero.

If the function fails, or if it flushes the cached version of the most recently accessed initialization file, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

If the *lpFileName* parameter does not contain a full path and file name for the file, **WritePrivateProfileString** searches the Windows directory for the file. If the file does not exist, this function creates the file in the Windows directory.

If *lpFileName* contains a full path and file name and the file does not exist, **WritePrivateProfileString** creates the file. The specified directory must already exist.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.

- **USR:** - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- **SYS:** - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

An application using the **WritePrivateProfileString** function to enter .ini file information into the registry should follow these guidelines:

- Ensure that no .ini file of the specified name exists on the system.
- Ensure that there is a key entry in the registry that specifies the .ini file. This entry should be under the path **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**.
- Specify a value for that .ini file key entry that specifies a section. That is to say, an application must specify a section name, as it would appear within an .ini file or registry entry. Here is an example: [My Section].
- For system files, specify SYS for an added value.
- For application files, specify USR within the added value. Here is an example: "My Section: USR: App Name\Section". And, since USR indicates a mapping under **HKEY_CURRENT_USER**, the application should also create a key under **HKEY_CURRENT_USER** that specifies the application name listed in the added value. For the example just given, that would be "App Name".
- After following the preceding steps, an application setup program should call **WritePrivateProfileString** with the first three parameters set to **NULL**, and the fourth parameter set to the INI file name. For example:

```
WritePrivateProfileString( NULL, NULL, NULL, L"appname.ini" );
```

- Such a call causes the mapping of an .ini file to the registry to take effect before the next system reboot. The system rereads the mapping information into shared memory. A user will not have to reboot their computer after installing an application in order to have future invocations of the application see the mapping of the .ini file to the registry.

Examples

The following sample code illustrates the preceding guidelines and is based on several assumptions:

- There is an application named App Name.
- That application uses an .ini file named AppName.ini.
- There is a section in the .ini file that we want to look like this:

syntax

```
[Section1]
FirstKey = It all worked out okay.
SecondKey = By golly, it works.
ThirdKey = Another test.
```

- The user will not have to reboot the system in order to have future invocations of the application see the mapping of the .ini file to the registry.

C++

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int main()
{
    TCHAR    inBuf[80];
    HKEY    hKey1, hKey2;
    DWORD   dwDisposition;
    LONG    lRetCode;
    TCHAR    szData[] = TEXT("USR:App Name\\Section1");

    // Create the .ini file key.
    lRetCode = RegCreateKeyEx ( HKEY_LOCAL_MACHINE,
        TEXT("SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion\\IniFileMapping\\appname.ini"),
        0,
        NULL,
        REG_OPTION_NON_VOLATILE,
        KEY_WRITE,
        NULL,
        &hKey1,
        &dwDisposition);

    if (lRetCode != ERROR_SUCCESS)
    {
        printf ("Error in creating appname.ini key (%d).\n", lRetCode);
        return (0) ;
    }

    // Set a section value
    lRetCode = RegSetValueEx ( hKey1,
        TEXT("Section1"),
        0,
        REG_SZ,
        (BYTE *)szData,
        sizeof(szData));

    if (lRetCode != ERROR_SUCCESS)
    {
        printf ("Error in setting Section1 value\n");
        // Close the key
        lRetCode = RegCloseKey( hKey1 );
    }
}
```

```

    if( lRetCode != ERROR_SUCCESS )
    {
        printf("Error in RegCloseKey (%d).\n", lRetCode);
        return (0) ;
    }
}

// Create an App Name key
lRetCode = RegCreateKeyEx ( HKEY_CURRENT_USER,
                           TEXT( "App Name" ),
                           0,
                           NULL,
                           REG_OPTION_NON_VOLATILE,
                           KEY_WRITE,
                           NULL,
                           &hKey2,
                           &dwDisposition);

if (lRetCode != ERROR_SUCCESS)
{
    printf ("Error in creating App Name key (%d).\n", lRetCode);

    // Close the key
    lRetCode = RegCloseKey( hKey2 );
    if( lRetCode != ERROR_SUCCESS )
    {
        printf("Error in RegCloseKey (%d).\n", lRetCode);
        return (0) ;
    }
}

// Force the system to read the mapping into shared memory
// so that future invocations of the application will see it
// without the user having to reboot the system
WritePrivateProfileStringW( NULL, NULL, NULL, L"appname.ini" );

// Write some added values
WritePrivateProfileString (TEXT( "Section1"),
                           TEXT( "FirstKey"),
                           TEXT( "It all worked out OK."),
                           TEXT( "appname.ini"));
WritePrivateProfileString (TEXT( "Section1"),
                           TEXT( "SecondKey"),
                           TEXT( "By golly, it works!"),
                           TEXT( "appname.ini"));
WritePrivateProfileString (TEXT( "Section1"),
                           TEXT( "ThirdKey"),
                           TEXT( "Another test..."),
                           TEXT( "appname.ini"));

// Test
GetPrivateProfileString (TEXT( "Section1"),
                           TEXT( "FirstKey"),
                           TEXT( "Error: GPPS failed"),
                           inBuf,

```

```

        80,
        TEXT("appname.ini"));
_tprintf (TEXT("Key: %s\n"), inBuf);

// Close the keys
lRetCode = RegCloseKey( hKey1 );
if( lRetCode != ERROR_SUCCESS )
{
    printf("Error in RegCloseKey (%d).\n", lRetCode);
    return(0);
}

lRetCode = RegCloseKey( hKey2 );
if( lRetCode != ERROR_SUCCESS )
{
    printf("Error in RegCloseKey (%d).\n", lRetCode);
    return(0);
}

return(1);
}

```

Note

The winbase.h header defines WritePrivateProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileString](#)

[WriteProfileString](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

WritePrivateProfileStringW function (winbase.h)

Article02/09/2023

Copies a string into the specified section of an initialization file.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WritePrivateProfileStringW(
    [in] LPCWSTR lpAppName,
    [in] LPCWSTR lpKeyName,
    [in] LPCWSTR lpString,
    [in] LPCWSTR lpFileName
);
```

Parameters

[in] `lpAppName`

The name of the section to which the string will be copied. If the section does not exist, it is created. The name of the section is case-independent; the string can be any combination of uppercase and lowercase letters.

[in] `lpKeyName`

The name of the key to be associated with a string. If the key does not exist in the specified section, it is created. If this parameter is **NULL**, the entire section, including all entries within the section, is deleted.

[in] `lpString`

A null-terminated string to be written to the file. If this parameter is **NULL**, the key pointed to by the `lpKeyName` parameter is deleted.

[in] *lpFileName*

The name of the initialization file.

If the file was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Return value

If the function successfully copies the string to the initialization file, the return value is nonzero.

If the function fails, or if it flushes the cached version of the most recently accessed initialization file, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

If the *lpFileName* parameter does not contain a full path and file name for the file, **WritePrivateProfileString** searches the Windows directory for the file. If the file does not exist, this function creates the file in the Windows directory.

If *lpFileName* contains a full path and file name and the file does not exist, **WritePrivateProfileString** creates the file. The specified directory must already exist.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.

- **USR:** - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- **SYS:** - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

An application using the **WritePrivateProfileString** function to enter .ini file information into the registry should follow these guidelines:

- Ensure that no .ini file of the specified name exists on the system.
- Ensure that there is a key entry in the registry that specifies the .ini file. This entry should be under the path **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\IniFileMapping**.
- Specify a value for that .ini file key entry that specifies a section. That is to say, an application must specify a section name, as it would appear within an .ini file or registry entry. Here is an example: [My Section].
- For system files, specify SYS for an added value.
- For application files, specify USR within the added value. Here is an example: "My Section: USR: App Name\Section". And, since USR indicates a mapping under **HKEY_CURRENT_USER**, the application should also create a key under **HKEY_CURRENT_USER** that specifies the application name listed in the added value. For the example just given, that would be "App Name".
- After following the preceding steps, an application setup program should call **WritePrivateProfileString** with the first three parameters set to **NULL**, and the fourth parameter set to the INI file name. For example:

```
WritePrivateProfileString( NULL, NULL, NULL, L"appname.ini" );
```

- Such a call causes the mapping of an .ini file to the registry to take effect before the next system reboot. The system rereads the mapping information into shared memory. A user will not have to reboot their computer after installing an application in order to have future invocations of the application see the mapping of the .ini file to the registry.

Examples

The following sample code illustrates the preceding guidelines and is based on several assumptions:

- There is an application named App Name.
- That application uses an .ini file named AppName.ini.
- There is a section in the .ini file that we want to look like this:

syntax

```
[Section1]
FirstKey = It all worked out okay.
SecondKey = By golly, it works.
ThirdKey = Another test.
```

- The user will not have to reboot the system in order to have future invocations of the application see the mapping of the .ini file to the registry.

C++

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int main()
{
    TCHAR    inBuf[80];
    HKEY    hKey1, hKey2;
    DWORD   dwDisposition;
    LONG    lRetCode;
    TCHAR    szData[] = TEXT("USR:App Name\\Section1");

    // Create the .ini file key.
    lRetCode = RegCreateKeyEx ( HKEY_LOCAL_MACHINE,
        TEXT("SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion\\IniFileMapping\\appname.ini"),
        0,
        NULL,
        REG_OPTION_NON_VOLATILE,
        KEY_WRITE,
        NULL,
        &hKey1,
        &dwDisposition);

    if (lRetCode != ERROR_SUCCESS)
    {
        printf ("Error in creating appname.ini key (%d).\n", lRetCode);
        return (0) ;
    }

    // Set a section value
    lRetCode = RegSetValueEx ( hKey1,
        TEXT("Section1"),
        0,
        REG_SZ,
        (BYTE *)szData,
        sizeof(szData));

    if (lRetCode != ERROR_SUCCESS)
    {
        printf ("Error in setting Section1 value\n");
        // Close the key
        lRetCode = RegCloseKey( hKey1 );
    }
}
```

```

    if( lRetCode != ERROR_SUCCESS )
    {
        printf("Error in RegCloseKey (%d).\n", lRetCode);
        return (0) ;
    }
}

// Create an App Name key
lRetCode = RegCreateKeyEx ( HKEY_CURRENT_USER,
                           TEXT( "App Name" ),
                           0,
                           NULL,
                           REG_OPTION_NON_VOLATILE,
                           KEY_WRITE,
                           NULL,
                           &hKey2,
                           &dwDisposition);

if (lRetCode != ERROR_SUCCESS)
{
    printf ("Error in creating App Name key (%d).\n", lRetCode);

    // Close the key
    lRetCode = RegCloseKey( hKey2 );
    if( lRetCode != ERROR_SUCCESS )
    {
        printf("Error in RegCloseKey (%d).\n", lRetCode);
        return (0) ;
    }
}

// Force the system to read the mapping into shared memory
// so that future invocations of the application will see it
// without the user having to reboot the system
WritePrivateProfileStringW( NULL, NULL, NULL, L"appname.ini" );

// Write some added values
WritePrivateProfileString (TEXT( "Section1"),
                           TEXT( "FirstKey"),
                           TEXT( "It all worked out OK."),
                           TEXT( "appname.ini"));
WritePrivateProfileString (TEXT( "Section1"),
                           TEXT( "SecondKey"),
                           TEXT( "By golly, it works!"),
                           TEXT( "appname.ini"));
WritePrivateProfileString (TEXT( "Section1"),
                           TEXT( "ThirdKey"),
                           TEXT( "Another test..."),
                           TEXT( "appname.ini"));

// Test
GetPrivateProfileString (TEXT( "Section1"),
                           TEXT( "FirstKey"),
                           TEXT( "Error: GPPS failed"),
                           inBuf,

```

```

        80,
        TEXT("appname.ini"));
_tprintf (TEXT("Key: %s\n"), inBuf);

// Close the keys
lRetCode = RegCloseKey( hKey1 );
if( lRetCode != ERROR_SUCCESS )
{
    printf("Error in RegCloseKey (%d).\n", lRetCode);
    return(0);
}

lRetCode = RegCloseKey( hKey2 );
if( lRetCode != ERROR_SUCCESS )
{
    printf("Error in RegCloseKey (%d).\n", lRetCode);
    return(0);
}

return(1);
}

```

ⓘ Note

The winbase.h header defines WritePrivateProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileString](#)

[WriteProfileString](#)

Feedback

Was this page helpful?



[Get help at Microsoft Q&A](#)

WritePrivateProfileStructA function (winbase.h)

Article02/09/2023

Copies data into a key in the specified section of an initialization file. As it copies the data, the function calculates a checksum and appends it to the end of the data. The [GetPrivateProfileStruct](#) function uses the checksum to ensure the integrity of the data.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WritePrivateProfileStructA(
    [in] LPCSTR lpszSection,
    [in] LPCSTR lpszKey,
    [in] LPVOID lpStruct,
    [in] UINT   uSizeStruct,
    [in] LPCSTR szFile
);
```

Parameters

[in] lpszSection

The name of the section to which the string will be copied. If the section does not exist, it is created. The name of the section is case independent, the string can be any combination of uppercase and lowercase letters.

[in] lpszKey

The name of the key to be associated with a string. If the key does not exist in the specified section, it is created. If this parameter is **NULL**, the entire section, including all keys and entries within the section, is deleted.

[in] lpStruct

The data to be copied. If this parameter is **NULL**, the key is deleted.

[in] *uSizeStruct*

The size of the buffer pointed to by the *lpStruct* parameter, in bytes.

[in] *szFile*

The name of the initialization file. If this parameter is **NULL**, the information is copied into the Win.ini file.

If the file was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Return value

If the function successfully copies the string to the initialization file, the return value is nonzero.

If the function fails, or if it flushes the cached version of the most recently accessed initialization file, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=string
.
.
.
```

If the *szFile* parameter does not contain a full path and file name for the file, [WritePrivateProfileString](#) searches the Windows directory for the file. If the file does not exist, this function creates the file in the Windows directory.

If *szFile* contains a full path and file name and the file does not exist, [WriteProfileString](#) creates the file. The specified directory must already exist.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the

system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines WritePrivateProfileStruct as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileString](#)

[WriteProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WritePrivateProfileStructW function (winbase.h)

Article02/09/2023

Copies data into a key in the specified section of an initialization file. As it copies the data, the function calculates a checksum and appends it to the end of the data. The [GetPrivateProfileStruct](#) function uses the checksum to ensure the integrity of the data.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WritePrivateProfileStructW(
    [in] LPCWSTR lpszSection,
    [in] LPCWSTR lpszKey,
    [in] LPVOID lpStruct,
    [in] UINT uSizeStruct,
    [in] LPCWSTR szFile
);
```

Parameters

[in] lpszSection

The name of the section to which the struct data will be copied. If the section does not exist, it is created. The name of the section is case independent.

[in] lpszKey

The name of the key to be associated with a struct. If the key does not exist in the specified section, it is created. If this parameter is **NULL**, the entire section, including all keys and entries within the section, is deleted.

[in] lpStruct

The data to be copied. If this parameter is **NULL**, the key is deleted.

[in] *uSizeStruct*

The size of the buffer pointed to by the *lpStruct* parameter, in bytes.

[in] *szFile*

The name of the initialization file. If this parameter is **NULL**, the information is copied into the Win.ini file.

If the file was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Return value

If the function successfully copies the struct to the initialization file, the return value is nonzero.

If the function fails, or if it flushes the cached version of the most recently accessed initialization file, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A section in the initialization file must have the following form:

syntax

```
[section]
key=struct
```

```
.
.
.
```

If the *szFile* parameter does not contain a full path and file name for the file, [WritePrivateProfileString](#) searches the Windows directory for the file. If the file does not exist, this function creates the file in the Windows directory.

If *szFile* contains a full path and file name and the file does not exist, [WriteProfileString](#) creates the file. The specified directory must already exist.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the

system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

This mapping is likely if an application modifies system-component initialization files, such as Control.ini, System.ini, and Winfile.ini. In this case, the function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines WritePrivateProfileStruct as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the **UNICODE** preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetPrivateProfileString](#)

[WriteProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WriteProfileSectionA function (winbase.h)

Article02/09/2023

Replaces the contents of the specified section in the Win.ini file with specified keys and values. If Win.ini uses Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WriteProfileSectionA(
    [in] LPCSTR lpAppName,
    [in] LPCSTR lpString
);
```

Parameters

[in] lpAppName

The name of the section. This section name is typically the name of the calling application.

[in] lpString

The new key names and associated values that are to be written to the named section. This string is limited to 65,535 bytes.

If the file exists and was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function creates a file using ANSI characters.

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

Keys and values in the *lpString* buffer consist of one or more **null**-terminated strings, followed by a final **null** character. Each string has the following form: *key=string*.

The **WriteProfileSection** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

WriteProfileSection deletes the existing keys and values for the named section and inserts the key names and values in the buffer pointed to by *lpString*. The function does not attempt to correlate old and new key names; if the new names appear in a different order from the old names, any comments associated with preexisting keys and values in the initialization file will probably be associated with incorrect keys and values.

This operation is atomic; no other operations that read from or write to the initialization file are allowed while the information is being written.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

When the operation has been mapped, the **WriteProfileSection** function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.

2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines WriteProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileSection](#)

[WritePrivateProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WriteProfileSectionW function (winbase.h)

Article02/09/2023

Replaces the contents of the specified section in the Win.ini file with specified keys and values. If Win.ini uses Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WriteProfileSectionW(  
    [in] LPCWSTR lpAppName,  
    [in] LPCWSTR lpString  
);
```

Parameters

[in] lpAppName

The name of the section. This section name is typically the name of the calling application.

[in] lpString

The new key names and associated values that are to be written to the named section. This string is limited to 65,535 bytes.

If the file exists and was created using Unicode characters, the function writes Unicode characters to the file. Otherwise, the function creates a file using ANSI characters.

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

Keys and values in the *lpString* buffer consist of one or more **null**-terminated strings, followed by a final **null** character. Each string has the following form: *key=string*.

The **WriteProfileSection** function is not case-sensitive; the strings can be a combination of uppercase and lowercase letters.

WriteProfileSection deletes the existing keys and values for the named section and inserts the key names and values in the buffer pointed to by *lpString*. The function does not attempt to correlate old and new key names; if the new names appear in a different order from the old names, any comments associated with preexisting keys and values in the initialization file will probably be associated with incorrect keys and values.

This operation is atomic; no other operations that read from or write to the initialization file are allowed while the information is being written.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

When the operation has been mapped, the **WriteProfileSection** function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.

2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines WriteProfileSection as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileSection](#)

[WritePrivateProfileSection](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WriteProfileStringA function (winbase.h)

Article02/09/2023

Copies a string into the specified section of the Win.ini file. If Win.ini uses Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WriteProfileStringA(
    [in] LPCSTR lpAppName,
    [in] LPCSTR lpKeyName,
    [in] LPCSTR lpString
);
```

Parameters

[in] *lpAppName*

The section to which the string is to be copied. If the section does not exist, it is created. The name of the section is not case-sensitive; the string can be any combination of uppercase and lowercase letters.

[in] *lpKeyName*

The key to be associated with the string. If the key does not exist in the specified section, it is created. If this parameter is **NULL**, the entire section, including all entries in the section, is deleted.

[in] *lpString*

A null-terminated string to be written to the file. If this parameter is **NULL**, the key pointed to by the *lpKeyName* parameter is deleted.

Return value

If the function successfully copies the string to the Win.ini file, the return value is nonzero.

If the function fails, or if it flushes the cached version of Win.ini, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A section in the Win.ini file must have the following form: *key=string*.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

When the operation has been mapped, the **WriteProfileString** function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.
4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an

unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.

5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines WriteProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileString](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WriteProfileStringW function (winbase.h)

Article02/09/2023

Copies a string into the specified section of the Win.ini file. If Win.ini uses Unicode characters, the function writes Unicode characters to the file. Otherwise, the function writes ANSI characters.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should store initialization information in the registry.

Syntax

C++

```
BOOL WriteProfileStringW(
    [in] LPCWSTR lpAppName,
    [in] LPCWSTR lpKeyName,
    [in] LPCWSTR lpString
);
```

Parameters

[in] lpAppName

The section to which the string is to be copied. If the section does not exist, it is created. The name of the section is not case-sensitive; the string can be any combination of uppercase and lowercase letters.

[in] lpKeyName

The key to be associated with the string. If the key does not exist in the specified section, it is created. If this parameter is **NULL**, the entire section, including all entries in the section, is deleted.

[in] lpString

A null-terminated string to be written to the file. If this parameter is **NULL**, the key pointed to by the *lpKeyName* parameter is deleted.

Return value

If the function successfully copies the string to the Win.ini file, the return value is nonzero.

If the function fails, or if it flushes the cached version of Win.ini, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

A section in the Win.ini file must have the following form: *key=string*.

The system keeps a cached version of the most recent registry file mapping to improve performance. If all parameters are **NULL**, the function flushes the cache. While the system is editing the cached version of the file, processes that edit the file itself will use the original file until the cache has been cleared.

The system maps most .ini file references to the registry, using the mapping defined under the following registry key:

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          IniFileMapping
```

When the operation has been mapped, the [WriteProfileString](#) function writes information to the registry, not to the initialization file; the change in the storage location has no effect on the function's behavior.

The profile functions use the following steps to locate initialization information:

1. Look in the registry for the name of the initialization file under the **IniFileMapping** key.
2. Look for the section name specified by *lpAppName*. This will be a named value under the key that has the name of the initialization file, or a subkey with this name, or the name will not exist as either a value or subkey.
3. If the section name specified by *lpAppName* is a named value, then that value specifies where in the registry you will find the keys for the section.

4. If the section name specified by *lpAppName* is a subkey, then named values under that subkey specify where in the registry you will find the keys for the section. If the key you are looking for does not exist as a named value, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the key.
5. If the section name specified by *lpAppName* does not exist as a named value or as a subkey, then there will be an unnamed value (shown as <No Name>) that specifies the default location in the registry where you will find the keys for the section.
6. If there is no subkey or entry for the section name, then look for the actual initialization file on the disk and read its contents.

When looking at values in the registry that specify other registry locations, there are several prefixes that change the behavior of the .ini file mapping:

- ! - this character forces all writes to go both to the registry and to the .ini file on disk.
- # - this character causes the registry value to be set to the value in the Windows 3.1 .ini file when a new user logs in for the first time after setup.
- @ - this character prevents any reads from going to the .ini file on disk if the requested data is not found in the registry.
- USR: - this prefix stands for **HKEY_CURRENT_USER**, and the text after the prefix is relative to that key.
- SYS: - this prefix stands for **HKEY_LOCAL_MACHINE\SOFTWARE**, and the text after the prefix is relative to that key.

Note

The winbase.h header defines WriteProfileString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[GetProfileString](#)

[WritePrivateProfileString](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WriteTapemark function (winbase.h)

Article 10/13/2021

The **WriteTapemark** function writes a specified number of filemarks, setmarks, short filemarks, or long filemarks to a tape device. These tapemarks divide a tape partition into smaller areas.

Syntax

C++

```
DWORD WriteTapemark(
    [in] HANDLE hDevice,
    [in] DWORD dwTapemarkType,
    [in] DWORD dwTapemarkCount,
    [in] BOOL bImmediate
);
```

Parameters

[in] `hDevice`

Handle to the device on which to write tapemarks. This handle is created by using the [CreateFile](#) function.

[in] `dwTapemarkType`

Type of tapemarks to write. This parameter can be one of the following values.

Value	Meaning
TAPE_FILEMARKS 1L	Writes the number of filemarks specified by the <i>dwTapemarkCount</i> parameter.
TAPE_LONG_FILEMARKS 3L	Writes the number of long filemarks specified by <i>dwTapemarkCount</i> .
TAPE_SETMARKS 0L	Writes the number of setmarks specified by <i>dwTapemarkCount</i> .
TAPE_SHORT_FILEMARKS 2L	Writes the number of short filemarks specified by <i>dwTapemarkCount</i> .

[in] `dwTapemarkCount`

Number of tapemarks to write.

[in] bImmediate

If this parameter is **TRUE**, the function returns immediately; if it is **FALSE**, the function does not return until the operation has been completed.

Return value

If the function succeeds, the return value is NO_ERROR.

If the function fails, it can return one of the following error codes.

Error	Description
ERROR_BEGINNING_OF_MEDIA 1102L	An attempt to access data before the beginning-of-medium marker failed.
ERROR_BUS_RESET 1111L	A reset condition was detected on the bus.
ERROR_DEVICE_NOT_PARTITIONED 1107L	The partition information could not be found when a tape was being loaded.
ERROR_END_OF_MEDIA 1100L	The end-of-tape marker was reached during an operation.
ERROR_FILEMARK_DETECTED 1101L	A filemark was reached during an operation.
ERROR_INVALID_BLOCK_LENGTH 1106L	The block size is incorrect on a new tape in a multivolume partition.
ERROR_MEDIA_CHANGED 1110L	The tape that was in the drive has been replaced or removed.
ERROR_NO_DATA_DETECTED 1104L	The end-of-data marker was reached during an operation.
ERROR_NO_MEDIA_IN_DRIVE 1112L	There is no media in the drive.
ERROR_NOT_SUPPORTED 50L	The tape driver does not support a requested function.
ERROR_PARTITION_FAILURE 1105L	The tape could not be partitioned.
ERROR_SETMARK_DETECTED 1103L	A setmark was reached during an operation.

ERROR_UNABLE_TO_LOCK_MEDIA 1108L	An attempt to lock the ejection mechanism failed.
ERROR_UNABLE_TO_UNLOAD_MEDIA 1109L	An attempt to unload the tape failed.
ERROR_WRITE_PROTECT 19L	The media is write protected.

Remarks

Filemarks, setmarks, short filemarks, and long filemarks are special recorded elements that denote the linear organization of the tape. None of these marks contain user data. Filemarks are the most general marks; setmarks provide a hierarchy not available with filemarks.

A short filemark contains a short erase gap that cannot be overwritten unless the write operation is performed from the beginning of the partition or from an earlier long filemark.

A long filemark contains a long erase gap that allows an application to position the tape at the beginning of the filemark and to overwrite the filemark and the erase gap.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[CreateFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

WTSGetActiveConsoleSessionId function (winbase.h)

Article 06/29/2021

Retrieves the session identifier of the console session. The console session is the session that is currently attached to the physical console. Note that it is not necessary that Remote Desktop Services be running for this function to succeed.

Syntax

C++

```
DWORD WTSGetActiveConsoleSessionId();
```

Return value

The session identifier of the session that is attached to the physical console. If there is no session attached to the physical console, (for example, if the physical console session is in the process of being attached or detached), this function returns 0xFFFFFFFF.

Remarks

The session identifier returned by this function is the identifier of the current physical console session. To monitor the state of the current physical console session, use the [WTSRegisterSessionNotification](#) function.

Requirements

Minimum supported client	Windows Vista
Minimum supported server	Windows Server 2008
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

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

See also

[ProcessIdToSessionId](#)

[WM_WTSSESSION_CHANGE](#)

[WTSQuerySessionInformation](#)

[WTSRegisterSessionNotification](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

ZombifyActCtx function (winbase.h)

Article10/13/2021

The **ZombifyActCtx** function deactivates the specified activation context, but does not deallocate it.

Syntax

C++

```
BOOL ZombifyActCtx(  
    [in] HANDLE hActCtx  
);
```

Parameters

[in] *hActCtx*

Handle to the activation context that is to be deactivated.

Return value

If the function succeeds, it returns **TRUE**. If a **null** handle is passed in the *hActCtx* parameter, **NONE_INVALID_PARAMETER** will be returned. Otherwise, it returns **FALSE**.

This function sets errors that can be retrieved by calling [GetLastError](#). For an example, see [Retrieving the Last-Error Code](#). For a complete list of error codes, see [System Error Codes](#).

Remarks

This function is intended for use in debugging threads using activation contexts. If the activation context deactivated by this function is subsequently accessed, the access fails and an assertion failure is shown in the debugger.

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)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[ACTCTX](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)