

Windows and Messages

Article01/24/2023

Overview of the Windows and Messages technology.

The Windows and Messages technology is not associated with any headers.

For programming guidance for this technology, see:

- [Windows and Messages](#)

Functions

AdjustWindowRect
Calculates the required size of the window rectangle, based on the desired client-rectangle size. The window rectangle can then be passed to the CreateWindow function to create a window whose client area is the desired size.
AdjustWindowRectEx
Calculates the required size of the window rectangle, based on the desired size of the client rectangle. The window rectangle can then be passed to the CreateWindowEx function to create a window whose client area is the desired size.
AllowSetForegroundWindow
Enables the specified process to set the foreground window using the SetForegroundWindow function. The calling process must already be able to set the foreground window. For more information, see Remarks later in this topic.
AnimateWindow
Enables you to produce special effects when showing or hiding windows. There are four types of animation:_roll, slide, collapse or expand, and alpha-blended fade.
AnyPopup
Indicates whether an owned, visible, top-level pop-up, or overlapped window exists on the screen. The function searches the entire screen, not just the calling application's client area.
ArrangeIconicWindows
Arranges all the minimized (iconic) child windows of the specified parent window.

[BeginDeferWindowPos](#)

Allocates memory for a multiple-window- position structure and returns the handle to the structure.

[BringWindowToTop](#)

Brings the specified window to the top of the Z order. If the window is a top-level window, it is activated. If the window is a child window, the top-level parent window associated with the child window is activated.

[BroadcastSystemMessage](#)

The BroadcastSystemMessage function sends a message to the specified recipients.
(BroadcastSystemMessage)

[BroadcastSystemMessageA](#)

Sends a message to the specified recipients. (BroadcastSystemMessageA)

[BroadcastSystemMessageExA](#)

Sends a message to the specified recipients. (BroadcastSystemMessageExA)

[BroadcastSystemMessageExW](#)

Sends a message to the specified recipients. (BroadcastSystemMessageExW)

[BroadcastSystemMessageW](#)

The BroadcastSystemMessageW (Unicode) function sends a message to the specified recipients.
(BroadcastSystemMessageW)

[CalculatePopupWindowPosition](#)

Calculates an appropriate pop-up window position using the specified anchor point, pop-up window size, flags, and the optional exclude rectangle.

[CallMsgFilterA](#)

Passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks. (ANSI)

[CallMsgFilterW](#)

Passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks. (Unicode)

[CallNextHookEx](#)

Passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.

[CallWindowProcA](#)

Passes message information to the specified window procedure. (ANSI)

[CallWindowProcW](#)

Passes message information to the specified window procedure. (Unicode)

[CascadeWindows](#)

Cascades the specified child windows of the specified parent window.

[ChangeWindowMessageFilter](#)

Adds or removes a message from the User Interface Privilege Isolation (UIPI) message filter.

[ChangeWindowMessageFilterEx](#)

Modifies the User Interface Privilege Isolation (UIPI) message filter for a specified window.

[ChildWindowFromPoint](#)

Determines which, if any, of the child windows belonging to a parent window contains the specified point. The search is restricted to immediate child windows. Grandchildren, and deeper descendant windows are not searched.

[ChildWindowFromPointEx](#)

Determines which, if any, of the child windows belonging to the specified parent window contains the specified point.

[CloseWindow](#)

Minimizes (but does not destroy) the specified window.

[CreateMDIWindowA](#)

Creates a multiple-document interface (MDI) child window. (ANSI)

[CreateMDIWindowW](#)

Creates a multiple-document interface (MDI) child window. (Unicode)

[CreateWindowA](#)

Creates an overlapped, pop-up, or child window. (ANSI)

[CreateWindowExA](#)

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the CreateWindow function. (ANSI)

[CreateWindowExW](#)

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the CreateWindow function. (Unicode)

[CreateWindowW](#)

Creates an overlapped, pop-up, or child window. (Unicode)

[DeferWindowPos](#)

Updates the specified multiple-window \diamond position structure for the specified window.

[DefFrameProcA](#)

Provides default processing for any window messages that the window procedure of a multiple-document interface (MDI) frame window does not process. (ANSI)

[DefFrameProcW](#)

Provides default processing for any window messages that the window procedure of a multiple-document interface (MDI) frame window does not process. (Unicode)

[DefMDIChildProcA](#)

Provides default processing for any window message that the window procedure of a multiple-document interface (MDI) child window does not process. (ANSI)

[DefMDIChildProcW](#)

Provides default processing for any window message that the window procedure of a multiple-document interface (MDI) child window does not process. (Unicode)

[DefWindowProcA](#)

Calls the default window procedure to provide default processing for any window messages that an application does not process. (ANSI)

[DefWindowProcW](#)

Calls the default window procedure to provide default processing for any window messages that an application does not process. (Unicode)

[DeregisterShellHookWindow](#)

Unregisters a specified Shell window that is registered to receive Shell hook messages.

[DestroyWindow](#)

Destroys the specified window.

[DispatchMessage](#)

The DispatchMessage function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

[DispatchMessageA](#)

Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function. (DispatchMessageA)

[DispatchMessageW](#)

The DispatchMessageW (Unicode) function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

[EndDeferWindowPos](#)

Simultaneously updates the position and size of one or more windows in a single screen-refreshing cycle.

[EndTask](#)

Forcibly closes the specified window.

[EnumChildWindows](#)

Enumerates the child windows that belong to the specified parent window by passing the handle to each child window, in turn, to an application-defined callback function.

[EnumPropsA](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. EnumProps continues until the last entry is enumerated or the callback function returns FALSE. (ANSI)

[EnumPropsExA](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. EnumPropsEx continues until the last entry is enumerated or the callback function returns FALSE. (ANSI)

[EnumPropsExW](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. EnumPropsEx continues until the last entry is enumerated or the callback function returns FALSE. (Unicode)

[EnumPropsW](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. EnumProps continues until the last entry is enumerated or the callback function returns FALSE. (Unicode)

[EnumThreadWindows](#)

Enumerates all nonchild windows associated with a thread by passing the handle to each window, in turn, to an application-defined callback function.

[EnumWindows](#)

Enumerates all top-level windows on the screen by passing the handle to each window, in turn, to an application-defined callback function. EnumWindows continues until the last top-level window is enumerated or the callback function returns FALSE.

[FindWindowA](#)

Retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search. (ANSI)

[FindWindowExA](#)

Retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search. (ANSI)

[FindWindowExW](#)

Retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search. (Unicode)

[FindWindowW](#)

Retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search. (Unicode)

[GET_X_LPARAM](#)

Retrieves the signed x-coordinate from the specified LPARAM value.

[GET_Y_LPARAM](#)

Retrieves the signed y-coordinate from the given LPARAM value.

[GetAltTabInfoA](#)

Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window. (ANSI)

[GetAltTabInfoW](#)

Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window. (Unicode)

[GetAncestor](#)

Retrieves the handle to the ancestor of the specified window.

[GetClassInfoA](#)

Retrieves information about a window class. (ANSI)

[GetClassInfoExA](#)

Retrieves information about a window class, including a handle to the small icon associated with the window class. The GetClassInfo function does not retrieve a handle to the small icon. (ANSI)

[GetClassInfoExW](#)

Retrieves information about a window class, including a handle to the small icon associated with the window class. The GetClassInfo function does not retrieve a handle to the small icon. (Unicode)

[GetClassInfoW](#)

Retrieves information about a window class. (Unicode)

[GetClassLongA](#)

Retrieves the specified 32-bit (DWORD) value from the WNDCLASSEX structure associated with the specified window. (ANSI)

[GetClassLongPtrA](#)

Retrieves the specified value from the WNDCLASSEX structure associated with the specified window. (ANSI)

[GetClassLongPtrW](#)

Retrieves the specified value from the WNDCLASSEX structure associated with the specified window. (Unicode)

[GetClassLongW](#)

Retrieves the specified 32-bit (DWORD) value from the WNDCLASSEX structure associated with the specified window. (Unicode)

[GetClassName](#)

The GetClassName function retrieves the name of the class to which the specified window belongs. (GetClassName)

[GetClassNameA](#)

Retrieves the name of the class to which the specified window belongs. (GetClassNameA)

[GetClassNameW](#)

The GetClassNameW (Unicode) function retrieves the name of the class to which the specified window belongs. (GetClassNameW)

[GetClassWord](#)

Retrieves the 16-bit (WORD) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

[GetClientRect](#)

Retrieves the coordinates of a window's client area.

[GetDesktopWindow](#)

Retrieves a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which other windows are painted.

[GetForegroundWindow](#)

Retrieves a handle to the foreground window (the window with which the user is currently working). The system assigns a slightly higher priority to the thread that creates the foreground window than it does to other threads.

[GetGUIThreadInfo](#)

Retrieves information about the active window or a specified GUI thread.

[GetInputState](#)

Determines whether there are mouse-button or keyboard messages in the calling thread's message queue.

[GetLastActivePopup](#)

Determines which pop-up window owned by the specified window was most recently active.

[GetLayeredWindowAttributes](#)

Retrieves the opacity and transparency color key of a layered window.

[GetMessage](#)

The GetMessage function retrieves a message from the calling thread's message queue.
([GetMessage](#))

[GetMessageA](#)

Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval. ([GetMessageA](#))

[GetMessageExtraInfo](#)

Retrieves the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue.

[GetMessagePos](#)

Retrieves the cursor position for the last message retrieved by the GetMessage function.

[GetMessageTime](#)

Retrieves the message time for the last message retrieved by the GetMessage function.

[GetMessageW](#)

The `GetMessageW` function (Unicode) retrieves a message from the calling thread's message queue. (`GetMessageW`)

[GetNextWindow](#)

Retrieves a handle to the next or previous window in the Z-Order. The next window is below the specified window; the previous window is above.

[GetParent](#)

Retrieves a handle to the specified window's parent or owner.

[GetProcessDefaultLayout](#)

Retrieves the default layout that is used when windows are created with no parent or owner.

[GetPropA](#)

Retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the `SetProp` function. (ANSI)

[GetPropW](#)

Retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the `SetProp` function. (Unicode)

[GetQueueStatus](#)

Retrieves the type of messages found in the calling thread's message queue.

[GetShellWindow](#)

Retrieves a handle to the Shell's desktop window.

[GetSysColor](#)

Retrieves the current color of the specified display element.

[GetSystemMetrics](#)

Retrieves the specified system metric or system configuration setting.

[GetTitleBarInfo](#)

Retrieves information about the specified title bar.

[GetTopWindow](#)

Examines the Z order of the child windows associated with the specified parent window and retrieves a handle to the child window at the top of the Z order.

[GetWindow](#)

Retrieves a handle to a window that has the specified relationship (Z-Order or owner) to the specified window.

[GetWindowDisplayAffinity](#)

Retrieves the current display affinity setting, from any process, for a given window.

[GetWindowInfo](#)

Retrieves information about the specified window. (GetWindowInfo)

[GetWindowLongA](#)

Retrieves information about the specified window. (GetWindowLongA)

[GetWindowLongPtrA](#)

Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory. (ANSI)

[GetWindowLongPtrW](#)

Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory. (Unicode)

[GetWindowLongW](#)

Retrieves information about the specified window. (GetWindowLongW)

[GetWindowModuleFileNameA](#)

Retrieves the full path and file name of the module associated with the specified window handle. (ANSI)

[GetWindowModuleFileNameW](#)

Retrieves the full path and file name of the module associated with the specified window handle. (Unicode)

[GetWindowPlacement](#)

Retrieves the show state and the restored, minimized, and maximized positions of the specified window.

[GetWindowRect](#)

Retrieves the dimensions of the bounding rectangle of the specified window. The dimensions are given in screen coordinates that are relative to the upper-left corner of the screen.

[GetWindowTextA](#)

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application. (ANSI)

[GetWindowTextLengthA](#)

Retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). (ANSI)

[GetWindowTextLengthW](#)

Retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). (Unicode)

[GetWindowTextW](#)

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application. (Unicode)

[GetWindowThreadProcessId](#)

Retrieves the identifier of the thread that created the specified window and, optionally, the identifier of the process that created the window.

[GetWindowWord](#)

Retrieves the 16-bit (DWORD) value at the specified offset into the extra window memor

[HOOKPROC](#)

An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function after the SendMessage function is called. The hook procedure can examine the message; it cannot modify it.

InSendMessage

Determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process) by a call to the `SendMessage` function.

InSendMessageEx

Determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process).

InternalGetWindowText

Copies the text of the specified window's title bar (if it has one) into a buffer.

IsChild

Determines whether a window is a child window or descendant window of a specified parent window.

IsGUIThread

Determines whether the calling thread is already a GUI thread. It can also optionally convert the thread to a GUI thread.

IsHungAppWindow

Determines whether the system considers that a specified application is not responding.

IsIconic

Determines whether the specified window is minimized (iconic).

IsProcessDPIAware

`IsProcessDPIAware` may be altered or unavailable. Instead, use `GetProcessDPIAwareness`.

IsWindow

Determines whether the specified window handle identifies an existing window.

IsWindowArranged

Determines whether the specified window is arranged (that is, whether it's snapped).

IsWindowUnicode

Determines whether the specified window is a native Unicode window.

[IsWindowVisible](#)

Determines the visibility state of the specified window.

[IsZoomed](#)

Determines whether a window is maximized.

[KillTimer](#)

Destroys the specified timer.

[LockSetForegroundWindow](#)

The foreground process can call the LockSetForegroundWindow function to disable calls to the SetForegroundWindow function.

[LogicalToPhysicalPoint](#)

Converts the logical coordinates of a point in a window to physical coordinates.

[MAKELPARAM](#)

Creates a value for use as an lParam parameter in a message. The macro concatenates the specified values.

[MAKELRESULT](#)

Creates a value for use as a return value from a window procedure. The macro concatenates the specified values.

[MAKEWPARAM](#)

Creates a value for use as a wParam parameter in a message. The macro concatenates the specified values.

[MoveWindow](#)

Changes the position and dimensions of the specified window.

[OpenIcon](#)

Restores a minimized (iconic) window to its previous size and position; it then activates the window.

[PeekMessageA](#)

Dispatches incoming nonqueued messages, checks the thread message queue for a posted message, and retrieves the message (if any exist). (ANSI)

[PeekMessageW](#)

Dispatches incoming nonqueued messages, checks the thread message queue for a posted message, and retrieves the message (if any exist). (Unicode)

[PhysicalToLogicalPoint](#)

Converts the physical coordinates of a point in a window to logical coordinates.

[PostMessageA](#)

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message. (ANSI)

[PostMessageW](#)

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message. (Unicode)

[PostQuitMessage](#)

Indicates to the system that a thread has made a request to terminate (quit). It is typically used in response to a WM_DESTROY message.

[PostThreadMessageA](#)

Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message. (ANSI)

[PostThreadMessageW](#)

Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message. (Unicode)

[PROOPENUMPROCA](#)

An application-defined callback function used with the EnumProps function. (ANSI)

[PROOPENUMPROCEXA](#)

Application-defined callback function used with the EnumPropsEx function. (ANSI)

[PROOPENUMPROCEWX](#)

Application-defined callback function used with the EnumPropsEx function. (Unicode)

[PROOPENUMPROCW](#)

An application-defined callback function used with the EnumProps function. (Unicode)

[RealChildWindowFromPoint](#)

Retrieves a handle to the child window at the specified point. The search is restricted to immediate child windows; grandchildren and deeper descendant windows are not searched.

[RealGetWindowClassA](#)

Retrieves a string that specifies the window type. (ANSI)

[RealGetWindowClassW](#)

Retrieves a string that specifies the window type. (Unicode)

[RegisterClassA](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassA)

[RegisterClassExA](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassExA)

[RegisterClassExW](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassExW)

[RegisterClassW](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassW)

[RegisterShellHookWindow](#)

Registers a specified Shell window to receive certain messages for events or notifications that are useful to Shell applications.

[RegisterWindowMessageA](#)

Defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages. (ANSI)

[RegisterWindowMessageW](#)

Defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages. (Unicode)

RemovePropA
Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed. (ANSI)
RemovePropW
Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed. (Unicode)
ReplyMessage
Replies to a message sent from another thread by the SendMessage function.
SENDASYNCPROC
An application-defined callback function used with the SendMessageCallback function.
SendMessage
The SendMessage function sends the specified message to a window or windows. (SendMessage function)
SendMessageA
Sends the specified message to a window or windows. The SendMessage function calls the window procedure for the specified window and does not return until the window procedure has processed the message. (SendMessageA)
SendMessageCallbackA
Sends the specified message to a window or windows. (SendMessageCallbackA)
SendMessageCallbackW
Sends the specified message to a window or windows. (SendMessageCallbackW)
SendMessageTimeoutA
Sends the specified message to one or more windows. (ANSI)
SendMessageTimeoutW
Sends the specified message to one or more windows. (Unicode)
SendMessageW
The SendMessageW (Unicode) function sends the specified message to a window or windows. (SendMessageW)

[SendNotifyMessageA](#)

Sends the specified message to a window or windows. (SendNotifyMessageA)

[SendNotifyMessageW](#)

Sends the specified message to a window or windows. (SendNotifyMessageW)

[SetAdditionalForegroundBoostProcesses](#)

`SetAdditionalForegroundBoostProcesses` is a performance assist API to help applications with a multi-process application model where multiple processes contribute to a foreground experience, either as data or rendering.

[SetClassLongA](#)

Replaces the specified 32-bit (long) value at the specified offset into the extra class memory or the `WNDCLASSEX` structure for the class to which the specified window belongs. (ANSI)

[SetClassLongPtrA](#)

Replaces the specified value at the specified offset in the extra class memory or the `WNDCLASSEX` structure for the class to which the specified window belongs. (ANSI)

[SetClassLongPtrW](#)

Replaces the specified value at the specified offset in the extra class memory or the `WNDCLASSEX` structure for the class to which the specified window belongs. (Unicode)

[SetClassLongW](#)

Replaces the specified 32-bit (long) value at the specified offset into the extra class memory or the `WNDCLASSEX` structure for the class to which the specified window belongs. (Unicode)

[SetClassWord](#)

Replaces the 16-bit (WORD) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

[SetCoalescableTimer](#)

Creates a timer with the specified time-out value and coalescing tolerance delay.

[SetForegroundWindow](#)

Brings the thread that created the specified window into the foreground and activates the window.

[SetLayeredWindowAttributes](#)

Sets the opacity and transparency color key of a layered window.

[SetMessageExtraInfo](#)

Sets the extra message information for the current thread.

[SetParent](#)

Changes the parent window of the specified child window.

[SetProcessDefaultLayout](#)

Changes the default layout when windows are created with no parent or owner only for the currently running process.

[SetProcessDPIAware](#)

`SetProcessDPIAware` may be altered or unavailable. Instead, use `SetProcessDPIAwareness`.

[SetPropA](#)

Adds a new entry or changes an existing entry in the property list of the specified window. (ANSI)

[SetPropW](#)

Adds a new entry or changes an existing entry in the property list of the specified window. (Unicode)

[SetSysColors](#)

Sets the colors for the specified display elements.

[SetTimer](#)

Creates a timer with the specified time-out value.

[SetWindowDisplayAffinity](#)

Stores the display affinity setting in kernel mode on the hWnd associated with the window.

[SetWindowLongA](#)

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory. (ANSI)

[SetWindowLongPtrA](#)

Changes an attribute of the specified window. (ANSI)

[SetWindowLongPtrW](#)

Changes an attribute of the specified window. (Unicode)

[SetWindowLongW](#)

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory. (Unicode)

[SetWindowPlacement](#)

Sets the show state and the restored, minimized, and maximized positions of the specified window.

[SetWindowPos](#)

Changes the size, position, and Z order of a child, pop-up, or top-level window. These windows are ordered according to their appearance on the screen. The topmost window receives the highest rank and is the first window in the Z order.

[SetWindowsHookExA](#)

Installs an application-defined hook procedure into a hook chain. (ANSI)

[SetWindowsHookExW](#)

Installs an application-defined hook procedure into a hook chain. (Unicode)

[SetWindowTextA](#)

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application. (ANSI)

[SetWindowTextW](#)

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application. (Unicode)

[ShowOwnedPopups](#)

Shows or hides all pop-up windows owned by the specified window.

[ShowWindow](#)

Sets the specified window's show state.

[ShowWindowAsync](#)

Sets the show state of a window without waiting for the operation to complete.

[SoundSentry](#)

Triggers a visual signal to indicate that a sound is playing.

[SwitchToThisWindow](#)

Switches focus to the specified window and brings it to the foreground.

[SystemParametersInfoA](#)

Retrieves or sets the value of one of the system-wide parameters. (ANSI)

[SystemParametersInfoW](#)

Retrieves or sets the value of one of the system-wide parameters. (Unicode)

[TileWindows](#)

Tiles the specified child windows of the specified parent window.

[TIMERPROC](#)

An application-defined callback function that processes WM_TIMER messages. The TIMERPROC type defines a pointer to this callback function. TimerProc is a placeholder for the application-defined function name.

[TranslateMDISysAccel](#)

Processes accelerator keystrokes for window menu commands of the multiple-document interface (MDI) child windows associated with the specified MDI client window.

[TranslateMessage](#)

Translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the GetMessage or PeekMessage function.

[UnhookWindowsHookEx](#)

Removes a hook procedure installed in a hook chain by the SetWindowsHookEx function.

[UnregisterClassA](#)

Unregisters a window class, freeing the memory required for the class. (ANSI)

[UnregisterClassW](#)

Unregisters a window class, freeing the memory required for the class. (Unicode)

[UpdateLayeredWindow](#)

Updates the position, size, shape, content, and translucency of a layered window.

[WaitMessage](#)

Yields control to other threads when a thread has no other messages in its message queue. The `WaitMessage` function suspends the thread and does not return until a new message is placed in the thread's message queue.

[WindowFromPhysicalPoint](#)

Retrieves a handle to the window that contains the specified physical point.

[WindowFromPoint](#)

Retrieves a handle to the window that contains the specified point.

[WinMain](#)

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

[WNDPROC](#)

A callback function, which you define in your application, that processes messages sent to a window.

Structures

[ALTTABINFO](#)

Contains status information for the application-switching (ALT+TAB) window.

[ANIMATIONINFO](#)

Describes the animation effects associated with user actions.

AUDIODESCRIPTION

Contains information associated with audio descriptions. This structure is used with the SystemParametersInfo function when the SPI_GETAUDIODESCRIPTION or SPI_SETAUDIODESCRIPTION action value is specified.

BSMINFO

Contains information about a window that denied a request from BroadcastSystemMessageEx.

CBT_CREATEWNDA

Contains information passed to a WH_CBT hook procedure, CBTProc, before a window is created. (ANSI)

CBT_CREATEWNDW

Contains information passed to a WH_CBT hook procedure, CBTProc, before a window is created. (Unicode)

CBTACTIVATESTRUCT

Contains information passed to a WH_CBT hook procedure, CBTProc, before a window is activated.

CHANGEFILTERSTRUCT

Contains extended result information obtained by calling the ChangeWindowMessageFilterEx function.

CLIENTCREATESTRUCT

Contains information about the menu and first multiple-document interface (MDI) child window of an MDI client window.

CREATESTRUCTA

Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the CreateWindowEx function. (ANSI)

CREATESTRUCTW

Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the CreateWindowEx function. (Unicode)

CWPRETSTRUCT

Defines the message parameters passed to a WH_CALLWNDPROCRET hook procedure, CallWndRetProc.

CWPSTRUCT	Defines the message parameters passed to a WH_CALLWNDPROC hook procedure, CallWndProc.
DEBUGHOOKINFO	Contains debugging information passed to a WH_DEBUG hook procedure, DebugProc.
EVENTMSG	Contains information about a hardware message sent to the system message queue. This structure is used to store message information for the JournalPlaybackProc callback function.
GUITHREADINFO	Contains information about a GUI thread.
KBDLLHOOKSTRUCT	Contains information about a low-level keyboard input event.
MDICREATESTRUCTA	Contains information about the class, title, owner, location, and size of a multiple-document interface (MDI) child window. (ANSI)
MDICREATESTRUCTW	Contains information about the class, title, owner, location, and size of a multiple-document interface (MDI) child window. (Unicode)
MINIMIZEDMETRICS	Contains the scalable metrics associated with minimized windows.
MINMAXINFO	Contains information about a window's maximized size and position and its minimum and maximum tracking size.
MOUSEHOOKSTRUCT	Contains information about a mouse event passed to a WH_MOUSE hook procedure, MouseProc.
MOUSEHOOKSTRUCTEX	Contains information about a mouse event passed to a WH_MOUSE hook procedure, MouseProc. This is an extension of the MOUSEHOOKSTRUCT structure that includes information about wheel movement or the use of the X button.

MSG
Contains message information from a thread's message queue.
MSLLHOOKSTRUCT
Contains information about a low-level mouse input event.
NCCALCSIZE_PARAMS
Contains information that an application can use while processing the WM_NCCALCSIZE message to calculate the size, position, and valid contents of the client area of a window.
NONCLIENTMETRICSA
Contains the scalable metrics associated with the nonclient area of a nonminimized window. (ANSI)
NONCLIENTMETRICSW
Contains the scalable metrics associated with the nonclient area of a nonminimized window. (Unicode)
STYLESTRUCT
Contains the styles for a window.
TITLEBARINFO
Contains title bar information.
TITLEBARINFOEX
Expands on the information described in the TITLEBARINFO structure by including the coordinates of each element of the title bar.
UPDATELAYEREDWINDOWINFO
Used by UpdateLayeredWindowIndirect to provide position, size, shape, content, and translucency information for a layered window.
WINDOWINFO
Contains window information.
WINDOWPLACEMENT
Contains information about the placement of a window on the screen.

[WINDOWPOS](#)

Contains information about the size and position of a window.

[WNDCLASSA](#)

Contains the window class attributes that are registered by the RegisterClass function. (ANSI)

[WNDCLASSEXA](#)

Contains window class information. (ANSI)

[WNDCLASSEXW](#)

Contains window class information. (Unicode)

[WNDCLASSW](#)

Contains the window class attributes that are registered by the RegisterClass function. (Unicode)

Feedback

Was this page helpful?

 Yes

 No

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

oleclt.h header

Article01/24/2023

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

- [Automation](#)
- [Component Object Model \(COM\)](#)
- [Windows and Messages](#)

oleclt.h contains the following programming interfaces:

Functions

DllRegisterServer	Instructs an in-process server to create its registry entries for all classes supported in this server module.
DllUnregisterServer	Instructs an in-process server to remove only those entries created through DllRegisterServer.
OleCreateFontIndirect	Creates and initializes a standard font object using an initial description of the font's properties in a FONTDESC structure.
OleCreatePictureIndirect	Creates a new picture object initialized according to a PICTDESC structure.
OleCreatePropertyFrame	Invokes a new property frame, that is, a property sheet dialog box, whose parent is hwndOwner, where the dialog is positioned at the point (x,y) in the parent window and has the caption lpszCaption.
OleCreatePropertyFrameIndirect	Creates a property frame, that is, a property sheet dialog box, based on a structure (OCPFIPARAMS) that contains the parameters, rather than specifying separate parameters as when calling OleCreatePropertyFrame.

[OleIconToCursor](#)

Converts an icon to a cursor.

[OleLoadPicture](#)

Creates a new picture object and initializes it from the contents of a stream. This is equivalent to calling OleCreatePictureIndirect with NULL as the first parameter, followed by a call to IPersistStream::Load. ([OleLoadPicture](#))

[OleLoadPictureEx](#)

Creates a new picture object and initializes it from the contents of a stream. This is equivalent to calling OleCreatePictureIndirect with NULL as the first parameter, followed by a call to IPersistStream::Load. ([OleLoadPictureEx](#))

[OleLoadPictureFile](#)

Creates an IPictureDisp object from a picture file on disk.

[OleLoadPictureFileEx](#)

Loads a picture from a file.

[OleLoadPicturePath](#)

Creates a new picture object and initializes it from the contents of a stream. This is equivalent to calling OleCreatePictureIndirect(NULL, ...) followed by IPersistStream::Load.

[OleSavePictureFile](#)

Saves a picture to a file.

[OleTranslateColor](#)

Converts an OLE_COLOR type to a COLORREF.

Structures

[FONTDESC](#)

Contains parameters used to create a font object through the OleCreateFontIndirect function.

OCPFIPARAMS

Contains parameters used to invoke a property sheet dialog box through the OleCreatePropertyFrameIndirect function.

PICTDESC

Contains parameters to create a picture object through the OleCreatePictureIndirect function.

Feedback

Was this page helpful?

 Yes

 No

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

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)

<p>EraseTape</p> <p>Erases all or part of a tape.</p>
<p>EscapeCommFunction</p> <p>Directs the specified communications device to perform an extended function.</p>
<p>ExecuteUmsThread</p> <p>Runs the specified UMS worker thread.</p>
<p>FatalExit</p> <p>Transfers execution control to the debugger. The behavior of the debugger thereafter is specific to the type of debugger used.</p>
<p>FileEncryptionStatusA</p> <p>Retrieves the encryption status of the specified file. (ANSI)</p>
<p>FileEncryptionStatusW</p> <p>Retrieves the encryption status of the specified file. (Unicode)</p>
<p>FileTimeToDosDateTime</p> <p>Converts a file time to MS-DOS date and time values.</p>
<p>FindActCtxSectionGuid</p> <p>The FindActCtxSectionGuid function retrieves information on a specific GUID in the current activation context and returns a ACTCTX_SECTION_KEYED_DATA structure.</p>
<p>FindActCtxSectionStringA</p> <p>The FindActCtxSectionString function retrieves information on a specific string in the current activation context and returns a ACTCTX_SECTION_KEYED_DATA structure. (ANSI)</p>
<p>FindActCtxSectionStringW</p> <p>The FindActCtxSectionString function retrieves information on a specific string in the current activation context and returns a ACTCTX_SECTION_KEYED_DATA structure. (Unicode)</p>
<p>FindAtomA</p> <p>Searches the local atom table for the specified character string and retrieves the atom associated with that string. (ANSI)</p>

[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.

[GetDIIIDirectoryA](#)

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

[GetDIIIDirectoryW](#)

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](#)

Retrieves the server process identifier for the specified named pipe.

[GetNamedPipeServerSessionId](#)

Retrieves the server session identifier for the specified named pipe.

[GetNextUmsListItem](#)

Returns the next user-mode scheduling (UMS) thread context in a list of thread contexts.

[GetNumaAvailableMemoryNode](#)

Retrieves the amount of memory available in the specified node.

[GetNumaAvailableMemoryNodeEx](#)

Retrieves the amount of memory that is available in a node specified as a USHORT value.

[GetNumaNodeNumberFromHandle](#)

Retrieves the NUMA node associated with the file or I/O device represented by the specified file handle.

<p>GetNumaNodeProcessorMask</p> <p>Retrieves the processor mask for the specified node.</p>
<p>GetNumaProcessorNode</p> <p>Retrieves the node number for the specified processor.</p>
<p>GetNumaProcessorNodeEx</p> <p>Retrieves the node number as a USHORT value for the specified logical processor.</p>
<p>GetNumaProximityNode</p> <p>Retrieves the NUMA node number that corresponds to the specified proximity domain identifier.</p>
<p>GetNumberOfEventLogRecords</p> <p>Retrieves the number of records in the specified event log.</p>
<p>GetOldestEventLogRecord</p> <p>Retrieves the absolute record number of the oldest record in the specified event log.</p>
<p>GetPrivateProfileInt</p> <p>The GetPrivateProfileInt function (winbase.h) retrieves an integer associated with a key in the specified section of an initialization file.</p>
<p>GetPrivateProfileIntA</p> <p>Retrieves an integer associated with a key in the specified section of an initialization file. (GetPrivateProfileIntA)</p>
<p>GetPrivateProfileIntW</p> <p>The GetPrivateProfileIntW (Unicode) function (winbase.h) retrieves an integer associated with a key in the specified section of an initialization file.</p>
<p>GetPrivateProfileSection</p> <p>The GetPrivateProfileSection function (winbase.h) retrieves all the keys and values for the specified section of an initialization file.</p>
<p>GetPrivateProfileSectionA</p> <p>Retrieves all the keys and values for the specified section of an initialization file. (GetPrivateProfileSectionA)</p>

[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. (GlobalMemoryStatus)
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
Decrements the lock count associated with a memory object that was allocated with GMEM_MOVEABLE.
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 CONTEXT 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.

[MapUserPhysicalPagesScatter](#)

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

[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.

<p>OperationStart</p> <p>Notifies the system that the application is about to start an operation.</p>
<p>PowerClearRequest</p> <p>Decrementsthe count of power requests of the specified type for a power request object.</p>
<p>PowerCreateRequest</p> <p>Creates a new power request object.</p>
<p>PowerSetRequest</p> <p>Increments the count of power requests of the specified type for a power request object.</p>
<p>PrepareTape</p> <p>Prepares the tape to be accessed or removed.</p>
<p>PrivilegedServiceAuditAlarmA</p> <p>Generates an audit message in the security event log. (PrivilegedServiceAuditAlarmA)</p>
<p>PulseEvent</p> <p>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.</p>
<p>PurgeComm</p> <p>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.</p>
<p>QueryActCtxSettingsW</p> <p>The QueryActCtxSettingsW function specifies the activation context, and the namespace and name of the attribute that is to be queried.</p>
<p>QueryActCtxW</p> <p>The QueryActCtxW function queries the activation context.</p>
<p>QueryDosDeviceA</p> <p>Retrieves information about MS-DOS device names. (QueryDosDeviceA)</p>

[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)

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

[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_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.

[OFSSTRUCT](#)

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?

 Yes

 No

Get help at Microsoft Q&A

WinMain function (winbase.h)

Article10/13/2021

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] HINSTANCE hPrevInstance,
    [in] LPSTR     lpCmdLine,
    [in] int        nShowCmd
);
```

Parameters

[in] hInstance

Type: **HINSTANCE**

A handle to the current instance of the application.

[in] 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](#)

[TranslateMessage](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

windef.h header

Article01/24/2023

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

- [Display Devices Reference](#)
- [High DPI](#)
- [Windows Accessibility Features](#)
- [Windows and Messages](#)
- [Windows GDI](#)

windef.h contains the following programming interfaces:

Structures

POINT
The POINT structure defines the x- and y-coordinates of a point.
POINTL
The POINTL structure defines the x- and y-coordinates of a point.
POINTS
The POINTS structure defines the x- and y-coordinates of a point.
RECT
The RECT structure defines a rectangle by the coordinates of its upper-left and lower-right corners.
RECTL
The RECTL structure defines a rectangle by the coordinates of its upper-left and lower-right corners.
SIZE
The SIZE structure defines the width and height of a rectangle.

Enumerations

[DPI_AWARENESS](#)

Identifies the dots per inch (dpi) setting for a thread, process, or window.

[DPI_HOSTING_BEHAVIOR](#)

Identifies the DPI hosting behavior for a window. This behavior allows windows created in the thread to host child windows with a different DPI_AWARENESS_CONTEXT.

Feedback

Was this page helpful?

 Yes

 No

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

windowsx.h header

Article01/24/2023

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

- [Developer Notes](#)
- [Windows and Messages](#)
- [Windows Controls](#)
- [Windows GDI](#)

windowsx.h contains the following programming interfaces:

Functions

<p>Button_Enable</p> <p>Enables or disables a button.</p>
<p>Button_GetCheck</p> <p>Gets the check state of a radio button or check box. You can use this macro or send the BM_GETCHECK message explicitly.</p>
<p>Button_GetState</p> <p>Retrieves the state of a button or check box. You can use this macro or send the BM_GETSTATE message explicitly.</p>
<p>Button_GetText</p> <p>Gets the text of a button.</p>
<p>Button_GetTextLength</p> <p>Gets the number of characters in the text of a button.</p>
<p>Button_SetCheck</p> <p>Sets the check state of a radio button or check box. You can use this macro or send the BM_SETCHECK message explicitly.</p>

[Button_SetState](#)

Sets the highlight state of a button. The highlight state indicates whether the button is highlighted as if the user had pushed it. You can use this macro or send the BM_SETSTATE message explicitly.

[Button_SetStyle](#)

Sets the style of a button. You can use this macro or send the BM_SETSTYLE message explicitly.

[Button_SetText](#)

Sets the text of a button.

[ComboBox_AddlItemData](#)

Adds item data to the list in a combo box at the specified location. You can use this macro or send the CB_ADDSTRING message explicitly.

[ComboBox_AddString](#)

Adds a string to a list in a combo box.

[ComboBox_DeleteString](#)

Deletes the item at the specified location in a list in a combo box. You can use this macro or send the CB_DELETESTRING message explicitly.

[ComboBox_Dir](#)

Adds names to the list displayed by a combo box.

[ComboBox_Enable](#)

Enables or disables a combo box control.

[ComboBox_FindlItemData](#)

Finds the first item in a combo box list that has the specified item data. You can use this macro or send the CB_FINDSTRING message explicitly.

[ComboBox_FindString](#)

Finds the first string in a combo box list that begins with the specified string. You can use this macro or send the CB_FINDSTRING message explicitly.

[ComboBox_FindStringExact](#)

Finds the first string in a combo box list that exactly matches the specified string, except that the search is not case sensitive. You can use this macro or send the CB_FINDSTRINGEXACT message explicitly.

[ComboBox_GetCount](#)

Gets the number of items in the list box of a combo box. You can use this macro or send the CB_GETCOUNT message explicitly.

[ComboBox_GetCurSel](#)

Gets the index of the currently selected item in a combo box. You can use this macro or send the CB_GETCURSEL message explicitly.

[ComboBox_GetDroppedControlRect](#)

Retrieves the screen coordinates of a combo box in its dropped-down state. You can use this macro or send the CB_GETDROPPEDCONTROLRECT message explicitly.

[ComboBox_GetDroppedState](#)

Ascertains whether the drop list in a combo box control is visible. You can use this macro or send the CB_GETDROPPEDSTATE message explicitly.

[ComboBox_GetExtendedUI](#)

Ascertains whether a combo box is using the default user interface (UI) or the extended UI. You can use this macro or send the CB_GETEXTENDEDUI message explicitly.

[ComboBox_GetItemData](#)

Gets the application-defined value associated with the specified list item in a combo box. You can use this macro or send the CB_GETITEMDATA message explicitly.

[ComboBox_GetItemHeight](#)

Retrieves the height of list items in a combo box. You can use this macro or send the CB_GETITEMHEIGHT message explicitly.

[ComboBox_GetLBText](#)

Gets a string from a list in a combo box. You can use this macro or send the CB_GETLBTEXT message explicitly.

[ComboBox_GetLBTextLen](#)

Gets the length of a string in the list in a combo box. You can use this macro or send the CB_GETLBTEXTLEN message explicitly.

[ComboBox_GetText](#)

Retrieves the text from a combo box control.

[ComboBox_GetTextLength](#)

Gets the number of characters in the text of a combo box.

[ComboBox_InsertItemData](#)

Inserts item data in a list in a combo box at the specified location. You can use this macro or send the CB_INSERTSTRING message explicitly.

[ComboBox_InsertString](#)

Adds a string to a list in a combo box at the specified location. You can use this macro or send the CB_INSERTSTRING message explicitly.

[ComboBox_LimitText](#)

Limits the length of the text the user may type into the edit control of a combo box. You can use this macro or send the CB_LIMITTEXT message explicitly.

[ComboBox_ResetContent](#)

Removes all items from the list box and edit control of a combo box. You can use this macro or send the CB_RESETCONTENT message explicitly.

[ComboBox_SelectItemData](#)

Searches a list in a combo box for an item that has the specified item data. If a matching item is found, the item is selected. You can use this macro or send the CB_SELECTSTRING message explicitly.

[ComboBox_SelectString](#)

Searches a list in a combo box for an item that begins with the characters in a specified string. If a matching item is found, the item is selected. You can use this macro or send the CB_SELECTSTRING message explicitly.

[ComboBox_SetCurSel](#)

Sets the currently selected item in a combo box. You can use this macro or send the CB_SETCURSEL message explicitly.

[ComboBox_SetExtendedUI](#)

Selects either the default user interface (UI) or the extended UI for a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style. You can use this macro or send the CB_SETEXTENDEDUI message explicitly.

[ComboBox_SetItemData](#)

Sets the application-defined value associated with the specified list item in a combo box. You can use this macro or send the CB_SETITEMDATA message explicitly.

[ComboBox_SetItemHeight](#)

Sets the height of list items or the selection field in a combo box. You can use this macro or send the CB_SETITEMHEIGHT message explicitly.

[ComboBox_SetText](#)

Sets the text of a combo box.

[ComboBox_ShowDropdown](#)

Shows or hides the list in a combo box. You can use this macro or send the CB_SHOWDROPDOWN message explicitly.

[DeleteFont](#)

The DeleteFont macro deletes a font object, freeing all system resources associated with the font object.

[Edit_CanUndo](#)

Determines whether there are any actions in the undo queue of an edit or rich edit control. You can use this macro or send the EM_CANUNDO message explicitly.

[Edit_EmptyUndoBuffer](#)

Resets the undo flag of an edit or rich edit control. The undo flag is set whenever an operation within the edit control can be undone. You can use this macro or send the EM_EMPTYUNDOBUFFER message explicitly.

[Edit_Enable](#)

Enables or disables an edit control.

[Edit_FmtLines](#)

Sets a flag that determines whether text retrieved from a multiline edit control includes soft line-break characters.

[Edit_GetFirstVisibleLine](#)

Gets the index of the uppermost visible line in a multiline edit or rich edit control. You can use this macro or send the EM_GETFIRSTVISIBLELINE message explicitly.

[Edit_GetHandle](#)

Gets a handle to the memory currently allocated for the text of a multiline edit control. You can use this macro or send the EM_GETHANDLE message explicitly.

[Edit_GetLine](#)

Retrieves a line of text from an edit or rich edit control. You can use this macro or send the EM_GETLINE message explicitly.

[Edit_GetLineCount](#)

Gets the number of lines in the text of an edit control. You can use this macro or send the EM_GETLINECOUNT message explicitly.

[Edit_GetModify](#)

Gets the state of an edit or rich edit control's modification flag. The flag indicates whether the contents of the control have been modified. You can use this macro or send the EM_GETMODIFY message explicitly.

[Edit_GetPasswordChar](#)

Gets the password character for an edit or rich edit control. You can use this macro or send the EM_GETPASSWORDCHAR message explicitly.

[Edit_GetRect](#)

Gets the formatting rectangle of an edit control. You can use this macro or send the EM_GETRECT message explicitly.

[Edit_GetSel](#)

Gets the starting and ending character positions of the current selection in an edit or rich edit control. You can use this macro or send the EM_GETSEL message explicitly.

[Edit_GetText](#)

Gets the text of an edit control.

[Edit_GetTextLength](#)

Gets the number of characters in the text of an edit control.

[Edit_GetWordBreakProc](#)

Retrieves the address of an edit or rich edit control's Wordwrap function. You can use this macro or send the EM_GETWORDBREAKPROC message explicitly.

[Edit_LimitText](#)

Limits the length of text that can be entered into an edit control. You can use this macro or send the EM_LIMITTEXT message explicitly.

[Edit_LineFromChar](#)

Gets the index of the line that contains the specified character index in a multiline edit or rich edit control. You can use this macro or send the EM_LINEFROMCHAR message explicitly.

[Edit_LineIndex](#)

Gets the character index of the first character of a specified line in a multiline edit or rich edit control. You can use this macro or send the EM_LINEINDEX message explicitly.

[Edit_LineLength](#)

Retrieves the length, in characters, of a line in an edit or rich edit control. You can use this macro or send the EM_LINELENGTH message explicitly.

[Edit_ReplaceSel](#)

Replaces the selected text in an edit control or a rich edit control with the specified text. You can use this macro or send the EM_REPLACESEL message explicitly.

[Edit_Scroll](#)

Scrolls the text vertically in a multiline edit or rich edit control. You can use this macro or send the EM_SCROLL message explicitly.

[Edit_ScrollCaret](#)

Scrolls the caret into view in an edit or rich edit control. You can use this macro or send the EM_SCROLLCARET message explicitly.

[Edit_SetHandle](#)

Sets the handle of the memory that will be used by a multiline edit control. You can use this macro or send the EM_SETHANDLE message explicitly.

[Edit_SetModify](#)

Sets or clears the modification flag for an edit control. The modification flag indicates whether the text within the edit control has been modified. You can use this macro or send the EM_SETMODIFY message explicitly.

[Edit_SetPasswordChar](#)

Sets or removes the password character for an edit or rich edit control. When a password character is set, that character is displayed in place of the characters typed by the user. You can use this macro or send the EM_SETPASSWORDCHAR message explicitly.

[Edit_SetReadOnly](#)

Sets or removes the read-only style (ES_READONLY) of an edit or rich edit control. You can use this macro or send the EM_SETREADONLY message explicitly.

[Edit_SetRect](#)

Sets the formatting rectangle of an edit control. You can use this macro or send the EM_SETRECT message explicitly.

[Edit_SetRectNoPaint](#)

Sets the formatting rectangle of a multiline edit control. This macro is equivalent to Edit_SetRect, except that it does not redraw the edit control window. You can use this macro or send the EM_SETRECTNP message explicitly.

[Edit_SetSel](#)

Selects a range of characters in an edit or rich edit control. You can use this macro or send the EM_SETSEL message explicitly.

[Edit_SetTabStops](#)

Sets the tab stops in a multiline edit or rich edit control. When text is copied to the control, any tab character in the text causes space to be generated up to the next tab stop. You can use this macro or send the EM_SETTABSTOPS message explicitly.

[Edit_SetText](#)

Sets the text of an edit control.

[Edit_SetWordBreakProc](#)

Replaces an edit control's default Wordwrap function with an application-defined Wordwrap function. You can use this macro or send the EM_SETWORDBREAKPROC message explicitly.

[Edit_Undo](#)

Undoes the last operation in the undo queue of an edit or rich edit control. You can use this macro or send the EM_UNDO message explicitly.

[GET_X_LPARAM](#)

Retrieves the signed x-coordinate from the specified LPARAM value.

[GET_Y_LPARAM](#)

Retrieves the signed y-coordinate from the given LPARAM value.

[ListBox_AddItemData](#)

Adds item data to the list box at the specified location. You can use this macro or send the LB_ADDSTRING message explicitly.

[ListBox_AddString](#)

Adds a string to a list box.

[ListBox_DeleteString](#)

Deletes the item at the specified location in a list box. You can use this macro or send the LB_DELETESTRING message explicitly.

[ListBox_Dir](#)

Adds names to the list displayed by a list box.

[ListBox_Enable](#)

Enables or disables a list box control.

[ListBox_FindItemData](#)

Finds the first item in a list box that has the specified item data. You can use this macro or send the LB_FINDSTRING message explicitly.

[ListBox_FindString](#)

Finds the first string in a list box that begins with the specified string. You can use this macro or send the LB_FINDSTRING message explicitly.

[ListBox_FindStringExact](#)

Finds the first list box string that exactly matches the specified string, except that the search is not case sensitive. You can use this macro or send the LB_FINDSTRINGEXACT message explicitly.

[ListBox_GetCaretIndex](#)

Retrieves the index of the list box item that has the focus rectangle in a multiple-selection list box. The item may or may not be selected. You can use this macro or send the LB_GETCARETINDEX message explicitly.

[ListBox_GetCount](#)

Gets the number of items in a list box. You can use this macro or send the LB_GETCOUNT message explicitly.

[ListBox_GetCurSel](#)

Gets the index of the currently selected item in a single-selection list box. You can use this macro or send the LB_GETCURSEL message explicitly.

[ListBox_GetHorizontalExtent](#)

Gets the width that a list box can be scrolled horizontally (the scrollable width) if the list box has a horizontal scroll bar. You can use this macro or send the LB_GETHORIZONTALEXTENT message explicitly.

[ListBox_GetItemData](#)

Gets the application-defined value associated with the specified list box item. You can use this macro or send the LB_GETITEMDATA message explicitly.

[ListBox_GetItemHeight](#)

Retrieves the height of items in a list box.

[ListBox_GetItemRect](#)

Gets the dimensions of the rectangle that bounds a list box item as it is currently displayed in the list box. You can use this macro or send the LB_GETITEMRECT message explicitly.

[ListBox_GetSel](#)

Gets the selection state of an item. You can use this macro or send the LB_GETSEL message explicitly.

[ListBox_GetSelCount](#)

Gets the count of selected items in a multiple-selection list box. You can use this macro or send the LB_GETSELCOUNT message explicitly.

[ListBox_GetSelItems](#)

Gets the indexes of selected items in a multiple-selection list box. You can use this macro or send the LB_GETSELITEMS message explicitly.

[ListBox_GetText](#)

Gets a string from a list box. You can use this macro or send the LB_GETTEXT message explicitly.

[ListBox_GetTextLen](#)

Gets the length of a string in a list box. You can use this macro or send the LB_GETTEXTLEN message explicitly.

[ListBox_GetTopIndex](#)

Gets the index of the first visible item in a list box. You can use this macro or send the LB_GETTOPINDEX message explicitly.

[ListBox_InsertItemData](#)

Inserts item data to a list box at the specified location. You can use this macro or send the LB_INSERTSTRING message explicitly.

[ListBox_InsertString](#)

Adds a string to a list box at the specified location. You can use this macro or send the LB_INSERTSTRING message explicitly.

[ListBox_ResetContent](#)

Removes all items from a list box. You can use this macro or send the LB_RESETCONTENT message explicitly.

[ListBox_SelectItemData](#)

Searches a list box for an item that has the specified item data. If a matching item is found, the item is selected. You can use this macro or send the LB_SELECTSTRING message explicitly.

[ListBox_SelectString](#)

Searches a list box for an item that begins with the characters in a specified string. If a matching item is found, the item is selected. You can use this macro or send the LB_SELECTSTRING message explicitly.

[ListBox_SetItemRange](#)

Selects or deselects one or more consecutive items in a multiple-selection list box. You can use this macro or send the LB_SELITEMRANGE message explicitly.

[ListBox_SetCaretIndex](#)

Sets the focus rectangle to the item at the specified index in a multiple-selection list box. If the item is not visible, it is scrolled into view. You can use this macro or send the LB_SETCARETINDEX message explicitly.

[ListBox_SetColumnWidth](#)

Sets the width of all columns in a multiple-column list box. You can use this macro or send the LB_SETCOLUMNWIDTH message explicitly.

[ListBox_SetCurSel](#)

Sets the currently selected item in a single-selection list box. You can use this macro or send the LB_SETSEL message explicitly.

[ListBox_SetHorizontalExtent](#)

Set the width by which a list box can be scrolled horizontally (the scrollable width).

[ListBox_SetItemData](#)

Sets the application-defined value associated with the specified list box item. You can use this macro or send the LB_SETITEMDATA message explicitly.

[ListBox_SetItemHeight](#)

Sets the height of items in a list box.

[ListBox_SetSel](#)

Selects or deselects an item in a multiple-selection list box. You can use this macro or send the LB_SETSEL message explicitly.

[ListBox_SetTabStops](#)

Sets the tab-stop positions in a list box. You can use this macro or send the LB_SETTABSTOPS message explicitly.

[ListBox_SetTopIndex](#)

Ensures that the specified item in a list box is visible. You can use this macro or send the LB_SETTOPINDEX message explicitly.

[ScrollBar_Enable](#)

Enables or disables a scroll bar control.

[ScrollBar_GetPos](#)

Retrieves the position of the scroll box (thumb) in the specified scroll bar.

[ScrollBar_GetRange](#)

Gets the range of a scroll bar.

[ScrollBar_SetPos](#)

Sets the position of the scroll box (thumb) in the specified scroll bar and, if requested, redraws the scroll bar to reflect the new position of the scroll box. ([ScrollBar_SetPos](#))

[ScrollBar_SetRange](#)

Sets the range of a scroll bar.

[ScrollBar_Show](#)

Shows or hides a scroll bar control.

[SelectFont](#)

The [SelectFont](#) macro selects a font object into the specified device context (DC). The new font object replaces the previous font object.

[Static_Enable](#)

Enables or disables a static control.

[Static_GetIcon](#)

Retrieves a handle to the icon associated with a static control that has the SS_ICON style. You can use this macro or send the STM_GETICON message explicitly.

[Static_GetText](#)

Gets the text of a static control.

[Static_GetTextLength](#)

Gets the number of characters in the text of a static control.

[Static_SetIcon](#)

Sets the icon for a static control. You can use this macro or send the STM_SETICON message explicitly.

Static_SetText

Sets the text of a static control.

Feedback

Was this page helpful?



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

GET_X_LPARAM macro (windowsx.h)

Article04/02/2021

Retrieves the signed x-coordinate from the specified LPARAM value.

Syntax

C++

```
void GET_X_LPARAM(  
    lp  
);
```

Parameters

lp

The data from which the x-coordinate is to be extracted.

Return value

Type: int

X-coordinate.

Remarks

Use GET_X_LPARAM instead of [LOWORD](#) to extract signed coordinate data. Negative screen coordinates may be returned on multiple monitor systems.

Requirements

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

Header

windowsx.h (include Windowsx.h)

See also

Conceptual

[GET_Y_LPARAM](#)

[LOWORD](#)

Reference

[Windows Data Types](#)

Feedback

Was this page helpful?

 Yes

 No

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

GET_Y_LPARAM macro (windowsx.h)

Article04/02/2021

Retrieves the signed y-coordinate from the given LPARAM value.

Syntax

C++

```
void GET_Y_LPARAM(  
    lp  
) ;
```

Parameters

lp

The data from which the y-coordinate is to be extracted.

Return value

Type: int

Y-coordinate.

Remarks

Use GET_Y_LPARAM instead of HIWORD to extract signed coordinate data. Negative screen coordinates may be returned on multiple monitor systems.

Requirements

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

Header

windowsx.h (include Windowsx.h)

See also

Conceptual

[GET_X_LPARAM](#)

[HIWORD](#)

Reference

[Windows Data Types](#)

Feedback

Was this page helpful?

 Yes

 No

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

winuser.h header

Article02/17/2023

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

- [Data Exchange](#)
- [Desktop Window Manager \(DWM\)](#)
- [Developer Notes](#)
- [Dialog Boxes](#)
- [Display Devices Reference](#)
- [High DPI](#)
- [Input Feedback Configuration](#)
- [Input Source Identification](#)
- [Internationalization for Windows Applications](#)
- [Keyboard and Mouse Input](#)
- [Menus and Other Resources](#)
- [Mobile Device Management Settings Provider](#)
- [Pointer Device Input Stack](#)
- [Pointer Input Messages and Notifications](#)
- [Remote Desktop Services](#)
- [Security and Identity](#)
- [System Services](#)
- [The Windows Shell](#)
- [Touch Hit Testing](#)
- [Touch Injection](#)
- [Touch Input](#)
- [Window Stations and Desktops](#)
- [Windows Accessibility Features](#)
- [Windows and Messages](#)
- [Windows Controls](#)
- [Windows GDI](#)

winuser.h contains the following programming interfaces:

Functions



[ActivateKeyboardLayout](#)

Sets the input locale identifier (formerly called the keyboard layout handle) for the calling thread or the current process. The input locale identifier specifies a locale as well as the physical layout of the keyboard.

[AddClipboardFormatListener](#)

Places the given window in the system-maintained clipboard format listener list.

[AdjustWindowRect](#)

Calculates the required size of the window rectangle, based on the desired client-rectangle size. The window rectangle can then be passed to the CreateWindow function to create a window whose client area is the desired size.

[AdjustWindowRectEx](#)

Calculates the required size of the window rectangle, based on the desired size of the client rectangle. The window rectangle can then be passed to the CreateWindowEx function to create a window whose client area is the desired size.

[AdjustWindowRectExForDpi](#)

Calculates the required size of the window rectangle, based on the desired size of the client rectangle and the provided DPI.

[AllowSetForegroundWindow](#)

Enables the specified process to set the foreground window using the SetForegroundWindow function. The calling process must already be able to set the foreground window. For more information, see Remarks later in this topic.

[AnimateWindow](#)

Enables you to produce special effects when showing or hiding windows. There are four types of animation:_roll, slide, collapse or expand, and alpha-blended fade.

[AnyPopup](#)

Indicates whether an owned, visible, top-level pop-up, or overlapped window exists on the screen. The function searches the entire screen, not just the calling application's client area.

[AppendMenuA](#)

Appends a new item to the end of the specified menu bar, drop-down menu, submenu, or shortcut menu. You can use this function to specify the content, appearance, and behavior of the menu item. (ANSI)

[AppendMenuW](#)

Appends a new item to the end of the specified menu bar, drop-down menu, submenu, or shortcut menu. You can use this function to specify the content, appearance, and behavior of the menu item. (Unicode)

[AreDpiAwarenessContextsEqual](#)

Determines whether two DPI_AWARENESS_CONTEXT values are identical.

[ArrangelconicWindows](#)

Arranges all the minimized (iconic) child windows of the specified parent window.

[AttachThreadInput](#)

Attaches or detaches the input processing mechanism of one thread to that of another thread.

[BeginDeferWindowPos](#)

Allocates memory for a multiple-window- position structure and returns the handle to the structure.

[BeginPaint](#)

The BeginPaint function prepares the specified window for painting and fills a PAINTSTRUCT structure with information about the painting.

[BlockInput](#)

Blocks keyboard and mouse input events from reaching applications.

[BringWindowToTop](#)

Brings the specified window to the top of the Z order. If the window is a top-level window, it is activated. If the window is a child window, the top-level parent window associated with the child window is activated.

[BroadcastSystemMessage](#)

The BroadcastSystemMessage function sends a message to the specified recipients. (BroadcastSystemMessage)

[BroadcastSystemMessageA](#)

Sends a message to the specified recipients. (BroadcastSystemMessageA)

BroadcastSystemMessageExA
Sends a message to the specified recipients. (BroadcastSystemMessageExA)
BroadcastSystemMessageExW
Sends a message to the specified recipients. (BroadcastSystemMessageExW)
BroadcastSystemMessageW
The BroadcastSystemMessageW (Unicode) function sends a message to the specified recipients. (BroadcastSystemMessageW)
CalculatePopupWindowPosition
Calculates an appropriate pop-up window position using the specified anchor point, pop-up window size, flags, and the optional exclude rectangle.
CallIMsgFilterA
Passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks. (ANSI)
CallIMsgFilterW
Passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks. (Unicode)
CallNextHookEx
Passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.
CallWindowProcA
Passes message information to the specified window procedure. (ANSI)
CallWindowProcW
Passes message information to the specified window procedure. (Unicode)
CascadeWindows
Cascades the specified child windows of the specified parent window.
ChangeClipboardChain
Removes a specified window from the chain of clipboard viewers.

[ChangeDisplaySettingsA](#)

The ChangeDisplaySettings function changes the settings of the default display device to the specified graphics mode. (ANSI)

[ChangeDisplaySettingsExA](#)

The ChangeDisplaySettingsEx function changes the settings of the specified display device to the specified graphics mode. (ANSI)

[ChangeDisplaySettingsExW](#)

The ChangeDisplaySettingsEx function changes the settings of the specified display device to the specified graphics mode. (Unicode)

[ChangeDisplaySettingsW](#)

The ChangeDisplaySettings function changes the settings of the default display device to the specified graphics mode. (Unicode)

[ChangeWindowMessageFilter](#)

Adds or removes a message from the User Interface Privilege Isolation (UIPI) message filter.

[ChangeWindowMessageFilterEx](#)

Modifies the User Interface Privilege Isolation (UIPI) message filter for a specified window.

[CharLowerA](#)

Converts a character string or a single character to lowercase. If the operand is a character string, the function converts the characters in place. (ANSI)

[CharLowerBuffA](#)

Converts uppercase characters in a buffer to lowercase characters. The function converts the characters in place. (ANSI)

[CharLowerBuffW](#)

Converts uppercase characters in a buffer to lowercase characters. The function converts the characters in place. (Unicode)

[CharLowerW](#)

Converts a character string or a single character to lowercase. If the operand is a character string, the function converts the characters in place. (Unicode)

[CharNextA](#)

Retrieves a pointer to the next character in a string. This function can handle strings consisting of either single- or multi-byte characters. (ANSI)

[CharNextExA](#)

Retrieves the pointer to the next character in a string. This function can handle strings consisting of either single- or multi-byte characters.

[CharNextW](#)

Retrieves a pointer to the next character in a string. This function can handle strings consisting of either single- or multi-byte characters. (Unicode)

[CharPrevA](#)

Retrieves a pointer to the preceding character in a string. This function can handle strings consisting of either single- or multi-byte characters. (ANSI)

[CharPrevExA](#)

Retrieves the pointer to the preceding character in a string. This function can handle strings consisting of either single- or multi-byte characters.

[CharPrevW](#)

Retrieves a pointer to the preceding character in a string. This function can handle strings consisting of either single- or multi-byte characters. (Unicode)

[CharToOemA](#)

Translates a string into the OEM-defined character set. Warning Do not use. (ANSI)

[CharToOemBuffA](#)

Translates a specified number of characters in a string into the OEM-defined character set. (ANSI)

[CharToOemBuffW](#)

Translates a specified number of characters in a string into the OEM-defined character set. (Unicode)

[CharToOemW](#)

Translates a string into the OEM-defined character set. Warning Do not use. (Unicode)

[CharUpperA](#)

Converts a character string or a single character to uppercase. If the operand is a character string, the function converts the characters in place. (ANSI)

[CharUpperBuffA](#)

Converts lowercase characters in a buffer to uppercase characters. The function converts the characters in place. (ANSI)

[CharUpperBuffW](#)

Converts lowercase characters in a buffer to uppercase characters. The function converts the characters in place. (Unicode)

[CharUpperW](#)

Converts a character string or a single character to uppercase. If the operand is a character string, the function converts the characters in place. (Unicode)

[CheckDlgButton](#)

Changes the check state of a button control.

[CheckMenuItem](#)

Sets the state of the specified menu item's check-mark attribute to either selected or clear.

[CheckMenuItem](#)

Checks a specified menu item and makes it a radio item. At the same time, the function clears all other menu items in the associated group and clears the radio-item type flag for those items.

[CheckRadioButton](#)

Adds a check mark to (checks) a specified radio button in a group and removes a check mark from (clears) all other radio buttons in the group.

[ChildWindowFromPoint](#)

Determines which, if any, of the child windows belonging to a parent window contains the specified point. The search is restricted to immediate child windows. Grandchildren, and deeper descendant windows are not searched.

[ChildWindowFromPointEx](#)

Determines which, if any, of the child windows belonging to the specified parent window contains the specified point.

[ClientToScreen](#)

The ClientToScreen function converts the client-area coordinates of a specified point to screen coordinates.

[ClipCursor](#)

Confines the cursor to a rectangular area on the screen.

[CloseClipboard](#)

Closes the clipboard.

[CloseDesktop](#)

Closes an open handle to a desktop object.

[CloseGestureInfoHandle](#)

Closes resources associated with a gesture information handle.

[CloseTouchInputHandle](#)

Closes a touch input handle, frees process memory associated with it, and invalidates the handle.

[CloseWindow](#)

Minimizes (but does not destroy) the specified window.

[CloseWindowStation](#)

Closes an open window station handle.

[CopyAcceleratorTableA](#)

Copies the specified accelerator table. This function is used to obtain the accelerator-table data that corresponds to an accelerator-table handle, or to determine the size of the accelerator-table data. (ANSI)

[CopyAcceleratorTableW](#)

Copies the specified accelerator table. This function is used to obtain the accelerator-table data that corresponds to an accelerator-table handle, or to determine the size of the accelerator-table data. (Unicode)

[CopyCursor](#)

Copies the specified cursor.

[CopyIcon](#)

Copies the specified icon from another module to the current module.

[CopyImage](#)

Creates a new image (icon, cursor, or bitmap) and copies the attributes of the specified image to the new one. If necessary, the function stretches the bits to fit the desired size of the new image.

[CopyRect](#)

The CopyRect function copies the coordinates of one rectangle to another.

[CountClipboardFormats](#)

Retrieves the number of different data formats currently on the clipboard.

[CreateAcceleratorTableA](#)

Creates an accelerator table. (ANSI)

[CreateAcceleratorTableW](#)

Creates an accelerator table. (Unicode)

[CreateCaret](#)

Creates a new shape for the system caret and assigns ownership of the caret to the specified window. The caret shape can be a line, a block, or a bitmap.

[CreateCursor](#)

Creates a cursor having the specified size, bit patterns, and hot spot.

[CreateDesktopA](#)

Creates a new desktop, associates it with the current window station of the calling process, and assigns it to the calling thread. (ANSI)

[CreateDesktopExA](#)

Creates a new desktop with the specified heap, associates it with the current window station of the calling process, and assigns it to the calling thread. (ANSI)

[CreateDesktopExW](#)

Creates a new desktop with the specified heap, associates it with the current window station of the calling process, and assigns it to the calling thread. (Unicode)

[CreateDesktopW](#)

Creates a new desktop, associates it with the current window station of the calling process, and assigns it to the calling thread. (Unicode)

[CreateDialogA](#)

Creates a modeless dialog box from a dialog box template resource. The CreateDialog macro uses the CreateDialogParam function. (ANSI)

[CreateDialogIndirectA](#)

Creates a modeless dialog box from a dialog box template in memory. The CreateDialogIndirect macro uses the CreateDialogIndirectParam function. (ANSI)

[CreateDialogIndirectParamA](#)

Creates a modeless dialog box from a dialog box template in memory. (ANSI)

[CreateDialogIndirectParamW](#)

Creates a modeless dialog box from a dialog box template in memory. (Unicode)

[CreateDialogIndirectW](#)

Creates a modeless dialog box from a dialog box template in memory. The CreateDialogIndirect macro uses the CreateDialogIndirectParam function. (Unicode)

[CreateDialogParamA](#)

Creates a modeless dialog box from a dialog box template resource. (ANSI)

[CreateDialogParamW](#)

Creates a modeless dialog box from a dialog box template resource. (Unicode)

[CreateDialogW](#)

Creates a modeless dialog box from a dialog box template resource. The CreateDialog macro uses the CreateDialogParam function. (Unicode)

[CreateIcon](#)

Creates an icon that has the specified size, colors, and bit patterns.

[CreateIconFromResource](#)

Creates an icon or cursor from resource bits describing the icon. (CreateIconFromResource)

[CreateIconFromResourceEx](#)

Creates an icon or cursor from resource bits describing the icon. ([CreateIconFromResourceEx](#))

[CreateIconIndirect](#)

Creates an icon or cursor from an ICONINFO structure.

[CreateMDIWindowA](#)

Creates a multiple-document interface (MDI) child window. (ANSI)

[CreateMDIWindowW](#)

Creates a multiple-document interface (MDI) child window. (Unicode)

[CreateMenu](#)

Creates a menu. The menu is initially empty, but it can be filled with menu items by using the [InsertMenuItem](#), [AppendMenu](#), and [InsertMenu](#) functions.

[CreatePopupMenu](#)

Creates a drop-down menu, submenu, or shortcut menu.

[CreateSyntheticPointerDevice](#)

Configures the pointer injection device for the calling application, and initializes the maximum number of simultaneous pointers that the app can inject.

[CreateWindowA](#)

Creates an overlapped, pop-up, or child window. (ANSI)

[CreateWindowExA](#)

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the [CreateWindow](#) function. (ANSI)

[CreateWindowExW](#)

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the [CreateWindow](#) function. (Unicode)

[CreateWindowStationA](#)

Creates a window station object, associates it with the calling process, and assigns it to the current session. (ANSI)

[CreateWindowStationW](#)

Creates a window station object, associates it with the calling process, and assigns it to the current session. (Unicode)

[CreateWindowW](#)

Creates an overlapped, pop-up, or child window. (Unicode)

[DefDlgProcA](#)

Calls the default dialog box window procedure to provide default processing for any window messages that a dialog box with a private window class does not process. (ANSI)

[DefDlgProcW](#)

Calls the default dialog box window procedure to provide default processing for any window messages that a dialog box with a private window class does not process. (Unicode)

[DeferWindowPos](#)

Updates the specified multiple-window  position structure for the specified window.

[DefFrameProcA](#)

Provides default processing for any window messages that the window procedure of a multiple-document interface (MDI) frame window does not process. (ANSI)

[DefFrameProcW](#)

Provides default processing for any window messages that the window procedure of a multiple-document interface (MDI) frame window does not process. (Unicode)

[DefMDIChildProcA](#)

Provides default processing for any window message that the window procedure of a multiple-document interface (MDI) child window does not process. (ANSI)

[DefMDIChildProcW](#)

Provides default processing for any window message that the window procedure of a multiple-document interface (MDI) child window does not process. (Unicode)

[DefRawInputProc](#)

Verifies that the size of the RAWINPUTHEADER structure is correct.

[DefWindowProcA](#)

Calls the default window procedure to provide default processing for any window messages that an application does not process. (ANSI)

[DefWindowProcW](#)

Calls the default window procedure to provide default processing for any window messages that an application does not process. (Unicode)

[DeleteMenu](#)

Deletes an item from the specified menu. If the menu item opens a menu or submenu, this function destroys the handle to the menu or submenu and frees the memory used by the menu or submenu.

[DeregisterShellHookWindow](#)

Unregisters a specified Shell window that is registered to receive Shell hook messages.

[DestroyAcceleratorTable](#)

Destroys an accelerator table.

[DestroyCaret](#)

Destroys the caret's current shape, frees the caret from the window, and removes the caret from the screen.

[DestroyCursor](#)

Destroys a cursor and frees any memory the cursor occupied. Do not use this function to destroy a shared cursor.

[DestroyIcon](#)

Destroys an icon and frees any memory the icon occupied.

[DestroyMenu](#)

Destroys the specified menu and frees any memory that the menu occupies.

[DestroySyntheticPointerDevice](#)

Destroys the specified pointer injection device.

[DestroyWindow](#)

Destroys the specified window.

[DialogBoxA](#)

Creates a modal dialog box from a dialog box template resource. DialogBox does not return control until the specified callback function terminates the modal dialog box by calling the EndDialog function. (ANSI)

[DialogBoxIndirectA](#)

Creates a modal dialog box from a dialog box template in memory. DialogBoxIndirect does not return control until the specified callback function terminates the modal dialog box by calling the EndDialog function. (ANSI)

[DialogBoxIndirectParamA](#)

Creates a modal dialog box from a dialog box template in memory. (ANSI)

[DialogBoxIndirectParamW](#)

Creates a modal dialog box from a dialog box template in memory. (Unicode)

[DialogBoxIndirectW](#)

Creates a modal dialog box from a dialog box template in memory. DialogBoxIndirect does not return control until the specified callback function terminates the modal dialog box by calling the EndDialog function. (Unicode)

[DialogBoxParamA](#)

Creates a modal dialog box from a dialog box template resource. (ANSI)

[DialogBoxParamW](#)

Creates a modal dialog box from a dialog box template resource. (Unicode)

[DialogBoxW](#)

Creates a modal dialog box from a dialog box template resource. DialogBox does not return control until the specified callback function terminates the modal dialog box by calling the EndDialog function. (Unicode)

[DisableProcessWindowsGhosting](#)

Disables the window ghosting feature for the calling GUI process. Window ghosting is a Windows Manager feature that lets the user minimize, move, or close the main window of an application that is not responding.

[DispatchMessage](#)

The DispatchMessage function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

[DispatchMessageA](#)

Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function. (DispatchMessageA)

[DispatchMessageW](#)

The DispatchMessageW (Unicode) function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

[DisplayConfigGetDeviceInfo](#)

The DisplayConfigGetDeviceInfo function retrieves display configuration information about the device.

[DisplayConfigSetDeviceInfo](#)

The DisplayConfigSetDeviceInfo function sets the properties of a target.

[DlgDirListA](#)

Replaces the contents of a list box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list can optionally include mapped drives. (ANSI)

[DlgDirListComboBoxA](#)

Replaces the contents of a combo box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list of names can include mapped drive letters. (ANSI)

[DlgDirListComboBoxW](#)

Replaces the contents of a combo box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list of names can include mapped drive letters. (Unicode)

[DlgDirListW](#)

Replaces the contents of a list box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list can optionally include mapped drives. (Unicode)

[DlgDirSelectComboBoxExA](#)

Retrieves the current selection from a combo box filled by using the DlgDirListComboBox function. The selection is interpreted as a drive letter, a file, or a directory name. (ANSI)

[DlgDirSelectComboBoxExW](#)

Retrieves the current selection from a combo box filled by using the DlgDirListComboBox function. The selection is interpreted as a drive letter, a file, or a directory name. (Unicode)

[DlgDirSelectExA](#)

Retrieves the current selection from a single-selection list box. It assumes that the list box has been filled by the DlgDirList function and that the selection is a drive letter, filename, or directory name. (ANSI)

[DlgDirSelectExW](#)

Retrieves the current selection from a single-selection list box. It assumes that the list box has been filled by the DlgDirList function and that the selection is a drive letter, filename, or directory name. (Unicode)

[DragDetect](#)

Captures the mouse and tracks its movement until the user releases the left button, presses the ESC key, or moves the mouse outside the drag rectangle around the specified point.

[DrawAnimatedRects](#)

Animates the caption of a window to indicate the opening of an icon or the minimizing or maximizing of a window.

[DrawCaption](#)

The DrawCaption function draws a window caption.

[DrawEdge](#)

The DrawEdge function draws one or more edges of rectangle.

[DrawFocusRect](#)

The DrawFocusRect function draws a rectangle in the style used to indicate that the rectangle has the focus.

[DrawFrameControl](#)

The DrawFrameControl function draws a frame control of the specified type and style.

[DrawIcon](#)

Draws an icon or cursor into the specified device context.

[DrawIconEx](#)

Draws an icon or cursor into the specified device context, performing the specified raster operations, and stretching or compressing the icon or cursor as specified.

[DrawMenuBar](#)

Redraws the menu bar of the specified window. If the menu bar changes after the system has created the window, this function must be called to draw the changed menu bar.

[DrawStateA](#)

The DrawState function displays an image and applies a visual effect to indicate a state, such as a disabled or default state. (ANSI)

[DrawStateW](#)

The DrawState function displays an image and applies a visual effect to indicate a state, such as a disabled or default state. (Unicode)

[DrawText](#)

The DrawText function draws formatted text in the specified rectangle. (DrawText function)

[DrawTextA](#)

The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth). (DrawTextA)

[DrawTextExA](#)

The DrawTextEx function draws formatted text in the specified rectangle. (ANSI)

[DrawTextExW](#)

The DrawTextEx function draws formatted text in the specified rectangle. (Unicode)

[DrawTextW](#)

The DrawTextW (Unicode) function draws formatted text in the specified rectangle. (DrawTextW function)

[EmptyClipboard](#)

Empties the clipboard and frees handles to data in the clipboard. The function then assigns ownership of the clipboard to the window that currently has the clipboard open.

[EnableMenuItem](#)

Enables, disables, or grays the specified menu item.

[EnableMouseInPointer](#)

Enables the mouse to act as a pointer input device and send WM_POINTER messages.

[EnableNonClientDpiScaling](#)

In high-DPI displays, enables automatic display scaling of the non-client area portions of the specified top-level window. Must be called during the initialization of that window.

[EnableScrollBar](#)

The EnableScrollBar function enables or disables one or both scroll bar arrows.

[EnableWindow](#)

Enables or disables mouse and keyboard input to the specified window or control. When input is disabled, the window does not receive input such as mouse clicks and key presses. When input is enabled, the window receives all input.

[EndDeferWindowPos](#)

Simultaneously updates the position and size of one or more windows in a single screen-refreshing cycle.

[EndDialog](#)

Destroys a modal dialog box, causing the system to end any processing for the dialog box.

[EndMenu](#)

Ends the calling thread's active menu.

[EndPaint](#)

The EndPaint function marks the end of painting in the specified window. This function is required for each call to the BeginPaint function, but only after painting is complete.

[EndTask](#)

Forcibly closes the specified window.

[EnumChildWindows](#)

Enumerates the child windows that belong to the specified parent window by passing the handle to each child window, in turn, to an application-defined callback function.

[EnumClipboardFormats](#)

Enumerates the data formats currently available on the clipboard.

[EnumDesktopsA](#)

Enumerates all desktops associated with the specified window station of the calling process. The function passes the name of each desktop, in turn, to an application-defined callback function. (ANSI)

[EnumDesktopsW](#)

Enumerates all desktops associated with the specified window station of the calling process. The function passes the name of each desktop, in turn, to an application-defined callback function. (Unicode)

[EnumDesktopWindows](#)

Enumerates all top-level windows associated with the specified desktop. It passes the handle to each window, in turn, to an application-defined callback function.

[EnumDisplayDevicesA](#)

The `EnumDisplayDevices` function lets you obtain information about the display devices in the current session. (ANSI)

[EnumDisplayDevicesW](#)

The `EnumDisplayDevices` function lets you obtain information about the display devices in the current session. (Unicode)

[EnumDisplayMonitors](#)

The `EnumDisplayMonitors` function enumerates display monitors (including invisible pseudo-monitors associated with the mirroring drivers) that intersect a region formed by the intersection of a specified clipping rectangle and the visible region of a device context. `EnumDisplayMonitors` calls an application-defined `MonitorEnumProc` callback function once for each monitor that is enumerated. Note that `GetSystemMetrics` (`SM_CMONITORS`) counts only the display monitors.

[EnumDisplaySettingsA](#)

The `EnumDisplaySettings` function retrieves information about one of the graphics modes for a display device. To retrieve information for all the graphics modes of a display device, make a series of calls to this function. (ANSI)

[EnumDisplaySettingsExA](#)

The `EnumDisplaySettingsEx` function retrieves information about one of the graphics modes for a display device. To retrieve information for all the graphics modes for a display device, make a series of calls to this function. (ANSI)

[EnumDisplaySettingsExW](#)

The `EnumDisplaySettingsEx` function retrieves information about one of the graphics modes for a display device. To retrieve information for all the graphics modes for a display device, make a series of calls to this function. (Unicode)

[EnumDisplaySettingsW](#)

The `EnumDisplaySettings` function retrieves information about one of the graphics modes for a display device. To retrieve information for all the graphics modes of a display device, make a series of calls to this function. (Unicode)

[EnumPropsA](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumProps` continues until the last entry is enumerated or the callback function returns FALSE. (ANSI)

[EnumPropsExA](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumPropsEx` continues until the last entry is enumerated or the callback function returns FALSE. (ANSI)

[EnumPropsExW](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumPropsEx` continues until the last entry is enumerated or the callback function returns FALSE. (Unicode)

[EnumPropsW](#)

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumProps` continues until the last entry is enumerated or the callback function returns FALSE. (Unicode)

EnumThreadWindows
Enumerates all nonchild windows associated with a thread by passing the handle to each window, in turn, to an application-defined callback function.
EnumWindows
Enumerates all top-level windows on the screen by passing the handle to each window, in turn, to an application-defined callback function. EnumWindows continues until the last top-level window is enumerated or the callback function returns FALSE.
EnumWindowStationsA
Enumerates all window stations in the current session. The function passes the name of each window station, in turn, to an application-defined callback function. (ANSI)
EnumWindowStationsW
Enumerates all window stations in the current session. The function passes the name of each window station, in turn, to an application-defined callback function. (Unicode)
EqualRect
The EqualRect function determines whether the two specified rectangles are equal by comparing the coordinates of their upper-left and lower-right corners.
EvaluateProximityToPolygon
Returns the score of a polygon as the probable touch target (compared to all other polygons that intersect the touch contact area) and an adjusted touch point within the polygon.
EvaluateProximityToRect
Returns the score of a rectangle as the probable touch target, compared to all other rectangles that intersect the touch contact area, and an adjusted touch point within the rectangle.
ExcludeUpdateRgn
The ExcludeUpdateRgn function prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.
ExitWindows
Calls the ExitWindowsEx function to log off the interactive user.
ExitWindowsEx
Logs off the interactive user, shuts down the system, or shuts down and restarts the system.

[FillRect](#)

The FillRect function fills a rectangle by using the specified brush. This function includes the left and top borders, but excludes the right and bottom borders of the rectangle.

[FindWindowA](#)

Retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search. (ANSI)

[FindWindowExA](#)

Retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search. (ANSI)

[FindWindowExW](#)

Retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search. (Unicode)

[FindWindowW](#)

Retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search. (Unicode)

[FlashWindow](#)

Flashes the specified window one time. It does not change the active state of the window.

[FlashWindowEx](#)

Flashes the specified window. It does not change the active state of the window.

[FrameRect](#)

The FrameRect function draws a border around the specified rectangle by using the specified brush. The width and height of the border are always one logical unit.

[GET_APPCOMMAND_LPARAM](#)

Retrieves the application command from the specified LPARAM value.

[GET_DEVICE_LPARAM](#)

Retrieves the input device type from the specified LPARAM value.

GET_FLAGS_LPARAM
Retrieves the state of certain virtual keys from the specified LPARAM value. (GET_FLAGS_LPARAM)
GET_KEYSTATE_LPARAM
Retrieves the state of certain virtual keys from the specified LPARAM value. (GET_KEYSTATE_LPARAM)
GET_KEYSTATE_WPARAM
Retrieves the state of certain virtual keys from the specified WPARAM value.
GET_NCHITTEST_WPARAM
Retrieves the hit-test value from the specified WPARAM value.
GET_POINTERID_WPARAM
Retrieves the pointer ID using the specified value.
GET_RAWINPUT_CODE_WPARAM
Retrieves the input code from wParam in WM_INPUT.
GET_WHEEL_DELTA_WPARAM
Retrieves the wheel-delta value from the specified WPARAM value.
GET_XBUTTON_WPARAM
Retrieves the state of certain buttons from the specified WPARAM value.
GetActiveWindow
Retrieves the window handle to the active window attached to the calling thread's message queue.
GetAltTabInfoA
Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window. (ANSI)
GetAltTabInfoW
Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window. (Unicode)

GetAncestor
Retrieves the handle to the ancestor of the specified window.
GetAsyncKeyState
Determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to GetAsyncKeyState.
GetAutoRotationState
Retrieves an AR_STATE value containing the state of screen auto-rotation for the system, for example whether auto-rotation is supported, and whether it is enabled by the user.
GetAwarenessFromDpiAwarenessContext
Retrieves the DPI_AWARENESS value from a DPI_AWARENESS_CONTEXT.
GetCapture
Retrieves a handle to the window (if any) that has captured the mouse. Only one window at a time can capture the mouse; this window receives mouse input whether or not the cursor is within its borders.
GetCaretBlinkTime
Retrieves the time required to invert the caret's pixels. The user can set this value.
GetCaretPos
Copies the caret's position to the specified POINT structure.
GetCIMSSM
Retrieves the source of the input message (GetCurrentInputMessageSourceInSendMessage).
GetClassInfoA
Retrieves information about a window class. (ANSI)
GetClassInfoExA
Retrieves information about a window class, including a handle to the small icon associated with the window class. The GetClassInfo function does not retrieve a handle to the small icon. (ANSI)

[GetClassInfoExW](#)

Retrieves information about a window class, including a handle to the small icon associated with the window class. The GetClassInfo function does not retrieve a handle to the small icon.
(Unicode)

[GetClassInfoW](#)

Retrieves information about a window class. (Unicode)

[GetClassLongA](#)

Retrieves the specified 32-bit (DWORD) value from the WNDCLASSEX structure associated with the specified window. (ANSI)

[GetClassLongPtrA](#)

Retrieves the specified value from the WNDCLASSEX structure associated with the specified window. (ANSI)

[GetClassLongPtrW](#)

Retrieves the specified value from the WNDCLASSEX structure associated with the specified window. (Unicode)

[GetClassLongW](#)

Retrieves the specified 32-bit (DWORD) value from the WNDCLASSEX structure associated with the specified window. (Unicode)

[GetClassName](#)

The GetClassName function retrieves the name of the class to which the specified window belongs. (GetClassName)

[GetClassNameA](#)

Retrieves the name of the class to which the specified window belongs. (GetClassNameA)

[GetClassNameW](#)

The GetClassNameW (Unicode) function retrieves the name of the class to which the specified window belongs. (GetClassNameW)

[GetClassWord](#)

Retrieves the 16-bit (WORD) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

GetClientRect
Retrieves the coordinates of a window's client area.
GetClipboardData
Retrieves data from the clipboard in a specified format. The clipboard must have been opened previously.
GetClipboardFormatNameA
Retrieves from the clipboard the name of the specified registered format. The function copies the name to the specified buffer. (ANSI)
GetClipboardFormatNameW
Retrieves from the clipboard the name of the specified registered format. The function copies the name to the specified buffer. (Unicode)
GetClipboardOwner
Retrieves the window handle of the current owner of the clipboard.
GetClipboardSequenceNumber
Retrieves the clipboard sequence number for the current window station.
GetClipboardViewer
Retrieves the handle to the first window in the clipboard viewer chain.
GetClipCursor
Retrieves the screen coordinates of the rectangular area to which the cursor is confined.
GetComboBoxInfo
Retrieves information about the specified combo box.
GetCurrentInputMessageSource
Retrieves the source of the input message.
GetCursor
Retrieves a handle to the current cursor.

GetCursorInfo
Retrieves information about the global cursor.
GetCursorPos
Retrieves the position of the mouse cursor, in screen coordinates.
GetDC
The GetDC function retrieves a handle to a device context (DC) for the client area of a specified window or for the entire screen.
GetDCEx
The GetDCEx function retrieves a handle to a device context (DC) for the client area of a specified window or for the entire screen.
GetDesktopWindow
Retrieves a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which other windows are painted.
GetDialogBaseUnits
Retrieves the system's dialog base units, which are the average width and height of characters in the system font.
GetDialogControlDpiChangeBehavior
Retrieves and per-monitor DPI scaling behavior overrides of a child window in a dialog.
GetDialogDpiChangeBehavior
Returns the flags that might have been set on a given dialog by an earlier call to SetDialogDpiChangeBehavior.
GetDisplayAutoRotationPreferences
Retrieves the screen auto-rotation preferences for the current process.
GetDisplayAutoRotationPreferencesByProcessId
Retrieves the screen auto-rotation preferences for the process indicated by the dwProcessId parameter.

[GetDisplayConfigBufferSizes](#)

The `GetDisplayConfigBufferSizes` function retrieves the size of the buffers that are required to call the `QueryDisplayConfig` function.

[GetDlgCtrlID](#)

Retrieves the identifier of the specified control.

[GetDlgItem](#)

Retrieves a handle to a control in the specified dialog box.

[GetDlgItemInt](#)

Translates the text of a specified control in a dialog box into an integer value.

[GetDlgItemTextA](#)

Retrieves the title or text associated with a control in a dialog box. (ANSI)

[GetDlgItemTextW](#)

Retrieves the title or text associated with a control in a dialog box. (Unicode)

[GetDoubleClickTime](#)

Retrieves the current double-click time for the mouse.

[GetDpiAwarenessContextForProcess](#)

Gets a `DPI_AWARENESS_CONTEXT` handle for the specified process.

[GetDpiForSystem](#)

Returns the system DPI.

[GetDpiForWindow](#)

Returns the dots per inch (dpi) value for the specified window.

[GetDpiFromDpiAwarenessContext](#)

Retrieves the DPI from a given `DPI_AWARENESS_CONTEXT` handle. This enables you to determine the DPI of a thread without needed to examine a window created within that thread.

GetFocus
Retrieves the handle to the window that has the keyboard focus, if the window is attached to the calling thread's message queue.
GetForegroundWindow
Retrieves a handle to the foreground window (the window with which the user is currently working). The system assigns a slightly higher priority to the thread that creates the foreground window than it does to other threads.
GetGestureConfig
Retrieves the configuration for which Windows Touch gesture messages are sent from a window.
GetGestureExtraArgs
Retrieves additional information about a gesture from its GESTUREINFO handle.
GetGestureInfo
Retrieves a GESTUREINFO structure given a handle to the gesture information.
GetGuiResources
Retrieves the count of handles to graphical user interface (GUI) objects in use by the specified process.
GetGUIThreadInfo
Retrieves information about the active window or a specified GUI thread.
GetIconInfo
Retrieves information about the specified icon or cursor.
GetIconInfoExW
Retrieves information about the specified icon or cursor. GetIconInfoEx extends GetIconInfo by using the newer ICONINFOEX structure. (Unicode)
GetInputState
Determines whether there are mouse-button or keyboard messages in the calling thread's message queue.
GetKBCodePage
Retrieves the current code page.

[GetKeyboardLayout](#)

Retrieves the active input locale identifier (formerly called the keyboard layout).

[GetKeyboardLayoutList](#)

Retrieves the input locale identifiers (formerly called keyboard layout handles) corresponding to the current set of input locales in the system. The function copies the identifiers to the specified buffer.

[GetKeyboardLayoutNameA](#)

Retrieves the name of the active input locale identifier (formerly called the keyboard layout) for the system. (ANSI)

[GetKeyboardLayoutNameW](#)

Retrieves the name of the active input locale identifier (formerly called the keyboard layout) for the system. (Unicode)

[GetKeyboardState](#)

Copies the status of the 256 virtual keys to the specified buffer.

[GetKeyboardType](#)

Retrieves information about the current keyboard.

[GetKeyNameTextA](#)

Retrieves a string that represents the name of a key. (ANSI)

[GetKeyNameTextW](#)

Retrieves a string that represents the name of a key. (Unicode)

[GetKeyState](#)

Retrieves the status of the specified virtual key. The status specifies whether the key is up, down, or toggled (on, off♦alternating each time the key is pressed).

[GetLastActivePopup](#)

Determines which pop-up window owned by the specified window was most recently active.

[GetLastInputInfo](#)

Retrieves the time of the last input event.

[GetLayeredWindowAttributes](#)

Retrieves the opacity and transparency color key of a layered window.

[GetListBoxInfo](#)

Retrieves the number of items per column in a specified list box.

[GetMenu](#)

Retrieves a handle to the menu assigned to the specified window.

[GetMenuBarInfo](#)

Retrieves information about the specified menu bar.

[GetMenuCheckMarkDimensions](#)

Retrieves the dimensions of the default check-mark bitmap.

[GetMenuContextHelpId](#)

Retrieves the Help context identifier associated with the specified menu.

[GetMenuItemDefault](#)

Determines the default menu item on the specified menu.

[GetMenuItemInfo](#)

Retrieves information about a specified menu.

[GetMenuItemCount](#)

Determines the number of items in the specified menu.

[GetMenuItemID](#)

Retrieves the menu item identifier of a menu item located at the specified position in a menu.

[GetMenuItemInfoA](#)

Retrieves information about a menu item. (ANSI)

[GetMenuItemInfoW](#)

Retrieves information about a menu item. (Unicode)

[GetMenuItemRect](#)

Retrieves the bounding rectangle for the specified menu item.

[GetMenuState](#)

Retrieves the menu flags associated with the specified menu item.

[GetMenuItemStringA](#)

Copies the text string of the specified menu item into the specified buffer. (ANSI)

[GetMenuItemStringW](#)

Copies the text string of the specified menu item into the specified buffer. (Unicode)

[GetMessage](#)

The GetMessage function retrieves a message from the calling thread's message queue.
([GetMessage](#))

[GetMessageA](#)

Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval. ([GetMessageA](#))

[GetMessageExtraInfo](#)

Retrieves the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue.

[GetMessagePos](#)

Retrieves the cursor position for the last message retrieved by the GetMessage function.

[GetMessageTime](#)

Retrieves the message time for the last message retrieved by the GetMessage function.

[GetMessageW](#)

The GetMessageW function (Unicode) retrieves a message from the calling thread's message queue. ([GetMessageW](#))

[GetMonitorInfoA](#)

The GetMonitorInfo function retrieves information about a display monitor. (ANSI)

[GetMonitorInfoW](#)

The GetMonitorInfo function retrieves information about a display monitor. (Unicode)

[GetMouseMovePointsEx](#)

Retrieves a history of up to 64 previous coordinates of the mouse or pen.

[GetNextDlgGroupItem](#)

Retrieves a handle to the first control in a group of controls that precedes (or follows) the specified control in a dialog box.

[GetNextDlgTabItem](#)

Retrieves a handle to the first control that has the WS_TABSTOP style that precedes (or follows) the specified control.

[GetNextWindow](#)

Retrieves a handle to the next or previous window in the Z-Order. The next window is below the specified window; the previous window is above.

[GetOpenClipboardWindow](#)

Retrieves the handle to the window that currently has the clipboard open.

[GetParent](#)

Retrieves a handle to the specified window's parent or owner.

[GetPhysicalCursorPos](#)

Retrieves the position of the cursor in physical coordinates.

[GetPointerCursorId](#)

Retrieves the cursor identifier associated with the specified pointer.

[GetPointerDevice](#)

Gets information about the pointer device.

[GetPointerDeviceCursors](#)

Gets the cursor IDs that are mapped to the cursors associated with a pointer device.

[GetPointerDeviceProperties](#)

Gets device properties that aren't included in the `POINTER_DEVICE_INFO` structure.

[GetPointerDeviceRects](#)

Gets the x and y range for the pointer device (in himetric) and the x and y range (current resolution) for the display that the pointer device is mapped to.

[GetPointerDevices](#)

Gets information about the pointer devices attached to the system.

[GetPointerFrameInfo](#)

Gets the entire frame of information for the specified pointers associated with the current message.

[GetPointerFrameInfoHistory](#)

Gets the entire frame of information (including coalesced input frames) for the specified pointers associated with the current message.

[GetPointerFramePenInfo](#)

Gets the entire frame of pen-based information for the specified pointers (of type `PT_PEN`) associated with the current message.

[GetPointerFramePenInfoHistory](#)

Gets the entire frame of pen-based information (including coalesced input frames) for the specified pointers (of type `PT_PEN`) associated with the current message.

[GetPointerFrameTouchInfo](#)

Gets the entire frame of touch-based information for the specified pointers (of type `PT_TOUCH`) associated with the current message.

[GetPointerFrameTouchInfoHistory](#)

Gets the entire frame of touch-based information (including coalesced input frames) for the specified pointers (of type `PT_TOUCH`) associated with the current message.

[GetPointerInfo](#)

Gets the information for the specified pointer associated with the current message.

GetPointerInfoHistory
Gets the information associated with the individual inputs, if any, that were coalesced into the current message for the specified pointer.
GetPointerInputTransform
Gets one or more transforms for the pointer information coordinates associated with the current message.
GetPointerPenInfo
Gets the pen-based information for the specified pointer (of type PT_PEN) associated with the current message.
GetPointerPenInfoHistory
Gets the pen-based information associated with the individual inputs, if any, that were coalesced into the current message for the specified pointer (of type PT_PEN).
GetPointerTouchInfo
Gets the touch-based information for the specified pointer (of type PT_TOUCH) associated with the current message.
GetPointerTouchInfoHistory
Gets the touch-based information associated with the individual inputs, if any, that were coalesced into the current message for the specified pointer (of type PT_TOUCH).
GetPointerType
Retrieves the pointer type for a specified pointer.
GetPriorityClipboardFormat
Retrieves the first available clipboard format in the specified list.
GetProcessDefaultLayout
Retrieves the default layout that is used when windows are created with no parent or owner.
GetProcessWindowStation
Retrieves a handle to the current window station for the calling process.

[GetPropA](#)

Retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the SetProp function. (ANSI)

[GetPropW](#)

Retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the SetProp function. (Unicode)

[GetQueueStatus](#)

Retrieves the type of messages found in the calling thread's message queue.

[GetRawInputBuffer](#)

Performs a buffered read of the raw input data.

[GetRawInputData](#)

Retrieves the raw input from the specified device.

[GetRawInputDeviceInfoA](#)

Retrieves information about the raw input device. (ANSI)

[GetRawInputDeviceInfoW](#)

Retrieves information about the raw input device. (Unicode)

[GetRawInputDeviceList](#)

Enumerates the raw input devices attached to the system.

[GetRawPointerDeviceData](#)

Gets the raw input data from the pointer device.

[GetRegisteredRawInputDevices](#)

Retrieves the information about the raw input devices for the current application.

[GetScrollBarInfo](#)

The GetScrollBarInfo function retrieves information about the specified scroll bar.

[GetScrollInfo](#)

The `GetScrollInfo` function retrieves the parameters of a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (thumb).

[GetScrollPos](#)

The `GetScrollPos` function retrieves the current position of the scroll box (thumb) in the specified scroll bar.

[GetScrollRange](#)

The `GetScrollRange` function retrieves the current minimum and maximum scroll box (thumb) positions for the specified scroll bar.

[GetShellWindow](#)

Retrieves a handle to the Shell's desktop window.

[GetSubMenu](#)

Retrieves a handle to the drop-down menu or submenu activated by the specified menu item.

[GetSysColor](#)

Retrieves the current color of the specified display element.

[GetSysColorBrush](#)

The `GetSysColorBrush` function retrieves a handle identifying a logical brush that corresponds to the specified color index.

[GetSystemDpiForProcess](#)

Retrieves the system DPI associated with a given process. This is useful for avoiding compatibility issues that arise from sharing DPI-sensitive information between multiple system-aware processes with different system DPI values.

[GetSystemMenu](#)

Enables the application to access the window menu (also known as the system menu or the control menu) for copying and modifying.

[GetSystemMetrics](#)

Retrieves the specified system metric or system configuration setting.

GetSystemMetricsForDpi
Retrieves the specified system metric or system configuration setting taking into account a provided DPI.
GetTabbedTextExtentA
The GetTabbedTextExtent function computes the width and height of a character string. (ANSI)
GetTabbedTextExtentW
The GetTabbedTextExtent function computes the width and height of a character string. (Unicode)
GetThreadDesktop
Retrieves a handle to the desktop assigned to the specified thread.
GetThreadDpiAwarenessContext
Gets the DPI_AWARENESS_CONTEXT for the current thread.
GetThreadDpiHostingBehavior
Retrieves the DPI_HOSTING_BEHAVIOR from the current thread.
GetTitleBarInfo
Retrieves information about the specified title bar.
GetTopWindow
Examines the Z order of the child windows associated with the specified parent window and retrieves a handle to the child window at the top of the Z order.
GetTouchInputInfo
Retrieves detailed information about touch inputs associated with a particular touch input handle.
GetUnpredictedMessagePos
Gets pointer data before it has gone through touch prediction processing.
GetUpdatedClipboardFormats
Retrieves the currently supported clipboard formats.

[GetUpdateRect](#)

The GetUpdateRect function retrieves the coordinates of the smallest rectangle that completely encloses the update region of the specified window.

[GetUpdateRgn](#)

The GetUpdateRgn function retrieves the update region of a window by copying it into the specified region. The coordinates of the update region are relative to the upper-left corner of the window (that is, they are client coordinates).

[GetUserObjectInformationA](#)

Retrieves information about the specified window station or desktop object. (ANSI)

[GetUserObjectInformationW](#)

Retrieves information about the specified window station or desktop object. (Unicode)

[GetUserObjectSecurity](#)

Retrieves security information for the specified user object.

[GetWindow](#)

Retrieves a handle to a window that has the specified relationship (Z-Order or owner) to the specified window.

[GetWindowContextHelpId](#)

Retrieves the Help context identifier, if any, associated with the specified window.

[GetWindowDC](#)

The GetWindowDC function retrieves the device context (DC) for the entire window, including title bar, menus, and scroll bars.

[GetWindowDisplayAffinity](#)

Retrieves the current display affinity setting, from any process, for a given window.

[GetWindowDpiAwarenessContext](#)

Returns the DPI_AWARENESS_CONTEXT associated with a window.

[GetWindowDpiHostingBehavior](#)

Returns the DPI_HOSTING_BEHAVIOR of the specified window.

[GetWindowFeedbackSetting](#)

Retrieves the feedback configuration for a window.

[GetWindowInfo](#)

Retrieves information about the specified window. (GetWindowInfo)

[GetWindowLongA](#)

Retrieves information about the specified window. (GetWindowLongA)

[GetWindowLongPtrA](#)

Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory. (ANSI)

[GetWindowLongPtrW](#)

Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory. (Unicode)

[GetWindowLongW](#)

Retrieves information about the specified window. (GetWindowLongW)

[GetWindowModuleFileNameA](#)

Retrieves the full path and file name of the module associated with the specified window handle. (ANSI)

[GetWindowModuleFileNameW](#)

Retrieves the full path and file name of the module associated with the specified window handle. (Unicode)

[GetWindowPlacement](#)

Retrieves the show state and the restored, minimized, and maximized positions of the specified window.

[GetWindowRect](#)

Retrieves the dimensions of the bounding rectangle of the specified window. The dimensions are given in screen coordinates that are relative to the upper-left corner of the screen.

[GetWindowRgn](#)

The GetWindowRgn function obtains a copy of the window region of a window.

[GetWindowRgnBox](#)

The GetWindowRgnBox function retrieves the dimensions of the tightest bounding rectangle for the window region of a window.

[GetWindowTextA](#)

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application. (ANSI)

[GetWindowTextLengthA](#)

Retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). (ANSI)

[GetWindowTextLengthW](#)

Retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). (Unicode)

[GetWindowTextW](#)

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application. (Unicode)

[GetWindowThreadProcessId](#)

Retrieves the identifier of the thread that created the specified window and, optionally, the identifier of the process that created the window.

[GetWindowWord](#)

Retrieves the 16-bit (DWORD) value at the specified offset into the extra window memor

[GID_ROTATE_ANGLE_FROM_ARGUMENT](#)

The GID_ROTATE_ANGLE_FROM_ARGUMENT macro is used to interpret the GID_ROTATE ullArgument value when receiving the value in the WM_GESTURE structure.

[GID_ROTATE_ANGLE_TO_ARGUMENT](#)

Converts a radian value to an argument for rotation gesture messages.

[GrayStringA](#)

The GrayString function draws gray text at the specified location. (ANSI)

[GrayStringW](#)

The GrayString function draws gray text at the specified location. (Unicode)

[HAS_POINTER_CONFIDENCE_WPARAM](#)

Checks whether the specified pointer message is considered intentional rather than accidental.

[HideCaret](#)

Removes the caret from the screen. Hiding a caret does not destroy its current shape or invalidate the insertion point.

[HiliteMenuItem](#)

Adds or removes highlighting from an item in a menu bar.

[InflateRect](#)

The InflateRect function increases or decreases the width and height of the specified rectangle.

[InheritWindowMonitor](#)

Causes a specified window to inherit the monitor of another window.

[InitializeTouchInjection](#)

Configures the touch injection context for the calling application and initializes the maximum number of simultaneous contacts that the app can inject.

[InjectSyntheticPointerInput](#)

Simulates pointer input (pen or touch).

[InjectTouchInput](#)

Simulates touch input.

[InSendMessage](#)

Determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process) by a call to the SendMessage function.

[InSendMessageEx](#)

Determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process).

[InsertMenuA](#)

Inserts a new menu item into a menu, moving other items down the menu. (ANSI)

[InsertMenuItemA](#)

Inserts a new menu item at the specified position in a menu. (ANSI)

[InsertMenuItemW](#)

Inserts a new menu item at the specified position in a menu. (Unicode)

[InsertMenuW](#)

Inserts a new menu item into a menu, moving other items down the menu. (Unicode)

[InternalGetWindowText](#)

Copies the text of the specified window's title bar (if it has one) into a buffer.

[IntersectRect](#)

The IntersectRect function calculates the intersection of two source rectangles and places the coordinates of the intersection rectangle into the destination rectangle.

[InvalidateRect](#)

The InvalidateRect function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn.

[InvalidateRgn](#)

The InvalidateRgn function invalidates the client area within the specified region by adding it to the current update region of a window.

[InvertRect](#)

The InvertRect function inverts a rectangle in a window by performing a logical NOT operation on the color values for each pixel in the rectangle's interior.

[IS_INTRESOURCE](#)

Determines whether a value is an integer identifier for a resource.

[IS_POINTER_CANCELED_WPARAM](#)

Checks whether the specified pointer input ended abruptly, or was invalid, indicating the interaction was not completed.

[IS_POINTER_FIFTHBUTTON_WPARAM](#)

Checks whether the specified pointer took fifth action.

[IS_POINTER_FIRSTBUTTON_WPARAM](#)

Checks whether the specified pointer took first action.

[IS_POINTER_FLAG_SET_WPARAM](#)

Checks whether a pointer macro sets the specified flag.

[IS_POINTER_FOURTHBUTTON_WPARAM](#)

Checks whether the specified pointer took fourth action.

[IS_POINTER_INCONTACT_WPARAM](#)

Checks whether the specified pointer is in contact.

[IS_POINTER_INRANGE_WPARAM](#)

Checks whether the specified pointer is in range.

[IS_POINTER_NEW_WPARAM](#)

Checks whether the specified pointer is a new pointer.

[IS_POINTER_SECONDBUTTON_WPARAM](#)

Checks whether the specified pointer took second action.

[IS_POINTER_THIRDBUTTON_WPARAM](#)

Checks whether the specified pointer took third action.

[IsCharAlphaA](#)

Determines whether a character is an alphabetical character. This determination is based on the semantics of the language selected by the user during setup or through Control Panel. (ANSI)

[IsCharAlphaNumerica](#)

Determines whether a character is either an alphabetical or a numeric character. This determination is based on the semantics of the language selected by the user during setup or through Control Panel. (ANSI)

[IsCharAlphaNumericW](#)

Determines whether a character is either an alphabetical or a numeric character. This determination is based on the semantics of the language selected by the user during setup or through Control Panel. (Unicode)

[IsCharAlphaW](#)

Determines whether a character is an alphabetical character. This determination is based on the semantics of the language selected by the user during setup or through Control Panel. (Unicode)

[IsCharLowerA](#)

Determines whether a character is lowercase. This determination is based on the semantics of the language selected by the user during setup or through Control Panel.

[IsCharLowerW](#)

The IsCharLowerW (Unicode) function determines whether a character is lowercase. (IsCharLowerW)

[IsCharUpperA](#)

Determines whether a character is uppercase. This determination is based on the semantics of the language selected by the user during setup or through Control Panel. (ANSI)

[IsCharUpperW](#)

Determines whether a character is uppercase. This determination is based on the semantics of the language selected by the user during setup or through Control Panel. (Unicode)

[IsChild](#)

Determines whether a window is a child window or descendant window of a specified parent window.

[IsClipboardFormatAvailable](#)

Determines whether the clipboard contains data in the specified format.

[IsDialogMessageA](#)

Determines whether a message is intended for the specified dialog box and, if it is, processes the message. (ANSI)

[IsDialogMessageW](#)

Determines whether a message is intended for the specified dialog box and, if it is, processes the message. (Unicode)

[IsDlgButtonChecked](#)

The `IsDlgButtonChecked` function determines whether a button control is checked or whether a three-state button control is checked, unchecked, or indeterminate.

[IsGUIThread](#)

Determines whether the calling thread is already a GUI thread. It can also optionally convert the thread to a GUI thread.

[IsHungAppWindow](#)

Determines whether the system considers that a specified application is not responding.

[IsIconic](#)

Determines whether the specified window is minimized (iconic).

[IsImmersiveProcess](#)

Determines whether the process belongs to a Windows Store app.

[IsMenu](#)

Determines whether a handle is a menu handle.

[IsMouseInPointerEnabled](#)

Indicates whether `EnableMouseInPointer` is set for the mouse to act as a pointer input device and send WM_POINTER messages.

[IsProcessDPIAware](#)

`IsProcessDPIAware` may be altered or unavailable. Instead, use `GetProcessDPIAwareness`.

[IsRectEmpty](#)

The `IsRectEmpty` function determines whether the specified rectangle is empty.

[IsTouchWindow](#)

Checks whether a specified window is touch-capable and, optionally, retrieves the modifier flags set for the window's touch capability.

[IsValidDpiAwarenessContext](#)

Determines if a specified `DPI_AWARENESS_CONTEXT` is valid and supported by the current system.

IsWindow
Determines whether the specified window handle identifies an existing window.
IsWindowArranged
Determines whether the specified window is arranged (that is, whether it's snapped).
IsWindowEnabled
Determines whether the specified window is enabled for mouse and keyboard input.
IsWindowUnicode
Determines whether the specified window is a native Unicode window.
IsWindowVisible
Determines the visibility state of the specified window.
IsWinEventHookInstalled
Determines whether there is an installed WinEvent hook that might be notified of a specified event.
IsWow64Message
Determines whether the last message read from the current thread's queue originated from a WOW64 process.
IsZoomed
Determines whether a window is maximized.
keybd_event
Synthesizes a keystroke.
KillTimer
Destroys the specified timer.
LoadAcceleratorsA
Loads the specified accelerator table. (ANSI)
LoadAcceleratorsW
Loads the specified accelerator table. (Unicode)

[LoadBitmapA](#)

The LoadBitmap function loads the specified bitmap resource from a module's executable file. (ANSI)

[LoadBitmapW](#)

The LoadBitmap function loads the specified bitmap resource from a module's executable file. (Unicode)

[LoadCursorA](#)

Loads the specified cursor resource from the executable (.EXE) file associated with an application instance. (ANSI)

[LoadCursorFromFileA](#)

Creates a cursor based on data contained in a file. (ANSI)

[LoadCursorFromFileW](#)

Creates a cursor based on data contained in a file. (Unicode)

[LoadCursorW](#)

Loads the specified cursor resource from the executable (.EXE) file associated with an application instance. (Unicode)

[LoadIconA](#)

Loads the specified icon resource from the executable (.exe) file associated with an application instance. (ANSI)

[LoadIconW](#)

Loads the specified icon resource from the executable (.exe) file associated with an application instance. (Unicode)

[LoadImageA](#)

Loads an icon, cursor, animated cursor, or bitmap. (ANSI)

[LoadImageW](#)

Loads an icon, cursor, animated cursor, or bitmap. (Unicode)

[LoadKeyboardLayoutA](#)

Loads a new input locale identifier (formerly called the keyboard layout) into the system. (ANSI)

[LoadKeyboardLayoutW](#)

Loads a new input locale identifier (formerly called the keyboard layout) into the system. (Unicode)

[LoadMenuA](#)

Loads the specified menu resource from the executable (.exe) file associated with an application instance. (ANSI)

[LoadMenuIndirectA](#)

Loads the specified menu template in memory. (ANSI)

[LoadMenuIndirectW](#)

Loads the specified menu template in memory. (Unicode)

[LoadMenuW](#)

Loads the specified menu resource from the executable (.exe) file associated with an application instance. (Unicode)

[LoadStringA](#)

Loads a string resource from the executable file associated with a specified module, copies the string into a buffer, and appends a terminating null character. (ANSI)

[LoadStringW](#)

Loads a string resource from the executable file associated with a specified module, copies the string into a buffer, and appends a terminating null character. (Unicode)

[LockSetForegroundWindow](#)

The foreground process can call the LockSetForegroundWindow function to disable calls to the SetForegroundWindow function.

[LockWindowUpdate](#)

The LockWindowUpdate function disables or enables drawing in the specified window. Only one window can be locked at a time.

[LockWorkStation](#)

Locks the workstation's display.

[LogicalToPhysicalPoint](#)

Converts the logical coordinates of a point in a window to physical coordinates.

[LogicalToPhysicalPointForPerMonitorDPI](#)

Converts a point in a window from logical coordinates into physical coordinates, regardless of the dots per inch (dpi) awareness of the caller.

[LookupIconIdFromDirectory](#)

Searches through icon or cursor data for the icon or cursor that best fits the current display device. (LookupIconIdFromDirectory)

[LookupIconIdFromDirectoryEx](#)

Searches through icon or cursor data for the icon or cursor that best fits the current display device. (LookupIconIdFromDirectoryEx)

[MAKEINTRESOURCEA](#)

Converts an integer value to a resource type compatible with the resource-management functions. This macro is used in place of a string containing the name of the resource. (ANSI)

[MAKEINTRESOURCEW](#)

Converts an integer value to a resource type compatible with the resource-management functions. This macro is used in place of a string containing the name of the resource. (Unicode)

[MAKELPARAM](#)

Creates a value for use as an lParam parameter in a message. The macro concatenates the specified values.

[MAKELRESULT](#)

Creates a value for use as a return value from a window procedure. The macro concatenates the specified values.

[MAKEWPARAM](#)

Creates a value for use as a wParam parameter in a message. The macro concatenates the specified values.

[MapDialogRect](#)

Converts the specified dialog box units to screen units (pixels).

[MapVirtualKeyA](#)

Translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. (ANSI)

[MapVirtualKeyExA](#)

Translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. The function translates the codes using the input language and an input locale identifier. (ANSI)

[MapVirtualKeyExW](#)

Translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. The function translates the codes using the input language and an input locale identifier. (Unicode)

[MapVirtualKeyW](#)

Translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. (Unicode)

[MapWindowPoints](#)

The MapWindowPoints function converts (maps) a set of points from a coordinate space relative to one window to a coordinate space relative to another window.

[MenuItemFromPoint](#)

Determines which menu item, if any, is at the specified location.

[MessageBeep](#)

Plays a waveform sound. The waveform sound for each sound type is identified by an entry in the registry.

[MessageBox](#)

The MessageBox function displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message.

[MessageBoxA](#)

Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked. (MessageBoxA)

[MessageBoxExA](#)

Creates, displays, and operates a message box. (ANSI)

[MessageBoxExW](#)

Creates, displays, and operates a message box. (Unicode)

[MessageBoxIndirectA](#)

Creates, displays, and operates a message box. The message box contains application-defined message text and title, any icon, and any combination of predefined push buttons. (ANSI)

[MessageBoxIndirectW](#)

Creates, displays, and operates a message box. The message box contains application-defined message text and title, any icon, and any combination of predefined push buttons. (Unicode)

[MessageBoxW](#)

The MessageBoxW (Unicode) function displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message.

[ModifyMenuA](#)

Changes an existing menu item. (ANSI)

[ModifyMenuW](#)

Changes an existing menu item. (Unicode)

[MonitorFromPoint](#)

The MonitorFromPoint function retrieves a handle to the display monitor that contains a specified point.

[MonitorFromRect](#)

The MonitorFromRect function retrieves a handle to the display monitor that has the largest area of intersection with a specified rectangle.

[MonitorFromWindow](#)

The MonitorFromWindow function retrieves a handle to the display monitor that has the largest area of intersection with the bounding rectangle of a specified window.

[mouse_event](#)

The mouse_event function synthesizes mouse motion and button clicks.

[MoveWindow](#)

Changes the position and dimensions of the specified window.

[MsgWaitForMultipleObjects](#)

Waits until one or all of the specified objects are in the signaled state or the time-out interval elapses. The objects can include input event objects.

[MsgWaitForMultipleObjectsEx](#)

Waits until one or all of the specified objects are in the signaled state, an I/O completion routine or asynchronous procedure call (APC) is queued to the thread, or the time-out interval elapses. The array of objects can include input event objects.

[NEXTRAWINPUTBLOCK](#)

Retrieves the location of the next structure in an array of RAWINPUT structures.

[NotifyWinEvent](#)

Signals the system that a predefined event occurred. If any client applications have registered a hook function for the event, the system calls the client's hook function.

[OemKeyScan](#)

Maps OEMASCII codes 0 through 0xFF into the OEM scan codes and shift states. The function provides information that allows a program to send OEM text to another program by simulating keyboard input.

[OemToCharA](#)

Translates a string from the OEM-defined character set into either an ANSI or a wide-character string.**Warning** Do not use. (ANSI)

[OemToCharBuffA](#)

Translates a specified number of characters in a string from the OEM-defined character set into either an ANSI or a wide-character string. (ANSI)

[OemToCharBuffW](#)

Translates a specified number of characters in a string from the OEM-defined character set into either an ANSI or a wide-character string. (Unicode)

[OemToCharW](#)

Translates a string from the OEM-defined character set into either an ANSI or a wide-character string.**Warning** Do not use. (Unicode)

<p>OffsetRect</p> <p>The OffsetRect function moves the specified rectangle by the specified offsets.</p>
<p>OpenClipboard</p> <p>Opens the clipboard for examination and prevents other applications from modifying the clipboard content.</p>
<p>OpenDesktopA</p> <p>Opens the specified desktop object. (ANSI)</p>
<p>OpenDesktopW</p> <p>Opens the specified desktop object. (Unicode)</p>
<p>OpenIcon</p> <p>Restores a minimized (iconic) window to its previous size and position; it then activates the window.</p>
<p>OpenInputDesktop</p> <p>Opens the desktop that receives user input.</p>
<p>OpenWindowStationA</p> <p>Opens the specified window station. (ANSI)</p>
<p>OpenWindowStationW</p> <p>Opens the specified window station. (Unicode)</p>
<p>PackTouchHitTestingProximityEvaluation</p> <p>Returns the proximity evaluation score and the adjusted touch-point coordinates as a packed value for the WM_TOUCHHITTESTING callback.</p>
<p>PaintDesktop</p> <p>The PaintDesktop function fills the clipping region in the specified device context with the desktop pattern or wallpaper. The function is provided primarily for shell desktops.</p>
<p>PeekMessageA</p> <p>Dispatches incoming nonqueued messages, checks the thread message queue for a posted message, and retrieves the message (if any exist). (ANSI)</p>

[PeekMessageW](#)

Dispatches incoming nonqueued messages, checks the thread message queue for a posted message, and retrieves the message (if any exist). (Unicode)

[PhysicalToLogicalPoint](#)

Converts the physical coordinates of a point in a window to logical coordinates.

[PhysicalToLogicalPointForPerMonitorDPI](#)

Converts a point in a window from physical coordinates into logical coordinates, regardless of the dots per inch (dpi) awareness of the caller.

[POINTSTOPOINT](#)

The POINTSTOPOINT macro copies the contents of a POINTS structure into a POINT structure.

[POINTTOPOINTS](#)

The POINTTOPOINTS macro converts a POINT structure to a POINTS structure.

[PostMessageA](#)

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message. (ANSI)

[PostMessageW](#)

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message. (Unicode)

[PostQuitMessage](#)

Indicates to the system that a thread has made a request to terminate (quit). It is typically used in response to a WM_DESTROY message.

[PostThreadMessageA](#)

Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message. (ANSI)

[PostThreadMessageW](#)

Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message. (Unicode)

[PrintWindow](#)

The PrintWindow function copies a visual window into the specified device context (DC), typically a printer DC.

[PrivateExtractIconsA](#)

Creates an array of handles to icons that are extracted from a specified file. (ANSI)

[PrivateExtractIconsW](#)

Creates an array of handles to icons that are extracted from a specified file. (Unicode)

[PtInRect](#)

The PtInRect function determines whether the specified point lies within the specified rectangle.

[QueryDisplayConfig](#)

The QueryDisplayConfig function retrieves information about all possible display paths for all display devices, or views, in the current setting.

[RealChildWindowFromPoint](#)

Retrieves a handle to the child window at the specified point. The search is restricted to immediate child windows; grandchildren and deeper descendant windows are not searched.

[RealGetWindowClassA](#)

Retrieves a string that specifies the window type. (ANSI)

[RealGetWindowClassW](#)

Retrieves a string that specifies the window type. (Unicode)

[RedrawWindow](#)

The RedrawWindow function updates the specified rectangle or region in a window's client area.

[RegisterClassA](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassA)

[RegisterClassExA](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassExA)

[RegisterClassExW](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassExW)

[RegisterClassW](#)

Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function. (RegisterClassW)

[RegisterClipboardFormatA](#)

Registers a new clipboard format. This format can then be used as a valid clipboard format. (ANSI)

[RegisterClipboardFormatW](#)

Registers a new clipboard format. This format can then be used as a valid clipboard format. (Unicode)

[RegisterDeviceNotificationA](#)

Registers the device or type of device for which a window will receive notifications. (ANSI)

[RegisterDeviceNotificationW](#)

Registers the device or type of device for which a window will receive notifications. (Unicode)

[RegisterForTooltipDismissNotification](#)

Lets apps or UI frameworks register and unregister windows to receive notification to dismiss their tooltip windows.

[RegisterHotKey](#)

Defines a system-wide hot key.

[RegisterPointerDeviceNotifications](#)

Registers a window to process the WM_POINTERDEVICECHANGE, WM_POINTERDEVICEINRANGE, and WM_POINTERDEVICEOUTOFRANGE pointer device notifications.

[RegisterPointerInputTarget](#)

Allows the caller to register a target window to which all pointer input of the specified type is redirected.

[RegisterPointerInputTargetEx](#)

RegisterPointerInputTargetEx may be altered or unavailable. Instead, use RegisterPointerInputTarget.

[RegisterPowerSettingNotification](#)

Registers the application to receive power setting notifications for the specific power setting event.

[RegisterRawInputDevices](#)

Registers the devices that supply the raw input data.

[RegisterShellHookWindow](#)

Registers a specified Shell window to receive certain messages for events or notifications that are useful to Shell applications.

[RegisterSuspendResumeNotification](#)

Registers to receive notification when the system is suspended or resumed. Similar to PowerRegisterSuspendResumeNotification, but operates in user mode and can take a window handle.

[RegisterTouchHitTestingWindow](#)

Registers a window to process the WM_TOUCHHITTESTING notification.

[RegisterTouchWindow](#)

Registers a window as being touch-capable.

[RegisterWindowMessageA](#)

Defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages. (ANSI)

[RegisterWindowMessageW](#)

Defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages. (Unicode)

[ReleaseCapture](#)

Releases the mouse capture from a window in the current thread and restores normal mouse input processing.

[ReleaseDC](#)

The ReleaseDC function releases a device context (DC), freeing it for use by other applications. The effect of the ReleaseDC function depends on the type of DC. It frees only common and window DCs. It has no effect on class or private DCs.

[RemoveClipboardFormatListener](#)

Removes the given window from the system-maintained clipboard format listener list.

[RemoveMenu](#)

Deletes a menu item or detaches a submenu from the specified menu.

[RemovePropA](#)

Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed. (ANSI)

[RemovePropW](#)

Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed. (Unicode)

[ReplyMessage](#)

Replies to a message sent from another thread by the SendMessage function.

[ScreenToClient](#)

The ScreenToClient function converts the screen coordinates of a specified point on the screen to client-area coordinates.

[ScrollDC](#)

The ScrollDC function scrolls a rectangle of bits horizontally and vertically.

[ScrollWindow](#)

The ScrollWindow function scrolls the contents of the specified window's client area.

[ScrollWindowEx](#)

The ScrollWindowEx function scrolls the contents of the specified window's client area.

[SendDlgItemMessageA](#)

Sends a message to the specified control in a dialog box. (ANSI)

[SendDlgItemMessageW](#)

Sends a message to the specified control in a dialog box. (Unicode)

[SendInput](#)

Synthesizes keystrokes, mouse motions, and button clicks.

[SendMessage](#)

The SendMessage function sends the specified message to a window or windows. (SendMessage function)

[SendMessageA](#)

Sends the specified message to a window or windows. The SendMessage function calls the window procedure for the specified window and does not return until the window procedure has processed the message. (SendMessageA)

[SendMessageCallbackA](#)

Sends the specified message to a window or windows. (SendMessageCallbackA)

[SendMessageCallbackW](#)

Sends the specified message to a window or windows. (SendMessageCallbackW)

[SendMessageTimeoutA](#)

Sends the specified message to one or more windows. (ANSI)

[SendMessageTimeoutW](#)

Sends the specified message to one or more windows. (Unicode)

[SendMessageW](#)

The SendMessageW (Unicode) function sends the specified message to a window or windows. (SendMessageW)

[SendNotifyMessageA](#)

Sends the specified message to a window or windows. (SendNotifyMessageA)

[SendNotifyMessageW](#)

Sends the specified message to a window or windows. (SendNotifyMessageW)

[SetActiveWindow](#)

Activates a window. The window must be attached to the calling thread's message queue.

[SetAdditionalForegroundBoostProcesses](#)

`SetAdditionalForegroundBoostProcesses` is a performance assist API to help applications with a multi-process application model where multiple processes contribute to a foreground experience, either as data or rendering.

[SetCapture](#)

Sets the mouse capture to the specified window belonging to the current thread.

[SetCaretBlinkTime](#)

Sets the caret blink time to the specified number of milliseconds. The blink time is the elapsed time, in milliseconds, required to invert the caret's pixels.

[SetCaretPos](#)

Moves the caret to the specified coordinates. If the window that owns the caret was created with the CS_OWNDC class style, then the specified coordinates are subject to the mapping mode of the device context associated with that window.

[SetClassLongA](#)

Replaces the specified 32-bit (long) value at the specified offset into the extra class memory or the WNDCLASSEX structure for the class to which the specified window belongs. (ANSI)

[SetClassLongPtrA](#)

Replaces the specified value at the specified offset in the extra class memory or the WNDCLASSEX structure for the class to which the specified window belongs. (ANSI)

[SetClassLongPtrW](#)

Replaces the specified value at the specified offset in the extra class memory or the WNDCLASSEX structure for the class to which the specified window belongs. (Unicode)

[SetClassLongW](#)

Replaces the specified 32-bit (long) value at the specified offset into the extra class memory or the WNDCLASSEX structure for the class to which the specified window belongs. (Unicode)

[SetClassWord](#)

Replaces the 16-bit (WORD) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

[SetClipboardData](#)

Places data on the clipboard in a specified clipboard format.

[SetClipboardViewer](#)

Adds the specified window to the chain of clipboard viewers. Clipboard viewer windows receive a WM_DRAWCLIPBOARD message whenever the content of the clipboard changes. This function is used for backward compatibility with earlier versions of Windows.

[SetCoalescableTimer](#)

Creates a timer with the specified time-out value and coalescing tolerance delay.

[SetCursor](#)

Sets the cursor shape.

[SetCursorPos](#)

Moves the cursor to the specified screen coordinates.

[SetDialogControlDpiChangeBehavior](#)

Overrides the default per-monitor DPI scaling behavior of a child window in a dialog.

[SetDialogDpiChangeBehavior](#)

Dialogs in Per-Monitor v2 contexts are automatically DPI scaled. This method lets you customize their DPI change behavior.

[SetDisplayAutoRotationPreferences](#)

Sets the screen auto-rotation preferences for the current process.

[SetDisplayConfig](#)

The SetDisplayConfig function modifies the display topology, source, and target modes by exclusively enabling the specified paths in the current session.

[SetDlgItemInt](#)

Sets the text of a control in a dialog box to the string representation of a specified integer value.

[SetDlgItemTextA](#)

Sets the title or text of a control in a dialog box. (ANSI)

[SetDlgItemTextW](#)

Sets the title or text of a control in a dialog box. (Unicode)

[SetDoubleClickTime](#)

Sets the double-click time for the mouse.

[SetFocus](#)

Sets the keyboard focus to the specified window. The window must be attached to the calling thread's message queue.

[SetForegroundWindow](#)

Brings the thread that created the specified window into the foreground and activates the window.

[SetGestureConfig](#)

Configures the messages that are sent from a window for Windows Touch gestures.

[SetKeyboardState](#)

Copies an array of keyboard key states into the calling thread's keyboard input-state table. This is the same table accessed by the GetKeyboardState and GetKeyState functions. Changes made to this table do not affect keyboard input to any other thread.

[SetLastErrorEx](#)

Sets the last-error code.

[SetLayeredWindowAttributes](#)

Sets the opacity and transparency color key of a layered window.

[SetMenu](#)

Assigns a new menu to the specified window.

[SetMenuItemContextHelpId](#)

Associates a Help context identifier with a menu.

[SetMenuItemDefault](#)

Sets the default menu item for the specified menu.

[SetMenuItemInfo](#)

Sets information for a specified menu.

[SetMenuItemBitmaps](#)

Associates the specified bitmap with a menu item. Whether the menu item is selected or clear, the system displays the appropriate bitmap next to the menu item.

[SetMenuItemInfoA](#)

Changes information about a menu item. (ANSI)

[SetMenuItemInfoW](#)

Changes information about a menu item. (Unicode)

[SetMessageExtraInfo](#)

Sets the extra message information for the current thread.

[SetParent](#)

Changes the parent window of the specified child window.

[SetPhysicalCursorPos](#)

Sets the position of the cursor in physical coordinates.

[SetProcessDefaultLayout](#)

Changes the default layout when windows are created with no parent or owner only for the currently running process.

[SetProcessDPIAware](#)

`SetProcessDPIAware` may be altered or unavailable. Instead, use `SetProcessDPIAwareness`.

[SetProcessDpiAwarenessContext](#)

Sets the current process to a specified dots per inch (dpi) awareness context. The DPI awareness contexts are from the `DPI_AWARENESS_CONTEXT` value.

[SetProcessRestrictionExemption](#)

Exempts the calling process from restrictions preventing desktop processes from interacting with the Windows Store app environment. This function is used by development and debugging tools.

[SetProcessWindowStation](#)

Assigns the specified window station to the calling process.

[SetPropA](#)

Adds a new entry or changes an existing entry in the property list of the specified window. (ANSI)

[SetPropW](#)

Adds a new entry or changes an existing entry in the property list of the specified window. (Unicode)

[SetRect](#)

The SetRect function sets the coordinates of the specified rectangle. This is equivalent to assigning the left, top, right, and bottom arguments to the appropriate members of the RECT structure.

[SetRectEmpty](#)

The SetRectEmpty function creates an empty rectangle in which all coordinates are set to zero.

[SetScrollInfo](#)

The SetScrollInfo function sets the parameters of a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (thumb). The function also redraws the scroll bar, if requested.

[SetScrollPos](#)

The SetScrollPos function sets the position of the scroll box (thumb) in the specified scroll bar and, if requested, redraws the scroll bar to reflect the new position of the scroll box.

[SetScrollRange](#)

The SetScrollRange function sets the minimum and maximum scroll box positions for the specified scroll bar.

[SetSysColors](#)

Sets the colors for the specified display elements.

[SetSystemCursor](#)

Enables an application to customize the system cursors. It replaces the contents of the system cursor specified by the id parameter with the contents of the cursor specified by the hcur parameter and then destroys hcur.

SetThreadCursorCreationScaling
Sets the DPI scale for which the cursors being created on this thread are intended. This value is taken into account when scaling the cursor for the specific monitor on which it is being shown.
SetThreadDesktop
Assigns the specified desktop to the calling thread. All subsequent operations on the desktop use the access rights granted to the desktop.
SetThreadDpiAwarenessContext
Set the DPI awareness for the current thread to the provided value.
SetThreadDpiHostingBehavior
Sets the thread's DPI_HOSTING_BEHAVIOR. This behavior allows windows created in the thread to host child windows with a different DPI_AWARENESS_CONTEXT.
SetTimer
Creates a timer with the specified time-out value.
SetUserObjectInformationA
Sets information about the specified window station or desktop object. (ANSI)
SetUserObjectInformationW
Sets information about the specified window station or desktop object. (Unicode)
SetUserObjectSecurity
Sets the security of a user object. This can be, for example, a window or a DDE conversation.
SetWindowContextHelpId
Associates a Help context identifier with the specified window.
SetWindowDisplayAffinity
Stores the display affinity setting in kernel mode on the hWnd associated with the window.
SetWindowFeedbackSetting
Sets the feedback configuration for a window.

[SetWindowLongA](#)

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory. (ANSI)

[SetWindowLongPtrA](#)

Changes an attribute of the specified window. (ANSI)

[SetWindowLongPtrW](#)

Changes an attribute of the specified window. (Unicode)

[SetWindowLongW](#)

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory. (Unicode)

[SetWindowPlacement](#)

Sets the show state and the restored, minimized, and maximized positions of the specified window.

[SetWindowPos](#)

Changes the size, position, and Z order of a child, pop-up, or top-level window. These windows are ordered according to their appearance on the screen. The topmost window receives the highest rank and is the first window in the Z order.

[SetWindowRgn](#)

The SetWindowRgn function sets the window region of a window.

[SetWindowsHookExA](#)

Installs an application-defined hook procedure into a hook chain. (ANSI)

[SetWindowsHookExW](#)

Installs an application-defined hook procedure into a hook chain. (Unicode)

[SetWindowTextA](#)

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application. (ANSI)

[SetWindowTextW](#)

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application. (Unicode)

[SetWinEventHook](#)

Sets an event hook function for a range of events.

[ShowCaret](#)

Makes the caret visible on the screen at the caret's current position. When the caret becomes visible, it begins flashing automatically.

[ShowCursor](#)

Displays or hides the cursor. (ShowCursor)

[ShowOwnedPopups](#)

Shows or hides all pop-up windows owned by the specified window.

[ShowScrollBar](#)

The ShowScrollBar function shows or hides the specified scroll bar.

[ShowWindow](#)

Sets the specified window's show state.

[ShowWindowAsync](#)

Sets the show state of a window without waiting for the operation to complete.

[ShutdownBlockReasonCreate](#)

Indicates that the system cannot be shut down and sets a reason string to be displayed to the user if system shutdown is initiated.

[ShutdownBlockReasonDestroy](#)

Indicates that the system can be shut down and frees the reason string.

[ShutdownBlockReasonQuery](#)

Retrieves the reason string set by the ShutdownBlockReasonCreate function.

[SkipPointerFrameMessages](#)

Determines which pointer input frame generated the most recently retrieved message for the specified pointer and discards any queued (unretrieved) pointer input messages generated from the same pointer input frame.

[SoundSentry](#)

Triggers a visual signal to indicate that a sound is playing.

[SubtractRect](#)

The SubtractRect function determines the coordinates of a rectangle formed by subtracting one rectangle from another.

[SwapMouseButton](#)

Reverses or restores the meaning of the left and right mouse buttons.

[SwitchDesktop](#)

Makes the specified desktop visible and activates it. This enables the desktop to receive input from the user.

[SwitchToThisWindow](#)

Switches focus to the specified window and brings it to the foreground.

[SystemParametersInfoA](#)

Retrieves or sets the value of one of the system-wide parameters. (ANSI)

[SystemParametersInfoForDpi](#)

Retrieves the value of one of the system-wide parameters, taking into account the provided DPI value.

[SystemParametersInfoW](#)

Retrieves or sets the value of one of the system-wide parameters. (Unicode)

[TabbedTextOutA](#)

The TabbedTextOut function writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions. Text is written in the currently selected font, background color, and text color. (ANSI)

[TabbedTextOutW](#)

The TabbedTextOut function writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions. Text is written in the currently selected font, background color, and text color. (Unicode)

[TileWindows](#)

Tiles the specified child windows of the specified parent window.

[ToAscii](#)

Translates the specified virtual-key code and keyboard state to the corresponding character or characters.

[ToAsciiEx](#)

Translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the input locale identifier.

[TOUCH_COORD_TO_PIXEL](#)

Converts touch coordinates to pixels.

[ToUnicode](#)

Translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters. (ToUnicode)

[ToUnicodeEx](#)

Translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters. (ToUnicodeEx)

[TrackMouseEvent](#)

Posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

[TrackPopupMenu](#)

Displays a shortcut menu at the specified location and tracks the selection of items on the menu. The shortcut menu can appear anywhere on the screen.

[TrackPopupMenuEx](#)

Displays a shortcut menu at the specified location and tracks the selection of items on the shortcut menu. The shortcut menu can appear anywhere on the screen.

[TranslateAcceleratorA](#)

Processes accelerator keys for menu commands. (ANSI)

[TranslateAcceleratorW](#)

Processes accelerator keys for menu commands. (Unicode)

[TranslateMDISysAccel](#)

Processes accelerator keystrokes for window menu commands of the multiple-document interface (MDI) child windows associated with the specified MDI client window.

[TranslateMessage](#)

Translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the GetMessage or PeekMessage function.

[UnhookWindowsHookEx](#)

Removes a hook procedure installed in a hook chain by the SetWindowsHookEx function.

[UnhookWinEvent](#)

Removes an event hook function created by a previous call to SetWinEventHook.

[UnionRect](#)

The UnionRect function creates the union of two rectangles. The union is the smallest rectangle that contains both source rectangles.

[UnloadKeyboardLayout](#)

Unloads an input locale identifier (formerly called a keyboard layout).

[UnregisterClassA](#)

Unregisters a window class, freeing the memory required for the class. (ANSI)

[UnregisterClassW](#)

Unregisters a window class, freeing the memory required for the class. (Unicode)

[UnregisterDeviceNotification](#)

Closes the specified device notification handle.

[UnregisterHotKey](#)

Frees a hot key previously registered by the calling thread.

[UnregisterPointerInputTarget](#)

Allows the caller to unregister a target window to which all pointer input of the specified type is redirected.

[UnregisterPointerInputTargetEx](#)

UnregisterPointerInputTargetEx may be altered or unavailable. Instead, use UnregisterPointerInputTarget.

[UnregisterPowerSettingNotification](#)

Unregisters the power setting notification.

[UnregisterSuspendResumeNotification](#)

Cancels a registration to receive notification when the system is suspended or resumed. Similar to PowerUnregisterSuspendResumeNotification but operates in user mode.

[UnregisterTouchWindow](#)

Registers a window as no longer being touch-capable.

[UpdateLayeredWindow](#)

Updates the position, size, shape, content, and translucency of a layered window.

[UpdateWindow](#)

The UpdateWindow function updates the client area of the specified window by sending a WM_PAINT message to the window if the window's update region is not empty.

[UserHandleGrantAccess](#)

Grants or denies access to a handle to a User object to a job that has a user-interface restriction.

[ValidateRect](#)

The ValidateRect function validates the client area within a rectangle by removing the rectangle from the update region of the specified window.

[ValidateRgn](#)

The ValidateRgn function validates the client area within a region by removing the region from the current update region of the specified window.

[VkKeyScanA](#)

Translates a character to the corresponding virtual-key code and shift state for the current keyboard. (ANSI)

[VkKeyScanExA](#)

Translates a character to the corresponding virtual-key code and shift state. The function translates the character using the input language and physical keyboard layout identified by the input locale identifier. (ANSI)

[VkKeyScanExW](#)

Translates a character to the corresponding virtual-key code and shift state. The function translates the character using the input language and physical keyboard layout identified by the input locale identifier. (Unicode)

[VkKeyScanW](#)

Translates a character to the corresponding virtual-key code and shift state for the current keyboard. (Unicode)

[WaitForInputIdle](#)

Waits until the specified process has finished processing its initial input and is waiting for user input with no input pending, or until the time-out interval has elapsed.

[WaitMessage](#)

Yields control to other threads when a thread has no other messages in its message queue. The WaitMessage function suspends the thread and does not return until a new message is placed in the thread's message queue.

[WindowFromDC](#)

The WindowFromDC function returns a handle to the window associated with the specified display device context (DC). Output functions that use the specified device context draw into this window.

[WindowFromPhysicalPoint](#)

Retrieves a handle to the window that contains the specified physical point.

[WindowFromPoint](#)

Retrieves a handle to the window that contains the specified point.

[WinHelpA](#)

Launches Windows Help (Winhelp.exe) and passes additional data that indicates the nature of the help requested by the application. (ANSI)

[WinHelpW](#)

Launches Windows Help (Winhelp.exe) and passes additional data that indicates the nature of the help requested by the application. (Unicode)

[wsprintfA](#)

Writes formatted data to the specified buffer. (ANSI)

[wsprintfW](#)

Writes formatted data to the specified buffer. (Unicode)

[vwsprintfA](#)

Writes formatted data to the specified buffer using a pointer to a list of arguments. (ANSI)

[vwsprintfW](#)

Writes formatted data to the specified buffer using a pointer to a list of arguments. (Unicode)

Callback functions

[DLGPROC](#)

Application-defined callback function used with the CreateDialog and DialogBox families of functions.

[DRAWSTATEPROC](#)

The DrawStateProc function is an application-defined callback function that renders a complex image for the DrawState function.

[EDITWORDBREAKPROCA](#)

An application-defined callback function used with the EM_SETWORDBREAKPROC message. (ANSI)

[EDITWORDBREAKPROCW](#)

An application-defined callback function used with the EM_SETWORDBREAKPROC message. (Unicode)

[GRAYSTRINGPROC](#)

The OutputProc function is an application-defined callback function used with the GrayString function.

[HOOKPROC](#)

An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function after the SendMessage function is called. The hook procedure can examine the message; it cannot modify it.

[MONITORENUMPROC](#)

A MonitorEnumProc function is an application-defined callback function that is called by the EnumDisplayMonitors function.

[MSGBOXCALLBACK](#)

A callback function, which you define in your application, that processes help events for the message box.

[PROOPENUMPROCA](#)

An application-defined callback function used with the EnumProps function. (ANSI)

[PROOPENUMPROCEXA](#)

Application-defined callback function used with the EnumPropsEx function. (ANSI)

[PROOPENUMPROCEWX](#)

Application-defined callback function used with the EnumPropsEx function. (Unicode)

[PROOPENUMPROCW](#)

An application-defined callback function used with the EnumProps function. (Unicode)

[SENDASYNCPROC](#)

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

TIMERPROC

An application-defined callback function that processes WM_TIMER messages. The TIMERPROC type defines a pointer to this callback function. TimerProc is a placeholder for the application-defined function name.

WINEVENTPROC

An application-defined callback (or hook) function that the system calls in response to events generated by an accessible object.

WNDPROC

A callback function, which you define in your application, that processes messages sent to a window.

Structures

ACCEL

Defines an accelerator key used in an accelerator table.

ACCESSTIMEOUT

Contains information about the time-out period associated with the accessibility features.

ALTTABINFO

Contains status information for the application-switching (ALT+TAB) window.

ANIMATIONINFO

Describes the animation effects associated with user actions.

AUDIODESCRIPTION

Contains information associated with audio descriptions. This structure is used with the SystemParametersInfo function when the SPI_GETAUDIODESCRIPTION or SPI_SETAUDIODESCRIPTION action value is specified.

BSMINFO

Contains information about a window that denied a request from BroadcastSystemMessageEx.

CBT_CREATEWNDA
Contains information passed to a WH_CBT hook procedure, CBTProc, before a window is created. (ANSI)
CBT_CREATEWNDW
Contains information passed to a WH_CBT hook procedure, CBTProc, before a window is created. (Unicode)
CBTACTIVATESTRUCT
Contains information passed to a WH_CBT hook procedure, CBTProc, before a window is activated.
CHANGEFILTERSTRUCT
Contains extended result information obtained by calling the ChangeWindowMessageFilterEx function.
CLIENTCREATESTRUCT
Contains information about the menu and first multiple-document interface (MDI) child window of an MDI client window.
COMBOBOXINFO
Contains combo box status information.
COMPAREITEMSTRUCT
Supplies the identifiers and application-supplied data for two items in a sorted, owner-drawn list box or combo box.
COPYDATASTRUCT
Contains data to be passed to another application by the WM_COPYDATA message.
CREATESTRUCTA
Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the CreateWindowEx function. (ANSI)
CREATESTRUCTW
Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the CreateWindowEx function. (Unicode)

CURSORINFO
Contains global cursor information.
CURSORSHAPE
Contains information about a cursor.
CWPRETSTRUCT
Defines the message parameters passed to a WH_CALLWNDPROCRET hook procedure, CallWndRetProc.
CWPSTRUCT
Defines the message parameters passed to a WH_CALLWNDPROC hook procedure, CallWndProc.
DEBUGHOOKINFO
Contains debugging information passed to a WH_DEBUG hook procedure, DebugProc.
DELETEITEMSTRUCT
Describes a deleted list box or combo box item.
DLGITEMTEMPLATE
Defines the dimensions and style of a control in a dialog box. One or more of these structures are combined with a DLGTEMPLATE structure to form a standard template for a dialog box.
DLGTEMPLATE
Defines the dimensions and style of a dialog box.
DRAWITEMSTRUCT
Provides information that the owner window uses to determine how to paint an owner-drawn control or menu item.
DRAWTEXTPARAMS
The DRAWTEXTPARAMS structure contains extended formatting options for the DrawTextEx function.
EVENTMSG
Contains information about a hardware message sent to the system message queue. This structure is used to store message information for the JournalPlaybackProc callback function.

FILTERKEYS	Contains information about the FilterKeys accessibility feature, which enables a user with disabilities to set the keyboard repeat rate (RepeatKeys), acceptance delay (SlowKeys), and bounce rate (BounceKeys).
FLASHINFO	Contains the flash status for a window and the number of times the system should flash the window.
GESTURECONFIG	Gets and sets the configuration for enabling gesture messages and the type of this configuration.
GESTUREINFO	Stores information about a gesture.
GESTURENOTIFYSTRUCT	When transmitted with WM_GESTURENOTIFY messages, passes information about a gesture.
GUITHREADINFO	Contains information about a GUI thread.
HARDWAREINPUT	Contains information about a simulated message generated by an input device other than a keyboard or mouse.
HELPINFO	Contains information about an item for which context-sensitive help has been requested.
HELPWININFOA	Contains the size and position of either a primary or secondary Help window. An application can set this information by calling the WinHelp function with the HELP_SETWINPOS value. (ANSI)
HELPWININFOW	Contains the size and position of either a primary or secondary Help window. An application can set this information by calling the WinHelp function with the HELP_SETWINPOS value. (Unicode)
HIGHCONTRASTA	Contains information about the high contrast accessibility feature. (ANSI)

[HIGHCONTRASTW](#)

Contains information about the high contrast accessibility feature. (Unicode)

[ICONINFO](#)

Contains information about an icon or a cursor.

[ICONINFOEXA](#)

Contains information about an icon or a cursor. Extends ICONINFO. Used by GetIconInfoEx. (ANSI)

[ICONINFOEXW](#)

Contains information about an icon or a cursor. Extends ICONINFO. Used by GetIconInfoEx. (Unicode)

[ICONMETRICS](#)

Contains the scalable metrics associated with icons. This structure is used with the SystemParametersInfo function when the SPI_GETICONMETRICS or SPI_SETICONMETRICS action is specified. (ANSI)

[ICONMETRICSW](#)

Contains the scalable metrics associated with icons. This structure is used with the SystemParametersInfo function when the SPI_GETICONMETRICS or SPI_SETICONMETRICS action is specified. (Unicode)

[INPUT](#)

Used by SendInput to store information for synthesizing input events such as keystrokes, mouse movement, and mouse clicks.

[INPUT_INJECTION_VALUE](#)

Contains the input injection details.

[INPUT_MESSAGE_SOURCE](#)

Contains information about the source of the input message.

[INPUT_TRANSFORM](#)

Defines the matrix that represents a transform on a message consumer.

[KBDLLHOOKSTRUCT](#)

Contains information about a low-level keyboard input event.

[KEYBDINPUT](#)

Contains information about a simulated keyboard event.

[LASTINPUTINFO](#)

Contains the time of the last input.

[MDICREATESTRUCTA](#)

Contains information about the class, title, owner, location, and size of a multiple-document interface (MDI) child window. (ANSI)

[MDICREATESTRUCTW](#)

Contains information about the class, title, owner, location, and size of a multiple-document interface (MDI) child window. (Unicode)

[MDINEXTMENU](#)

Contains information about the menu to be activated.

[MEASUREITEMSTRUCT](#)

Informs the system of the dimensions of an owner-drawn control or menu item. This allows the system to process user interaction with the control correctly.

[MENUBARINFO](#)

Contains menu bar information.

[MENUGETOBJECTINFO](#)

Contains information about the menu that the mouse cursor is on.

[MENUINFO](#)

Contains information about a menu.

[MENUITEMINFOA](#)

Contains information about a menu item. (MENUITEMINFOA)

[MENUITEMINFOW](#)

Contains information about a menu item. (MENUITEMINFOW)

MENUITEMTEMPLATE

Defines a menu item in a menu template.

MENUITEMTEMPLATEHEADER

Defines the header for a menu template. A complete menu template consists of a header and one or more menu item lists.

MINIMIZEDMETRICS

Contains the scalable metrics associated with minimized windows.

MINMAXINFO

Contains information about a window's maximized size and position and its minimum and maximum tracking size.

MONITORINFO

The MONITORINFO structure contains information about a display monitor. The GetMonitorInfo function stores information in a MONITORINFO structure or a MONITORINFOEX structure. The MONITORINFO structure is a subset of the MONITORINFOEX structure.

MONITORINFOEXA

The MONITORINFOEX structure contains information about a display monitor. The GetMonitorInfo function stores information into a MONITORINFOEX structure or a MONITORINFO structure. The MONITORINFOEX structure is a superset of the MONITORINFO structure. (ANSI)

MONITORINFOEXW

The MONITORINFOEX structure contains information about a display monitor. The GetMonitorInfo function stores information into a MONITORINFOEX structure or a MONITORINFO structure. The MONITORINFOEX structure is a superset of the MONITORINFO structure. (Unicode)

MOUSEHOOKSTRUCT

Contains information about a mouse event passed to a WH_MOUSE hook procedure, MouseProc.

MOUSEHOOKSTRUCTEX

Contains information about a mouse event passed to a WH_MOUSE hook procedure, MouseProc. This is an extension of the MOUSEHOOKSTRUCT structure that includes information about wheel movement or the use of the X button.

MOUSEINPUT

Contains information about a simulated mouse event.

[MOUSEKEYS](#)

Contains information about the MouseKeys accessibility feature.

[MOUSEMOVEPOINT](#)

Contains information about the mouse's location in screen coordinates.

[MSG](#)

Contains message information from a thread's message queue.

[MSGBOXPARAMSA](#)

Contains information used to display a message box. The MessageBoxIndirect function uses this structure. (ANSI)

[MSGBOXPARAMSW](#)

Contains information used to display a message box. The MessageBoxIndirect function uses this structure. (Unicode)

[MSLLHOOKSTRUCT](#)

Contains information about a low-level mouse input event.

[MULTIKEYHELPA](#)

Specifies a keyword to search for and the keyword table to be searched by Windows Help. (ANSI)

[MULTIKEYHELPW](#)

Specifies a keyword to search for and the keyword table to be searched by Windows Help. (Unicode)

[NCCALCSIZE_PARAMS](#)

Contains information that an application can use while processing the WM_NCCALCSIZE message to calculate the size, position, and valid contents of the client area of a window.

[NMHDR](#)

The NMHDR structure contains information about a notification message. (NMHDR structure)

[NONCLIENTMETRICS](#)

Contains the scalable metrics associated with the nonclient area of a nonminimized window. (ANSI)

[NONCLIENTMETRICSW](#)

Contains the scalable metrics associated with the nonclient area of a nonminimized window.
(Unicode)

[PAINTSTRUCT](#)

The PAINTSTRUCT structure contains information for an application. This information can be used to paint the client area of a window owned by that application.

[POINTER_DEVICE_CURSOR_INFO](#)

Contains cursor ID mappings for pointer devices.

[POINTER_DEVICE_INFO](#)

Contains information about a pointer device. An array of these structures is returned from the GetPointerDevices function. A single structure is returned from a call to the GetPointerDevice function.

[POINTER_DEVICE_PROPERTY](#)

Contains pointer-based device properties (Human Interface Device (HID) global items that correspond to HID usages).

[POINTER_INFO](#)

Contains basic pointer information common to all pointer types. Applications can retrieve this information using the GetPointerInfo, GetPointerFrameInfo, GetPointerInfoHistory and GetPointerFrameInfoHistory functions.

[POINTER_PEN_INFO](#)

Defines basic pen information common to all pointer types.

[POINTER_TOUCH_INFO](#)

Defines basic touch information common to all pointer types.

[POINTER_TYPE_INFO](#)

Contains information about the pointer input type.

[POWERBROADCAST_SETTING](#)

Sent with a power setting event and contains data about the specific change.

RAWHID
Describes the format of the raw input from a Human Interface Device (HID).
RAWINPUT
Contains the raw input from a device.
RAWINPUTDEVICE
Defines information for the raw input devices.
RAWINPUTDEVICELIST
Contains information about a raw input device.
RAWINPUTHEADER
Contains the header information that is part of the raw input data.
RAWKEYBOARD
Contains information about the state of the keyboard.
RAWMOUSE
Contains information about the state of the mouse.
RID_DEVICE_INFO
Defines the raw input data coming from any device.
RID_DEVICE_INFO_HID
Defines the raw input data coming from the specified Human Interface Device (HID).
RID_DEVICE_INFO_KEYBOARD
Defines the raw input data coming from the specified keyboard.
RID_DEVICE_INFO_MOUSE
Defines the raw input data coming from the specified mouse.
SCROLLBARINFO
The SCROLLBARINFO structure contains scroll bar information.

[SCROLLINFO](#)

The SCROLLINFO structure contains scroll bar parameters to be set by the SetScrollInfo function (or SBM_SETSCROLLINFO message), or retrieved by the GetScrollInfo function (or SBM_GETSCROLLINFO message).

[SERIALKEYSA](#)

Contains information about the SerialKeys accessibility feature, which interprets data from a communication aid attached to a serial port as commands causing the system to simulate keyboard and mouse input. (ANSI)

[SERIALKEYSW](#)

Contains information about the SerialKeys accessibility feature, which interprets data from a communication aid attached to a serial port as commands causing the system to simulate keyboard and mouse input. (Unicode)

[SOUNDSENTRYA](#)

Contains information about the SoundSentry accessibility feature. When the SoundSentry feature is on, the computer displays a visual indication only when a sound is generated. (ANSI)

[SOUNDSENTRYW](#)

Contains information about the SoundSentry accessibility feature. When the SoundSentry feature is on, the computer displays a visual indication only when a sound is generated. (Unicode)

[STICKYKEYS](#)

Contains information about the StickyKeys accessibility feature.

[STYLESTRUCT](#)

Contains the styles for a window.

[TITLEBARINFO](#)

Contains title bar information.

[TITLEBARINFOEX](#)

Expands on the information described in the TITLEBARINFO structure by including the coordinates of each element of the title bar.

[TOGGLEKEYS](#)

Contains information about the ToggleKeys accessibility feature.

[TOUCH_HIT_TESTING_INPUT](#)

Contains information about the touch contact area reported by the touch digitizer.

[TOUCH_HIT_TESTING_PROXIMITY_EVALUATION](#)

Contains the hit test score that indicates whether the object is the likely target of the touch contact area, relative to other objects that intersect the touch contact area.

[TOUCHINPUT](#)

Encapsulates data for touch input.

[TOUCHPREDICTIONPARAMETERS](#)

Contains hardware input details that can be used to predict touch targets and help compensate for hardware latency when processing touch and gesture input that contains distance and velocity data.

[TPMPARAMS](#)

Contains extended parameters for the TrackPopupMenuEx function.

[TRACKMOUSEEVENT](#)

Used by the TrackMouseEvent function to track when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

[UPDATELAYEREDWINDOWINFO](#)

Used by UpdateLayeredWindowIndirect to provide position, size, shape, content, and translucency information for a layered window.

[USAGE_PROPERTIES](#)

Contains device properties (Human Interface Device (HID) global items that correspond to HID usages) for any type of HID input device.

[USEROBJECTFLAGS](#)

Contains information about a window station or desktop handle.

[WINDOWINFO](#)

Contains window information.

[WINDOWPLACEMENT](#)

Contains information about the placement of a window on the screen.

[WINDOWPOS](#)

Contains information about the size and position of a window.

[WNDCLASSA](#)

Contains the window class attributes that are registered by the RegisterClass function. (ANSI)

[WNDCLASSEXA](#)

Contains window class information. (ANSI)

[WNDCLASSEXW](#)

Contains window class information. (Unicode)

[WNDCLASSW](#)

Contains the window class attributes that are registered by the RegisterClass function. (Unicode)

[WTSSESSION_NOTIFICATION](#)

Provides information about the session change notification. A service receives this structure in its HandlerEx function in response to a session change event.

Enumerations

[AR_STATE](#)

Indicates the state of screen auto-rotation for the system. For example, whether auto-rotation is supported, and whether it is enabled by the user.

[DIALOG_CONTROL_DPI_CHANGE_BEHAVIORS](#)

Describes per-monitor DPI scaling behavior overrides for child windows within dialogs. The values in this enumeration are bitfields and can be combined.

[DIALOG_DPI_CHANGE_BEHAVIORS](#)

In Per Monitor v2 contexts, dialogs will automatically respond to DPI changes by resizing themselves and re-computing the positions of their child windows (here referred to as re-laying out).

FEEDBACK_TYPE
Specifies the visual feedback associated with an event.
INPUT_MESSAGE_DEVICE_TYPE
The type of device that sent the input message.
INPUT_MESSAGE_ORIGIN_ID
The ID of the input message source.
ORIENTATION_PREFERENCE
Indicates the screen orientation preference for a desktop app process.
POINTER_BUTTON_CHANGE_TYPE
Identifies a change in the state of a button associated with a pointer.
POINTER_DEVICE_CURSOR_TYPE
Identifies the pointer device cursor types.
POINTER_DEVICE_TYPE
Identifies the pointer device types.
POINTER_FEEDBACK_MODE
Identifies the visual feedback behaviors available to CreateSyntheticPointerDevice.
tagPOINTER_INPUT_TYPE
Identifies the pointer input types.

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

AdjustWindowRect function (winuser.h)

Article 10/13/2021

Calculates the required size of the window rectangle, based on the desired client-rectangle size. The window rectangle can then be passed to the [CreateWindow](#) function to create a window whose client area is the desired size.

To specify an extended window style, use the [AdjustWindowRectEx](#) function.

Syntax

C++

```
BOOL AdjustWindowRect(
    [in, out] LPRECT lpRect,
    [in]      DWORD dwStyle,
    [in]      BOOL bMenu
);
```

Parameters

[in, out] lpRect

Type: **LPRECT**

A pointer to a [RECT](#) structure that contains the coordinates of the top-left and bottom-right corners of the desired client area. When the function returns, the structure contains the coordinates of the top-left and bottom-right corners of the window to accommodate the desired client area.

[in] dwStyle

Type: **DWORD**

The [window style](#) of the window whose required size is to be calculated. Note that you cannot specify the **WS_OVERLAPPED** style.

[in] bMenu

Type: **BOOL**

Indicates whether the window has a menu.

Return value

Type: **BOOL**

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 client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window, which includes the client area and the nonclient area.

The **AdjustWindowRect** function does not add extra space when a menu bar wraps to two or more rows.

The **AdjustWindowRect** function does not take the **WS_VSCROLL** or **WS_HSCROLL** styles into account. To account for the scroll bars, call the [GetSystemMetrics](#) function with **SM_CXVSCROLL** or **SM_CYHSCROLL**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[AdjustWindowRectEx](#)

[Conceptual](#)

[CreateWindow](#)

[GetSystemMetrics](#)

Other Resources

[RECT](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

AdjustWindowRectEx function (winuser.h)

Article 10/13/2021

Calculates the required size of the window rectangle, based on the desired size of the client rectangle. The window rectangle can then be passed to the [CreateWindowEx](#) function to create a window whose client area is the desired size.

Syntax

C++

```
BOOL AdjustWindowRectEx(
    [in, out] LPRECT lpRect,
    [in]      DWORD dwStyle,
    [in]      BOOL bMenu,
    [in]      DWORD dwExStyle
);
```

Parameters

[in, out] lpRect

Type: **LPRECT**

A pointer to a [RECT](#) structure that contains the coordinates of the top-left and bottom-right corners of the desired client area. When the function returns, the structure contains the coordinates of the top-left and bottom-right corners of the window to accommodate the desired client area.

[in] dwStyle

Type: **DWORD**

The [window style](#) of the window whose required size is to be calculated. Note that you cannot specify the **WS_OVERLAPPED** style.

[in] bMenu

Type: **BOOL**

Indicates whether the window has a menu.

[in] dwExStyle

Type: **DWORD**

The [extended window style](#) of the window whose required size is to be calculated.

Return value

Type: **BOOL**

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 client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window, which includes the client area and the nonclient area.

The **AdjustWindowRectEx** function does not add extra space when a menu bar wraps to two or more rows.

The **AdjustWindowRectEx** function does not take the **WS_VSCROLL** or **WS_HSCROLL** styles into account. To account for the scroll bars, call the [GetSystemMetrics](#) function with **SM_CXVSCROLL** or **SM_CYHSCROLL**.

This API is not DPI aware, and should not be used if the calling thread is per-monitor DPI aware. For the DPI-aware version of this API, see [AdjustWindowsRectExForDPI](#). For more information on DPI awareness, see [the Windows High DPI documentation](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[AdjustWindowsRectExForDPI](#)

[Conceptual](#)

[CreateWindowEx](#)

[Other Resources](#)

[RECT](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

AllowSetForegroundWindow function (winuser.h)

Article03/11/2023

Enables the specified process to set the foreground window using the [SetForegroundWindow](#) function. The calling process must already be able to set the foreground window. For more information, see Remarks later in this topic.

Syntax

C++

```
BOOL AllowSetForegroundWindow(  
    [in] DWORD dwProcessId  
);
```

Parameters

[in] dwProcessId

Type: **DWORD**

The identifier of the process that will be enabled to set the foreground window. If this parameter is **ASFW_ANY**, all processes will be enabled to set the foreground window.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function will fail if the calling process cannot set the foreground window. To get extended error information, call [GetLastError](#).

Remarks

The system restricts which processes can set the foreground window. Normally, a process can set the foreground window by calling the [SetForegroundWindow](#) function only if:

- All of the following conditions are true:
 - The calling process belongs to a desktop application, not a UWP app or a Windows Store app designed for Windows 8 or 8.1.
 - The foreground process has not disabled calls to **SetForegroundWindow** by a previous call to the **LockSetForegroundWindow** function.
 - The foreground lock time-out has expired (see [SPI_GETFOREGROUNDLOCKTIMEOUT](#) in **SystemParametersInfo**).
 - No menus are active.
- Additionally, at least one of the following conditions is true:
 - The calling process is the foreground process.
 - The calling process was started by the foreground process.
 - There is currently no foreground window, and thus no foreground process.
 - The calling process received the last input event.
 - Either the foreground process or the calling process is being debugged.

A process that can set the foreground window can enable another process to set the foreground window by calling **AllowSetForegroundWindow**. The process specified by the *dwProcessId* parameter loses the ability to set the foreground window the next time that either the user generates input, unless the input is directed at that process, or the next time a process calls **AllowSetForegroundWindow**, unless the same process is specified as in the previous call to **AllowSetForegroundWindow**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[LockSetForegroundWindow](#)

[Reference](#)

[SetForegroundWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ALTTABINFO structure (winuser.h)

Article11/19/2022

Contains status information for the application-switching (ALT+TAB) window.

Syntax

C++

```
typedef struct tagALTTABINFO {
    DWORD cbSize;
    int   cItems;
    int   cColumns;
    int   cRows;
    int   iColFocus;
    int   iRowFocus;
    int   cxItem;
    int   cyItem;
    POINT ptStart;
} ALTTABINFO, *PALTTABINFO, *LPALTTABINFO;
```

Members

`cbSize`

Type: `DWORD`

The size, in bytes, of the structure. The caller must set this to `sizeof(ALTTABINFO)`.

`cItems`

Type: `int`

The number of items in the window.

`cColumns`

Type: `int`

The number of columns in the window.

`cRows`

Type: `int`

The number of rows in the window.

`iColFocus`

Type: `int`

The column of the item that has the focus.

`iRowFocus`

Type: `int`

The row of the item that has the focus.

`cxItem`

Type: `int`

The width of each icon in the application-switching window.

`cyItem`

Type: `int`

The height of each icon in the application-switching window.

`ptStart`

Type: `POINT`

The top-left corner of the first icon.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[GetAltTabInfo](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

AnimateWindow function (winuser.h)

Article 10/13/2021

Enables you to produce special effects when showing or hiding windows. There are four types of animation: roll, slide, collapse or expand, and alpha-blended fade.

Syntax

C++

```
BOOL AnimateWindow(
    [in] HWND hWnd,
    [in] DWORD dwTime,
    [in] DWORD dwFlags
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to animate. The calling thread must own this window.

[in] dwTime

Type: **DWORD**

The time it takes to play the animation, in milliseconds. Typically, an animation takes 200 milliseconds to play.

[in] dwFlags

Type: **DWORD**

The type of animation. This parameter can be one or more of the following values. Note that, by default, these flags take effect when showing a window. To take effect when hiding a window, use **AW_HIDE** and a logical OR operator with the appropriate flags.

Value	Meaning
AW_ACTIVATE 0x00020000	Activates the window. Do not use this value with AW_HIDE .

AW_BLEND 0x00080000	Uses a fade effect. This flag can be used only if <i>hwnd</i> is a top-level window.
AW_CENTER 0x00000010	Makes the window appear to collapse inward if AW_HIDE is used or expand outward if the AW_HIDE is not used. The various direction flags have no effect.
AW_HIDE 0x00010000	Hides the window. By default, the window is shown.
AW_HOR_POSITIVE 0x00000001	Animates the window from left to right. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND .
AW_HOR_NEGATIVE 0x00000002	Animates the window from right to left. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND .
AW_SLIDE 0x00040000	Uses slide animation. By default, roll animation is used. This flag is ignored when used with AW_CENTER .
AW_VER_POSITIVE 0x00000004	Animates the window from top to bottom. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND .
AW_VER_NEGATIVE 0x00000008	Animates the window from bottom to top. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND .

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function will fail in the following situations:

- If the window is already visible and you are trying to show the window.
- If the window is already hidden and you are trying to hide the window.
- If there is no direction specified for the slide or roll animation.
- When trying to animate a child window with **AW_BLEND**.
- If the thread does not own the window. Note that, in this case, **AnimateWindow** fails but [GetLastError](#) returns **ERROR_SUCCESS**.

To get extended error information, call the [GetLastError](#) function.

Remarks

To show or hide a window without special effects, use [ShowWindow](#).

When using slide or roll animation, you must specify the direction. It can be either **AW_HOR_POSITIVE**, **AW_HOR_NEGATIVE**, **AW_VER_POSITIVE**, or **AW_VER_NEGATIVE**.

You can combine **AW_HOR_POSITIVE** or **AW_HOR_NEGATIVE** with **AW_VER_POSITIVE** or **AW_VER_NEGATIVE** to animate a window diagonally.

The window procedures for the window and its child windows should handle any **WM_PRINT** or **WM_PRINTCLIENT** messages. Dialog boxes, controls, and common controls already handle **WM_PRINTCLIENT**. The default window procedure already handles **WM_PRINT**.

If a child window is displayed partially clipped, when it is animated it will have holes where it is clipped.

AnimateWindow supports RTL windows.

Avoid animating a window that has a drop shadow because it produces visually distracting, jerky animations.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

Other Resources

Reference

[ShowWindow](#)

[WM_PRINT](#)

[WM_PRINTCLIENT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ANIMATIONINFO structure (winuser.h)

Article04/02/2021

Describes the animation effects associated with user actions. This structure is used with the [SystemParametersInfo](#) function when the SPI_GETANIMATION or SPI_SETANIMATION action value is specified.

Syntax

C++

```
typedef struct tagANIMATIONINFO {
    UINT cbSize;
    int iMinAnimate;
} ANIMATIONINFO, *LPANIMATIONINFO;
```

Members

`cbSize`

The size of the structure, in bytes. The caller must set this to `sizeof(ANIMATIONINFO)`.

`iMinAnimate`

If this member is nonzero, minimize and restore animation is enabled; otherwise it is disabled.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

AnyPopup function (winuser.h)

Article 06/29/2021

Indicates whether an owned, visible, top-level pop-up, or overlapped window exists on the screen. The function searches the entire screen, not just the calling application's client area.

This function is provided only for compatibility with 16-bit versions of Windows. It is generally not useful.

Syntax

C++

```
BOOL AnyPopup();
```

Return value

Type: **BOOL**

If a pop-up window exists, the return value is nonzero, even if the pop-up window is completely covered by other windows.

If a pop-up window does not exist, the return value is zero.

Remarks

This function does not detect unowned pop-up windows or windows that do not have the **WS_VISIBLE** style bit set.

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[GetLastActivePopup](#)

Reference

[ShowOwnedPopups](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ArrangeIconicWindows function (winuser.h)

Article 10/13/2021

Arranges all the minimized (iconic) child windows of the specified parent window.

Syntax

C++

```
UINT ArrangeIconicWindows(
    [in] HWND hWnd
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the parent window.

Return value

Type: **UINT**

If the function succeeds, the return value is the height of one row of icons.

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

Remarks

An application that maintains its own minimized child windows can use the **ArrangeIconicWindows** function to arrange icons in a parent window. This function can also arrange icons on the desktop. To retrieve the window handle to the desktop window, use the [GetDesktopWindow](#) function.

An application sends the [WM_MDIICONARRANGE](#) message to the multiple-document interface (MDI) client window to prompt the client window to arrange its minimized MDI child windows.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[CloseWindow](#)

[Conceptual](#)

[GetDesktopWindow](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

AUDIODESCRIPTION structure (winuser.h)

Article04/02/2021

Contains information associated with audio descriptions. This structure is used with the [SystemParametersInfo](#) function when the SPI_GETAUDIODESCRIPTION or SPI_SETAUDIODESCRIPTION action value is specified.

Syntax

C++

```
typedef struct tagAUDIODESCRIPTION {
    UINT cbSize;
    BOOL Enabled;
    LCID Locale;
} AUDIODESCRIPTION, *LPAUDIODESCRIPTION;
```

Members

`cbSize`

The size of the structure, in bytes. The caller must set this member to `sizeof(AUDIODESCRIPTION)`.

`Enabled`

If this member is **TRUE**, audio descriptions are enabled; Otherwise, this member is **FALSE**.

`Locale`

The locale identifier (LCID) of the language for the audio description. For more information, see [Locales and Languages](#).

Remarks

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

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

BeginDeferWindowPos function (winuser.h)

Article 10/03/2022

Allocates memory for a multiple-window-position structure and returns the handle to the structure.

Syntax

C++

```
HDWP BeginDeferWindowPos(  
    [in] int nNumWindows  
);
```

Parameters

[in] nNumWindows

Type: **int**

The initial number of windows for which to store position information. The [DeferWindowPos](#) function increases the size of the structure, if necessary.

Return value

Type: **HDWP**

If the function succeeds, the return value identifies the multiple-window-position structure. If insufficient system resources are available to allocate the structure, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The multiple-window-position structure is an internal structure; an application cannot access it directly.

[DeferWindowPos](#) fills the multiple-window-position structure with information about the target position for one or more windows about to be moved. The

[EndDeferWindowPos](#) function accepts the handle to this structure and repositions the windows by using the information stored in the structure.

If the system must increase the size of the multiple-window- position structure beyond the initial size specified by the *nNumWindows* parameter but cannot allocate enough memory to do so, the system fails the entire window positioning sequence ([BeginDeferWindowPos](#), [DeferWindowPos](#), and [EndDeferWindowPos](#)). By specifying the maximum size needed, an application can detect and process failure early in the process.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[DeferWindowPos](#)

[EndDeferWindowPos](#)

[Reference](#)

[SetWindowPos](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

BringWindowToTop function (winuser.h)

Article10/13/2021

Brings the specified window to the top of the Z order. If the window is a top-level window, it is activated. If the window is a child window, the top-level parent window associated with the child window is activated.

Syntax

C++

```
BOOL BringWindowToTop(
    [in] HWND hWnd
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to bring to the top of the Z order.

Return value

Type: **BOOL**

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 the **BringWindowToTop** function to uncover any window that is partially or completely obscured by other windows.

Calling this function is similar to calling the [SetWindowPos](#) function to change a window's position in the Z order. **BringWindowToTop** does not make a window a top-level window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Reference](#)

[SetWindowPos](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

BroadcastSystemMessage function (winuser.h)

Article 08/02/2022

Sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

To receive additional information if the request is defined, use the [BroadcastSystemMessageEx](#) function.

Syntax

C++

```
long BroadcastSystemMessage(
    [in]           DWORD   flags,
    [in, out, optional] LPDWORD lpInfo,
    [in]           UINT    Msg,
    [in]           WPARAM  wParam,
    [in]           LPARAM  lParam
);
```

Parameters

[in] flags

Type: **DWORD**

The broadcast option. This parameter can be one or more of the following values.

Value	Meaning
BSF_ALLOWFW 0x00000080	Enables the recipient to set the foreground window while processing the message.
BSF_FLUSHDISK 0x00000004	Flushes the disk after each recipient processes the message.
BSF_FORCEIFHUNG 0x00000020	Continues to broadcast the message, even if the time-out period elapses or one of the recipients is not responding.
BSF_IGNORECURRENTTASK	Does not send the message to windows that belong to

0x00000002	the current task. This prevents an application from receiving its own message.
BSF_NOHANG 0x00000008	Forces a nonresponsive application to time out. If one of the recipients times out, do not continue broadcasting the message.
BSF_NOTIMEOUTIFNOTHUNG 0x00000040	Waits for a response to the message, as long as the recipient is not being unresponsive. Does not time out.
BSF_POSTMESSAGE 0x00000010	Posts the message. Do not use in combination with BSF_QUERY .
BSF_QUERY 0x00000001	Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE.
BSF_SENDNOTIFYMESSAGE 0x00000100	Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY .

[in, out, optional] lpInfo

Type: **LPDWORD**

A pointer to a variable that contains and receives information about the recipients of the message.

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is **NULL**, the function broadcasts to all components.

This parameter can be one or more of the following values.

Value	Meaning
BSM_ALLCOMPONENTS 0x00000000	Broadcast to all system components.
BSM_ALLDESKTOPS 0x00000010	Broadcast to all desktops. Requires the SE_TCB_NAME privilege.
BSM_APPLICATIONS 0x00000008	Broadcast to applications.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **long**

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is –1.

If the *dwFlags* parameter is **BSF_QUERY** and at least one recipient returned **BROADCAST_QUERY_DENY** to the corresponding message, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If **BSF_QUERY** is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= **WM_USER**) to another process, you must do custom marshalling.

Examples

For an example, see [Terminating a Process](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[BroadcastSystemMessageEx](#)

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

BroadcastSystemMessageA function (winuser.h)

Article 07/27/2022

Sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

To receive additional information if the request is defined, use the [BroadcastSystemMessageEx](#) function.

Syntax

C++

```
long BroadcastSystemMessageA(
    [in]           DWORD   flags,
    [in, out, optional] LPDWORD lpInfo,
    [in]           UINT    Msg,
    [in]           WPARAM  wParam,
    [in]           LPARAM  lParam
);
```

Parameters

[in] flags

Type: **DWORD**

The broadcast option. This parameter can be one or more of the following values.

Value	Meaning
BSF_ALLOWFW 0x00000080	Enables the recipient to set the foreground window while processing the message.
BSF_FLUSHDISK 0x00000004	Flushes the disk after each recipient processes the message.
BSF_FORCEIFHUNG 0x00000020	Continues to broadcast the message, even if the time-out period elapses or one of the recipients is not responding.
BSF_IGNORECURRENTTASK	Does not send the message to windows that belong to

0x00000002	the current task. This prevents an application from receiving its own message.
BSF_NOHANG 0x00000008	Forces a nonresponsive application to time out. If one of the recipients times out, do not continue broadcasting the message.
BSF_NOTIMEOUTIFNOTHUNG 0x00000040	Waits for a response to the message, as long as the recipient is not being unresponsive. Does not time out.
BSF_POSTMESSAGE 0x00000010	Posts the message. Do not use in combination with BSF_QUERY .
BSF_QUERY 0x00000001	Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE.
BSF_SENDNOTIFYMESSAGE 0x00000100	Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY .

[in, out, optional] lpInfo

Type: **LPDWORD**

A pointer to a variable that contains and receives information about the recipients of the message.

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is **NULL**, the function broadcasts to all components.

This parameter can be one or more of the following values.

Value	Meaning
BSM_ALLCOMPONENTS 0x00000000	Broadcast to all system components.
BSM_ALLDESKTOPS 0x00000010	Broadcast to all desktops. Requires the SE_TCB_NAME privilege.
BSM_APPLICATIONS 0x00000008	Broadcast to applications.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **long**

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is –1.

If the *dwFlags* parameter is **BSF_QUERY** and at least one recipient returned **BROADCAST_QUERY_DENY** to the corresponding message, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If **BSF_QUERY** is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= **WM_USER**) to another process, you must do custom marshalling.

For an example, see [Terminating a Process](#).

ⓘ Note

The winuser.h header defines **BroadcastSystemMessage** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[BroadcastSystemMessageEx](#)

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

BroadcastSystemMessageExA function (winuser.h)

Article 02/09/2023

Sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

This function is similar to [BroadcastSystemMessage](#) except that this function can return more information from the recipients.

Syntax

C++

```
long BroadcastSystemMessageExA(
    [in]          DWORD      flags,
    [in, out, optional] LPDWORD   lpInfo,
    [in]          UINT       Msg,
    [in]          WPARAM     wParam,
    [in]          LPARAM     lParam,
    [out, optional] PBSMINFO  pbsmInfo
);
```

Parameters

[in] flags

Type: **DWORD**

The broadcast option. This parameter can be one or more of the following values.

Value	Meaning
BSF_ALLOWSFW 0x00000080	Enables the recipient to set the foreground window while processing the message.
BSF_FLUSHDISK 0x00000004	Flushes the disk after each recipient processes the message.
BSF_FORCEIFHUNG 0x00000020	Continues to broadcast the message, even if the time-out period elapses or one of the recipients is not responding.

BSF_IGNORECURRENTTASK 0x00000002	Does not send the message to windows that belong to the current task. This prevents an application from receiving its own message.
BSF_LUID 0x00000400	If BSF_LUID is set, the message is sent to the window that has the same LUID as specified in the luid member of the BSMINFO structure.
	Windows 2000: This flag is not supported.
BSF_NOHANG 0x00000008	Forces a nonresponsive application to time out. If one of the recipients times out, do not continue broadcasting the message.
BSF_NOTIMEOUTIFNOTHUNG 0x00000040	Waits for a response to the message, as long as the recipient is not being unresponsive. Does not time out.
BSF_POSTMESSAGE 0x00000010	Posts the message. Do not use in combination with BSF_QUERY .
BSF_RETURNNHDESK 0x00000200	If access is denied and both this and BSF_QUERY are set, BSMINFO returns both the desktop handle and the window handle. If access is denied and only BSF_QUERY is set, only the window handle is returned by BSMINFO .
	Windows 2000: This flag is not supported.
BSF_QUERY 0x00000001	Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE.
BSF_SENDNOTIFYMESSAGE 0x00000100	Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY .

[in, out, optional] lpInfo

Type: **LPDWORD**

A pointer to a variable that contains and receives information about the recipients of the message.

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is **NULL**, the function broadcasts to all components.

This parameter can be one or more of the following values.

Value	Meaning

BSM_ALLCOMPONENTS 0x00000000	Broadcast to all system components.
BSM_ALLDESKTOPS 0x00000010	Broadcast to all desktops. Requires the SE_TCB_NAME privilege.
BSM_APPLICATIONS 0x00000008	Broadcast to applications.

[in] `Msg`

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] `wParam`

Type: **WPARAM**

Additional message-specific information.

[in] `lParam`

Type: **LPARAM**

Additional message-specific information.

[out, optional] `pbsmInfo`

Type: **PBSMINFO**

A pointer to a [BSMINFO](#) structure that contains additional information if the request is denied and *dwFlags* is set to **BSF_QUERY**.

Return value

Type: **long**

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is -1.

If the *dwFlags* parameter is **BSF_QUERY** and at least one recipient returned **BROADCAST_QUERY_DENY** to the corresponding message, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If **BSF_QUERY** is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

If the caller's thread is on a desktop other than that of the window that denied the request, the caller must call [SetThreadDesktop\(hdesk\)](#) to query anything on that window. Also, the caller must call [CloseDesktop](#) on the returned **hdesk** handle.

The system only does marshalling for system messages (those in the range 0 to [WM_USER-1](#)). To send other messages (those >= [WM_USER](#)) to another process, you must do custom marshalling.

ⓘ Note

The winuser.h header defines BroadcastSystemMessageEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[BSMINFO](#)

[BroadcastSystemMessage](#)

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

BroadcastSystemMessageExW function (winuser.h)

Article 02/09/2023

Sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

This function is similar to [BroadcastSystemMessage](#) except that this function can return more information from the recipients.

Syntax

C++

```
long BroadcastSystemMessageExW(
    [in]          DWORD      flags,
    [in, out, optional] LPDWORD   lpInfo,
    [in]          UINT       Msg,
    [in]          WPARAM     wParam,
    [in]          LPARAM     lParam,
    [out, optional] PBSMINFO  pbsmInfo
);
```

Parameters

[in] flags

Type: **DWORD**

The broadcast option. This parameter can be one or more of the following values.

Value	Meaning
BSF_ALLOWSFW 0x00000080	Enables the recipient to set the foreground window while processing the message.
BSF_FLUSHDISK 0x00000004	Flushes the disk after each recipient processes the message.
BSF_FORCEIFHUNG 0x00000020	Continues to broadcast the message, even if the time-out period elapses or one of the recipients is not responding.

BSF_IGNORECURRENTTASK 0x00000002	Does not send the message to windows that belong to the current task. This prevents an application from receiving its own message.
BSF_LUID 0x00000400	If BSF_LUID is set, the message is sent to the window that has the same LUID as specified in the luid member of the BSMINFO structure. Windows 2000: This flag is not supported.
BSF_NOHANG 0x00000008	Forces a nonresponsive application to time out. If one of the recipients times out, do not continue broadcasting the message.
BSF_NOTIMEOUTIFNOTHUNG 0x00000040	Waits for a response to the message, as long as the recipient is not being unresponsive. Does not time out.
BSF_POSTMESSAGE 0x00000010	Posts the message. Do not use in combination with BSF_QUERY .
BSF_RETURNNHDESK 0x00000200	If access is denied and both this and BSF_QUERY are set, BSMINFO returns both the desktop handle and the window handle. If access is denied and only BSF_QUERY is set, only the window handle is returned by BSMINFO . Windows 2000: This flag is not supported.
BSF_QUERY 0x00000001	Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE.
BSF_SENDNOTIFYMESSAGE 0x00000100	Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY .

[in, out, optional] lpInfo

Type: **LPDWORD**

A pointer to a variable that contains and receives information about the recipients of the message.

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is **NULL**, the function broadcasts to all components.

This parameter can be one or more of the following values.

Value	Meaning

BSM_ALLCOMPONENTS 0x00000000	Broadcast to all system components.
BSM_ALLDESKTOPS 0x00000010	Broadcast to all desktops. Requires the SE_TCB_NAME privilege.
BSM_APPLICATIONS 0x00000008	Broadcast to applications.

[in] `Msg`

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] `wParam`

Type: **WPARAM**

Additional message-specific information.

[in] `lParam`

Type: **LPARAM**

Additional message-specific information.

[out, optional] `pbsmInfo`

Type: **PBSMINFO**

A pointer to a [BSMINFO](#) structure that contains additional information if the request is denied and *dwFlags* is set to **BSF_QUERY**.

Return value

Type: **long**

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is –1.

If the *dwFlags* parameter is **BSF_QUERY** and at least one recipient returned **BROADCAST_QUERY_DENY** to the corresponding message, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If **BSF_QUERY** is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

If the caller's thread is on a desktop other than that of the window that denied the request, the caller must call [SetThreadDesktop\(hdesk\)](#) to query anything on that window. Also, the caller must call [CloseDesktop](#) on the returned **hdesk** handle.

The system only does marshalling for system messages (those in the range 0 to [WM_USER-1](#)). To send other messages (those >= [WM_USER](#)) to another process, you must do custom marshalling.

ⓘ Note

The winuser.h header defines BroadcastSystemMessageEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[BSMINFO](#)

[BroadcastSystemMessage](#)

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

BroadcastSystemMessageW function (winuser.h)

Article02/09/2023

Sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

To receive additional information if the request is defined, use the [BroadcastSystemMessageEx](#) function.

Syntax

C++

```
long BroadcastSystemMessageW(
    [in]           DWORD   flags,
    [in, out, optional] LPDWORD lpInfo,
    [in]           UINT    Msg,
    [in]           WPARAM  wParam,
    [in]           LPARAM  lParam
);
```

Parameters

[in] flags

Type: **DWORD**

The broadcast option. This parameter can be one or more of the following values.

Value	Meaning
BSF_ALLOWFW 0x00000080	Enables the recipient to set the foreground window while processing the message.
BSF_FLUSHDISK 0x00000004	Flushes the disk after each recipient processes the message.
BSF_FORCEIFHUNG 0x00000020	Continues to broadcast the message, even if the time-out period elapses or one of the recipients is not responding.
BSF_IGNORECURRENTTASK	Does not send the message to windows that belong to

0x00000002	the current task. This prevents an application from receiving its own message.
BSF_NOHANG 0x00000008	Forces a nonresponsive application to time out. If one of the recipients times out, do not continue broadcasting the message.
BSF_NOTIMEOUTIFNOTHUNG 0x00000040	Waits for a response to the message, as long as the recipient is not being unresponsive. Does not time out.
BSF_POSTMESSAGE 0x00000010	Posts the message. Do not use in combination with BSF_QUERY .
BSF_QUERY 0x00000001	Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE.
BSF_SENDNOTIFYMESSAGE 0x00000100	Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY .

[in, out, optional] lpInfo

Type: **LPDWORD**

A pointer to a variable that contains and receives information about the recipients of the message.

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is **NULL**, the function broadcasts to all components.

This parameter can be one or more of the following values.

Value	Meaning
BSM_ALLCOMPONENTS 0x00000000	Broadcast to all system components.
BSM_ALLDESKTOPS 0x00000010	Broadcast to all desktops. Requires the SE_TCB_NAME privilege.
BSM_APPLICATIONS 0x00000008	Broadcast to applications.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **long**

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is –1.

If the *dwFlags* parameter is **BSF_QUERY** and at least one recipient returned **BROADCAST_QUERY_DENY** to the corresponding message, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

If **BSF_QUERY** is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= **WM_USER**) to another process, you must do custom marshalling.

Examples

For an example, see [Terminating a Process](#).

ⓘ Note

The winuser.h header defines BroadcastSystemMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[BroadcastSystemMessageEx](#)

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

BSMINFO structure (winuser.h)

Article 03/13/2023

Contains information about a window that denied a request from [BroadcastSystemMessageEx](#).

Syntax

C++

```
typedef struct {
    UINT    cbSize;
    HDESK   hdesk;
    HWND    hwnd;
    LUID    luid;
} BSMINFO, *PBSMINFO;
```

Members

`cbSize`

Type: **UINT**

The size, in bytes, of this structure.

`hdesk`

Type: **HDESK**

A desktop handle to the window specified by `hwnd`. This value is returned only if [BroadcastSystemMessageEx](#) specifies **BSF_RETURNHDESK** and **BSF_QUERY**.

`hwnd`

Type: **HWND**

A handle to the window that denied the request. This value is returned if [BroadcastSystemMessageEx](#) specifies **BSF_QUERY**.

`luid`

Type: **LUID**

A locally unique identifier (LUID) for the window.

Requirements

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

See also

[BroadcastSystemMessageEx](#)

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

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

CalculatePopupWindowPosition function (winuser.h)

Article 10/13/2021

Calculates an appropriate pop-up window position using the specified anchor point, pop-up window size, flags, and the optional exclude rectangle. When the specified pop-up window size is smaller than the desktop window size, use the **CalculatePopupWindowPosition** function to ensure that the pop-up window is fully visible on the desktop window, regardless of the specified anchor point.

Syntax

C++

```
BOOL CalculatePopupWindowPosition(
    [in]          const POINT *anchorPoint,
    [in]          const SIZE   *windowSize,
    [in]          UINT        flags,
    [in, optional] RECT       *excludeRect,
    [out]         RECT       *popupWindowPosition
);
```

Parameters

[in] **anchorPoint**

Type: **const POINT***

The specified anchor point.

[in] **windowSize**

Type: **const SIZE***

The specified window size.

[in] **flags**

Type: **UINT**

Use one of the following flags to specify how the function positions the pop-up window horizontally and vertically. The flags are the same as the vertical and horizontal

positioning flags of the [TrackPopupMenuEx](#) function.

Use one of the following flags to specify how the function positions the pop-up window horizontally.

Value	Meaning
TPM_CENTERALIGN 0x0004L	Centers pop-up window horizontally relative to the coordinate specified by the anchorPoint->x parameter.
TPM_LEFTALIGN 0x0000L	Positions the pop-up window so that its left edge is aligned with the coordinate specified by the anchorPoint->x parameter.
TPM_RIGHTALIGN 0x0008L	Positions the pop-up window so that its right edge is aligned with the coordinate specified by the anchorPoint->x parameter.

Uses one of the following flags to specify how the function positions the pop-up window vertically.

Value	Meaning
TPM_BOTTOMALIGN 0x0020L	Positions the pop-up window so that its bottom edge is aligned with the coordinate specified by the anchorPoint->y parameter.
TPM_TOPALIGN 0x0000L	Positions the pop-up window so that its top edge is aligned with the coordinate specified by the anchorPoint->y parameter.
TPM_VCENTERALIGN 0x0010L	Centers the pop-up window vertically relative to the coordinate specified by the anchorPoint->y parameter.

Use one of the following flags to specify whether to accommodate horizontal or vertical alignment.

Value	Meaning
TPM_HORIZONTAL 0x0000L	If the pop-up window cannot be shown at the specified location without overlapping the excluded rectangle, the system tries to accommodate the requested horizontal alignment before the requested vertical alignment.
TPM_VERTICAL 0x0040L	If the pop-up window cannot be shown at the specified location without overlapping the excluded rectangle, the

system tries to accommodate the requested vertical alignment before the requested horizontal alignment.

The following flag is available starting with Windows 7.

Value	Meaning
TPM_WORKAREA 0x10000L	Restricts the pop-up window to within the work area. If this flag is not set, the pop-up window is restricted to the work area only if the input point is within the work area. For more information, see the rcWork and rcMonitor members of the MONITORINFO structure.

`[in, optional] excludeRect`

Type: [RECT*](#)

A pointer to a structure that specifies the exclude rectangle. It can be **NULL**.

`[out] popupWindowPosition`

Type: [RECT*](#)

A pointer to a structure that specifies the pop-up window position.

Return value

Type: [BOOL](#)

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

Remarks

TPM_WORKAREA is supported for the [TrackPopupMenu](#) and [TrackPopupMenuEx](#) functions.

Requirements

Minimum supported client

Windows 7 [desktop apps only]

Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Reference

[TrackPopupMenu](#)

[TrackPopupMenuEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

CallMsgFilterA function (winuser.h)

Article 02/09/2023

Passes the specified message and hook code to the hook procedures associated with the [WH_SYSMSGFILTER](#) and [WH_MSGFILTER](#) hooks. A [WH_SYSMSGFILTER](#) or [WH_MSGFILTER](#) hook procedure is an application-defined callback function that examines and, optionally, modifies messages for a dialog box, message box, menu, or scroll bar.

Syntax

C++

```
BOOL CallMsgFilterA(  
    [in] LPMSG lpMsg,  
    [in] int nCode  
>;
```

Parameters

[in] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that contains the message to be passed to the hook procedures.

[in] nCode

Type: [int](#)

An application-defined code used by the hook procedure to determine how to process the message. The code must not have the same value as system-defined hook codes (MSGF_ and HC_) associated with the [WH_SYSMSGFILTER](#) and [WH_MSGFILTER](#) hooks.

Return value

Type: [BOOL](#)

If the application should process the message further, the return value is zero.

If the application should not process the message further, the return value is nonzero.

Remarks

The system calls **CallMsgFilter** to enable applications to examine and control the flow of messages during internal processing of dialog boxes, message boxes, menus, and scroll bars, or when the user activates a different window by pressing the ALT+TAB key combination.

Install this hook procedure by using the [SetWindowsHookEx](#) function.

Examples

For an example, see [WH_MSGFILTER and WH_SYSMSGFILTER Hooks](#).

ⓘ Note

The winuser.h header defines CallMsgFilter 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Hooks](#)

[MSG](#)

[MessageProc](#)

[Reference](#)

[SetWindowsHookEx](#)

[SysMsgProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

CallMsgFilterW function (winuser.h)

Article 02/09/2023

Passes the specified message and hook code to the hook procedures associated with the [WH_SYSMSGFILTER](#) and [WH_MSGFILTER](#) hooks. A [WH_SYSMSGFILTER](#) or [WH_MSGFILTER](#) hook procedure is an application-defined callback function that examines and, optionally, modifies messages for a dialog box, message box, menu, or scroll bar.

Syntax

C++

```
BOOL CallMsgFilterW(
    [in] LPMSG lpMsg,
    [in] int    nCode
);
```

Parameters

[in] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that contains the message to be passed to the hook procedures.

[in] nCode

Type: [int](#)

An application-defined code used by the hook procedure to determine how to process the message. The code must not have the same value as system-defined hook codes (MSGF_ and HC_) associated with the [WH_SYSMSGFILTER](#) and [WH_MSGFILTER](#) hooks.

Return value

Type: [BOOL](#)

If the application should process the message further, the return value is zero.

If the application should not process the message further, the return value is nonzero.

Remarks

The system calls **CallMsgFilter** to enable applications to examine and control the flow of messages during internal processing of dialog boxes, message boxes, menus, and scroll bars, or when the user activates a different window by pressing the ALT+TAB key combination.

Install this hook procedure by using the [SetWindowsHookEx](#) function.

Examples

For an example, see [WH_MSGFILTER and WH_SYSMSGFILTER Hooks](#).

ⓘ Note

The winuser.h header defines CallMsgFilter 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Hooks](#)

[MSG](#)

[MessageProc](#)

[Reference](#)

[SetWindowsHookEx](#)

[SysMsgProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

CallNextHookEx function (winuser.h)

Article10/13/2021

Passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.

Syntax

C++

```
LRESULT CallNextHookEx(
    [in, optional] HHOOK hhk,
    [in]           int   nCode,
    [in]           WPARAM wParam,
    [in]           LPARAM lParam
);
```

Parameters

[in, optional] hhk

Type: **HHOOK**

This parameter is ignored.

[in] nCode

Type: **int**

The hook code passed to the current hook procedure. The next hook procedure uses this code to determine how to process the hook information.

[in] wParam

Type: **WPARAM**

The *wParam* value passed to the current hook procedure. The meaning of this parameter depends on the type of hook associated with the current hook chain.

[in] lParam

Type: **LPARAM**

The *lParam* value passed to the current hook procedure. The meaning of this parameter depends on the type of hook associated with the current hook chain.

Return value

Type: LRESULT

This value is returned by the next hook procedure in the chain. The current hook procedure must also return this value. The meaning of the return value depends on the hook type. For more information, see the descriptions of the individual hook procedures.

Remarks

Hook procedures are installed in chains for particular hook types. **CallNextHookEx** calls the next hook in the chain.

Calling **CallNextHookEx** is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call **CallNextHookEx** unless you absolutely need to prevent the notification from being seen by other applications.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Hooks](#)

[Reference](#)

[SetWindowsHookEx](#)

[UnhookWindowsHookEx function](#)

Feedback

Was this page helpful?

 Yes

 No

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

CallWindowProcA function (winuser.h)

Article 02/09/2023

Passes message information to the specified window procedure.

Syntax

C++

```
LRESULT CallWindowProcA(
    [in] WNDPROC lpPrevWndFunc,
    [in] HWND     hWnd,
    [in] UINT      Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] lpPrevWndFunc

Type: **WNDPROC**

The previous window procedure. If this value is obtained by calling the [GetWindowLong](#) function with the *nIndex* parameter set to **GWL_WNDPROC** or **DWL_DLGPROC**, it is actually either the address of a window or dialog box procedure, or a special internal value meaningful only to **CallWindowProc**.

[in] hWnd

Type: **HWND**

A handle to the window procedure to receive the message.

[in] Msg

Type: **UINT**

The message.

[in] wParam

Type: **WPARAM**

Additional message-specific information. The contents of this parameter depend on the value of the *Msg* parameter.

[in] lParam

Type: **LPARAM**

Additional message-specific information. The contents of this parameter depend on the value of the *Msg* parameter.

Return value

Type: **LRESULT**

The return value specifies the result of the message processing and depends on the message sent.

Remarks

Use the **CallWindowProc** function for window subclassing. Usually, all windows with the same class share one window procedure. A subclass is a window or set of windows with the same class whose messages are intercepted and processed by another window procedure (or procedures) before being passed to the window procedure of the class.

The **SetWindowLong** function creates the subclass by changing the window procedure associated with a particular window, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling **CallWindowProc**. This allows the application to create a chain of window procedures.

If **STRICT** is defined, the *lpPrevWndFunc* parameter has the data type **WNDPROC**. The **WNDPROC** type is declared as follows:

syntax

```
LRESULT (CALLBACK* WNDPROC) (HWND, UINT, WPARAM, LPARAM);
```

If **STRICT** is not defined, the *lpPrevWndFunc* parameter has the data type **FARPROC**. The **FARPROC** type is declared as follows:

syntax

```
int (FAR WINAPI * FARPROC) ()
```

In C, the **FARPROC** declaration indicates a callback function that has an unspecified parameter list. In C++, however, the empty parameter list in the declaration indicates that a function has no parameters. This subtle distinction can break careless code.

Following is one way to handle this situation:

syntax

```
#ifdef STRICT  
    WNDPROC MyWindowProcedure  
#else  
    FARPROC MyWindowProcedure  
#endif  
...  
lResult = CallWindowProc(MyWindowProcedure, ...);
```

For further information about functions declared with empty argument lists, refer to *The C++ Programming Language, Second Edition*, by Bjarne Stroustrup.

The **CallWindowProc** function handles Unicode-to-ANSI conversion. You cannot take advantage of this conversion if you call the window procedure directly.

Examples

For an example, see [Subclassing a Window](#)

ⓘ Note

The winuser.h header defines CallWindowProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[GetWindowLong](#)

[Reference](#)

[SetClassLong](#)

[SetWindowLong](#)

[Window Procedures](#)

Feedback

Was this page helpful?

 Yes

 No

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

CallWindowProcW function (winuser.h)

Article 02/09/2023

Passes message information to the specified window procedure.

Syntax

C++

```
LRESULT CallWindowProcW(
    [in] WNDPROC lpPrevWndFunc,
    [in] HWND     hWnd,
    [in] UINT      Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] lpPrevWndFunc

Type: **WNDPROC**

The previous window procedure. If this value is obtained by calling the [GetWindowLong](#) function with the *nIndex* parameter set to **GWL_WNDPROC** or **DWL_DLGPROC**, it is actually either the address of a window or dialog box procedure, or a special internal value meaningful only to **CallWindowProc**.

[in] hWnd

Type: **HWND**

A handle to the window procedure to receive the message.

[in] Msg

Type: **UINT**

The message.

[in] wParam

Type: **WPARAM**

Additional message-specific information. The contents of this parameter depend on the value of the *Msg* parameter.

[in] lParam

Type: **LPARAM**

Additional message-specific information. The contents of this parameter depend on the value of the *Msg* parameter.

Return value

Type: **LRESULT**

The return value specifies the result of the message processing and depends on the message sent.

Remarks

Use the **CallWindowProc** function for window subclassing. Usually, all windows with the same class share one window procedure. A subclass is a window or set of windows with the same class whose messages are intercepted and processed by another window procedure (or procedures) before being passed to the window procedure of the class.

The **SetWindowLong** function creates the subclass by changing the window procedure associated with a particular window, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling **CallWindowProc**. This allows the application to create a chain of window procedures.

If **STRICT** is defined, the *lpPrevWndFunc* parameter has the data type **WNDPROC**. The **WNDPROC** type is declared as follows:

syntax

```
LRESULT (CALLBACK* WNDPROC) (HWND, UINT, WPARAM, LPARAM);
```

If **STRICT** is not defined, the *lpPrevWndFunc* parameter has the data type **FARPROC**. The **FARPROC** type is declared as follows:

syntax

```
int (FAR WINAPI * FARPROC) ()
```

In C, the **FARPROC** declaration indicates a callback function that has an unspecified parameter list. In C++, however, the empty parameter list in the declaration indicates that a function has no parameters. This subtle distinction can break careless code.

Following is one way to handle this situation:

syntax

```
#ifdef STRICT  
    WNDPROC MyWindowProcedure  
#else  
    FARPROC MyWindowProcedure  
#endif  
...  
lResult = CallWindowProc(MyWindowProcedure, ...);
```

For further information about functions declared with empty argument lists, refer to *The C++ Programming Language, Second Edition*, by Bjarne Stroustrup.

The **CallWindowProc** function handles Unicode-to-ANSI conversion. You cannot take advantage of this conversion if you call the window procedure directly.

Examples

For an example, see [Subclassing a Window](#)

ⓘ Note

The winuser.h header defines CallWindowProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[GetWindowLong](#)

[Reference](#)

[SetClassLong](#)

[SetWindowLong](#)

[Window Procedures](#)

Feedback

Was this page helpful?

 Yes

 No

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

CascadeWindows function (winuser.h)

Article 10/13/2021

Cascades the specified child windows of the specified parent window.

Syntax

C++

```
WORD CascadeWindows(
    [in, optional] HWND      hwndParent,
    [in]          UINT       wHow,
    [in, optional] const RECT *lpRect,
    [in]          UINT       cKids,
    [in, optional] const HWND *lpKids
);
```

Parameters

[in, optional] hwndParent

Type: **HWND**

A handle to the parent window. If this parameter is **NULL**, the desktop window is assumed.

[in] wHow

Type: **UINT**

A cascade flag. This parameter can be one or more of the following values.

Value	Meaning
MDITILE_SKIPDISABLED 0x0002	Prevents disabled MDI child windows from being cascaded.
MDITILE_ZORDER 0x0004	Arranges the windows in Z order. If this value is not specified, the windows are arranged using the order specified in the <i>lpKids</i> array.

[in, optional] lpRect

Type: **const RECT***

A pointer to a structure that specifies the rectangular area, in client coordinates, within which the windows are arranged. This parameter can be **NULL**, in which case the client area of the parent window is used.

[in] cKids

Type: **UINT**

The number of elements in the array specified by the *lpKids* parameter. This parameter is ignored if *lpKids* is **NULL**.

[in, optional] lpKids

Type: **const HWND***

An array of handles to the child windows to arrange. If a specified child window is a top-level window with the style **WS_EX_TOPMOST** or **WS_EX_TOOLWINDOW**, the child window is not arranged. If this parameter is **NULL**, all child windows of the specified parent window (or of the desktop window) are arranged.

Return value

Type: **WORD**

If the function succeeds, the return value is the number of windows arranged.

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

Remarks

By default, **CascadeWindows** arranges the windows in the order provided by the *lpKids* array, but preserves the [Z-Order](#). If you specify the **MDITILE_ZORDER** flag, **CascadeWindows** arranges the windows in Z order.

Calling **CascadeWindows** causes all maximized windows to be restored to their previous size.

Requirements

Minimum supported client

Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

CBT_CREATEWND structure (winuser.h)

Article 07/27/2022

Contains information passed to a **WH_CBT** hook procedure, [CBTProc](#), before a window is created.

Syntax

C++

```
typedef struct tagCBT_CREATEWNDA {
    struct tagCREATESTRUCTA *lpcs;
    HWND                 hwndInsertAfter;
} CBT_CREATEWNDA, *LPCBT_CREATEWNDA;
```

Members

`lpcs`

Type: [LPCREATESTRUCT](#)

A pointer to a [CREATESTRUCT](#) structure that contains initialization parameters for the window about to be created.

`hwndInsertAfter`

Type: [HWND](#)

A handle to the window whose position in the Z order precedes that of the window being created. This member can also be **NULL**.

Remarks

Note

The winuser.h header defines **CBT_CREATEWND** 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	winuser.h (include Windows.h)

See also

[CBTProc](#)

[CREATESTRUCT](#)

[Conceptual](#)

[Hooks](#)

[Reference](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

CBT_CREATEWNDW structure (winuser.h)

Article 07/27/2022

Contains information passed to a **WH_CBT** hook procedure, [CBTProc](#), before a window is created.

Syntax

C++

```
typedef struct tagCBT_CREATEWNDW {
    struct tagCREATESTRUCTW *lpcs;
    HWND                 hwndInsertAfter;
} CBT_CREATEWNDW, *LPCBT_CREATEWNDW;
```

Members

`lpcs`

Type: [LPCREATESTRUCT](#)

A pointer to a [CREATESTRUCT](#) structure that contains initialization parameters for the window about to be created.

`hwndInsertAfter`

Type: [HWND](#)

A handle to the window whose position in the Z order precedes that of the window being created. This member can also be **NULL**.

Remarks

Note

The winuser.h header defines **CBT_CREATEWND** 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	winuser.h (include Windows.h)

See also

[CBTProc](#)

[CREATESTRUCT](#)

[Conceptual](#)

[Hooks](#)

[Reference](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

CBTACTIVATESTRUCT structure (winuser.h)

Article 04/02/2021

Contains information passed to a **WH_CBT** hook procedure, [CBTProc](#), before a window is activated.

Syntax

C++

```
typedef struct tagCBTACTIVATESTRUCT {
    BOOL fMouse;
    HWND hWndActive;
} CBTACTIVATESTRUCT, *LPCBTACTIVATESTRUCT;
```

Members

fMouse

Type: **BOOL**

This member is **TRUE** if a mouse click is causing the activation or **FALSE** if it is not.

hWndActive

Type: **HWND**

A handle to the active window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[CBTProc](#)

[Conceptual](#)

[Hooks](#)

[Reference](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

CHANGEFILTERSTRUCT structure (winuser.h)

Article 04/02/2021

Contains extended result information obtained by calling the [ChangeWindowMessageFilterEx](#) function.

Syntax

C++

```
typedef struct tagCHANGEFILTERSTRUCT {
    DWORD cbSize;
    DWORD ExtStatus;
} CHANGEFILTERSTRUCT, *PCHANGEFILTERSTRUCT;
```

Members

`cbSize`

Type: `DWORD`

The size of the structure, in bytes. Must be set to `sizeof(CHANGEFILTERSTRUCT)`, otherwise the function fails with `ERROR_INVALID_PARAMETER`.

`ExtStatus`

Type: `DWORD`

If the function succeeds, this field contains one of the following values.

Value	Meaning
<code>MSGFLTINFO_NONE</code> 0	See the Remarks section. Applies to <code>MSGFLT_ALLOW</code> and <code>MSGFLT_DISALLOW</code> .
<code>MSGFLTINFO_ALLOWED_HIGHER</code> 3	The message is allowed at a scope higher than the window. Applies to <code>MSGFLT_DISALLOW</code> .
<code>MSGFLTINFO_ALREADYALLOWED_FORWND</code> 1	The message has already been allowed by this window's message filter, and the function thus succeeded with no change to the window's message filter. Applies to <code>MSGFLT_ALLOW</code> .

MSGFLTINFO_ALREADYDISALLOWED_FORWND	The message has already been blocked by this window's message filter, and the function thus succeeded with no change to the window's message filter. Applies to MSGFLT_DISALLOW .
2	

Remarks

Certain messages whose value is smaller than **WM_USER** are required to pass through the filter, regardless of the filter setting. There will be no effect when you attempt to use this function to allow or block such messages.

An application may use the [ChangeWindowMessageFilter](#) function to allow or block a message in a process-wide manner. If the message is allowed by either the process message filter or the window message filter, it will be delivered to the window.

The following table lists the possible values returned in **ExtStatus**.

Message already allowed at higher scope	Message already allowed by window's message filter	Operation requested	Indicator returned in ExtStatus on success
No	No	MSGFLT_ALLOW	MSGFLTINFO_NONE
No	No	MSGFLT_DISALLOW	MSGFLTINFO_ALREADYDISALLOWED_FORWND
No	No	MSGFLT_RESET	MSGFLTINFO_NONE
No	Yes	MSGFLT_ALLOW	MSGFLTINFO_ALREADYALLOWED_FORWND
No	Yes	MSGFLT_DISALLOW	MSGFLTINFO_NONE
No	Yes	MSGFLT_RESET	MSGFLTINFO_NONE
Yes	No	MSGFLT_ALLOW	MSGFLTINFO_NONE
Yes	No	MSGFLT_DISALLOW	MSGFLTINFO_ALLOWED_HIGHER
Yes	No	MSGFLT_RESET	MSGFLTINFO_NONE
Yes	Yes	MSGFLT_ALLOW	MSGFLTINFO_ALREADYALLOWED_FORWND
Yes	Yes	MSGFLT_DISALLOW	MSGFLTINFO_ALLOWED_HIGHER
Yes	Yes	MSGFLT_RESET	MSGFLTINFO_NONE

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[ChangeWindowMessageFilterEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

ChangeWindowMessageFilter function (winuser.h)

Article 10/13/2021

[Using the **ChangeWindowMessageFilter** function is not recommended, as it has process-wide scope. Instead, use the [ChangeWindowMessageFilterEx](#) function to control access to specific windows as needed. **ChangeWindowMessageFilter** may not be supported in future versions of Windows.]

Adds or removes a message from the User Interface Privilege Isolation (UIPI) message filter.

Syntax

C++

```
BOOL ChangeWindowMessageFilter(
    [in] UINT message,
    [in] DWORD dwFlag
);
```

Parameters

[in] message

Type: **UINT**

The message to add to or remove from the filter.

[in] dwFlag

Type: **DWORD**

The action to be performed. One of the following values.

Value	Meaning
MSGFLT_ADD 1	Adds the <i>message</i> to the filter. This has the effect of allowing the message to be received.
MSGFLT_REMOVE 2	Removes the <i>message</i> from the filter. This has the effect of blocking the message.

Return value

Type: **BOOL**

TRUE if successful; otherwise, **FALSE**. To get extended error information, call [GetLastError](#).

Note A message can be successfully removed from the filter, but that is not a guarantee that the message will be blocked. See the Remarks section for more details.

Remarks

UIPI is a security feature that prevents messages from being received from a lower integrity level sender. All such messages with a value above **WM_USER** are blocked by default. The filter, somewhat contrary to intuition, is a list of messages that are allowed through. Therefore, adding a message to the filter allows that message to be received from a lower integrity sender, while removing a message blocks that message from being received.

Certain messages with a value less than **WM_USER** are required to pass through the filter regardless of the filter setting. You can call this function to remove one of those messages from the filter and it will return **TRUE**. However, the message will still be received by the calling process.

Processes at or below **SECURITY_MANDATORY_LOW_RID** are not allowed to change the filter. If those processes call this function, it will fail.

For more information on integrity levels, see [Understanding and Working in Protected Mode Internet Explorer](#).

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

Feedback

Was this page helpful?

 Yes

 No

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

ChangeWindowMessageFilterEx function (winuser.h)

Article 10/13/2021

Modifies the User Interface Privilege Isolation (UIPI) message filter for a specified window.

Syntax

C++

```
BOOL ChangeWindowMessageFilterEx(
    [in]             HWND      hwnd,
    [in]             UINT      message,
    [in]             DWORD     action,
    [in, out, optional] PCHANGEFILTERSTRUCT pChangeFilterStruct
);
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the window whose UIPI message filter is to be modified.

[in] `message`

Type: **UINT**

The message that the message filter allows through or blocks.

[in] `action`

Type: **DWORD**

The action to be performed, and can take one of the following values:

Value	Meaning
MSGFLT_ALLOW 1	Allows the message through the filter. This enables the message to be received by <i>hWnd</i> , regardless of the

	source of the message, even it comes from a lower privileged process.
MSGFLT_DISALLOW 2	Blocks the message to be delivered to <i>hWnd</i> if it comes from a lower privileged process, unless the message is allowed process-wide by using the ChangeWindowMessageFilter function or globally.
MSGFLT_RESET 0	Resets the window message filter for <i>hWnd</i> to the default. Any message allowed globally or process-wide will get through, but any message not included in those two categories, and which comes from a lower privileged process, will be blocked.

[in, out, optional] *pChangeFilterStruct*

Type: **PCHANGEFILTERSTRUCT**

Optional pointer to a [CHANGEFILTERSTRUCT](#) structure.

Return value

Type: **BOOL**

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

Remarks

UIPI is a security feature that prevents messages from being received from a lower-integrity-level sender. You can use this function to allow specific messages to be delivered to a window even if the message originates from a process at a lower integrity level. Unlike the [ChangeWindowMessageFilter](#) function, which controls the process message filter, the [ChangeWindowMessageFilterEx](#) function controls the window message filter.

An application may use the [ChangeWindowMessageFilter](#) function to allow or block a message in a process-wide manner. If the message is allowed by either the process message filter or the window message filter, it will be delivered to the window.

Note that processes at or below **SECURITY_MANDATORY_LOW_RID** are not allowed to change the message filter. If those processes call this function, it will fail and generate the extended error code, **ERROR_ACCESS_DENIED**.

Certain messages whose value is smaller than WM_USER are required to be passed through the filter, regardless of the filter setting. There will be no effect when you attempt to use this function to allow or block such messages.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-gui-l1-3-0 (introduced in Windows 10, version 10.0.10240)

See also

[ChangeWindowMessageFilter](#)

[Conceptual](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ChildWindowFromPoint function (winuser.h)

Article 11/19/2022

Determines which, if any, of the child windows belonging to a parent window contains the specified point. The search is restricted to immediate child windows. Grandchildren, and deeper descendant windows are not searched.

To skip certain child windows, use the [ChildWindowFromPointEx](#) function.

Syntax

C++

```
HWND ChildWindowFromPoint(  
    [in] HWND hWndParent,  
    [in] POINT Point  
) ;
```

Parameters

[in] hWndParent

Type: **HWND**

A handle to the parent window.

[in] Point

Type: **POINT**

A structure that defines the client coordinates, relative to *hWndParent*, of the point to be checked.

Return value

Type: **HWND**

The return value is a handle to the child window that contains the point, even if the child window is hidden or disabled. If the point lies outside the parent window, the return

value is **NULL**. If the point is within the parent window but not within any child window, the return value is a handle to the parent window.

Remarks

The system maintains an internal list, containing the handles of the child windows associated with a parent window. The order of the handles in the list depends on the Z order of the child windows. If more than one child window contains the specified point, the system returns a handle to the first window in the list that contains the point.

ChildWindowFromPoint treats an **HTTRANSPARENT** area of a standard control the same as other parts of the control. In contrast, **RealChildWindowFromPoint** treats an **HTTRANSPARENT** area differently; it returns the child window behind a transparent area of a control. For example, if the point is in a transparent area of a groupbox, **ChildWindowFromPoint** returns the groupbox while **RealChildWindowFromPoint** returns the child window behind the groupbox. However, both APIs return a static field, even though it, too, returns **HTTRANSPARENT**.

Examples

For an example, see "Creating a Combo Box Toolbar" in [Using Combo Boxes](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[ChildWindowFromPointEx](#)

[Conceptual](#)

[Other Resources](#)

[POINT](#)

[RealChildWindowFromPoint](#)

[Reference](#)

[WindowFromPoint](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ChildWindowFromPointEx function (winuser.h)

Article 11/19/2022

Determines which, if any, of the child windows belonging to the specified parent window contains the specified point. The function can ignore invisible, disabled, and transparent child windows. The search is restricted to immediate child windows. Grandchildren and deeper descendants are not searched.

Syntax

C++

```
HWND ChildWindowFromPointEx(  
    [in] HWND  hwnd,  
    [in] POINT pt,  
    [in] UINT   flags  
) ;
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the parent window.

[in] `pt`

Type: **POINT**

A structure that defines the client coordinates (relative to *hwndParent*) of the point to be checked.

[in] `flags`

Type: **UINT**

The child windows to be skipped. This parameter can be one or more of the following values.

Value	Meaning
-------	---------

CWP_ALL 0x0000	Does not skip any child windows
CWP_SKIPDISABLED 0x0002	Skips disabled child windows
CWP_SKIPINVISIBLE 0x0001	Skips invisible child windows
CWP_SKIPTRANSPARENT 0x0004	Skips transparent child windows

Return value

Type: **HWND**

The return value is a handle to the first child window that contains the point and meets the criteria specified by *uFlags*. If the point is within the parent window but not within any child window that meets the criteria, the return value is a handle to the parent window. If the point lies outside the parent window or if the function fails, the return value is **NULL**.

Remarks

The system maintains an internal list that contains the handles of the child windows associated with a parent window. The order of the handles in the list depends on the Z order of the child windows. If more than one child window contains the specified point, the system returns a handle to the first window in the list that contains the point and meets the criteria specified by *uFlags*.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[Other Resources](#)

[POINT](#)

[Reference](#)

[WindowFromPoint](#)

[Windows](#)

Feedback

Was this page helpful?

[!\[\]\(34dd7a5be9e234bae713d7ccebd621f9_img.jpg\) Yes](#)

[!\[\]\(d3fecaeb4806eba227c9ab054df5b8bf_img.jpg\) No](#)

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

CLIENTCREATESTRUCT structure (winuser.h)

Article04/02/2021

Contains information about the menu and first multiple-document interface (MDI) child window of an MDI client window. An application passes a pointer to this structure as the *lpParam* parameter of the [CreateWindow](#) function when creating an MDI client window.

Syntax

C++

```
typedef struct tagCLIENTCREATESTRUCT {
    HANDLE hWindowMenu;
    UINT    idFirstChild;
} CLIENTCREATESTRUCT, *LPCCLIENTCREATESTRUCT;
```

Members

`hWindowMenu`

Type: **HANDLE**

A handle to the MDI application's window menu. An MDI application can retrieve this handle from the menu of the MDI frame window by using the [GetSubMenu](#) function.

`idFirstChild`

Type: **UINT**

The child window identifier of the first MDI child window created. The system increments the identifier for each additional MDI child window the application creates, and reassigns identifiers when the application destroys a window to keep the range of identifiers contiguous. These identifiers are used in [WM_COMMAND](#) messages sent to the application's MDI frame window when a child window is chosen from the window menu; they should not conflict with any other command identifiers.

Remarks

When the MDI client window is created by calling [CreateWindow](#), the system sends a **WM_CREATE** message to the window. The *lParam* parameter of **WM_CREATE** contains a pointer to a [CREATESTRUCT](#) structure. The **lpCreateParams** member of this structure contains a pointer to a [CLIENTCREATESTRUCT](#) structure.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[About the Multiple Document Interface](#)

[Conceptual](#)

[CreateWindow](#)

[GetSubMenu](#)

[MDICREATESTRUCT](#)

[Reference](#)

[WM_COMMAND](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

CloseWindow function (winuser.h)

Article10/13/2021

Minimizes (but does not destroy) the specified window.

Syntax

C++

```
BOOL CloseWindow(  
    [in] HWND hWnd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be minimized.

Return value

Type: **BOOL**

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 destroy a window, an application must use the [DestroyWindow](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
--------------------------	---

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[ArrangeIconicWindows](#)

[Conceptual](#)

[DestroyWindow](#)

[ISIconic](#)

[OpenIcon](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateMDIWindowA function (winuser.h)

Article 02/09/2023

Creates a multiple-document interface (MDI) child window.

Syntax

C++

```
HWND CreateMDIWindowA(
    [in]          LPCSTR    lpClassName,
    [in]          LPCSTR    lpWindowName,
    [in]          DWORD     dwStyle,
    [in]          int       X,
    [in]          int       Y,
    [in]          int       nWidth,
    [in]          int       nHeight,
    [in, optional] HWND     hWndParent,
    [in, optional] HINSTANCE hInstance,
    [in]          LPARAM    lParam
);
```

Parameters

[in] lpClassName

Type: **LPCTSTR**

The window class of the MDI child window. The class name must have been registered by a call to the [RegisterClassEx](#) function.

[in] lpWindowName

Type: **LPCTSTR**

The window name. The system displays the name in the title bar of the child window.

[in] dwStyle

Type: **DWORD**

The style of the MDI child window. If the MDI client window is created with the **MDIS_ALLCHILDSTYLES** window style, this parameter can be any combination of the window styles listed in the [Window Styles](#) page. Otherwise, this parameter is limited to one or more of the following values.

Value	Meaning
WS_MINIMIZE 0x20000000L	Creates an MDI child window that is initially minimized.
WS_MAXIMIZE 0x01000000L	Creates an MDI child window that is initially maximized.
WS_HSCROLL 0x00100000L	Creates an MDI child window that has a horizontal scroll bar.
WS_VSCROLL 0x00200000L	Creates an MDI child window that has a vertical scroll bar.

[in] X

Type: int

The initial horizontal position, in client coordinates, of the MDI child window. If this parameter is **CW_USEDEFAULT** ((int)0x80000000), the MDI child window is assigned the default horizontal position.

[in] Y

Type: int

The initial vertical position, in client coordinates, of the MDI child window. If this parameter is **CW_USEDEFAULT**, the MDI child window is assigned the default vertical position.

[in] nWidth

Type: int

The initial width, in device units, of the MDI child window. If this parameter is **CW_USEDEFAULT**, the MDI child window is assigned the default width.

[in] nHeight

Type: int

The initial height, in device units, of the MDI child window. If this parameter is set to **CW_USEDEFAULT**, the MDI child window is assigned the default height.

[in, optional] `hWndParent`

Type: **HWND**

A handle to the MDI client window that will be the parent of the new MDI child window.

[in, optional] `hInstance`

Type: **HINSTANCE**

A handle to the instance of the application creating the MDI child window.

[in] `lParam`

Type: **LPARAM**

An application-defined value.

Return value

Type: **HWND**

If the function succeeds, the return value is the handle to the created window.

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

Remarks

Note

The winuser.h header defines `CreateMDIWindow` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[CreateWindow](#)

[Multiple Document Interface](#)

Reference

[RegisterClassEx](#)

[WM_MDICREATE](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateMDIWindowW function (winuser.h)

Article 02/09/2023

Creates a multiple-document interface (MDI) child window.

Syntax

C++

```
HWND CreateMDIWindowW(
    [in]          LPCWSTR    lpClassName,
    [in]          LPCWSTR    lpWindowName,
    [in]          DWORD      dwStyle,
    [in]          int        X,
    [in]          int        Y,
    [in]          int        nWidth,
    [in]          int        nHeight,
    [in, optional] HWND      hWndParent,
    [in, optional] HINSTANCE hInstance,
    [in]          LPARAM     lParam
);
```

Parameters

[in] lpClassName

Type: **LPCTSTR**

The window class of the MDI child window. The class name must have been registered by a call to the [RegisterClassEx](#) function.

[in] lpWindowName

Type: **LPCTSTR**

The window name. The system displays the name in the title bar of the child window.

[in] dwStyle

Type: **DWORD**

The style of the MDI child window. If the MDI client window is created with the **MDIS_ALLCHILDSTYLES** window style, this parameter can be any combination of the window styles listed in the [Window Styles](#) page. Otherwise, this parameter is limited to one or more of the following values.

Value	Meaning
WS_MINIMIZE 0x20000000L	Creates an MDI child window that is initially minimized.
WS_MAXIMIZE 0x01000000L	Creates an MDI child window that is initially maximized.
WS_HSCROLL 0x00100000L	Creates an MDI child window that has a horizontal scroll bar.
WS_VSCROLL 0x00200000L	Creates an MDI child window that has a vertical scroll bar.

[in] X

Type: int

The initial horizontal position, in client coordinates, of the MDI child window. If this parameter is **CW_USEDEFAULT** ((int)0x80000000), the MDI child window is assigned the default horizontal position.

[in] Y

Type: int

The initial vertical position, in client coordinates, of the MDI child window. If this parameter is **CW_USEDEFAULT**, the MDI child window is assigned the default vertical position.

[in] nWidth

Type: int

The initial width, in device units, of the MDI child window. If this parameter is **CW_USEDEFAULT**, the MDI child window is assigned the default width.

[in] nHeight

Type: int

The initial height, in device units, of the MDI child window. If this parameter is set to **CW_USEDEFAULT**, the MDI child window is assigned the default height.

[in, optional] `hWndParent`

Type: **HWND**

A handle to the MDI client window that will be the parent of the new MDI child window.

[in, optional] `hInstance`

Type: **HINSTANCE**

A handle to the instance of the application creating the MDI child window.

[in] `lParam`

Type: **LPARAM**

An application-defined value.

Return value

Type: **HWND**

If the function succeeds, the return value is the handle to the created window.

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

Remarks

Note

The winuser.h header defines `CreateMDIWindow` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[CreateWindow](#)

[Multiple Document Interface](#)

Reference

[RegisterClassEx](#)

[WM_MDICREATE](#)

Feedback

Was this page helpful?

 Yes

 No

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

CREATESTRUCTA structure (winuser.h)

Article07/27/2022

Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the [CreateWindowEx](#) function.

Syntax

C++

```
typedef struct tagCREATESTRUCTA {
    LPVOID    lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCSTR    lpszName;
    LPCSTR    lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCTA, *LPCREATESTRUCTA;
```

Members

`lpCreateParams`

Type: **LPVOID**

Contains additional data which may be used to create the window. If the window is being created as a result of a call to the [CreateWindow](#) or [CreateWindowEx](#) function, this member contains the value of the *lpParam* parameter specified in the function call.

If the window being created is a MDI client window, this member contains a pointer to a [CLIENTCREATESTRUCT](#) structure. If the window being created is a MDI child window, this member contains a pointer to an [MDICREATESTRUCT](#) structure.

If the window is being created from a dialog template, this member is the address of a **SHORT** value that specifies the size, in bytes, of the window creation data. The value is immediately followed by the creation data. For more information, see the following Remarks section.

`hInstance`

Type: **HINSTANCE**

A handle to the module that owns the new window.

`hMenu`

Type: **HMENU**

A handle to the menu to be used by the new window.

`hwndParent`

Type: **HWND**

A handle to the parent window, if the window is a child window. If the window is owned, this member identifies the owner window. If the window is not a child or owned window, this member is **NULL**.

`cy`

Type: **int**

The height of the new window, in pixels.

`cx`

Type: **int**

The width of the new window, in pixels.

`y`

Type: **int**

The y-coordinate of the upper left corner of the new window. If the new window is a child window, coordinates are relative to the parent window. Otherwise, the coordinates are relative to the screen origin.

`x`

Type: **int**

The x-coordinate of the upper left corner of the new window. If the new window is a child window, coordinates are relative to the parent window. Otherwise, the coordinates are relative to the screen origin.

`style`

Type: **LONG**

The style for the new window. For a list of possible values, see [Window Styles](#).

`lpszName`

Type: **LPCTSTR**

The name of the new window.

`lpszClass`

Type: **LPCTSTR**

A pointer to a null-terminated string or an atom that specifies the class name of the new window.

`dwExStyle`

Type: **DWORD**

The extended window style for the new window. For a list of possible values, see [Extended Window Styles](#).

Remarks

Because the `lpszClass` member can contain a pointer to a local (and thus inaccessible) atom, do not obtain the class name by using this member. Use the [GetClassName](#) function instead.

You should access the data represented by the `lpCreateParams` member using a pointer that has been declared using the **UNALIGNED** type, because the pointer may not be **DWORD** aligned. This is demonstrated in the following example:

```
typedef struct tagMyData
{
    // Define creation data here.
} MYDATA;

typedef struct tagMyDlgData
{
    SHORT    cbExtra;
    MYDATA   myData;
```

```
} MYDLGDATA, UNALIGNED *PMYDLGDATA;  
  
PMYDLGDATA pMyDlgdata = (PMYDLGDATA) (((LPCREATESTRUCT) lParam)-  
>lpCreateParams);
```

ⓘ Note

The winuser.h header defines CREATESTRUCT 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	winuser.h (include Windows.h)

See also

[About the Multiple Document Interface](#)

[Conceptual](#)

[CreateWindow](#)

[CreateWindowEx](#)

[MDICREATESTRUCT](#)

[Reference](#)

[Windows](#)

Feedback



Was this page helpful? [!\[\]\(a83dd616dc676ff946d2788a1840e496_img.jpg\) Yes](#) [!\[\]\(ade506b31e3c48ed793b1c4ba19040fa_img.jpg\) No](#)

Get help at Microsoft Q&A

CREATESTRUCTW structure (winuser.h)

Article07/27/2022

Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the [CreateWindowEx](#) function.

Syntax

C++

```
typedef struct tagCREATESTRUCTW {
    LPVOID    lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCWSTR   lpszName;
    LPCWSTR   lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCTW, *LPCREATESTRUCTW;
```

Members

`lpCreateParams`

Type: **LPVOID**

Contains additional data which may be used to create the window. If the window is being created as a result of a call to the [CreateWindow](#) or [CreateWindowEx](#) function, this member contains the value of the *lpParam* parameter specified in the function call.

If the window being created is a MDI client window, this member contains a pointer to a [CLIENTCREATESTRUCT](#) structure. If the window being created is a MDI child window, this member contains a pointer to an [MDICREATESTRUCT](#) structure.

If the window is being created from a dialog template, this member is the address of a **SHORT** value that specifies the size, in bytes, of the window creation data. The value is immediately followed by the creation data. For more information, see the following Remarks section.

`hInstance`

Type: **HINSTANCE**

A handle to the module that owns the new window.

`hMenu`

Type: **HMENU**

A handle to the menu to be used by the new window.

`hwndParent`

Type: **HWND**

A handle to the parent window, if the window is a child window. If the window is owned, this member identifies the owner window. If the window is not a child or owned window, this member is **NULL**.

`cy`

Type: **int**

The height of the new window, in pixels.

`cx`

Type: **int**

The width of the new window, in pixels.

`y`

Type: **int**

The y-coordinate of the upper left corner of the new window. If the new window is a child window, coordinates are relative to the parent window. Otherwise, the coordinates are relative to the screen origin.

`x`

Type: **int**

The x-coordinate of the upper left corner of the new window. If the new window is a child window, coordinates are relative to the parent window. Otherwise, the coordinates are relative to the screen origin.

`style`

Type: **LONG**

The style for the new window. For a list of possible values, see [Window Styles](#).

`lpszName`

Type: **LPCTSTR**

The name of the new window.

`lpszClass`

Type: **LPCTSTR**

A pointer to a null-terminated string or an atom that specifies the class name of the new window.

`dwExStyle`

Type: **DWORD**

The extended window style for the new window. For a list of possible values, see [Extended Window Styles](#).

Remarks

Because the `lpszClass` member can contain a pointer to a local (and thus inaccessible) atom, do not obtain the class name by using this member. Use the [GetClassName](#) function instead.

You should access the data represented by the `lpCreateParams` member using a pointer that has been declared using the **UNALIGNED** type, because the pointer may not be **DWORD** aligned. This is demonstrated in the following example:

```
typedef struct tagMyData
{
    // Define creation data here.
} MYDATA;

typedef struct tagMyDlgData
{
    SHORT    cbExtra;
    MYDATA   myData;
```

```
} MYDLGDATA, UNALIGNED *PMYDLGDATA;  
  
PMYDLGDATA pMyDlgdata = (PMYDLGDATA) (((LPCREATESTRUCT) lParam)-  
>lpCreateParams);
```

ⓘ Note

The winuser.h header defines CREATESTRUCT 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	winuser.h (include Windows.h)

See also

[About the Multiple Document Interface](#)

[Conceptual](#)

[CreateWindow](#)

[CreateWindowEx](#)

[MDICREATESTRUCT](#)

[Reference](#)

[Windows](#)

Feedback



Was this page helpful? [!\[\]\(ad3f3c89b7dc5dcd010a44b65dd60648_img.jpg\) Yes](#) [!\[\]\(245e56c1eb332538dd370a93459fd938_img.jpg\) No](#)

Get help at Microsoft Q&A

CreateWindowA macro (winuser.h)

Article02/09/2023

Creates an overlapped, pop-up, or child window. It specifies the window class, window title, window style, and (optionally) the initial position and size of the window. The function also specifies the window's parent or owner, if any, and the window's menu.

To use extended window styles in addition to the styles supported by [CreateWindow](#), use the [CreateWindowEx](#) function.

Syntax

C++

```
void CreateWindowA(
    [in, optional]  lpClassName,
    [in, optional]  lpWindowName,
    [in]            dwStyle,
    [in]            x,
    [in]            y,
    [in]            nWidth,
    [in]            nHeight,
    [in, optional]  hWndParent,
    [in, optional]  hMenu,
    [in, optional]  hInstance,
    [in, optional]  lpParam
);
```

Parameters

[in, optional] lpClassName

Type: [LPCTSTR](#)

A null-terminated string or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero. If *lpClassName* is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined system class names. For a list of system class names, see the Remarks section.

[in, optional] lpWindowName

Type: LPCTSTR

The window name. If the window style specifies a title bar, the window title pointed to by *lpWindowName* is displayed in the title bar. When using [CreateWindow](#) to create controls, such as buttons, check boxes, and static controls, use *lpWindowName* to specify the text of the control. When creating a static control with the **SS_ICON** style, use *lpWindowName* to specify the icon name or identifier. To specify an identifier, use the syntax "#*num*".

[in] dwStyle

Type: DWORD

The style of the window being created. This parameter can be a combination of the [window style values](#), plus the control styles indicated in the Remarks section.

[in] x

Type: int

The initial horizontal position of the window. For an overlapped or pop-up window, the *x* parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, *x* is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area. If this parameter is set to **CW_USEDEFAULT**, the system selects the default position for the window's upper-left corner and ignores the *y* parameter. **CW_USEDEFAULT** is valid only for overlapped windows; if it is specified for a pop-up or child window, the *x* and *y* parameters are set to zero.

[in] y

Type: int

The initial vertical position of the window. For an overlapped or pop-up window, the *y* parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, *y* is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box, *y* is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

If an overlapped window is created with the **WS_VISIBLE** style bit set and the *x* parameter is set to **CW_USEDEFAULT**, then the *y* parameter determines how the window is shown. If the *y* parameter is **CW_USEDEFAULT**, then the window manager calls [ShowWindow](#) with the **SW_SHOW** flag after the window has been created. If the *y*

parameter is some other value, then the window manager calls **ShowWindow** with that value as the *nCmdShow* parameter.

[in] *nWidth*

Type: **int**

The width, in device units, of the window. For overlapped windows, *nWidth* is either the window's width, in screen coordinates, or **CW_USEDEFAULT**. If *nWidth* is **CW_USEDEFAULT**, the system selects a default width and height for the window; the default width extends from the initial x-coordinate to the right edge of the screen, and the default height extends from the initial y-coordinate to the top of the icon area. **CW_USEDEFAULT** is valid only for overlapped windows; if **CW_USEDEFAULT** is specified for a pop-up or child window, *nWidth* and *nHeight* are set to zero.

[in] *nHeight*

Type: **int**

The height, in device units, of the window. For overlapped windows, *nHeight* is the window's height, in screen coordinates. If *nWidth* is set to **CW_USEDEFAULT**, the system ignores *nHeight*.

[in, optional] *hWndParent*

Type: **HWND**

A handle to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows.

To create a [message-only window](#), supply **HWND_MESSAGE** or a handle to an existing message-only window.

[in, optional] *hMenu*

Type: **HMENU**

A handle to a menu, or specifies a child-window identifier depending on the window style. For an overlapped or pop-up window, *hMenu* identifies the menu to be used with the window; it can be **NULL** if the class menu is to be used. For a child window, *hMenu* specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

[in, optional] hInstance

Type: **HINSTANCE**

A handle to the instance of the module to be associated with the window.

[in, optional] lpParam

Type: **LPVOID**

A pointer to a value to be passed to the window through the [CREATESTRUCT](#) structure (*lpCreateParams* member) pointed to by the *lParam* param of the [WM_CREATE](#) message. This message is sent to the created window by this function before it returns.

If an application calls [CreateWindow](#) to create a MDI client window, *lpParam* should point to a [CLIENTCREATESTRUCT](#) structure. If an MDI client window calls [CreateWindow](#) to create an MDI child window, *lpParam* should point to a [MDICREATESTRUCT](#) structure. *lpParam* may be **NULL** if no additional data is needed.

Returns

Type: **HWND**

If the function succeeds, the return value is a handle to the new window.

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

Return value

None

Remarks

Before returning, [CreateWindow](#) sends a [WM_CREATE](#) message to the window procedure. For overlapped, pop-up, and child windows, [CreateWindow](#) sends [WM_CREATE](#), [WM_GETMINMAXINFO](#), and [WM_NCCREATE](#) messages to the window. The *lParam* parameter of the [WM_CREATE](#) message contains a pointer to a [CREATESTRUCT](#) structure. If the [WS_VISIBLE](#) style is specified, [CreateWindow](#) sends the window all the messages required to activate and show the window.

If the created window is a child window, its default position is at the bottom of the Z-order. If the created window is a top-level window, its default position is at the top of

the Z-order (but beneath all topmost windows unless the created window is itself topmost).

For information on controlling whether the Taskbar displays a button for the created window, see [Managing Taskbar Buttons](#).

For information on removing a window, see the [DestroyWindow](#) function.

The following predefined system classes can be specified in the *lpClassName* parameter. Note the corresponding control styles you can use in the *dwStyle* parameter.

System class	Meaning
BUTTON	<p>Designates a small rectangular child window that represents a button the user can click to turn it on or off. Button controls can be used alone or in groups, and they can either be labeled or appear without text. Button controls typically change appearance when the user clicks them. For more information, see Buttons</p> <p>For a table of the button styles you can specify in the <i>dwStyle</i> parameter, see Button Styles.</p>
COMBOBOX	<p>Designates a control consisting of a list box and a selection field similar to an edit control. When using this style, an application should either display the list box at all times or enable a drop-down list box. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. Conversely, selecting an item in the list box displays the selected text in the selection field.</p> <p>For more information, see Combo Boxes. For a table of the combo box styles you can specify in the <i>dwStyle</i> parameter, see Combo Box Styles.</p>
EDIT	<p>Designates a rectangular child window into which the user can type text from the keyboard. The user selects the control and gives it the keyboard focus by clicking it or moving to it by pressing the TAB key. The user can type text when the edit control displays a flashing caret; use the mouse to move the cursor, select characters to be replaced, or position the cursor for inserting characters; or use the BACKSPACE key to delete characters. For more information, see Edit Controls.</p> <p>For a table of the edit control styles you can specify in the <i>dwStyle</i> parameter, see Edit Control Styles.</p>
LISTBOX	<p>Designates a list of character strings. Specify this control whenever an application must present a list of names, such as file names, from which the user can choose. The user can select a string by clicking it. A selected string is highlighted, and a notification message is passed to the parent window. For more information, see List Boxes.</p> <p>For a table of the list box styles you can specify in the <i>dwStyle</i> parameter, see List Box Styles.</p>

MDICLIENT	Designates an MDI client window. This window receives messages that control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD . Specify the WS_HSCROLL and WS_VSCROLL styles to create an MDI client window that allows the user to scroll MDI child windows into view. For more information, see Multiple Document Interface .
RichEdit	Designates a Microsoft Rich Edit 1.0 control. This window lets the user view and edit text with character and paragraph formatting, and can include embedded Component Object Model (COM) objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
RICHEDIT_CLASS	Designates a Microsoft Rich Edit 2.0 control. This controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
SCROLLBAR	Designates a rectangle that contains a scroll box and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the position of the scroll box, if necessary. For more information, see Scroll Bars . For a table of the scroll bar control styles you can specify in the <i>dwStyle</i> parameter, see Scroll Bar Control Styles .
STATIC	Designates a simple text field, box, or rectangle used to label, box, or separate other controls. Static controls take no input and provide no output. For more information, see Static Controls . For a table of the static control styles you can specify in the <i>dwStyle</i> parameter, see Static Control Styles .

CreateWindow is implemented as a call to the [CreateWindowEx](#) function, as shown below.

syntax

```
#define CreateWindowA(lpClassName, lpWindowName, dwStyle, x, y, nWidth,
nHeight, hWndParent, hMenu, hInstance, lpParam)\nCreateWindowExA(0L, lpClassName, lpWindowName, dwStyle, x, y, nWidth,
nHeight, hWndParent, hMenu, hInstance, lpParam)

#define CreateWindowW(lpClassName, lpWindowName, dwStyle, x, y, nWidth,
```

```
nHeight, hWndParent, hMenu, hInstance, lpParam)＼  
CreateWindowExW(0L, lpClassName, lpWindowName, dwStyle, x, y, nWidth,  
nHeight, hWndParent, hMenu, hInstance, lpParam)  
  
#ifdef UNICODE  
#define CreateWindow CreateWindowW  
#else  
#define CreateWindow CreateWindowA  
#endif
```

Examples

For an example, see [Using Window Classes](#).

ⓘ Note

The winuser.h header defines CreateWindow 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	winuser.h (include Windows.h)

See also

[About the Multiple Document Interface](#)

[Common Control Window Classes](#)

[Conceptual](#)

[CreateWindowEx](#)

[DestroyWindow](#)

[EnableWindow](#)

Other Resources

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[ShowWindow](#)

[WM_COMMAND](#)

[WM_CREATE](#)

[WM_GETMINMAXINFO](#)

[WM_NCCREATE](#)

[WM_PAINT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateWindowExA function (winuser.h)

Article 02/09/2023

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the [CreateWindow](#) function. For more information about creating a window and for full descriptions of the other parameters of [CreateWindowEx](#), see [CreateWindow](#).

Syntax

C++

```
HWND CreateWindowExA(
    [in]           DWORD      dwExStyle,
    [in, optional] LPCSTR    lpClassName,
    [in, optional] LPCSTR    lpWindowName,
    [in]           DWORD      dwStyle,
    [in]           int        X,
    [in]           int        Y,
    [in]           int        nWidth,
    [in]           int        nHeight,
    [in, optional] HWND       hWndParent,
    [in, optional] HMENU     hMenu,
    [in, optional] HINSTANCE hInstance,
    [in, optional] LPVOID    lpParam
);
```

Parameters

[in] dwExStyle

Type: **DWORD**

The extended window style of the window being created. For a list of possible values, see [Extended Window Styles](#).

[in, optional] lpClassName

Type: **LPCTSTR**

A null-terminated string or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero. If *lpClassName* is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or

`RegisterClassEx`, provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined [system class names](#).

[in, optional] `lpWindowName`

Type: **LPCTSTR**

The window name. If the window style specifies a title bar, the window title pointed to by `lpWindowName` is displayed in the title bar. When using [CreateWindow](#) to create controls, such as buttons, check boxes, and static controls, use `lpWindowName` to specify the text of the control. When creating a static control with the **SS_ICON** style, use `lpWindowName` to specify the icon name or identifier. To specify an identifier, use the syntax "#*num*".

[in] `dwStyle`

Type: **DWORD**

The style of the window being created. This parameter can be a combination of the [window style values](#), plus the control styles indicated in the Remarks section.

[in] `x`

Type: **int**

The initial horizontal position of the window. For an overlapped or pop-up window, the `x` parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, `x` is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area. If `x` is set to **CW_USEDEFAULT**, the system selects the default position for the window's upper-left corner and ignores the `y` parameter. **CW_USEDEFAULT** is valid only for overlapped windows; if it is specified for a pop-up or child window, the `x` and `y` parameters are set to zero.

[in] `y`

Type: **int**

The initial vertical position of the window. For an overlapped or pop-up window, the `y` parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, `y` is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box `y` is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

If an overlapped window is created with the **WS_VISIBLE** style bit set and the *x* parameter is set to **CW_USEDEFAULT**, then the *y* parameter determines how the window is shown. If the *y* parameter is **CW_USEDEFAULT**, then the window manager calls [ShowWindow](#) with the **SW_SHOW** flag after the window has been created. If the *y* parameter is some other value, then the window manager calls [ShowWindow](#) with that value as the *nCmdShow* parameter.

[in] *nWidth*

Type: **int**

The width, in device units, of the window. For overlapped windows, *nWidth* is the window's width, in screen coordinates, or **CW_USEDEFAULT**. If *nWidth* is **CW_USEDEFAULT**, the system selects a default width and height for the window; the default width extends from the initial x-coordinates to the right edge of the screen; the default height extends from the initial y-coordinate to the top of the icon area. **CW_USEDEFAULT** is valid only for overlapped windows; if **CW_USEDEFAULT** is specified for a pop-up or child window, the *nWidth* and *nHeight* parameter are set to zero.

[in] *nHeight*

Type: **int**

The height, in device units, of the window. For overlapped windows, *nHeight* is the window's height, in screen coordinates. If the *nWidth* parameter is set to **CW_USEDEFAULT**, the system ignores *nHeight*.

[in, optional] *hWndParent*

Type: **HWND**

A handle to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows.

To create a [message-only window](#), supply **HWND_MESSAGE** or a handle to an existing message-only window.

[in, optional] *hMenu*

Type: **HMENU**

A handle to a menu, or specifies a child-window identifier, depending on the window style. For an overlapped or pop-up window, *hMenu* identifies the menu to be used with the window; it can be **NULL** if the class menu is to be used. For a child window, *hMenu*

specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

[in, optional] hInstance

Type: **HINSTANCE**

A handle to the instance of the module to be associated with the window.

[in, optional] lpParam

Type: **LPVOID**

Pointer to a value to be passed to the window through the [CREATESTRUCT](#) structure (*lpCreateParams* member) pointed to by the *lParam* param of the [WM_CREATE](#) message. This message is sent to the created window by this function before it returns.

If an application calls [CreateWindow](#) to create a MDI client window, *lpParam* should point to a [CLIENTCREATESTRUCT](#) structure. If an MDI client window calls [CreateWindow](#) to create an MDI child window, *lpParam* should point to a [MDICREATESTRUCT](#) structure. *lpParam* may be **NULL** if no additional data is needed.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the new window.

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

This function typically fails for one of the following reasons:

- an invalid parameter value
- the system class was registered by a different module
- The **WH_CBT** hook is installed and returns a failure code
- if one of the controls in the dialog template is not registered, or its window window procedure fails [WM_CREATE](#) or [WM_NCCREATE](#)

Remarks

The [CreateWindowEx](#) function sends [WM_NCCREATE](#), [WM_NCCALCSIZE](#), and [WM_CREATE](#) messages to the window being created.

If the created window is a child window, its default position is at the bottom of the Z-order. If the created window is a top-level window, its default position is at the top of the Z-order (but beneath all topmost windows unless the created window is itself topmost).

For information on controlling whether the Taskbar displays a button for the created window, see [Managing Taskbar Buttons](#).

For information on removing a window, see the [DestroyWindow](#) function.

The following predefined control classes can be specified in the *lpClassName* parameter. Note the corresponding control styles you can use in the *dwStyle* parameter.

Class	Meaning
BUTTON	<p>Designates a small rectangular child window that represents a button the user can click to turn it on or off. Button controls can be used alone or in groups, and they can either be labeled or appear without text. Button controls typically change appearance when the user clicks them. For more information, see Buttons.</p> <p>For a table of the button styles you can specify in the <i>dwStyle</i> parameter, see Button Styles.</p>
COMBOBOX	<p>Designates a control consisting of a list box and a selection field similar to an edit control. When using this style, an application should either display the list box at all times or enable a drop-down list box. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. Conversely, selecting an item in the list box displays the selected text in the selection field. For more information, see Combo Boxes.</p> <p>For a table of the combo box styles you can specify in the <i>dwStyle</i> parameter, see Combo Box Styles.</p>
EDIT	<p>Designates a rectangular child window into which the user can type text from the keyboard. The user selects the control and gives it the keyboard focus by clicking it or moving to it by pressing the TAB key. The user can type text when the edit control displays a flashing caret; use the mouse to move the cursor, select characters to be replaced, or position the cursor for inserting characters; or use the key to delete characters. For more information, see Edit Controls.</p> <p>For a table of the edit control styles you can specify in the <i>dwStyle</i> parameter, see Edit Control Styles.</p>
LISTBOX	<p>Designates a list of character strings. Specify this control whenever an application must present a list of names, such as filenames, from which the user can choose. The user can select a string by clicking it. A selected string is</p>

highlighted, and a notification message is passed to the parent window. For more information, see [List Boxes](#).

For a table of the list box styles you can specify in the *dwStyle* parameter, see [List Box Styles](#).

MDICLIENT	Designates an MDI client window. This window receives messages that control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD . Specify the WS_HSCROLL and WS_VSCROLL styles to create an MDI client window that allows the user to scroll MDI child windows into view. For more information, see Multiple Document Interface .
RichEdit	Designates a Microsoft Rich Edit 1.0 control. This window lets the user view and edit text with character and paragraph formatting, and can include embedded Component Object Model (COM) objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
RICHEDIT_CLASS	Designates a Microsoft Rich Edit 2.0 control. This controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
SCROLLBAR	Designates a rectangle that contains a scroll box and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the position of the scroll box, if necessary. For more information, see Scroll Bars . For a table of the scroll bar control styles you can specify in the <i>dwStyle</i> parameter, see Scroll Bar Control Styles .
STATIC	Designates a simple text field, box, or rectangle used to label, box, or separate other controls. Static controls take no input and provide no output. For more information, see Static Controls . For a table of the static control styles you can specify in the <i>dwStyle</i> parameter, see Static Control Styles .

The **WS_EX_NOACTIVATE** value for *dwExStyle* prevents foreground activation by the system. To prevent queue activation when the user clicks on the window, you must process the **WM_MOUSEACTIVATE** message appropriately. To bring the window to the foreground or to activate it programmatically, use [SetForegroundWindow](#) or

`SetActiveWindow`. Returning `FALSE` to `WM_NCACTIVATE` prevents the window from losing queue activation. However, the return value is ignored at activation time.

With `WS_EX_COMPOSITED` set, all descendants of a window get bottom-to-top painting order using double-buffering. Bottom-to-top painting order allows a descendent window to have translucency (alpha) and transparency (color-key) effects, but only if the descendent window also has the `WS_EX_TRANSPARENT` bit set. Double-buffering allows the window and its descendants to be painted without flicker.

Example

The following sample code illustrates the use of `CreateWindowExA`.

C++

```
BOOL Create(
    PCWSTR lpWindowName,
    DWORD dwStyle,
    DWORD dwExStyle = 0,
    int x = CW_USEDEFAULT,
    int y = CW_USEDEFAULT,
    int nWidth = CW_USEDEFAULT,
    int nHeight = CW_USEDEFAULT,
    HWND hWndParent = 0,
    HMENU hMenu = 0
)
{
    WNDCLASS wc = {0};

    wc.lpfnWndProc = DERIVED_TYPE::WindowProc;
    wc.hInstance = GetModuleHandle(NULL);
    wc.lpszClassName = ClassName();

    RegisterClass(&wc);

    m_hwnd = CreateWindowEx(
        dwExStyle, ClassName(), lpWindowName, dwStyle, x, y,
        nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this
    );

    return (m_hwnd ? TRUE : FALSE);
}
```

ⓘ Note

The `winuser.h` header defines `CreateWindowEx` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[About the Multiple Document Interface](#)

[CLIENTCREATESTRUCT](#)

[CREATESTRUCT](#)

[Conceptual](#)

[CreateWindow](#)

[DestroyWindow](#)

[EnableWindow](#)

[Other Resources](#)

[Reference](#)

[RegisterClass](#)

[RegisterClassEx](#)

[SetActiveWindow](#)

[SetForegroundWindow](#)

[SetWindowLong](#)

[SetWindowPos](#)

[ShowWindow](#)

[WM_CREATE](#)

[WM_NCCALCSIZE](#)

[WM_NCCREATE](#)

[WM_PAINT](#)

[WM_PARENTNOTIFY](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateWindowExW function (winuser.h)

Article 02/09/2023

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the [CreateWindow](#) function. For more information about creating a window and for full descriptions of the other parameters of [CreateWindowEx](#), see [CreateWindow](#).

Syntax

C++

```
HWND CreateWindowExW(
    [in]           DWORD      dwExStyle,
    [in, optional] LPCWSTR   lpClassName,
    [in, optional] LPCWSTR   lpWindowName,
    [in]           DWORD      dwStyle,
    [in]           int        X,
    [in]           int        Y,
    [in]           int        nWidth,
    [in]           int        nHeight,
    [in, optional] HWND       hWndParent,
    [in, optional] HMENU     hMenu,
    [in, optional] HINSTANCE hInstance,
    [in, optional] LPVOID    lpParam
);
```

Parameters

[in] dwExStyle

Type: **DWORD**

The extended window style of the window being created. For a list of possible values, see [Extended Window Styles](#).

[in, optional] lpClassName

Type: **LPCTSTR**

A null-terminated string or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero. If *lpClassName* is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or

`RegisterClassEx`, provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined [system class names](#).

[in, optional] `lpWindowName`

Type: **LPCTSTR**

The window name. If the window style specifies a title bar, the window title pointed to by `lpWindowName` is displayed in the title bar. When using [CreateWindow](#) to create controls, such as buttons, check boxes, and static controls, use `lpWindowName` to specify the text of the control. When creating a static control with the **SS_ICON** style, use `lpWindowName` to specify the icon name or identifier. To specify an identifier, use the syntax "#*num*".

[in] `dwStyle`

Type: **DWORD**

The style of the window being created. This parameter can be a combination of the [window style values](#), plus the control styles indicated in the Remarks section.

[in] `x`

Type: **int**

The initial horizontal position of the window. For an overlapped or pop-up window, the `x` parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, `x` is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area. If `x` is set to **CW_USEDEFAULT**, the system selects the default position for the window's upper-left corner and ignores the `y` parameter. **CW_USEDEFAULT** is valid only for overlapped windows; if it is specified for a pop-up or child window, the `x` and `y` parameters are set to zero.

[in] `y`

Type: **int**

The initial vertical position of the window. For an overlapped or pop-up window, the `y` parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, `y` is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box `y` is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

If an overlapped window is created with the **WS_VISIBLE** style bit set and the *x* parameter is set to **CW_USEDEFAULT**, then the *y* parameter determines how the window is shown. If the *y* parameter is **CW_USEDEFAULT**, then the window manager calls [ShowWindow](#) with the **SW_SHOW** flag after the window has been created. If the *y* parameter is some other value, then the window manager calls [ShowWindow](#) with that value as the *nCmdShow* parameter.

[in] *nWidth*

Type: **int**

The width, in device units, of the window. For overlapped windows, *nWidth* is the window's width, in screen coordinates, or **CW_USEDEFAULT**. If *nWidth* is **CW_USEDEFAULT**, the system selects a default width and height for the window; the default width extends from the initial x-coordinates to the right edge of the screen; the default height extends from the initial y-coordinate to the top of the icon area. **CW_USEDEFAULT** is valid only for overlapped windows; if **CW_USEDEFAULT** is specified for a pop-up or child window, the *nWidth* and *nHeight* parameter are set to zero.

[in] *nHeight*

Type: **int**

The height, in device units, of the window. For overlapped windows, *nHeight* is the window's height, in screen coordinates. If the *nWidth* parameter is set to **CW_USEDEFAULT**, the system ignores *nHeight*.

[in, optional] *hWndParent*

Type: **HWND**

A handle to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows.

To create a [message-only window](#), supply **HWND_MESSAGE** or a handle to an existing message-only window.

[in, optional] *hMenu*

Type: **HMENU**

A handle to a menu, or specifies a child-window identifier, depending on the window style. For an overlapped or pop-up window, *hMenu* identifies the menu to be used with the window; it can be **NULL** if the class menu is to be used. For a child window, *hMenu*

specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

[in, optional] hInstance

Type: **HINSTANCE**

A handle to the instance of the module to be associated with the window.

[in, optional] lpParam

Type: **LPVOID**

Pointer to a value to be passed to the window through the [CREATESTRUCT](#) structure (*lpCreateParams* member) pointed to by the *lParam* param of the [WM_CREATE](#) message. This message is sent to the created window by this function before it returns.

If an application calls [CreateWindow](#) to create a MDI client window, *lpParam* should point to a [CLIENTCREATESTRUCT](#) structure. If an MDI client window calls [CreateWindow](#) to create an MDI child window, *lpParam* should point to a [MDICREATESTRUCT](#) structure. *lpParam* may be **NULL** if no additional data is needed.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the new window.

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

This function typically fails for one of the following reasons:

- an invalid parameter value
- the system class was registered by a different module
- The **WH_CBT** hook is installed and returns a failure code
- if one of the controls in the dialog template is not registered, or its window window procedure fails [WM_CREATE](#) or [WM_NCCREATE](#)

Remarks

The [CreateWindowEx](#) function sends [WM_NCCREATE](#), [WM_NCCALCSIZE](#), and [WM_CREATE](#) messages to the window being created.

If the created window is a child window, its default position is at the bottom of the Z-order. If the created window is a top-level window, its default position is at the top of the Z-order (but beneath all topmost windows unless the created window is itself topmost).

For information on controlling whether the Taskbar displays a button for the created window, see [Managing Taskbar Buttons](#).

For information on removing a window, see the [DestroyWindow](#) function.

The following predefined control classes can be specified in the *lpClassName* parameter. Note the corresponding control styles you can use in the *dwStyle* parameter.

Class	Meaning
BUTTON	<p>Designates a small rectangular child window that represents a button the user can click to turn it on or off. Button controls can be used alone or in groups, and they can either be labeled or appear without text. Button controls typically change appearance when the user clicks them. For more information, see Buttons.</p> <p>For a table of the button styles you can specify in the <i>dwStyle</i> parameter, see Button Styles.</p>
COMBOBOX	<p>Designates a control consisting of a list box and a selection field similar to an edit control. When using this style, an application should either display the list box at all times or enable a drop-down list box. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. Conversely, selecting an item in the list box displays the selected text in the selection field. For more information, see Combo Boxes.</p> <p>For a table of the combo box styles you can specify in the <i>dwStyle</i> parameter, see Combo Box Styles.</p>
EDIT	<p>Designates a rectangular child window into which the user can type text from the keyboard. The user selects the control and gives it the keyboard focus by clicking it or moving to it by pressing the TAB key. The user can type text when the edit control displays a flashing caret; use the mouse to move the cursor, select characters to be replaced, or position the cursor for inserting characters; or use the key to delete characters. For more information, see Edit Controls.</p> <p>For a table of the edit control styles you can specify in the <i>dwStyle</i> parameter, see Edit Control Styles.</p>
LISTBOX	<p>Designates a list of character strings. Specify this control whenever an application must present a list of names, such as filenames, from which the user can choose. The user can select a string by clicking it. A selected string is</p>

highlighted, and a notification message is passed to the parent window. For more information, see [List Boxes](#).

For a table of the list box styles you can specify in the *dwStyle* parameter, see [List Box Styles](#).

MDICLIENT	Designates an MDI client window. This window receives messages that control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD . Specify the WS_HSCROLL and WS_VSCROLL styles to create an MDI client window that allows the user to scroll MDI child windows into view. For more information, see Multiple Document Interface .
RichEdit	Designates a Microsoft Rich Edit 1.0 control. This window lets the user view and edit text with character and paragraph formatting, and can include embedded Component Object Model (COM) objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
RICHEDIT_CLASS	Designates a Microsoft Rich Edit 2.0 control. This controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
SCROLLBAR	Designates a rectangle that contains a scroll box and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the position of the scroll box, if necessary. For more information, see Scroll Bars . For a table of the scroll bar control styles you can specify in the <i>dwStyle</i> parameter, see Scroll Bar Control Styles .
STATIC	Designates a simple text field, box, or rectangle used to label, box, or separate other controls. Static controls take no input and provide no output. For more information, see Static Controls . For a table of the static control styles you can specify in the <i>dwStyle</i> parameter, see Static Control Styles .

The **WS_EX_NOACTIVATE** value for *dwExStyle* prevents foreground activation by the system. To prevent queue activation when the user clicks on the window, you must process the **WM_MOUSEACTIVATE** message appropriately. To bring the window to the foreground or to activate it programmatically, use [SetForegroundWindow](#) or

`SetActiveWindow`. Returning `FALSE` to `WM_NCACTIVATE` prevents the window from losing queue activation. However, the return value is ignored at activation time.

With `WS_EX_COMPOSITED` set, all descendants of a window get bottom-to-top painting order using double-buffering. Bottom-to-top painting order allows a descendent window to have translucency (alpha) and transparency (color-key) effects, but only if the descendent window also has the `WS_EX_TRANSPARENT` bit set. Double-buffering allows the window and its descendants to be painted without flicker.

Example

The following sample code illustrates the use of `CreateWindowExA`.

C++

```
BOOL Create(
    PCWSTR lpWindowName,
    DWORD dwStyle,
    DWORD dwExStyle = 0,
    int x = CW_USEDEFAULT,
    int y = CW_USEDEFAULT,
    int nWidth = CW_USEDEFAULT,
    int nHeight = CW_USEDEFAULT,
    HWND hWndParent = 0,
    HMENU hMenu = 0
)
{
    WNDCLASS wc = {0};

    wc.lpfnWndProc = DERIVED_TYPE::WindowProc;
    wc.hInstance = GetModuleHandle(NULL);
    wc.lpszClassName = ClassName();

    RegisterClass(&wc);

    m_hwnd = CreateWindowEx(
        dwExStyle, ClassName(), lpWindowName, dwStyle, x, y,
        nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this
    );

    return (m_hwnd ? TRUE : FALSE);
}
```

ⓘ Note

The `winuser.h` header defines `CreateWindowEx` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[About the Multiple Document Interface](#)

[CLIENTCREATESTRUCT](#)

[CREATESTRUCT](#)

[Conceptual](#)

[CreateWindow](#)

[DestroyWindow](#)

[EnableWindow](#)

[Other Resources](#)

[Reference](#)

[RegisterClass](#)

[RegisterClassEx](#)

[SetActiveWindow](#)

[SetForegroundWindow](#)

[SetWindowLong](#)

[SetWindowPos](#)

[ShowWindow](#)

[WM_CREATE](#)

[WM_NCCALCSIZE](#)

[WM_NCCREATE](#)

[WM_PAINT](#)

[WM_PARENTNOTIFY](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateWindowW macro (winuser.h)

Article03/11/2023

Creates an overlapped, pop-up, or child window. It specifies the window class, window title, window style, and (optionally) the initial position and size of the window. The function also specifies the window's parent or owner, if any, and the window's menu.

To use extended window styles in addition to the styles supported by [CreateWindow](#), use the [CreateWindowEx](#) function.

Syntax

C++

```
HWND CreateWindowW(
    [in, optional]  lpClassName,
    [in, optional]  lpWindowName,
    [in]            dwStyle,
    [in]            x,
    [in]            y,
    [in]            nWidth,
    [in]            nHeight,
    [in, optional]  hWndParent,
    [in, optional]  hMenu,
    [in, optional]  hInstance,
    [in, optional]  lpParam
);
```

Parameters

[in, optional] lpClassName

Type: [LPCWSTR](#)

A null-terminated string or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero. If *lpClassName* is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined system class names. For a list of system class names, see the Remarks section.

[in, optional] lpWindowName

Type: **LPCWSTR**

The window name. If the window style specifies a title bar, the window title pointed to by *lpWindowName* is displayed in the title bar. When using [CreateWindow](#) to create controls, such as buttons, check boxes, and static controls, use *lpWindowName* to specify the text of the control. When creating a static control with the **SS_ICON** style, use *lpWindowName* to specify the icon name or identifier. To specify an identifier, use the syntax "#*num*".

[in] **dwStyle**

Type: **DWORD**

The style of the window being created. This parameter can be a combination of the [window style values](#), plus the control styles indicated in the Remarks section.

[in] **x**

Type: **int**

The initial horizontal position of the window. For an overlapped or pop-up window, the *x* parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, *x* is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area. If this parameter is set to **CW_USEDEFAULT**, the system selects the default position for the window's upper-left corner and ignores the *y* parameter. **CW_USEDEFAULT** is valid only for overlapped windows; if it is specified for a pop-up or child window, the *x* and *y* parameters are set to zero.

[in] **y**

Type: **int**

The initial vertical position of the window. For an overlapped or pop-up window, the *y* parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, *y* is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box, *y* is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

If an overlapped window is created with the **WS_VISIBLE** style bit set and the *x* parameter is set to **CW_USEDEFAULT**, then the *y* parameter determines how the window is shown. If the *y* parameter is **CW_USEDEFAULT**, then the window manager calls [ShowWindow](#) with the **SW_SHOW** flag after the window has been created. If the *y*

parameter is some other value, then the window manager calls **ShowWindow** with that value as the *nCmdShow* parameter.

[in] *nWidth*

Type: int

The width, in device units, of the window. For overlapped windows, *nWidth* is either the window's width, in screen coordinates, or **CW_USEDEFAULT**. If *nWidth* is **CW_USEDEFAULT**, the system selects a default width and height for the window; the default width extends from the initial x-coordinate to the right edge of the screen, and the default height extends from the initial y-coordinate to the top of the icon area. **CW_USEDEFAULT** is valid only for overlapped windows; if **CW_USEDEFAULT** is specified for a pop-up or child window, *nWidth* and *nHeight* are set to zero.

[in] *nHeight*

Type: int

The height, in device units, of the window. For overlapped windows, *nHeight* is the window's height, in screen coordinates. If *nWidth* is set to **CW_USEDEFAULT**, the system ignores *nHeight*.

[in, optional] *hWndParent*

Type: HWND

A handle to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows.

To create a [message-only window](#), supply **HWND_MESSAGE** or a handle to an existing message-only window.

[in, optional] *hMenu*

Type: HMENU

A handle to a menu, or specifies a child-window identifier depending on the window style. For an overlapped or pop-up window, *hMenu* identifies the menu to be used with the window; it can be **NULL** if the class menu is to be used. For a child window, *hMenu* specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

[in, optional] hInstance

Type: **HINSTANCE**

A handle to the instance of the module to be associated with the window.

[in, optional] lpParam

Type: **LPVOID**

A pointer to a value to be passed to the window through the [CREATESTRUCT](#) structure (*lpCreateParams* member) pointed to by the *lParam* param of the [WM_CREATE](#) message. This message is sent to the created window by this function before it returns.

If an application calls [CreateWindow](#) to create a MDI client window, *lpParam* should point to a [CLIENTCREATESTRUCT](#) structure. If an MDI client window calls [CreateWindow](#) to create an MDI child window, *lpParam* should point to a [MDICREATESTRUCT](#) structure. *lpParam* may be **NULL** if no additional data is needed.

Returns

Type: **HWND**

If the function succeeds, the return value is a handle to the new window.

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

Return value

None

Remarks

Before returning, [CreateWindow](#) sends a [WM_CREATE](#) message to the window procedure. For overlapped, pop-up, and child windows, [CreateWindow](#) sends [WM_CREATE](#), [WM_GETMINMAXINFO](#), and [WM_NCCREATE](#) messages to the window. The *lParam* parameter of the [WM_CREATE](#) message contains a pointer to a [CREATESTRUCT](#) structure. If the [WS_VISIBLE](#) style is specified, [CreateWindow](#) sends the window all the messages required to activate and show the window.

If the created window is a child window, its default position is at the bottom of the Z-order. If the created window is a top-level window, its default position is at the top of

the Z-order (but beneath all topmost windows unless the created window is itself topmost).

For information on controlling whether the Taskbar displays a button for the created window, see [Managing Taskbar Buttons](#).

For information on removing a window, see the [DestroyWindow](#) function.

The following predefined system classes can be specified in the *lpClassName* parameter. Note the corresponding control styles you can use in the *dwStyle* parameter.

System class	Meaning
BUTTON	<p>Designates a small rectangular child window that represents a button the user can click to turn it on or off. Button controls can be used alone or in groups, and they can either be labeled or appear without text. Button controls typically change appearance when the user clicks them. For more information, see Buttons</p> <p>For a table of the button styles you can specify in the <i>dwStyle</i> parameter, see Button Styles.</p>
COMBOBOX	<p>Designates a control consisting of a list box and a selection field similar to an edit control. When using this style, an application should either display the list box at all times or enable a drop-down list box. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. Conversely, selecting an item in the list box displays the selected text in the selection field.</p> <p>For more information, see Combo Boxes. For a table of the combo box styles you can specify in the <i>dwStyle</i> parameter, see Combo Box Styles.</p>
EDIT	<p>Designates a rectangular child window into which the user can type text from the keyboard. The user selects the control and gives it the keyboard focus by clicking it or moving to it by pressing the TAB key. The user can type text when the edit control displays a flashing caret; use the mouse to move the cursor, select characters to be replaced, or position the cursor for inserting characters; or use the BACKSPACE key to delete characters. For more information, see Edit Controls.</p> <p>For a table of the edit control styles you can specify in the <i>dwStyle</i> parameter, see Edit Control Styles.</p>
LISTBOX	<p>Designates a list of character strings. Specify this control whenever an application must present a list of names, such as file names, from which the user can choose. The user can select a string by clicking it. A selected string is highlighted, and a notification message is passed to the parent window. For more information, see List Boxes.</p> <p>For a table of the list box styles you can specify in the <i>dwStyle</i> parameter, see List Box Styles.</p>

MDICLIENT	Designates an MDI client window. This window receives messages that control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD . Specify the WS_HSCROLL and WS_VSCROLL styles to create an MDI client window that allows the user to scroll MDI child windows into view. For more information, see Multiple Document Interface .
RichEdit	Designates a Microsoft Rich Edit 1.0 control. This window lets the user view and edit text with character and paragraph formatting, and can include embedded Component Object Model (COM) objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
RICHEDIT_CLASS	Designates a Microsoft Rich Edit 2.0 control. This controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see Rich Edit Controls . For a table of the rich edit control styles you can specify in the <i>dwStyle</i> parameter, see Rich Edit Control Styles .
SCROLLBAR	Designates a rectangle that contains a scroll box and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the position of the scroll box, if necessary. For more information, see Scroll Bars . For a table of the scroll bar control styles you can specify in the <i>dwStyle</i> parameter, see Scroll Bar Control Styles .
STATIC	Designates a simple text field, box, or rectangle used to label, box, or separate other controls. Static controls take no input and provide no output. For more information, see Static Controls . For a table of the static control styles you can specify in the <i>dwStyle</i> parameter, see Static Control Styles .

CreateWindow is implemented as a call to the [CreateWindowEx](#) function, as shown below.

syntax

```
#define CreateWindowA(lpClassName, lpWindowName, dwStyle, x, y, nWidth,
nHeight, hWndParent, hMenu, hInstance, lpParam)\nCreateWindowExA(0L, lpClassName, lpWindowName, dwStyle, x, y, nWidth,
nHeight, hWndParent, hMenu, hInstance, lpParam)

#define CreateWindowW(lpClassName, lpWindowName, dwStyle, x, y, nWidth,
```

```
nHeight, hWndParent, hMenu, hInstance, lpParam)＼  
CreateWindowExW(0L, lpClassName, lpWindowName, dwStyle, x, y, nWidth,  
nHeight, hWndParent, hMenu, hInstance, lpParam)  
  
#ifdef UNICODE  
#define CreateWindow CreateWindowW  
#else  
#define CreateWindow CreateWindowA  
#endif
```

Examples

For an example, see [Using Window Classes](#).

ⓘ Note

The winuser.h header defines CreateWindow 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	winuser.h (include Windows.h)

See also

[About the Multiple Document Interface](#)

[Common Control Window Classes](#)

[Conceptual](#)

[CreateWindowEx](#)

[DestroyWindow](#)

[EnableWindow](#)

Other Resources

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[ShowWindow](#)

[WM_COMMAND](#)

[WM_CREATE](#)

[WM_GETMINMAXINFO](#)

[WM_NCCREATE](#)

[WM_PAINT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

CWPRETSTRUCT structure (winuser.h)

Article 06/30/2023

Defines the message parameters passed to a [WH_CALLWNDPROCRET hook procedure](#), [HOOKPROC callback function](#).

Syntax

C++

```
typedef struct tagCWPRETSTRUCT {
    LRESULT lResult;
    LPARAM lParam;
    WPARAM wParam;
    UINT    message;
    HWND    hwnd;
} CWPRETSTRUCT, *PCWPRETSTRUCT, *NPCWPRETSTRUCT, *LPCWPRETSTRUCT;
```

Members

`lResult`

Type: **LRESULT**

The return value of the window procedure that processed the message specified by the **message** value.

`lParam`

Type: **LPARAM**

Additional information about the message. The exact meaning depends on the **message** value.

`wParam`

Type: **WPARAM**

Additional information about the message. The exact meaning depends on the **message** value.

`message`

Type: **UINT**

The message.

`hwnd`

Type: **HWND**

A handle to the window that processed the message specified by the **message** value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[HOOKPROC](#)

[Conceptual](#)

[Hooks](#)

[Reference](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

CWPSTRUCT structure (winuser.h)

Article 06/30/2023

Defines the message parameters passed to a [WH_CALLWNDPROC hook procedure](#), [CallWindowProcW function](#)/[CallWindowProcA function](#).

Syntax

C++

```
typedef struct tagCWPSTRUCT {
    LPARAM lParam;
    WPARAM wParam;
    UINT    message;
    HWND    hwnd;
} CWPSTRUCT, *PCWPSTRUCT, *NPCWPSTRUCT, *LPCWPSTRUCT;
```

Members

lParam

Type: **LPARAM**

Additional information about the message. The exact meaning depends on the **message** value.

wParam

Type: **WPARAM**

Additional information about the message. The exact meaning depends on the **message** value.

message

Type: **UINT**

The message.

hwnd

Type: **HWND**

A handle to the window to receive the message.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Hooks](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

DEBUGHOOKINFO structure (winuser.h)

Article 06/30/2023

Contains debugging information passed to a WH_DEBUG hook procedure, [DebugProc](#).

Syntax

C++

```
typedef struct tagDEBUGHOOKINFO {
    DWORD idThread;
    DWORD idThreadInstaller;
    LPARAM lParam;
    WPARAM wParam;
    int code;
} DEBUGHOOKINFO, *PDEBUGHOOKINFO, *NPDEBUGHOOKINFO, *LPDEBUGHOOKINFO;
```

Members

`idThread`

Type: **DWORD**

A handle to the thread containing the filter function.

`idThreadInstaller`

Type: **DWORD**

A handle to the thread that installed the debugging filter function.

`lParam`

Type: **LPARAM**

The value to be passed to the hook in the *lParam* parameter of the [DebugProc](#) callback function.

`wParam`

Type: **WPARAM**

The value to be passed to the hook in the *wParam* parameter of the [DebugProc](#) callback function.

code

Type: `int`

The value to be passed to the hook in the *nCode* parameter of the [*DebugProc*](#) callback function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[*DebugProc*](#)

[Hooks](#)

[*SetWindowsHookEx*](#)

Feedback

Was this page helpful?

 Yes

 No

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

DeferWindowPos function (winuser.h)

Article10/13/2021

Updates the specified multiple-window – position structure for the specified window. The function then returns a handle to the updated structure. The [EndDeferWindowPos](#) function uses the information in this structure to change the position and size of a number of windows simultaneously. The [BeginDeferWindowPos](#) function creates the structure.

Syntax

C++

```
HDWP DeferWindowPos(
    [in]          HDWP hWinPosInfo,
    [in]          HWND hWnd,
    [in, optional] HWND hWndInsertAfter,
    [in]          int   x,
    [in]          int   y,
    [in]          int   cx,
    [in]          int   cy,
    [in]          UINT  uFlags
);
```

Parameters

[in] hWinPosInfo

Type: **HDWP**

A handle to a multiple-window – position structure that contains size and position information for one or more windows. This structure is returned by [BeginDeferWindowPos](#) or by the most recent call to [DeferWindowPos](#).

[in] hWnd

Type: **HWND**

A handle to the window for which update information is stored in the structure. All windows in a multiple-window – position structure must have the same parent.

[in, optional] hWndInsertAfter

Type: **HWND**

A handle to the window that precedes the positioned window in the Z order. This parameter must be a window handle or one of the following values. This parameter is ignored if the **SWP_NOZORDER** flag is set in the *uFlags* parameter.

Value	Meaning
HWND_BOTTOM ((HWND)1)	Places the window at the bottom of the Z order. If the <i>hWnd</i> parameter identifies a topmost window, the window loses its topmost status and is placed at the bottom of all other windows.
HWND_NOTOPMOST ((HWND)-2)	Places the window above all non-topmost windows (that is, behind all topmost windows). This flag has no effect if the window is already a non-topmost window.
HWND_TOP ((HWND)0)	Places the window at the top of the Z order.
HWND_TOPMOST ((HWND)-1)	Places the window above all non-topmost windows. The window maintains its topmost position even when it is deactivated.

[in] `x`

Type: **int**

The x-coordinate of the window's upper-left corner.

[in] `y`

Type: **int**

The y-coordinate of the window's upper-left corner.

[in] `cx`

Type: **int**

The window's new width, in pixels.

[in] `cy`

Type: **int**

The window's new height, in pixels.

[in] `uFlags`

Type: **UINT**

A combination of the following values that affect the size and position of the window.

Value	Meaning
SWP_DRAWFRAME 0x0020	Draws a frame (defined in the window's class description) around the window.
SWP_FRAMECHANGED 0x0020	Sends a WM_NCCALCSIZE message to the window, even if the window's size is not being changed. If this flag is not specified, WM_NCCALCSIZE is sent only when the window's size is being changed.
SWP_HIDEWINDOW 0x0080	Hides the window.
SWP_NOACTIVATE 0x0010	Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the topmost or non-topmost group (depending on the setting of the <i>hWndInsertAfter</i> parameter).
SWP_NOCOPYBITS 0x0100	Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
SWP NOMOVE 0x0002	Retains the current position (ignores the <i>x</i> and <i>y</i> parameters).
SWP_NOOWNERZORDER 0x0200	Does not change the owner window's position in the Z order.
SWP_NOREDRAW 0x0008	Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of the window being moved. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.
SWP_NOREPOSITION 0x0200	Same as the SWP_NOOWNERZORDER flag.
SWP_NOSENDCHANGING 0x0400	Prevents the window from receiving the WM_WINDOWPOSCHANGING message.
SWP_NOSIZE 0x0001	Retains the current size (ignores the <i>cx</i> and <i>cy</i> parameters).
SWP_NOZORDER	Retains the current Z order (ignores the <i>hWndInsertAfter</i>

0x0004	parameter).
SWP_SHOWWINDOW 0x0040	Displays the window.

Return value

Type: **HDWP**

The return value identifies the updated multiple-window – position structure. The handle returned by this function may differ from the handle passed to the function. The new handle that this function returns should be passed during the next call to the [DeferWindowPos](#) or [EndDeferWindowPos](#) function.

If insufficient system resources are available for the function to succeed, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

If a call to [DeferWindowPos](#) fails, the application should abandon the window-positioning operation and not call [EndDeferWindowPos](#).

If **SWP_NOZORDER** is not specified, the system places the window identified by the *hWnd* parameter in the position following the window identified by the *hWndInsertAfter* parameter. If *hWndInsertAfter* is **NULL** or **HWND_TOP**, the system places the *hWnd* window at the top of the Z order. If *hWndInsertAfter* is set to **HWND_BOTTOM**, the system places the *hWnd* window at the bottom of the Z order.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

A window can be made a topmost window either by setting *hWndInsertAfter* to the **HWND_TOPMOST** flag and ensuring that the **SWP_NOZORDER** flag is not set, or by setting the window's position in the Z order so that it is above any existing topmost windows. When a non-topmost window is made topmost, its owned windows are also made topmost. Its owners, however, are not changed.

If neither the **SWP_NOACTIVATE** nor **SWP_NOZORDER** flag is specified (that is, when the application requests that a window be simultaneously activated and its position in the Z order changed), the value specified in *hWndInsertAfter* is used only in the following circumstances:

- Neither the **HWND_TOPMOST** nor **HWND_NOTOPMOST** flag is specified in *hWndInsertAfter*.
- The window identified by *hWnd* is not the active window.

An application cannot activate an inactive window without also bringing it to the top of the Z order. An application can change an activated window's position in the Z order without restrictions, or it can activate a window and then move it to the top of the topmost or non-topmost windows.

A topmost window is no longer topmost if it is repositioned to the bottom (**HWND_BOTTOM**) of the Z order or after any non-topmost window. When a topmost window is made non-topmost, its owners and its owned windows are also made non-topmost windows.

A non-topmost window may own a topmost window, but not vice versa. Any window (for example, a dialog box) owned by a topmost window is itself made a topmost window to ensure that all owned windows stay above their owner.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[BeginDeferWindowPos](#)

[Conceptual](#)

[EndDeferWindowPos](#)

[Reference](#)

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

DefFrameProcA function (winuser.h)

Article 02/09/2023

Provides default processing for any window messages that the window procedure of a multiple-document interface (MDI) frame window does not process. All window messages that are not explicitly processed by the window procedure must be passed to the [DefFrameProc](#) function, not the [DefWindowProc](#) function.

Syntax

C++

```
LRESULT DefFrameProcA(
    [in] HWND     hWnd,
    [in] HWND     hWndMDIClient,
    [in] UINT     uMsg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the MDI frame window.

[in] hWndMDIClient

Type: **HWND**

A handle to the MDI client window.

[in] uMsg

Type: **UINT**

The message to be processed.

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: LPARAM

Additional message-specific information.

Return value

Type: LRESULT

The return value specifies the result of the message processing and depends on the message. If the *hWndMDIClient* parameter is **NULL**, the return value is the same as for the [DefWindowProc](#) function.

Remarks

When an application's window procedure does not handle a message, it typically passes the message to the [DefWindowProc](#) function to process the message. MDI applications use the [DefFrameProc](#) and [DefMDIChildProc](#) functions instead of [DefWindowProc](#) to provide default message processing. All messages that an application would usually pass to [DefWindowProc](#) (such as nonclient messages and the [WM_SETTEXT](#) message) should be passed to [DefFrameProc](#) instead. The [DefFrameProc](#) function also handles the following messages.

Message	Response
WM_COMMAND	Activates the MDI child window that the user chooses. This message is sent when the user chooses an MDI child window from the window menu of the MDI frame window. The window identifier accompanying this message identifies the MDI child window to be activated.
WM_MENUCHAR	Opens the window menu of the active MDI child window when the user presses the ALT+ – (minus) key combination.
WM_SETFOCUS	Passes the keyboard focus to the MDI client window, which in turn passes it to the active MDI child window.
WM_SIZE	Resizes the MDI client window to fit in the new frame window's client area. If the frame window procedure sizes the MDI client window to a different size, it should not pass the message to the DefWindowProc function.

Note

The winuser.h header defines DefFrameProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[DefMDIChildProc](#)

[DefWindowProc](#)

[Multiple Document Interface](#)

Reference

[WM_SETTEXT](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DefFrameProcW function (winuser.h)

Article 02/09/2023

Provides default processing for any window messages that the window procedure of a multiple-document interface (MDI) frame window does not process. All window messages that are not explicitly processed by the window procedure must be passed to the [DefFrameProc](#) function, not the [DefWindowProc](#) function.

Syntax

C++

```
LRESULT DefFrameProcW(
    [in] HWND     hWnd,
    [in] HWND     hWndMDIClient,
    [in] UINT     uMsg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the MDI frame window.

[in] hWndMDIClient

Type: **HWND**

A handle to the MDI client window.

[in] uMsg

Type: **UINT**

The message to be processed.

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: LPARAM

Additional message-specific information.

Return value

Type: LRESULT

The return value specifies the result of the message processing and depends on the message. If the *hWndMDIClient* parameter is **NULL**, the return value is the same as for the [DefWindowProc](#) function.

Remarks

When an application's window procedure does not handle a message, it typically passes the message to the [DefWindowProc](#) function to process the message. MDI applications use the [DefFrameProc](#) and [DefMDIChildProc](#) functions instead of [DefWindowProc](#) to provide default message processing. All messages that an application would usually pass to [DefWindowProc](#) (such as nonclient messages and the [WM_SETTEXT](#) message) should be passed to [DefFrameProc](#) instead. The [DefFrameProc](#) function also handles the following messages.

Message	Response
WM_COMMAND	Activates the MDI child window that the user chooses. This message is sent when the user chooses an MDI child window from the window menu of the MDI frame window. The window identifier accompanying this message identifies the MDI child window to be activated.
WM_MENUCHAR	Opens the window menu of the active MDI child window when the user presses the ALT+ – (minus) key combination.
WM_SETFOCUS	Passes the keyboard focus to the MDI client window, which in turn passes it to the active MDI child window.
WM_SIZE	Resizes the MDI client window to fit in the new frame window's client area. If the frame window procedure sizes the MDI client window to a different size, it should not pass the message to the DefWindowProc function.

Note

The winuser.h header defines DefFrameProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[DefMDIChildProc](#)

[DefWindowProc](#)

[Multiple Document Interface](#)

Reference

[WM_SETTEXT](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DefMDIChildProcA function (winuser.h)

Article 02/09/2023

Provides default processing for any window message that the window procedure of a multiple-document interface (MDI) child window does not process. A window message not processed by the window procedure must be passed to the [DefMDIChildProc](#) function, not to the [DefWindowProc](#) function.

Syntax

C++

```
LRESULT LRESULT DefMDIChildProcA(  
    [in] HWND     hWnd,  
    [in] UINT     uMsg,  
    [in] WPARAM   wParam,  
    [in] LPARAM   lParam  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the MDI child window.

[in] uMsg

Type: **UINT**

The message to be processed.

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: LRESULT

The return value specifies the result of the message processing and depends on the message.

Remarks

The **DefMDIChildProc** function assumes that the parent window of the MDI child window identified by the *hWnd* parameter was created with the **MDICLIENT** class.

When an application's window procedure does not handle a message, it typically passes the message to the **DefWindowProc** function to process the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions instead of **DefWindowProc** to provide default message processing. All messages that an application would usually pass to **DefWindowProc** (such as nonclient messages and the **WM_SETTEXT** message) should be passed to **DefMDIChildProc** instead. In addition, **DefMDIChildProc** also handles the following messages.

Message	Response
WM_CHILDACTIVATE	Performs activation processing when MDI child windows are sized, moved, or displayed. This message must be passed.
WM_GETMINMAXINFO	Calculates the size of a maximized MDI child window, based on the current size of the MDI client window.
WM_MENUCHAR	Passes the message to the MDI frame window.
WM_MOVE	Recalculates MDI client scroll bars if they are present.
WM_SETFOCUS	Activates the child window if it is not the active MDI child window.
WM_SIZE	Performs operations necessary for changing the size of a window, especially for maximizing or restoring an MDI child window. Failing to pass this message to the DefMDIChildProc function produces highly undesirable results.
WM_SYSCOMMAND	Handles window menu commands: SC_NEXTWINDOW , SC_PREVWINDOW , SC_MOVE , SC_SIZE , and SC_MAXIMIZE .

ⓘ Note

The winuser.h header defines DefMDIChildProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[DefFrameProc](#)

[DefWindowProc](#)

[Multiple Document Interface](#)

Reference

[WM_CHILDACTIVATE](#)

[WM_GETMINMAXINFO](#)

[WM_MENUCHAR](#)

[WM_MOVE](#)

[WM_SETFOCUS](#)

[WM_SETTEXT](#)

WM_SIZE

WM_SYSCOMMAND

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DefMDIChildProcW function (winuser.h)

Article 02/09/2023

Provides default processing for any window message that the window procedure of a multiple-document interface (MDI) child window does not process. A window message not processed by the window procedure must be passed to the [DefMDIChildProc](#) function, not to the [DefWindowProc](#) function.

Syntax

C++

```
LRESULT LRESULT DefMDIChildProcW(
    [in] HWND     hWnd,
    [in] UINT     uMsg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the MDI child window.

[in] uMsg

Type: **UINT**

The message to be processed.

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: LRESULT

The return value specifies the result of the message processing and depends on the message.

Remarks

The **DefMDIChildProc** function assumes that the parent window of the MDI child window identified by the *hWnd* parameter was created with the **MDICLIENT** class.

When an application's window procedure does not handle a message, it typically passes the message to the **DefWindowProc** function to process the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions instead of **DefWindowProc** to provide default message processing. All messages that an application would usually pass to **DefWindowProc** (such as nonclient messages and the **WM_SETTEXT** message) should be passed to **DefMDIChildProc** instead. In addition, **DefMDIChildProc** also handles the following messages.

Message	Response
WM_CHILDACTIVATE	Performs activation processing when MDI child windows are sized, moved, or displayed. This message must be passed.
WM_GETMINMAXINFO	Calculates the size of a maximized MDI child window, based on the current size of the MDI client window.
WM_MENUCHAR	Passes the message to the MDI frame window.
WM_MOVE	Recalculates MDI client scroll bars if they are present.
WM_SETFOCUS	Activates the child window if it is not the active MDI child window.
WM_SIZE	Performs operations necessary for changing the size of a window, especially for maximizing or restoring an MDI child window. Failing to pass this message to the DefMDIChildProc function produces highly undesirable results.
WM_SYSCOMMAND	Handles window menu commands: SC_NEXTWINDOW , SC_PREVWINDOW , SC_MOVE , SC_SIZE , and SC_MAXIMIZE .

ⓘ Note

The winuser.h header defines DefMDIChildProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[DefFrameProc](#)

[DefWindowProc](#)

[Multiple Document Interface](#)

Reference

[WM_CHILDACTIVATE](#)

[WM_GETMINMAXINFO](#)

[WM_MENUCHAR](#)

[WM_MOVE](#)

[WM_SETFOCUS](#)

[WM_SETTEXT](#)

WM_SIZE

WM_SYSCOMMAND

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DefWindowProcA function (winuser.h)

Article 02/09/2023

Calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed. **DefWindowProc** is called with the same parameters received by the window procedure.

Syntax

C++

```
LRESULT DefWindowProcA(  
    [in] HWND    hWnd,  
    [in] UINT    Msg,  
    [in] WPARAM wParam,  
    [in] LPARAM lParam  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window procedure that received the message.

[in] Msg

Type: **UINT**

The message.

[in] wParam

Type: **WPARAM**

Additional message information. The content of this parameter depends on the value of the *Msg* parameter.

[in] lParam

Type: **LPARAM**

Additional message information. The content of this parameter depends on the value of the *Msg* parameter.

Return value

Type: LRESULT

The return value is the result of the message processing and depends on the message.

Remarks

ⓘ Note

The winuser.h header defines DefWindowProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

DefDlgProc

Reference

[Window Procedures](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

DefWindowProcW function (winuser.h)

Article 02/09/2023

Calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed. **DefWindowProc** is called with the same parameters received by the window procedure.

Syntax

C++

```
LRESULT DefWindowProcW(
    [in] HWND     hWnd,
    [in] UINT     Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window procedure that received the message.

[in] Msg

Type: **UINT**

The message.

[in] wParam

Type: **WPARAM**

Additional message information. The content of this parameter depends on the value of the *Msg* parameter.

[in] lParam

Type: **LPARAM**

Additional message information. The content of this parameter depends on the value of the *Msg* parameter.

Return value

Type: LRESULT

The return value is the result of the message processing and depends on the message.

Remarks

ⓘ Note

The winuser.h header defines DefWindowProc 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

DefDlgProc

Reference

[Window Procedures](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

DeregisterShellHookWindow function (winuser.h)

Article 10/13/2021

[This function is not intended for general use. It may be altered or unavailable in subsequent versions of Windows.]

Unregisters a specified Shell window that is registered to receive Shell hook messages.

Syntax

C++

```
BOOL DeregisterShellHookWindow(  
    [in] HWND hwnd  
);
```

Parameters

`[in] hwnd`

Type: **HWND**

A handle to the window to be unregistered. The window was registered with a call to the [RegisterShellHookWindow](#) function.

Return value

Type: **BOOL**

TRUE if the function succeeds; **FALSE** if the function fails.

Remarks

This function was not included in the SDK headers and libraries until Windows XP with Service Pack 1 (SP1) and Windows Server 2003. If you do not have a header file and import library for this function, you can call the function using [LoadLibrary](#) and [GetProcAddress](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[Reference](#)

[RegisterShellHookWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

DestroyWindow function (winuser.h)

Article 02/23/2022

Destroys the specified window. The function sends [WM_DESTROY](#) and [WM_NCDESTROY](#) messages to the window to deactivate it and remove the keyboard focus from it. The function also destroys the window's menu, flushes the thread message queue, destroys timers, removes clipboard ownership, and breaks the clipboard viewer chain (if the window is at the top of the viewer chain).

If the specified window is a parent or owner window, **DestroyWindow** automatically destroys the associated child or owned windows when it destroys the parent or owner window. The function first destroys child or owned windows, and then it destroys the parent or owner window.

DestroyWindow also destroys modeless dialog boxes created by the [CreateDialog](#) function.

Syntax

C++

```
BOOL DestroyWindow(  
    [in] HWND hWnd  
>;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be destroyed.

Return value

Type: **BOOL**

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 cannot use **DestroyWindow** to destroy a window created by a different thread.

If the window being destroyed is a child window that does not have the **WS_EX_NOPARENTNOTIFY** style, a **WM_PARENTNOTIFY** message is sent to the parent.

Examples

For an example, see [Destroying a Window](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[CreateDialog](#)

[CreateWindow](#)

[CreateWindowEx](#)

Reference

[WM_DESTROY](#)

[WM_NCDESTROY](#)

[WM_PARENTNOTIFY](#)

Feedback

Was this page helpful?

 Yes

 No

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

DispatchMessage function (winuser.h)

Article 08/02/2022

Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the [GetMessage](#) function.

Syntax

C++

```
LRESULT DispatchMessage(  
    [in] const MSG *lpMsg  
);
```

Parameters

[in] lpMsg

Type: **const MSG***

A pointer to a structure that contains the message.

Return value

Type: **LRESULT**

The return value specifies the value returned by the window procedure. Although its meaning depends on the message being dispatched, the return value generally is ignored.

Remarks

The [MSG](#) structure must contain valid message values. If the *lpmmsg* parameter points to a [WM_TIMER](#) message and the *lParam* parameter of the [WM_TIMER](#) message is not **NULL**, *lParam* points to a function that is called instead of the window procedure.

Note that the application is responsible for retrieving and dispatching input messages to the dialog box. Most applications use the main message loop for this. However, to permit the user to move to and to select controls by using the keyboard, the application must call [IsDialogMessage](#). For more information, see [Dialog Box Keyboard Interface](#).

Examples

For an example, see [Creating a Message Loop](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetMessage](#)

[IsDialogMessage](#)

[MSG](#)

[Messages and Message Queues](#)

[PeekMessage](#)

Reference

[TranslateMessage](#)

[WM_TIMER](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

DispatchMessageA function (winuser.h)

Article 02/09/2023

Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the [GetMessage](#) function.

Syntax

C++

```
LRESULT DispatchMessageA(  
    [in] const MSG *lpMsg  
);
```

Parameters

[in] lpMsg

Type: **const MSG***

A pointer to a structure that contains the message.

Return value

Type: **LRESULT**

The return value specifies the value returned by the window procedure. Although its meaning depends on the message being dispatched, the return value generally is ignored.

Remarks

The [MSG](#) structure must contain valid message values. If the *lParam* parameter points to a [WM_TIMER](#) message and the *lParam* parameter of the [WM_TIMER](#) message is not **NULL**, *lParam* points to a function that is called instead of the window procedure.

Note that the application is responsible for retrieving and dispatching input messages to the dialog box. Most applications use the main message loop for this. However, to permit the user to move to and to select controls by using the keyboard, the application must call [IsDialogMessage](#). For more information, see [Dialog Box Keyboard Interface](#).

Examples

For an example, see [Creating a Message Loop](#).

Note

The winuser.h header defines DispatchMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetMessage](#)

[IsDialogMessage](#)

[MSG](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[Reference](#)

TranslateMessage

WM_TIMER

Feedback

Was this page helpful?

 Yes

 No

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

DispatchMessageW function (winuser.h)

Article 02/09/2023

Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the [GetMessage](#) function.

Syntax

C++

```
LRESULT DispatchMessageW(
    [in] const MSG *lpMsg
);
```

Parameters

[in] lpMsg

Type: **const MSG***

A pointer to a structure that contains the message.

Return value

Type: **LRESULT**

The return value specifies the value returned by the window procedure. Although its meaning depends on the message being dispatched, the return value generally is ignored.

Remarks

The [MSG](#) structure must contain valid message values. If the *lpmmsg* parameter points to a [WM_TIMER](#) message and the *lParam* parameter of the [WM_TIMER](#) message is not **NULL**, *lParam* points to a function that is called instead of the window procedure.

Note that the application is responsible for retrieving and dispatching input messages to the dialog box. Most applications use the main message loop for this. However, to permit the user to move to and to select controls by using the keyboard, the application must call [IsDialogMessage](#). For more information, see [Dialog Box Keyboard Interface](#).

Examples

For an example, see [Creating a Message Loop](#).

Note

The winuser.h header defines DispatchMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetMessage](#)

[IsDialogMessage](#)

[MSG](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[Reference](#)

TranslateMessage

WM_TIMER

Feedback

Was this page helpful?

 Yes

 No

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

EndDeferWindowPos function (winuser.h)

Article 10/13/2021

Simultaneously updates the position and size of one or more windows in a single screen-refreshing cycle.

Syntax

C++

```
BOOL EndDeferWindowPos(  
    [in] HDWP hWinPosInfo  
);
```

Parameters

[in] hWinPosInfo

Type: **HDWP**

A handle to a multiple-window – position structure that contains size and position information for one or more windows. This internal structure is returned by the [BeginDeferWindowPos](#) function or by the most recent call to the [DeferWindowPos](#) function.

Return value

Type: **BOOL**

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 **EndDeferWindowPos** function sends the [WM_WINDOWPOSCHANGING](#) and [WM_WINDOWPOSCHANGED](#) messages to each window identified in the internal

structure.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[BeginDeferWindowPos](#)

[Conceptual](#)

[DeferWindowPos](#)

[Reference](#)

[WM_WINDOWPOSCHANGED](#)

[WM_WINDOWPOSCHANGING](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

EndTask function (winuser.h)

Article10/13/2021

[This function is not intended for general use. It may be altered or unavailable in subsequent versions of Windows.]

Forcibly closes the specified window.

Syntax

C++

```
BOOL EndTask(
    [in] HWND hWnd,
    [in] BOOL fShutdown,
    [in] BOOL fForce
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be closed.

[in] fShutdown

Type: **BOOL**

Ignored. Must be **FALSE**.

[in] fForce

Type: **BOOL**

A **TRUE** for this parameter will force the destruction of the window if an initial attempt fails to gently close the window using [WM_CLOSE](#). With a **FALSE** for this parameter, only the close with [WM_CLOSE](#) is attempted.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

This function was not included in the SDK headers and libraries until Windows XP with Service Pack 1 (SP1) and Windows Server 2003. If you do not have a header file and import library for this function, you can call the function using [LoadLibrary](#) and [GetProcAddress](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[CloseWindow](#)

[Conceptual](#)

[DestroyWindow](#)

[Reference](#)

[WM_CLOSE](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

EnumChildWindows function (winuser.h)

Article 10/13/2021

Enumerates the child windows that belong to the specified parent window by passing the handle to each child window, in turn, to an application-defined callback function.

EnumChildWindows continues until the last child window is enumerated or the callback function returns **FALSE**.

Syntax

C++

```
BOOL EnumChildWindows(  
    [in, optional] HWND      hWndParent,  
    [in]          WNDENUMPROC lpEnumFunc,  
    [in]          LPARAM     lParam  
)
```

Parameters

[in, optional] **hWndParent**

Type: **HWND**

A handle to the parent window whose child windows are to be enumerated. If this parameter is **NULL**, this function is equivalent to [EnumWindows](#).

[in] **lpEnumFunc**

Type: **WNDENUMPROC**

A pointer to an application-defined callback function. For more information, see [EnumChildProc](#).

[in] **lParam**

Type: **LPARAM**

An application-defined value to be passed to the callback function.

Return value

Type: **BOOL**

The return value is not used.

Remarks

If a child window has created child windows of its own, **EnumChildWindows** enumerates those windows as well.

A child window that is moved or repositioned in the Z order during the enumeration process will be properly enumerated. The function does not enumerate a child window that is destroyed before being enumerated or that is created during the enumeration process.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[EnumChildProc](#)

[EnumThreadWindows](#)

[EnumWindows](#)

[GetWindow](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

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

EnumPropsA function (winuser.h)

Article 02/09/2023

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumProps` continues until the last entry is enumerated or the callback function returns **FALSE**.

To pass application-defined data to the callback function, use [EnumPropsEx](#) function.

Syntax

C++

```
int EnumPropsA(
    [in] HWND         hWnd,
    [in] PROOPENUMPROCA lpEnumFunc
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be enumerated.

[in] lpEnumFunc

Type: **PROOPENUMPROC**

A pointer to the callback function. For more information about the callback function, see the [PropEnumProc](#) function.

Return value

Type: **int**

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

 **Note**

The winuser.h header defines `EnumProps` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[EnumPropsEx](#)

[PropEnumProc](#)

Reference

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

EnumPropsExA function (winuser.h)

Article 02/09/2023

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. **EnumPropsEx** continues until the last entry is enumerated or the callback function returns **FALSE**.

Syntax

C++

```
int EnumPropsExA(
    [in] HWND           hWnd,
    [in] PROOPENUMPROCEXA lpEnumFunc,
    [in] LPARAM          lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be enumerated.

[in] lpEnumFunc

Type: **PROOPENUMPROCEX**

A pointer to the callback function. For more information about the callback function, see the [PropEnumProcEx](#) function.

[in] lParam

Type: **LPARAM**

Application-defined data to be passed to the callback function.

Return value

Type: **int**

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

Examples

For an example, see [Listing Window Properties for a Given Window](#).

ⓘ Note

The winuser.h header defines `EnumPropsEx` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[PropEnumProcEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

EnumPropsExW function (winuser.h)

Article 02/09/2023

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumPropsEx` continues until the last entry is enumerated or the callback function returns **FALSE**.

Syntax

C++

```
int EnumPropsExW(
    [in] HWND           hWnd,
    [in] PROOPENUMPROCEXW lpEnumFunc,
    [in] LPARAM          lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be enumerated.

[in] lpEnumFunc

Type: **PROOPENUMPROCEX**

A pointer to the callback function. For more information about the callback function, see the [PropEnumProcEx](#) function.

[in] lParam

Type: **LPARAM**

Application-defined data to be passed to the callback function.

Return value

Type: **int**

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

Examples

For an example, see [Listing Window Properties for a Given Window](#).

ⓘ Note

The winuser.h header defines `EnumPropsEx` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[PropEnumProcEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

EnumPropsW function (winuser.h)

Article 02/09/2023

Enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. `EnumProps` continues until the last entry is enumerated or the callback function returns **FALSE**.

To pass application-defined data to the callback function, use [EnumPropsEx](#) function.

Syntax

C++

```
int EnumPropsW(
    [in] HWND         hWnd,
    [in] PROOPENUMPROCW lpEnumFunc
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be enumerated.

[in] lpEnumFunc

Type: **PROOPENUMPROC**

A pointer to the callback function. For more information about the callback function, see the [PropEnumProc](#) function.

Return value

Type: **int**

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

 **Note**

The winuser.h header defines `EnumProps` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[EnumPropsEx](#)

[PropEnumProc](#)

Reference

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

EnumThreadWindows function (winuser.h)

Article 10/13/2021

Enumerates all nonchild windows associated with a thread by passing the handle to each window, in turn, to an application-defined callback function. **EnumThreadWindows** continues until the last window is enumerated or the callback function returns **FALSE**. To enumerate child windows of a particular window, use the [EnumChildWindows](#) function.

Syntax

C++

```
BOOL EnumThreadWindows(  
    [in] DWORD      dwThreadId,  
    [in] WNDENUMPROC lpfn,  
    [in] LPARAM     lParam  
) ;
```

Parameters

[in] dwThreadId

Type: **DWORD**

The identifier of the thread whose windows are to be enumerated.

[in] lpfn

Type: **WNDENUMPROC**

A pointer to an application-defined callback function. For more information, see [EnumThreadWndProc](#).

[in] lParam

Type: **LPARAM**

An application-defined value to be passed to the callback function.

Return value

Type: **BOOL**

If the callback function returns **TRUE** for all windows in the thread specified by *dwThreadId*, the return value is **TRUE**. If the callback function returns **FALSE** on any enumerated window, or if there are no windows found in the thread specified by *dwThreadId*, the return value is **FALSE**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[EnumChildWindows](#)

[EnumThreadWndProc](#)

[EnumWindows](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

EnumWindows function (winuser.h)

Article10/13/2021

Enumerates all top-level windows on the screen by passing the handle to each window, in turn, to an application-defined callback function. **EnumWindows** continues until the last top-level window is enumerated or the callback function returns **FALSE**.

Syntax

C++

```
BOOL EnumWindows(  
    [in] WNDENUMPROC lpEnumFunc,  
    [in] LPARAM     lParam  
) ;
```

Parameters

[in] **lpEnumFunc**

Type: **WNDENUMPROC**

A pointer to an application-defined callback function. For more information, see [EnumWindowsProc](#).

[in] **lParam**

Type: **LPARAM**

An application-defined value to be passed to the callback function.

Return value

Type: **BOOL**

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 [EnumWindowsProc](#) returns zero, the return value is also zero. In this case, the callback function should call [SetLastError](#) to obtain a meaningful error code to be returned to the

caller of [EnumWindows](#).

Remarks

The [EnumWindows](#) function does not enumerate child windows, with the exception of a few top-level windows owned by the system that have the [WS_CHILD](#) style.

This function is more reliable than calling the [GetWindow](#) function in a loop. An application that calls [GetWindow](#) to perform this task risks being caught in an infinite loop or referencing a handle to a window that has been destroyed.

Note For Windows 8 and later, [EnumWindows](#) enumerates only top-level windows of desktop apps.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[EnumChildWindows](#)

[EnumWindowsProc](#)

[GetWindow](#)

Feedback

Was this page helpful?

 Yes

 No

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

EVENTMSG structure (winuser.h)

Article 04/02/2021

Contains information about a hardware message sent to the system message queue. This structure is used to store message information for the [JournalPlaybackProc](#) callback function.

Syntax

C++

```
typedef struct tagEVENTMSG {
    UINT    message;
    UINT    paramL;
    UINT    paramH;
    DWORD   time;
    HWND    hwnd;
} EVENTMSG, *PEVENTMSGMSG, *NPEVENTMSGMSG, *LPEVENTMSGMSG, *PEVENTMSG,
*NPEVENTMSG, *LPEVENTMSG;
```

Members

`message`

Type: **UINT**

The message.

`paramL`

Type: **UINT**

Additional information about the message. The exact meaning depends on the **message** value.

`paramH`

Type: **UINT**

Additional information about the message. The exact meaning depends on the **message** value.

`time`

Type: **DWORD**

The time at which the message was posted.

hwnd

Type: **HWND**

A handle to the window to which the message was posted.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Hooks](#)

[JournalPlaybackProc](#)

[Reference](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

FindWindowA function (winuser.h)

Article02/09/2023

Retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search.

To search child windows, beginning with a specified child window, use the [FindWindowEx](#) function.

Syntax

C++

```
HWND FindWindowA(
    [in, optional] LPCSTR lpClassName,
    [in, optional] LPCSTR lpWindowName
);
```

Parameters

[in, optional] lpClassName

Type: [LPCTSTR](#)

The class name or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

If *lpClassName* points to a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined control-class names.

If *lpClassName* is **NULL**, it finds any window whose title matches the *lpWindowName* parameter.

[in, optional] lpWindowName

Type: [LPCTSTR](#)

The window name (the window's title). If this parameter is **NULL**, all window names match.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the window that has the specified class name and window name.

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

Remarks

If the *lpWindowName* parameter is not **NULL**, **FindWindow** calls the [GetWindowText](#) function to retrieve the window name for comparison. For a description of a potential problem that can arise, see the Remarks for [GetWindowText](#).

Examples

For an example, see [Retrieving the Number of Mouse Wheel Scroll Lines](#).

ⓘ Note

The winuser.h header defines **FindWindow** 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	winuser.h (include Windows.h)
Library	User32.lib

DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[EnumWindows](#)

[FindWindowEx](#)

[GetClassName](#)

[GetWindowText](#)

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

FindWindowExA function (winuser.h)

Article 02/09/2023

Retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search.

Syntax

C++

```
HWND FindWindowExA(
    [in, optional] HWND     hWndParent,
    [in, optional] HWND     hWndChildAfter,
    [in, optional] LPCSTR  lpszClass,
    [in, optional] LPCSTR  lpszWindow
);
```

Parameters

[in, optional] hWndParent

Type: **HWND**

A handle to the parent window whose child windows are to be searched.

If *hwndParent* is **NULL**, the function uses the desktop window as the parent window. The function searches among windows that are child windows of the desktop.

If *hwndParent* is **HWND_MESSAGE**, the function searches all [message-only windows](#).

[in, optional] hWndChildAfter

Type: **HWND**

A handle to a child window. The search begins with the next child window in the Z order. The child window must be a direct child window of *hwndParent*, not just a descendant window.

If *hwndChildAfter* is **NULL**, the search begins with the first child window of *hwndParent*.

Note that if both *hwndParent* and *hwndChildAfter* are **NULL**, the function searches all top-level and message-only windows.

[in, optional] *lpszClass*

Type: **LPCTSTR**

The class name or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be placed in the low-order word of *lpszClass*; the high-order word must be zero.

If *lpszClass* is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined control-class names, or it can be `MAKEINTATOM(0x8000)`. In this latter case, 0x8000 is the atom for a menu class. For more information, see the Remarks section of this topic.

[in, optional] *lpszWindow*

Type: **LPCTSTR**

The window name (the window's title). If this parameter is **NULL**, all window names match.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the window that has the specified class and window names.

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

Remarks

The [FindWindowEx](#) function searches only direct child windows. It does not search other descendants.

If the *lpszWindow* parameter is not **NULL**, [FindWindowEx](#) calls the [GetWindowText](#) function to retrieve the window name for comparison. For a description of a potential problem that can arise, see the Remarks section of [GetWindowText](#).

An application can call this function in the following way.

```
FindWindowEx( NULL, NULL, MAKEINTATOM(0x8000), NULL );
```

Note that 0x8000 is the atom for a menu class. When an application calls this function, the function checks whether a context menu is being displayed that the application created.

ⓘ Note

The winuser.h header defines FindWindowEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[EnumWindows](#)

[FindWindow](#)

[GetClassName](#)

[GetWindowText](#)

Reference

[RegisterClass](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

FindWindowExW function (winuser.h)

Article 02/09/2023

Retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search.

Syntax

C++

```
HWND FindWindowExW(
    [in, optional] HWND     hWndParent,
    [in, optional] HWND     hWndChildAfter,
    [in, optional] LPCWSTR lpszClass,
    [in, optional] LPCWSTR lpszWindow
);
```

Parameters

[in, optional] hWndParent

Type: **HWND**

A handle to the parent window whose child windows are to be searched.

If *hwndParent* is **NULL**, the function uses the desktop window as the parent window. The function searches among windows that are child windows of the desktop.

If *hwndParent* is **HWND_MESSAGE**, the function searches all [message-only windows](#).

[in, optional] hWndChildAfter

Type: **HWND**

A handle to a child window. The search begins with the next child window in the Z order. The child window must be a direct child window of *hwndParent*, not just a descendant window.

If *hwndChildAfter* is **NULL**, the search begins with the first child window of *hwndParent*.

Note that if both *hwndParent* and *hwndChildAfter* are **NULL**, the function searches all top-level and message-only windows.

[in, optional] *lpszClass*

Type: **LPCTSTR**

The class name or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be placed in the low-order word of *lpszClass*; the high-order word must be zero.

If *lpszClass* is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined control-class names, or it can be `MAKEINTATOM(0x8000)`. In this latter case, 0x8000 is the atom for a menu class. For more information, see the Remarks section of this topic.

[in, optional] *lpszWindow*

Type: **LPCTSTR**

The window name (the window's title). If this parameter is **NULL**, all window names match.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the window that has the specified class and window names.

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

Remarks

The [FindWindowEx](#) function searches only direct child windows. It does not search other descendants.

If the *lpszWindow* parameter is not **NULL**, [FindWindowEx](#) calls the [GetWindowText](#) function to retrieve the window name for comparison. For a description of a potential problem that can arise, see the Remarks section of [GetWindowText](#).

An application can call this function in the following way.

```
FindWindowEx( NULL, NULL, MAKEINTATOM(0x8000), NULL );
```

Note that 0x8000 is the atom for a menu class. When an application calls this function, the function checks whether a context menu is being displayed that the application created.

ⓘ Note

The winuser.h header defines FindWindowEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[EnumWindows](#)

[FindWindow](#)

[GetClassName](#)

[GetWindowText](#)

Reference

[RegisterClass](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

FindWindowW function (winuser.h)

Article02/09/2023

Retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search.

To search child windows, beginning with a specified child window, use the [FindWindowEx](#) function.

Syntax

C++

```
HWND FindWindowW(
    [in, optional] LPCWSTR lpClassName,
    [in, optional] LPCWSTR lpWindowName
);
```

Parameters

[in, optional] *lpClassName*

Type: [LPCTSTR](#)

The class name or a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

If *lpClassName* points to a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined control-class names.

If *lpClassName* is **NULL**, it finds any window whose title matches the *lpWindowName* parameter.

[in, optional] *lpWindowName*

Type: [LPCTSTR](#)

The window name (the window's title). If this parameter is **NULL**, all window names match.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the window that has the specified class name and window name.

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

Remarks

If the *lpWindowName* parameter is not **NULL**, **FindWindow** calls the [GetWindowText](#) function to retrieve the window name for comparison. For a description of a potential problem that can arise, see the Remarks for [GetWindowText](#).

Examples

For an example, see [Retrieving the Number of Mouse Wheel Scroll Lines](#).

ⓘ Note

The winuser.h header defines **FindWindow** 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	winuser.h (include Windows.h)
Library	User32.lib

DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[EnumWindows](#)

[FindWindowEx](#)

[GetClassName](#)

[GetWindowText](#)

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetAltTabInfoA function (winuser.h)

Article 02/09/2023

Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window.

Syntax

C++

```
BOOL GetAltTabInfoA(
    [in, optional] HWND      hwnd,
    [in]          int       iItem,
    [in, out]      PALTTABINFO pati,
    [out, optional] LPSTR     pszItemText,
    [in]          UINT      cchItemText
);
```

Parameters

[in, optional] `hwnd`

Type: **HWND**

A handle to the window for which status information will be retrieved. This window must be the application-switching window.

[in] `iItem`

Type: **int**

The index of the icon in the application-switching window. If the `pszItemText` parameter is not **NULL**, the name of the item is copied to the `pszItemText` string. If this parameter is **-1**, the name of the item is not copied.

[in, out] `pati`

Type: **PALTTABINFO**

A pointer to an [ALTTABINFO](#) structure to receive the status information. Note that you must set the `csSize` member to `sizeof(ALTTABINFO)` before calling this function.

[out, optional] `pszItemText`

Type: **LPTSTR**

The name of the item. If this parameter is **NULL**, the name of the item is not copied.

[in] **cchItemText**

Type: **UINT**

The size, in characters, of the *pszItemText* buffer.

Return value

Type: **BOOL**

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 application-switching window enables you to switch to the most recently used application window. To display the application-switching window, press ALT+TAB. To select an application from the list, continue to hold ALT down and press TAB to move through the list. Add SHIFT to reverse direction through the list.

ⓘ Note

The winuser.h header defines GetAltTabInfo 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[ALTTABINFO](#)

[Conceptual](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetAltTabInfoW function (winuser.h)

Article 02/09/2023

Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window.

Syntax

C++

```
BOOL GetAltTabInfoW(
    [in, optional] HWND      hwnd,
    [in]          int       iItem,
    [in, out]      PALTTABINFO pati,
    [out, optional] LPWSTR    pszItemText,
    [in]          UINT      cchItemText
);
```

Parameters

[in, optional] `hwnd`

Type: **HWND**

A handle to the window for which status information will be retrieved. This window must be the application-switching window.

[in] `iItem`

Type: **int**

The index of the icon in the application-switching window. If the `pszItemText` parameter is not **NULL**, the name of the item is copied to the `pszItemText` string. If this parameter is **-1**, the name of the item is not copied.

[in, out] `pati`

Type: **PALTTABINFO**

A pointer to an [ALTTABINFO](#) structure to receive the status information. Note that you must set the `csSize` member to `sizeof(ALTTABINFO)` before calling this function.

[out, optional] `pszItemText`

Type: **LPTSTR**

The name of the item. If this parameter is **NULL**, the name of the item is not copied.

[in] **cchItemText**

Type: **UINT**

The size, in characters, of the *pszItemText* buffer.

Return value

Type: **BOOL**

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 application-switching window enables you to switch to the most recently used application window. To display the application-switching window, press ALT+TAB. To select an application from the list, continue to hold ALT down and press TAB to move through the list. Add SHIFT to reverse direction through the list.

Note

The winuser.h header defines GetAltTabInfo 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[ALTTABINFO](#)

[Conceptual](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetAncestor function (winuser.h)

Article 10/13/2021

Retrieves the handle to the ancestor of the specified window.

Syntax

C++

```
HWND GetAncestor(  
    [in] HWND hwnd,  
    [in] UINT gaFlags  
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the window whose ancestor is to be retrieved. If this parameter is the desktop window, the function returns **NULL**.

[in] gaFlags

Type: **UINT**

The ancestor to be retrieved. This parameter can be one of the following values.

Value	Meaning
GA_PARENT 1	Retrieves the parent window. This does not include the owner, as it does with the GetParent function.
GA_ROOT 2	Retrieves the root window by walking the chain of parent windows.
GA_ROOTOWNER 3	Retrieves the owned root window by walking the chain of parent and owner windows returned by GetParent .

Return value

Type: **HWND**

The return value is the handle to the ancestor window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[Conceptual](#)

[GetParent](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassInfoA function (winuser.h)

Article02/09/2023

Retrieves information about a window class.

Note The **GetClassInfo** function has been superseded by the **GetClassInfoEx** function. You can still use **GetClassInfo**, however, if you do not need information about the class small icon.

Syntax

C++

```
BOOL GetClassInfoA(  
    [in, optional] HINSTANCE hInstance,  
    [in]           LPCSTR    lpClassName,  
    [out]          LPWNDCLASSA lpWndClass  
) ;
```

Parameters

[in, optional] **hInstance**

Type: **HINSTANCE**

A handle to the instance of the application that created the class. To retrieve information about classes defined by the system (such as buttons or list boxes), set this parameter to **NULL**.

[in] **lpClassName**

Type: **LPCTSTR**

The class name. The name must be that of a preregistered class or a class registered by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function.

Alternatively, this parameter can be an atom. If so, it must be a class atom created by a previous call to [RegisterClass](#) or [RegisterClassEx](#). The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

[out] `lpWndClass`

Type: `WNDCLASS`

A pointer to a `WNDCLASS` structure that receives the information about the class.

Return value

Type: `BOOL`

If the function finds a matching class and successfully copies the data, the return value is nonzero.

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

Remarks

Note

The `winuser.h` header defines `GetClassInfo` 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	<code>winuser.h</code> (include <code>Windows.h</code>)
Library	<code>User32.lib</code>
DLL	<code>User32.dll</code>
API set	<code>ext-ms-win-ntuser-windowclass-l1-1-0</code> (introduced in Windows 8)

See also

Conceptual

[GetClassInfoEx](#)

[GetClassLong](#)

[GetClassName](#)

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[WNDCLASS](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassInfoExA function (winuser.h)

Article02/09/2023

Retrieves information about a window class, including a handle to the small icon associated with the window class. The [GetClassInfo](#) function does not retrieve a handle to the small icon.

Syntax

C++

```
BOOL GetClassInfoExA(
    [in, optional] HINSTANCE     hInstance,
    [in]           LPCSTR       lpszClass,
    [out]          LPWNDCLASSEX lpwcx
);
```

Parameters

[in, optional] `hInstance`

Type: `HINSTANCE`

A handle to the instance of the application that created the class. To retrieve information about classes defined by the system (such as buttons or list boxes), set this parameter to `NULL`.

[in] `lpszClass`

Type: `LPCTSTR`

The class name. The name must be that of a preregistered class or a class registered by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. Alternatively, this parameter can be a class atom created by a previous call to [RegisterClass](#) or [RegisterClassEx](#). The atom must be in the low-order word of `lpszClass`; the high-order word must be zero.

[out] `lpwcx`

Type: `LPWNDCLASSEX`

A pointer to a [WNDCLASSEX](#) structure that receives the information about the class.

Return value

Type: **BOOL**

If the function finds a matching class and successfully copies the data, the return value is nonzero.

If the function does not find a matching class and successfully copy the data, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Class atoms are created using the [RegisterClass](#) or [RegisterClassEx](#) function, not the [GlobalAddAtom](#) function.

Note

The winuser.h header defines `GetClassInfoEx` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetClassLong](#)

[GetClassName](#)

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassInfoExW function (winuser.h)

Article02/09/2023

Retrieves information about a window class, including a handle to the small icon associated with the window class. The [GetClassInfo](#) function does not retrieve a handle to the small icon.

Syntax

C++

```
BOOL GetClassInfoExW(
    [in, optional] HINSTANCE     hInstance,
    [in]           LPCWSTR       lpszClass,
    [out]          LPWNDCLASSEXW lpwcx
);
```

Parameters

[in, optional] `hInstance`

Type: `HINSTANCE`

A handle to the instance of the application that created the class. To retrieve information about classes defined by the system (such as buttons or list boxes), set this parameter to `NULL`.

[in] `lpszClass`

Type: `LPCTSTR`

The class name. The name must be that of a preregistered class or a class registered by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. Alternatively, this parameter can be a class atom created by a previous call to [RegisterClass](#) or [RegisterClassEx](#). The atom must be in the low-order word of `lpszClass`; the high-order word must be zero.

[out] `lpwcx`

Type: `LPWNDCLASSEX`

A pointer to a [WNDCLASSEX](#) structure that receives the information about the class.

Return value

Type: **BOOL**

If the function finds a matching class and successfully copies the data, the return value is nonzero.

If the function does not find a matching class and successfully copy the data, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Class atoms are created using the [RegisterClass](#) or [RegisterClassEx](#) function, not the [GlobalAddAtom](#) function.

Note

The winuser.h header defines `GetClassInfoEx` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetClassLong](#)

[GetClassName](#)

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassInfoW function (winuser.h)

Article02/09/2023

Retrieves information about a window class.

Note The **GetClassInfo** function has been superseded by the **GetClassInfoEx** function. You can still use **GetClassInfo**, however, if you do not need information about the class small icon.

Syntax

C++

```
BOOL GetClassInfoW(
    [in, optional] HINSTANCE hInstance,
    [in]           LPCWSTR   lpClassName,
    [out]          LPWNDCLASSW lpWndClass
);
```

Parameters

[in, optional] `hInstance`

Type: **HINSTANCE**

A handle to the instance of the application that created the class. To retrieve information about classes defined by the system (such as buttons or list boxes), set this parameter to **NULL**.

[in] `lpClassName`

Type: **LPCTSTR**

The class name. The name must be that of a preregistered class or a class registered by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function.

Alternatively, this parameter can be an atom. If so, it must be a class atom created by a previous call to [RegisterClass](#) or [RegisterClassEx](#). The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

[out] `lpWndClass`

Type: `LPWNDCLASS`

A pointer to a [WNDCLASS](#) structure that receives the information about the class.

Return value

Type: `BOOL`

If the function finds a matching class and successfully copies the data, the return value is nonzero.

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

Remarks

Note

The winuser.h header defines `GetClassInfo` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetClassInfoEx](#)

[GetClassLong](#)

[GetClassName](#)

Reference

[RegisterClass](#)

[RegisterClassEx](#)

[WNDCLASS](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassLongA function (winuser.h)

Article02/09/2023

Retrieves the specified 32-bit (DWORD) value from the [WNDCLASSEX](#) structure associated with the specified window.

Note If you are retrieving a pointer or a handle, this function has been superseded by the [GetClassLongPtr](#) function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.)

Syntax

C++

```
DWORD GetClassLongA(  
    [in] HWND hWnd,  
    [in] int nIndex  
);
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be retrieved. To retrieve a value from the extra class memory, specify the positive, zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third integer. To retrieve any other value from the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning

GCW_ATOM -32	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the RegisterClassEx function returns.
GCL_CBCLSEXTRA -20	Retrieves the size, in bytes, of the extra memory associated with the class.
GCL_CBWNDEXTRA -18	Retrieves the size, in bytes, of the extra window memory associated with each window in the class. For information on how to access this memory, see GetWindowLong .
GCL_HBRBACKGROUND -10	Retrieves a handle to the background brush associated with the class.
GCL_HCURSOR -12	Retrieves a handle to the cursor associated with the class.
GCL_HICON -14	Retrieves a handle to the icon associated with the class.
GCL_HICONSM -34	Retrieves a handle to the small icon associated with the class.
GCL_HMODULE -16	Retrieves a handle to the module that registered the class.
GCL_MENUNAME -8	Retrieves the address of the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Retrieves the window-class style bits.
GCL_WNDPROC -24	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.

Return value

Type: **DWORD**

If the function succeeds, the return value is the requested value.

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

Remarks

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

ⓘ Note

The winuser.h header defines GetClassLong 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[GetClassLongPtr](#)

[GetWindowLong](#)

Reference

[RegisterClassEx](#)

[SetClassLong](#)

[WNDCLASSEX](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassLongPtrA function (winuser.h)

Article02/09/2023

Retrieves the specified value from the [WNDCLASSEX](#) structure associated with the specified window.

Note To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [GetClassLongPtr](#). When compiling for 32-bit Windows, [GetClassLongPtr](#) is defined as a call to the [GetClassLong](#) function.

Syntax

C++

```
ULONG_PTR GetClassLongPtrA(  
    [in] HWND hWnd,  
    [in] int nIndex  
>;
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be retrieved. To retrieve a value from the extra class memory, specify the positive, zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of extra class memory, minus eight; for example, if you specified 24 or more bytes of extra class memory, a value of 16 would be an index to the third integer. To retrieve any other value from the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning

GCW_ATOM -32	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the RegisterClassEx function returns.
GCL_CBCLSEXTRA -20	Retrieves the size, in bytes, of the extra memory associated with the class.
GCL_CBWNDEXTRA -18	Retrieves the size, in bytes, of the extra window memory associated with each window in the class. For information on how to access this memory, see GetWindowLongPtr .
GCLP_HBRBACKGROUND -10	Retrieves a handle to the background brush associated with the class.
GCLP_HCURSOR -12	Retrieves a handle to the cursor associated with the class.
GCLP_HICON -14	Retrieves a handle to the icon associated with the class.
GCLP_HICONSM -34	Retrieves a handle to the small icon associated with the class.
GCLP_HMODULE -16	Retrieves a handle to the module that registered the class.
GCLP_MENUNAME -8	Retrieves the pointer to the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Retrieves the window-class style bits.
GCLP_WNDPROC -24	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.

Return value

Type: **ULONG_PTR**

If the function succeeds, the return value is the requested value.

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

Remarks

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

 **Note**

The winuser.h header defines `GetClassLongPtr` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[GetWindowLongPtr](#)

[Reference](#)

[RegisterClassEx](#)

[SetClassLongPtr](#)

[WNDCLASSEX](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassLongPtrW function (winuser.h)

Article02/09/2023

Retrieves the specified value from the [WNDCLASSEX](#) structure associated with the specified window.

Note To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [GetClassLongPtr](#). When compiling for 32-bit Windows, [GetClassLongPtr](#) is defined as a call to the [GetClassLong](#) function.

Syntax

C++

```
ULONG_PTR GetClassLongPtrW(  
    [in] HWND hWnd,  
    [in] int nIndex  
>;
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be retrieved. To retrieve a value from the extra class memory, specify the positive, zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of extra class memory, minus eight; for example, if you specified 24 or more bytes of extra class memory, a value of 16 would be an index to the third integer. To retrieve any other value from the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning

GCW_ATOM -32	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the RegisterClassEx function returns.
GCL_CBCLSEXTRA -20	Retrieves the size, in bytes, of the extra memory associated with the class.
GCL_CBWNDEXTRA -18	Retrieves the size, in bytes, of the extra window memory associated with each window in the class. For information on how to access this memory, see GetWindowLongPtr .
GCLP_HBRBACKGROUND -10	Retrieves a handle to the background brush associated with the class.
GCLP_HCURSOR -12	Retrieves a handle to the cursor associated with the class.
GCLP_HICON -14	Retrieves a handle to the icon associated with the class.
GCLP_HICONSM -34	Retrieves a handle to the small icon associated with the class.
GCLP_HMODULE -16	Retrieves a handle to the module that registered the class.
GCLP_MENUNAME -8	Retrieves the pointer to the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Retrieves the window-class style bits.
GCLP_WNDPROC -24	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.

Return value

Type: **ULONG_PTR**

If the function succeeds, the return value is the requested value.

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

Remarks

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

 **Note**

The winuser.h header defines `GetClassLongPtr` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[GetWindowLongPtr](#)

[Reference](#)

[RegisterClassEx](#)

[SetClassLongPtr](#)

[WNDCLASSEX](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassLongW function (winuser.h)

Article02/09/2023

Retrieves the specified 32-bit (DWORD) value from the [WNDCLASSEX](#) structure associated with the specified window.

Note If you are retrieving a pointer or a handle, this function has been superseded by the [GetClassLongPtr](#) function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.)

Syntax

C++

```
DWORD GetClassLongW(
    [in] HWND hWnd,
    [in] int nIndex
);
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be retrieved. To retrieve a value from the extra class memory, specify the positive, zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third integer. To retrieve any other value from the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning

GCW_ATOM -32	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the RegisterClassEx function returns.
GCL_CBCLSEXTRA -20	Retrieves the size, in bytes, of the extra memory associated with the class.
GCL_CBWNDEXTRA -18	Retrieves the size, in bytes, of the extra window memory associated with each window in the class. For information on how to access this memory, see GetWindowLong .
GCL_HBRBACKGROUND -10	Retrieves a handle to the background brush associated with the class.
GCL_HCURSOR -12	Retrieves a handle to the cursor associated with the class.
GCL_HICON -14	Retrieves a handle to the icon associated with the class.
GCL_HICONSM -34	Retrieves a handle to the small icon associated with the class.
GCL_HMODULE -16	Retrieves a handle to the module that registered the class.
GCL_MENUNAME -8	Retrieves the address of the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Retrieves the window-class style bits.
GCL_WNDPROC -24	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.

Return value

Type: **DWORD**

If the function succeeds, the return value is the requested value.

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

Remarks

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

ⓘ Note

The winuser.h header defines GetClassLong 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[GetClassLongPtr](#)

[GetWindowLong](#)

Reference

[RegisterClassEx](#)

[SetClassLong](#)

[WNDCLASSEX](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassName function (winuser.h)

Article 08/02/2022

Retrieves the name of the class to which the specified window belongs.

Syntax

C++

```
int GetClassName(
    [in] HWND     hWnd,
    [out] LPTSTR  lpClassName,
    [in] int      nMaxCount
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[out] lpClassName

Type: **LPTSTR**

The class name string.

[in] nMaxCount

Type: **int**

The length of the *lpClassName* buffer, in characters. The buffer must be large enough to include the terminating null character; otherwise, the class name string is truncated to *nMaxCount-1* characters.

Return value

Type: **int**

If the function succeeds, the return value is the number of characters copied 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 function](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[FindWindowA function](#), [GetClassInfoA function](#), [GetClassLongA function](#), [GetClassWord function](#), [Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassNameA function (winuser.h)

Article 02/09/2023

Retrieves the name of the class to which the specified window belongs.

Syntax

C++

```
int GetClassNameA(
    [in]  HWND  hWnd,
    [out] LPSTR lpClassName,
    [in]  int   nMaxCount
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[out] lpClassName

Type: **LPTSTR**

The class name string.

[in] nMaxCount

Type: **int**

The length of the *lpClassName* buffer, in characters. The buffer must be large enough to include the terminating null character; otherwise, the class name string is truncated to *nMaxCount-1* characters.

Return value

Type: **int**

If the function succeeds, the return value is the number of characters copied 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](#).

Remarks

Note

The winuser.h header defines GetClassName 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[FindWindow](#)

[GetClassInfo](#)

[GetClassLong](#)

[GetClassWord](#)

[Reference](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassNameW function (winuser.h)

Article 02/09/2023

Retrieves the name of the class to which the specified window belongs.

Syntax

C++

```
int GetClassNameW(
    [in] HWND     hWnd,
    [out] LPWSTR  lpClassName,
    [in] int      nMaxCount
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[out] lpClassName

Type: **LPTSTR**

The class name string.

[in] nMaxCount

Type: **int**

The length of the *lpClassName* buffer, in characters. The buffer must be large enough to include the terminating null character; otherwise, the class name string is truncated to *nMaxCount-1* characters.

Return value

Type: **int**

If the function succeeds, the return value is the number of characters copied 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](#).

Remarks

Note

The winuser.h header defines GetClassName 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[FindWindow](#)

[GetClassInfo](#)

[GetClassLong](#)

[GetClassWord](#)

[Reference](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClassWord function (winuser.h)

Article10/13/2021

Retrieves the 16-bit (WORD) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

Note This function is deprecated for any use other than *nIndex* set to **GCW_ATOM**. The function is provided only for compatibility with 16-bit versions of Windows. Applications should use the **GetClassLong** or **GetClassLongPtr** function.

Syntax

C++

```
WORD GetClassWord(  
    [in] HWND hWnd,  
    [in] int nIndex  
)
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: **int**

The zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of class memory, minus two; for example, if you specified 10 or more bytes of extra class memory, a value of eight would be an index to the fifth 16-bit integer. There is an additional valid value as shown in the following table.

Value	Meaning
GCW_ATOM	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the
-32	

Return value

Type: WORD

If the function succeeds, the return value is the requested 16-bit value.

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

Remarks

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASS](#) structure used with the [RegisterClass](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[GetClassLong](#)

[Reference](#)

[RegisterClass](#)

[RegisterClassEx](#)

[SetClassWord](#)

[WNDCLASS](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetClientRect function (winuser.h)

Article 10/13/2021

Retrieves the coordinates of a window's client area. The client coordinates specify the upper-left and lower-right corners of the client area. Because client coordinates are relative to the upper-left corner of a window's client area, the coordinates of the upper-left corner are (0,0).

Syntax

C++

```
BOOL GetClientRect(  
    [in] HWND hWnd,  
    [out] LPRECT lpRect  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose client coordinates are to be retrieved.

[out] lpRect

Type: **LPRECT**

A pointer to a [RECT](#) structure that receives the client coordinates. The **left** and **top** members are zero. The **right** and **bottom** members contain the width and height of the window.

Return value

Type: **BOOL**

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 conformance with conventions for the [RECT](#) structure, the bottom-right coordinates of the returned rectangle are exclusive. In other words, the pixel at (**right**, **bottom**) lies immediately outside the rectangle.

Examples

For example, see [Creating, Enumerating, and Sizing Child Windows](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetWindowRect](#)

[Other Resources](#)

[RECT](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetDesktopWindow function (winuser.h)

Article 06/29/2021

Retrieves a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which other windows are painted.

Syntax

C++

```
HWND GetDesktopWindow();
```

Return value

Type: **HWND**

The return value is a handle to the desktop window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetWindow](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetForegroundWindow function (winuser.h)

Article 06/29/2021

Retrieves a handle to the foreground window (the window with which the user is currently working). The system assigns a slightly higher priority to the thread that creates the foreground window than it does to other threads.

Syntax

C++

```
HWND GetForegroundWindow();
```

Return value

Type: **HWND**

The return value is a handle to the foreground window. The foreground window can be **NULL** in certain circumstances, such as when a window is losing activation.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

Reference

[SetForegroundWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetGUIThreadInfo function (winuser.h)

Article 10/13/2021

Retrieves information about the active window or a specified GUI thread.

Syntax

C++

```
BOOL GetGUIThreadInfo(
    [in]      DWORD      idThread,
    [in, out] PGUITHREADINFO pgui
);
```

Parameters

[in] idThread

Type: **DWORD**

The identifier for the thread for which information is to be retrieved. To retrieve this value, use the [GetWindowThreadProcessId](#) function. If this parameter is **NULL**, the function returns information for the foreground thread.

[in, out] pgui

Type: **LPGUITHREADINFO**

A pointer to a [GUITHREADINFO](#) structure that receives information describing the thread. Note that you must set the **cbSize** member to `sizeof(GUITHREADINFO)` before calling this function.

Return value

Type: **BOOL**

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 succeeds even if the active window is not owned by the calling process. If the specified thread does not exist or have an input queue, the function will fail.

This function is useful for retrieving out-of-context information about a thread. The information retrieved is the same as if an application retrieved the information about itself.

For an edit control, the returned **rcCaret** rectangle contains the caret plus information on text direction and padding. Thus, it may not give the correct position of the cursor. The Sans Serif font uses four characters for the cursor:

Cursor character	Unicode code point
CURSOR_LTR	0xf00c
CURSOR_RTL	0xf00d
CURSOR_THAI	0xf00e
CURSOR_USA	0xffff (this is a marker value with no associated glyph)

To get the actual insertion point in the **rcCaret** rectangle, perform the following steps.

1. Call [GetKeyboardLayout](#) to retrieve the current input language.
2. Determine the character used for the cursor, based on the current input language.
3. Call [CreateFont](#) using Sans Serif for the font, the height given by **rcCaret**, and a width of `zero`. For *fnWeight*, call `SystemParametersInfo(SPI_GETCARETWIDTH, 0, &pvParam, 0)`. If *pvParam* is greater than 1, set *fnWeight* to 700, otherwise set *fnWeight* to 400.
4. Select the font into a device context (DC) and use [GetCharABCWidths](#) to get the **B** width of the appropriate cursor character.
5. Add the **B** width to **rcCaret.left** to obtain the actual insertion point.

The function may not return valid window handles in the [GUITHREADINFO](#) structure when called to retrieve information for the foreground thread, such as when a window is losing activation.

DPI Virtualization

The coordinates returned in the **rcCaret** rect of the [GUITHREADINFO](#) struct are logical coordinates in terms of the window associated with the caret. They are not virtualized

into the mode of the calling thread.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[Conceptual](#)

[GUILTHREADINFO](#)

[GetCursorInfo](#)

[GetWindowThreadProcessId](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetInputState function (winuser.h)

Article 06/29/2021

Determines whether there are mouse-button or keyboard messages in the calling thread's message queue.

Syntax

C++

```
BOOL GetInputState();
```

Return value

Type: **BOOL**

If the queue contains one or more new mouse-button or keyboard messages, the return value is nonzero.

If there are no new mouse-button or keyboard messages in the queue, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[GetQueueStatus](#)

[Messages and Message Queues](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetLastActivePopup function (winuser.h)

Article10/13/2021

Determines which pop-up window owned by the specified window was most recently active.

Syntax

C++

```
HWND GetLastActivePopup(  
    [in] HWND hWnd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the owner window.

Return value

Type: **HWND**

The return value identifies the most recently active pop-up window. The return value is the same as the *hWnd* parameter, if any of the following conditions are met:

- The window identified by hWnd was most recently active.
- The window identified by hWnd does not own any pop-up windows.
- The window identified by hWnd is not a top-level window, or it is owned by another window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
--------------------------	---

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[AnyPopup](#)

[Conceptual](#)

[Reference](#)

[ShowOwnedPopups](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetLayeredWindowAttributes function (winuser.h)

Article 10/13/2021

Retrieves the opacity and transparency color key of a layered window.

Syntax

C++

```
BOOL GetLayeredWindowAttributes(
    [in]             HWND      hwnd,
    [out, optional] COLORREF *pcrKey,
    [out, optional] BYTE     *pbAlpha,
    [out, optional] DWORD    *pdwFlags
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the layered window. A layered window is created by specifying **WS_EX_LAYERED** when creating the window with the [CreateWindowEx](#) function or by setting **WS_EX_LAYERED** using [SetWindowLong](#) after the window has been created.

[out, optional] pcrKey

Type: [**COLORREF***](#)

A pointer to a [**COLORREF**](#) value that receives the transparency color key to be used when composing the layered window. All pixels painted by the window in this color will be transparent. This can be **NULL** if the argument is not needed.

[out, optional] pbAlpha

Type: [**BYTE***](#)

The Alpha value used to describe the opacity of the layered window. Similar to the **SourceConstantAlpha** member of the [**BLENDFUNCTION**](#) structure. When the variable referred to by *pbAlpha* is 0, the window is completely transparent. When the variable

referred to by *pbAlpha* is 255, the window is opaque. This can be **NULL** if the argument is not needed.

[out, optional] *pdwFlags*

Type: **DWORD***

A layering flag. This parameter can be **NULL** if the value is not needed. The layering flag can be one or more of the following values.

Value	Meaning
LWA_ALPHA 0x00000002	Use <i>pbAlpha</i> to determine the opacity of the layered window.
LWA_COLORKEY 0x00000001	Use <i>pcrKey</i> as the transparency color.

Return value

Type: **BOOL**

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

[GetLayeredWindowAttributes](#) can be called only if the application has previously called [SetLayeredWindowAttributes](#) on the window. The function will fail if the layered window was setup with [UpdateLayeredWindow](#).

For more information, see [Using Layered Windows](#).

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[CreateWindowEx](#)

Reference

[SetLayeredWindowAttributes](#)

[SetWindowLong](#)

[Using Windows](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetMessage function (winuser.h)

Article03/11/2023

Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval.

GetMessage functions like [PeekMessage](#), however, **GetMessage** blocks until a message is posted before returning.

Syntax

C++

```
BOOL GetMessage(
    [out]     LPMMSG lpMsg,
    [in, optional] HWND hWnd,
    [in]      UINT wMsgFilterMin,
    [in]      UINT wMsgFilterMax
);
```

Parameters

[out] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that receives message information from the thread's message queue.

[in, optional] hWnd

Type: [HWND](#)

A handle to the window whose messages are to be retrieved. The window must belong to the current thread.

If *hWnd* is **NULL**, **GetMessage** retrieves messages for any window that belongs to the current thread, and any messages on the current thread's message queue whose *hwnd* value is **NULL** (see the [MSG](#) structure). Therefore if *hWnd* is **NULL**, both window messages and thread messages are processed.

If *hWnd* is **-1**, **GetMessage** retrieves only messages on the current thread's message queue whose *hwnd* value is **NULL**, that is, thread messages as posted by [PostMessage](#)

(when the *hWnd* parameter is **NULL**) or [PostThreadMessage](#).

[in] *wMsgFilterMin*

Type: **UINT**

The integer value of the lowest message value to be retrieved. Use **WM_KEYFIRST** (0x0100) to specify the first keyboard message or **WM_MOUSEFIRST** (0x0200) to specify the first mouse message.

Use [WM_INPUT](#) here and in *wMsgFilterMax* to specify only the **WM_INPUT** messages.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed).

[in] *wMsgFilterMax*

Type: **UINT**

The integer value of the highest message value to be retrieved. Use **WM_KEYLAST** to specify the last keyboard message or **WM_MOUSELAST** to specify the last mouse message.

Use [WM_INPUT](#) here and in *wMsgFilterMin* to specify only the **WM_INPUT** messages.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed).

Return value

Type: **BOOL**

If the function retrieves a message other than [WM_QUIT](#), the return value is nonzero.

If the function retrieves the [WM_QUIT](#) message, the return value is zero.

If there is an error, the return value is -1. For example, the function fails if *hWnd* is an invalid window handle or *lpMsg* is an invalid pointer. To get extended error information, call [GetLastError](#).

Because the return value can be nonzero, zero, or -1, avoid code like this:

```
while (GetMessage( lpMsg, hWnd, 0, 0)) ...
```

The possibility of a -1 return value in the case that *hWnd* is an invalid parameter (such as referring to a window that has already been destroyed) means that such code can lead to fatal application errors. Instead, use code like this:

```
BOOL bRet;

while( (bRet = GetMessage( &msg, hWnd, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Remarks

An application typically uses the return value to determine whether to end the main message loop and exit the program.

The **GetMessage** function retrieves messages associated with the window identified by the *hWnd* parameter or any of its children, as specified by the **IsChild** function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. Note that an application can only use the low word in the *wMsgFilterMin* and *wMsgFilterMax* parameters; the high word is reserved for the system.

Note that **GetMessage** always retrieves **WM_QUIT** messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

During this call, the system delivers pending, nonqueued messages, that is, messages sent to windows owned by the calling thread using the **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, or **SendNotifyMessage** function. Then the first queued message that matches the specified filter is retrieved. The system may also process internal events. If no filter is specified, messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events

- Sent messages (again)
- [WM_PAINT](#) messages
- [WM_TIMER](#) messages

To retrieve input messages before posted messages, use the *wMsgFilterMin* and *wMsgFilterMax* parameters.

[GetMessage](#) does not remove [WM_PAINT](#) messages from the queue. The messages remain in the queue until processed.

If a top-level window stops responding to messages for more than several seconds, the system considers the window to be not responding and replaces it with a ghost window that has the same z-order, location, size, and visual attributes. This allows the user to move it, resize it, or even close the application. However, these are the only actions available because the application is actually not responding. When in the debugger mode, the system does not generate a ghost window.

DPI Virtualization

This API does not participate in DPI virtualization. The output is in the mode of the window that the message is targeting. The calling thread is not taken into consideration.

Examples

For an example, see [Creating a Message Loop](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[IsChild](#)

[MSG](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[PostMessage](#)

[PostThreadMessage](#)

Reference

[WaitMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetMessageA function (winuser.h)

Article 02/09/2023

Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval.

Unlike [GetMessage](#), the [PeekMessage](#) function does not wait for a message to be posted before returning.

Syntax

C++

```
BOOL GetMessageA(
    [out]         LPMMSG lpMsg,
    [in, optional] HWND hWnd,
    [in]          UINT  wMsgFilterMin,
    [in]          UINT  wMsgFilterMax
);
```

Parameters

[out] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that receives message information from the thread's message queue.

[in, optional] hWnd

Type: [HWND](#)

A handle to the window whose messages are to be retrieved. The window must belong to the current thread.

If *hWnd* is **NULL**, [GetMessage](#) retrieves messages for any window that belongs to the current thread, and any messages on the current thread's message queue whose *hwnd* value is **NULL** (see the [MSG](#) structure). Therefore if *hWnd* is **NULL**, both window messages and thread messages are processed.

If *hWnd* is **-1**, [GetMessage](#) retrieves only messages on the current thread's message queue whose *hwnd* value is **NULL**, that is, thread messages as posted by [PostMessage](#)

(when the *hWnd* parameter is **NULL**) or [PostThreadMessage](#).

[in] *wMsgFilterMin*

Type: **UINT**

The integer value of the lowest message value to be retrieved. Use **WM_KEYFIRST** (0x0100) to specify the first keyboard message or **WM_MOUSEFIRST** (0x0200) to specify the first mouse message.

Use [WM_INPUT](#) here and in *wMsgFilterMax* to specify only the **WM_INPUT** messages.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed).

[in] *wMsgFilterMax*

Type: **UINT**

The integer value of the highest message value to be retrieved. Use **WM_KEYLAST** to specify the last keyboard message or **WM_MOUSELAST** to specify the last mouse message.

Use [WM_INPUT](#) here and in *wMsgFilterMin* to specify only the **WM_INPUT** messages.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed).

Return value

Type: **BOOL**

If the function retrieves a message other than [WM_QUIT](#), the return value is nonzero.

If the function retrieves the [WM_QUIT](#) message, the return value is zero.

If there is an error, the return value is -1. For example, the function fails if *hWnd* is an invalid window handle or *lpMsg* is an invalid pointer. To get extended error information, call [GetLastError](#).

Because the return value can be nonzero, zero, or -1, avoid code like this:

```
while (GetMessage( lpMsg, hWnd, 0, 0)) ...
```

The possibility of a -1 return value in the case that *hWnd* is an invalid parameter (such as referring to a window that has already been destroyed) means that such code can lead to fatal application errors. Instead, use code like this:

```
BOOL bRet;

while( (bRet = GetMessage( &msg, hWnd, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Remarks

An application typically uses the return value to determine whether to end the main message loop and exit the program.

The **GetMessage** function retrieves messages associated with the window identified by the *hWnd* parameter or any of its children, as specified by the **IsChild** function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. Note that an application can only use the low word in the *wMsgFilterMin* and *wMsgFilterMax* parameters; the high word is reserved for the system.

Note that **GetMessage** always retrieves **WM_QUIT** messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

During this call, the system delivers pending, nonqueued messages, that is, messages sent to windows owned by the calling thread using the **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, or **SendNotifyMessage** function. Then the first queued message that matches the specified filter is retrieved. The system may also process internal events. If no filter is specified, messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events

- Sent messages (again)
- [WM_PAINT](#) messages
- [WM_TIMER](#) messages

To retrieve input messages before posted messages, use the *wMsgFilterMin* and *wMsgFilterMax* parameters.

[GetMessage](#) does not remove [WM_PAINT](#) messages from the queue. The messages remain in the queue until processed.

If a top-level window stops responding to messages for more than several seconds, the system considers the window to be not responding and replaces it with a ghost window that has the same z-order, location, size, and visual attributes. This allows the user to move it, resize it, or even close the application. However, these are the only actions available because the application is actually not responding. When in the debugger mode, the system does not generate a ghost window.

DPI Virtualization

This API does not participate in DPI virtualization. The output is in the mode of the window that the message is targeting. The calling thread is not taken into consideration.

Examples

For an example, see [Creating a Message Loop](#).

Note

The winuser.h header defines [GetMessage](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[IsChild](#)

[MSG](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[PostMessage](#)

[PostThreadMessage](#)

[Reference](#)

[WaitMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetMessageExtraInfo function (winuser.h)

Article 06/29/2021

Retrieves the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue.

Syntax

C++

```
LPARAM GetMessageExtraInfo();
```

Return value

Type: LPARAM

The return value specifies the extra information. The meaning of the extra information is device specific.

Remarks

To set a thread's extra message information, use the [SetMessageExtraInfo](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib

DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetMessage](#)

[Messages and Message Queues](#)

[PeekMessage](#)

Reference

[SetMessageExtraInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetMessagePos function (winuser.h)

Article11/19/2022

Retrieves the cursor position for the last message retrieved by the [GetMessage](#) function.

To determine the current position of the cursor, use the [GetCursorPos](#) function.

Syntax

C++

```
DWORD GetMessagePos();
```

Return value

Type: **DWORD**

The return value specifies the x- and y-coordinates of the cursor position. The x-coordinate is the low order **short** and the y-coordinate is the high-order **short**.

Remarks

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the [MAKEPOINTS](#) macro to obtain a [POINTS](#) structure from the return value. You can also use the [GET_X_LPARAM](#) or [GET_Y_LPARAM](#) macro to extract the x- or y-coordinate.

Important Do not use the **LOWORD** or **HIGHWORD** macros to extract the x- and y-coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[GetCursorPos](#)

[GetMessage](#)

[GetMessageTime](#)

[HIWORD](#)

[LOWORD](#)

[MAKEPOINTS](#)

[Messages and Message Queues](#)

Other Resources

[POINTS](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

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

GetMessageTime function (winuser.h)

Article 06/29/2021

Retrieves the message time for the last message retrieved by the [GetMessage](#) function. The time is a long integer that specifies the elapsed time, in milliseconds, from the time the system was started to the time the message was created (that is, placed in the thread's message queue).

Syntax

C++

```
LONG GetMessageTime();
```

Return value

Type: **LONG**

The return value specifies the message time.

Remarks

The return value from the **GetMessageTime** function does not necessarily increase between subsequent messages, because the value wraps to the minimum value for a long integer if the timer count exceeds the maximum value for a long integer.

To calculate time delays between messages, subtract the time of the first message from the time of the second message (ignoring overflow) and compare the result of the subtraction against the desired delay amount.

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-1 (introduced in Windows 8.1)

See also

Conceptual

[GetMessage](#)

[GetMessagePos](#)

[Messages and Message Queues](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

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

GetMessageW function (winuser.h)

Article 02/09/2023

Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval.

Unlike [GetMessage](#), the [PeekMessage](#) function does not wait for a message to be posted before returning.

Syntax

C++

```
BOOL GetMessage(
    [out]     LPMMSG lpMsg,
    [in, optional] HWND hWnd,
    [in]      UINT wMsgFilterMin,
    [in]      UINT wMsgFilterMax
);
```

Parameters

[out] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that receives message information from the thread's message queue.

[in, optional] hWnd

Type: [HWND](#)

A handle to the window whose messages are to be retrieved. The window must belong to the current thread.

If *hWnd* is **NULL**, [GetMessage](#) retrieves messages for any window that belongs to the current thread, and any messages on the current thread's message queue whose *hwnd* value is **NULL** (see the [MSG](#) structure). Therefore if *hWnd* is **NULL**, both window messages and thread messages are processed.

If *hWnd* is **-1**, [GetMessage](#) retrieves only messages on the current thread's message queue whose *hwnd* value is **NULL**, that is, thread messages as posted by [PostMessage](#)

(when the *hWnd* parameter is **NULL**) or [PostThreadMessage](#).

[in] *wMsgFilterMin*

Type: **UINT**

The integer value of the lowest message value to be retrieved. Use **WM_KEYFIRST** (0x0100) to specify the first keyboard message or **WM_MOUSEFIRST** (0x0200) to specify the first mouse message.

Use [WM_INPUT](#) here and in *wMsgFilterMax* to specify only the **WM_INPUT** messages.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed).

[in] *wMsgFilterMax*

Type: **UINT**

The integer value of the highest message value to be retrieved. Use **WM_KEYLAST** to specify the last keyboard message or **WM_MOUSELAST** to specify the last mouse message.

Use [WM_INPUT](#) here and in *wMsgFilterMin* to specify only the **WM_INPUT** messages.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed).

Return value

Type: **BOOL**

If the function retrieves a message other than [WM_QUIT](#), the return value is nonzero.

If the function retrieves the [WM_QUIT](#) message, the return value is zero.

If there is an error, the return value is -1. For example, the function fails if *hWnd* is an invalid window handle or *lpMsg* is an invalid pointer. To get extended error information, call [GetLastError](#).

Because the return value can be nonzero, zero, or -1, avoid code like this:

```
while (GetMessage( lpMsg, hWnd, 0, 0)) ...
```

The possibility of a -1 return value in the case that *hWnd* is an invalid parameter (such as referring to a window that has already been destroyed) means that such code can lead to fatal application errors. Instead, use code like this:

```
BOOL bRet;

while( (bRet = GetMessage( &msg, hWnd, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Remarks

An application typically uses the return value to determine whether to end the main message loop and exit the program.

The **GetMessage** function retrieves messages associated with the window identified by the *hWnd* parameter or any of its children, as specified by the **IsChild** function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. Note that an application can only use the low word in the *wMsgFilterMin* and *wMsgFilterMax* parameters; the high word is reserved for the system.

Note that **GetMessage** always retrieves **WM_QUIT** messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

During this call, the system delivers pending, nonqueued messages, that is, messages sent to windows owned by the calling thread using the **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, or **SendNotifyMessage** function. Then the first queued message that matches the specified filter is retrieved. The system may also process internal events. If no filter is specified, messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events

- Sent messages (again)
- [WM_PAINT](#) messages
- [WM_TIMER](#) messages

To retrieve input messages before posted messages, use the *wMsgFilterMin* and *wMsgFilterMax* parameters.

[GetMessage](#) does not remove [WM_PAINT](#) messages from the queue. The messages remain in the queue until processed.

If a top-level window stops responding to messages for more than several seconds, the system considers the window to be not responding and replaces it with a ghost window that has the same z-order, location, size, and visual attributes. This allows the user to move it, resize it, or even close the application. However, these are the only actions available because the application is actually not responding. When in the debugger mode, the system does not generate a ghost window.

DPI Virtualization

This API does not participate in DPI virtualization. The output is in the mode of the window that the message is targeting. The calling thread is not taken into consideration.

Examples

For an example, see [Creating a Message Loop](#).

Note

The winuser.h header defines [GetMessage](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[IsChild](#)

[MSG](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[PostMessage](#)

[PostThreadMessage](#)

[Reference](#)

[WaitMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetNextWindow macro (winuser.h)

Article 10/13/2021

Retrieves a handle to the next or previous window in the [Z-Order](#). The next window is below the specified window; the previous window is above.

If the specified window is a topmost window, the function searches for a topmost window. If the specified window is a top-level window, the function searches for a top-level window. If the specified window is a child window, the function searches for a child window.

Syntax

C++

```
void GetNextWindow(
    [in] hWnd,
    [in] wCmd
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to a window. The window handle retrieved is relative to this window, based on the value of the *wCmd* parameter.

[in] wCmd

Type: **UINT**

Indicates whether the function returns a handle to the next window or the previous window. This parameter can be either of the following values.

Value	Meaning
GW_HWNDNEXT 2	Returns a handle to the window below the given window.
GW_HWNDPREV 3	Returns a handle to the window above the given window.

Return value

None

Remarks

This function is implemented as a call to the [GetWindow](#) function.

syntax

```
#define GetNextWindow(hWnd, wCmd) GetWindow(hWnd, wCmd)
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[GetTopWindow](#)

[GetWindow](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetParent function (winuser.h)

Article10/13/2021

Retrieves a handle to the specified window's parent or owner.

To retrieve a handle to a specified ancestor, use the [GetAncestor](#) function.

Syntax

C++

```
HWND GetParent(  
    [in] HWND hWnd  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose parent window handle is to be retrieved.

Return value

Type: **HWND**

If the window is a child window, the return value is a handle to the parent window. If the window is a top-level window with the **WS_POPUP** style, the return value is a handle to the owner window.

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

This function typically fails for one of the following reasons:

- The window is a top-level window that is unowned or does not have the **WS_POPUP** style.
- The owner window has **WS_POPUP** style.

Remarks

To obtain a window's owner window, instead of using [GetParent](#), use [GetWindow](#) with the **GW_OWNER** flag. To obtain the parent window and not the owner, instead of using [GetParent](#), use [GetAncestor](#) with the **GA_PARENT** flag.

Examples

For an example, see [Initializing a Dialog Box](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetAncestor](#)

[GetWindow](#)

Reference

[SetParent](#)

[Windows](#)

[Windows Styles](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetProcessDefaultLayout function (winuser.h)

Article 10/13/2021

Retrieves the default layout that is used when windows are created with no parent or owner.

Syntax

C++

```
BOOL GetProcessDefaultLayout(
    [out] DWORD *pdwDefaultLayout
);
```

Parameters

[out] pdwDefaultLayout

Type: **DWORD***

The current default process layout. For a list of values, see [SetProcessDefaultLayout](#).

Return value

Type: **BOOL**

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 layout specifies how text and graphics are laid out in a window; the default is left to right. The **GetProcessDefaultLayout** function lets you know if the default layout has changed, from using [SetProcessDefaultLayout](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[Reference](#)

[SetProcessDefaultLayout](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetPropA function (winuser.h)

Article 02/09/2023

Retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the [SetProp](#) function.

Syntax

C++

```
HANDLE GetPropA(
    [in] HWND    hWnd,
    [in] LPCSTR lpString
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be searched.

[in] lpString

Type: **LPCTSTR**

An atom that identifies a string. If this parameter is an atom, it must have been created by using the [GlobalAddAtom](#) function. The atom, a 16-bit value, must be placed in the low-order word of the *lpString* parameter; the high-order word must be zero.

Return value

Type: **HANDLE**

If the property list contains the string, the return value is the associated data handle. Otherwise, the return value is **NULL**.

Remarks

ⓘ Note

The winuser.h header defines GetProp 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GlobalAddAtom](#)

[Reference](#)

[SetProp](#)

[Window Properties](#)

[ITaskbarList2::MarkFullscreenWindow](#)

Feedback



Was this page helpful? [!\[\]\(e5f24984e6cc7de6e28602972ac25d00_img.jpg\) Yes](#) [!\[\]\(f5ac6990a9f32ec4721d8771757be80f_img.jpg\) No](#)

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

GetPropW function (winuser.h)

Article 02/09/2023

Retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the [SetProp](#) function.

Syntax

C++

```
HANDLE GetPropW(
    [in] HWND     hWnd,
    [in] LPCWSTR lpString
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be searched.

[in] lpString

Type: **LPCTSTR**

An atom that identifies a string. If this parameter is an atom, it must have been created by using the [GlobalAddAtom](#) function. The atom, a 16-bit value, must be placed in the low-order word of the *lpString* parameter; the high-order word must be zero.

Return value

Type: **HANDLE**

If the property list contains the string, the return value is the associated data handle. Otherwise, the return value is **NULL**.

Remarks

ⓘ Note

The winuser.h header defines GetProp 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GlobalAddAtom](#)

[Reference](#)

[SetProp](#)

[Window Properties](#)

[ITaskbarList2::MarkFullscreenWindow](#)

Feedback



Was this page helpful? [!\[\]\(d19121b7a08da8c171c6bfe2445a1c97_img.jpg\) Yes](#) [!\[\]\(e03937753152a2b0363a27eadcda0bc8_img.jpg\) No](#)

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

GetQueueStatus function (winuser.h)

Article03/17/2023

Retrieves the type of messages found in the calling thread's message queue.

Syntax

C++

```
DWORD GetQueueStatus(  
    [in] UINT flags  
);
```

Parameters

[in] flags

Type: **UINT**

The types of messages for which to check. This parameter can be one or more of the following values.

Value	Meaning
QS_KEY 0x0001	A WM_KEYUP , WM_KEYDOWN , WM_SYSKEYUP , or WM_SYSKEYDOWN message is in the queue.
QS_MOUSEMOVE 0x0002	A WM_MOUSEMOVE message is in the queue.
QS_MOUSEBUTTON 0x0004	A mouse-button message (WM_LBUTTONDOWN , WM_RBUTTONDOWN , and so on).
QS_POSTMESSAGE 0x0008	A posted message (other than those listed here) is in the queue. For more information, see PostMessage . This value is cleared when you call GetMessage or PeekMessage , whether or not you are filtering messages.
QS_TIMER 0x0010	A WM_TIMER message is in the queue.
QS_PAINT 0x0020	A WM_PAINT message is in the queue.

Value	Meaning
QS_SENDSMESSAGE 0x0040	A message sent by another thread or application is in the queue. For more information, see SendMessage .
QS_HOTKEY 0x0080	A WM_HOTKEY message is in the queue.
QS_ALLPOSTMESSAGE 0x0100	A posted message (other than those listed here) is in the queue. For more information, see PostMessage . This value is cleared when you call GetMessage or PeekMessage without filtering messages.
QS_RAWINPUT 0x0400	Windows XP and newer: A raw input message is in the queue. For more information, see Raw Input .
QS_TOUCH 0x0800	Windows 8 and newer: A touch input message is in the queue. For more information, see Touch Input .
QS_POINTER 0x1000	Windows 8 and newer: A pointer input message is in the queue. For more information, see Pointer Input .
QS_MOUSE (QS_MOUSEMOVE QS_MOUSEBUTTON)	A WM_MOUSEMOVE message or mouse-button message (WM_LBUTTONDOWN , WM_RBUTTONDOWN , and so on).
QS_INPUT (QS_MOUSE QS_KEY QS_RAWINPUT QS_TOUCH QS_POINTER)	An input message is in the queue.
QS_ALLEVENTS (QS_INPUT QS_POSTMESSAGE QS_TIMER QS_PAINT QS_HOTKEY)	An input, WM_TIMER , WM_PAINT , WM_HOTKEY , or posted message is in the queue.
QS_ALLINPUT (QS_INPUT QS_POSTMESSAGE QS_TIMER QS_PAINT QS_HOTKEY QS_SENDSMESSAGE)	Any message is in the queue.

Return value

Type: **DWORD**

The high-order word of the return value indicates the types of messages currently in the queue. The low-order word indicates the types of messages that have been added to the queue and that are still in the queue since the last call to the [GetQueueStatus](#), [GetMessage](#), or [PeekMessage](#) function.

Remarks

The presence of a QS_ flag in the return value does not guarantee that a subsequent call to the [GetMessage](#) or [PeekMessage](#) function will return a message. [GetMessage](#) and [PeekMessage](#) perform some internal filtering that may cause the message to be processed internally. For this reason, the return value from [GetQueueStatus](#) should be considered only a hint as to whether [GetMessage](#) or [PeekMessage](#) should be called.

The QS_ALLPOSTMESSAGE and QS_POSTMESSAGE flags differ in when they are cleared. QS_POSTMESSAGE is cleared when you call [GetMessage](#) or [PeekMessage](#), whether or not you are filtering messages. QS_ALLPOSTMESSAGE is cleared when you call [GetMessage](#) or [PeekMessage](#) without filtering messages (*wMsgFilterMin* and *wMsgFilterMax* are 0). This can be useful when you call [PeekMessage](#) multiple times to get messages in different ranges.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetInputState](#)

[GetMessage](#)

[Messages and Message Queues](#)

[PeekMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetShellWindow function (winuser.h)

Article 06/29/2021

Retrieves a handle to the Shell's desktop window.

Syntax

C++

```
HWND GetShellWindow();
```

Return value

Type: HWND

The return value is the handle of the Shell's desktop window. If no Shell process is present, the return value is **NULL**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetDesktopWindow](#)

[GetWindow](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetSysColor function (winuser.h)

Article11/10/2021

Retrieves the current color of the specified display element. Display elements are the parts of a window and the display that appear on the system display screen.

Syntax

C++

```
DWORD GetSysColor(  
    [in] int nIndex  
);
```

Parameters

[in] nIndex

Type: int

The display element whose color is to be retrieved. This parameter can be one of the following values.

Value	Meaning
COLOR_3DDKSHADOW 21	Dark shadow for three-dimensional display elements. Windows 10 or greater: This value is not supported.
COLOR_3DFACE 15	Face color for three-dimensional display elements and for dialog box backgrounds.
COLOR_3DHIGHLIGHT 20	Highlight color for three-dimensional display elements (for edges facing the light source.) Windows 10 or greater: This value is not supported.
COLOR_3DHILIGHT 20	Highlight color for three-dimensional display elements (for edges facing the light source.) Windows 10 or greater: This value is not supported.
COLOR_3DLIGHT 22	Light color for three-dimensional display elements (for edges facing the light source.) Windows 10 or greater: This value is not supported.
COLOR_3DSHADOW 16	Shadow color for three-dimensional display elements (for edges facing away from the light source).

		Windows 10 or greater: This value is not supported.
COLOR_ACTIVEBORDER 10	Active window border. Windows 10 or greater: This value is not supported.	
COLOR_ACTIVECAPTION 2	Active window title bar. The associated foreground color is COLOR_CAPTIONTEXT . Specifies the left side color in the color gradient of an active window's title bar if the gradient effect is enabled. Windows 10 or greater: This value is not supported.	
COLOR_APPWORKSPACE 12	Background color of multiple document interface (MDI) applications. Windows 10 or greater: This value is not supported.	
COLOR_BACKGROUND 1	Desktop. Windows 10 or greater: This value is not supported.	
COLOR_BTNFACE 15	Face color for three-dimensional display elements and for dialog box backgrounds. The associated foreground color is COLOR_BTNTTEXT . Windows 10 or greater: This value is not supported.	
COLOR_BTNHIGHLIGHT 20	Highlight color for three-dimensional display elements (for edges facing the light source.) Windows 10 or greater: This value is not supported.	
COLOR_BTNHILIGHT 20	Highlight color for three-dimensional display elements (for edges facing the light source.) Windows 10 or greater: This value is not supported.	
COLOR_BTNSHADOW 16	Shadow color for three-dimensional display elements (for edges facing away from the light source). Windows 10 or greater: This value is not supported.	
COLOR_BTNTTEXT 18	Text on push buttons. The associated background color is COLOR_BTNFACE .	
COLOR_CAPTIONTEXT 9	Text in caption, size box, and scroll bar arrow box. The associated background color is COLOR_ACTIVECAPTION . Windows 10 or greater: This value is not supported.	
COLOR_DESKTOP 1	Desktop. Windows 10 or greater: This value is not supported.	
COLOR_GRADIENTACTIVECAPTION 27	Right side color in the color gradient of an active window's title bar. COLOR_ACTIVECAPTION specifies the left side color. Use SPI_GETGRADIENTCAPTIONS with the	

		<p>SystemParametersInfo function to determine whether the gradient effect is enabled.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_GRADIENTINACTIVECAPTION 28		<p>Right side color in the color gradient of an inactive window's title bar. COLOR_INACTIVECAPTION specifies the left side color.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_GRAYTEXT 17		<p>Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.</p>
COLOR_HIGHLIGHT 13		<p>Item(s) selected in a control. The associated foreground color is COLOR_HIGHLIGHTTEXT.</p>
COLOR_HIGHLIGHTTEXT 14		<p>Text of item(s) selected in a control. The associated background color is COLOR_HIGHLIGHT.</p>
COLOR_HOTLIGHT 26		<p>Color for a hyperlink or hot-tracked item. The associated background color is COLOR_WINDOW.</p>
COLOR_INACTIVEBORDER 11		<p>Inactive window border.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_INACTIVECAPTION 3		<p>Inactive window caption. The associated foreground color is COLOR_INACTIVECAPTIONTEXT.</p>
		<p>Specifies the left side color in the color gradient of an inactive window's title bar if the gradient effect is enabled.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_INACTIVECAPTIONTEXT 19		<p>Color of text in an inactive caption. The associated background color is COLOR_INACTIVECAPTION.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_INFOBK 24		<p>Background color for tooltip controls. The associated foreground color is COLOR_INFOTEXT.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_INFOTEXT 23		<p>Text color for tooltip controls. The associated background color is COLOR_INFOBK.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_MENU 4		<p>Menu background. The associated foreground color is COLOR_MENUTEXT.</p>
		<p>Windows 10 or greater: This value is not supported.</p>
COLOR_MENUHILIGHT		<p>The color used to highlight menu items when the menu</p>

29	appears as a flat menu (see SystemParametersInfo). The highlighted menu item is outlined with COLOR_HIGHLIGHT.
	Windows 2000, Windows 10 or greater: This value is not supported.
COLOR_MENUBAR 30	The background color for the menu bar when menus appear as flat menus (see SystemParametersInfo). However, COLOR_MENU continues to specify the background color of the menu popup.
	Windows 2000, Windows 10 or greater: This value is not supported.
COLOR_MENUTEXT 7	Text in menus. The associated background color is COLOR_MENU. Windows 10 or greater: This value is not supported.
COLOR_SCROLLBAR 0	Scroll bar gray area. Windows 10 or greater: This value is not supported.
COLOR_WINDOW 5	Window background. The associated foreground colors are COLOR_WINDOWTEXT and COLOR_HOTLITE.
COLOR_WINDOWFRAME 6	Window frame. Windows 10 or greater: This value is not supported.
COLOR_WINDOWTEXT 8	Text in windows. The associated background color is COLOR_WINDOW.

Return value

Type: **DWORD**

The function returns the red, green, blue (RGB) color value of the given element.

If the *nIndex* parameter is out of range, the return value is zero. Because zero is also a valid RGB value, you cannot use **GetSysColor** to determine whether a system color is supported by the current platform. Instead, use the **GetSysColorBrush** function, which returns **NULL** if the color is not supported.

Remarks

To display the component of the RGB value, use the **GetRValue**, **GetGValue**, and **GetBValue** macros.

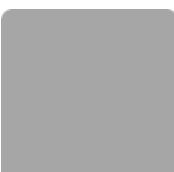
System colors for monochrome displays are usually interpreted as shades of gray.

To paint with a system color brush, an application should use `GetSysColorBrush(nIndex)`, instead of `CreateSolidBrush(GetSysColor(nIndex))`, because `GetSysColorBrush` returns a cached brush, instead of allocating a new one.

Color is an important visual element of most user interfaces. For guidelines about using color in your applications, see [Color - Win32](#) and [Color in Windows 11](#).

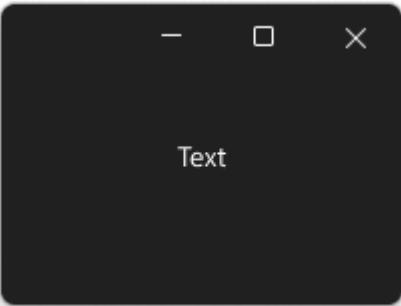
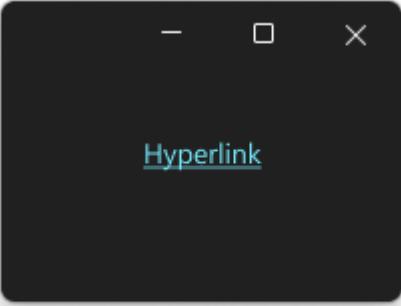
Windows 10/11 system colors

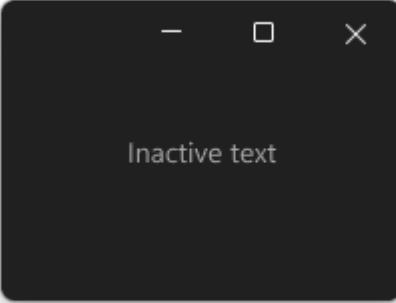
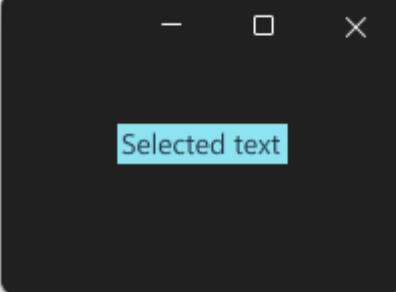
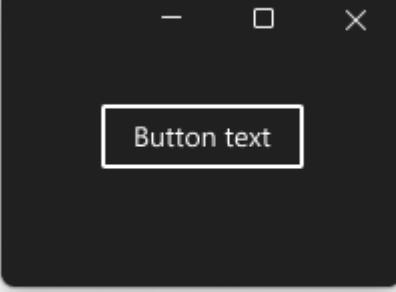
This table shows the values that are supported in Windows 10 and Windows 11 with color values from the Windows 11 *Aquatic* contrast theme.

Color swatch	Description
	COLOR_WINDOW Background of pages, panes, popups, and windows. Pair with COLOR_WINDOWTEXT
	COLOR_WINDOWTEXT Headings, body copy, lists, placeholder text, app and window borders, any UI that can't be interacted with. Pair with COLOR_WINDOW
	COLOR_HOTLIGHT Hyperlinks. Pair with COLOR_WINDOW
	COLOR_GRAYTEXT Inactive (disabled) UI. Pair with COLOR_WINDOW
	COLOR_HIGHLIGHTTEXT Foreground color for text or UI that is in selected, interacted with (hover, pressed), or in progress. Pair with COLOR_HIGHLIGHT

Color swatch	Description
	COLOR_HIGHLIGHT Background or accent color for UI that is selected, interacted with (hover, pressed), or in progress. Pair with COLOR_HIGHLIGHTTEXT
	COLOR_BTNTEXT Foreground color for buttons and any UI that can be interacted with. Pair with COLOR_3DFACE
	COLOR_3DFACE Background color for buttons and any UI that can be interacted with. Pair with COLOR_BTNTEXT

These images show how the colors appear when used on a background set to COLOR_WINDOW.

Example	Values
	COLOR_WINDOWTEXT
	COLOR_HOTLIGHT

Example	Values
 <p>Inactive text</p>	COLOR_GRAYTEXT
 <p>Selected text</p>	COLOR_HIGHLIGHTTEXT + HIGHLIGHT
 <p>Button text</p>	COLOR_BTNTEXT + COLOR_3DFACE

Examples

For an example, see [SetSysColors](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll

See also

[CreateSolidBrush](#)

[GetSysColorBrush](#)

[SetSysColors](#)

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetSystemMetrics function (winuser.h)

Article03/11/2023

Retrieves the specified system metric or system configuration setting.

Note that all dimensions retrieved by **GetSystemMetrics** are in pixels.

Syntax

C++

```
int GetSystemMetrics(  
    [in] int nIndex  
) ;
```

Parameters

[in] nIndex

Type: **int**

The system metric or configuration setting to be retrieved. This parameter can be one of the following values. Note that all SM_CX* values are widths and all SM_CY* values are heights. Also note that all settings designed to return Boolean data represent **TRUE** as any nonzero value, and **FALSE** as a zero value.

Value	Meaning
SM_ARRANGE 56	The flags that specify how the system arranged minimized windows. For more information, see the Remarks section in this topic.
SM_CLEANBOOT 67	<p>The value that specifies how the system is started:</p> <ul style="list-style-type: none">• 0 Normal boot• 1 Fail-safe boot• 2 Fail-safe with network boot <p>A fail-safe boot (also called SafeBoot, Safe Mode, or Clean Boot) bypasses the user startup files.</p>
SM_CMONITORS 80	The number of display monitors on a desktop. For more information, see the Remarks section in this topic.

SM_CMOUSEBUTTONS 43	The number of buttons on a mouse, or zero if no mouse is installed.
SM_CONVERTIBLESLATEMODE 0x2003	Reflects the state of the laptop or slate mode, 0 for Slate Mode and non-zero otherwise. When this system metric changes, the system sends a broadcast message via WM_SETTINGCHANGE with "ConvertibleSlateMode" in the LPARAM. Note that this system metric doesn't apply to desktop PCs. In that case, use GetAutoRotationState .
SM_CXBORDER 5	The width of a window border, in pixels. This is equivalent to the SM_CXEDGE value for windows with the 3-D look.
SM_CXCURSOR 13	The nominal width of a cursor, in pixels.
SM_CXDLGFRAME 7	This value is the same as SM_CXFIXEDFRAME.
SM_CXDOUBLECLK 36	<p>The width of the rectangle around the location of a first click in a double-click sequence, in pixels. The second click must occur within the rectangle that is defined by SM_CXDOUBLECLK and SM_CYDOUBLECLK for the system to consider the two clicks a double-click. The two clicks must also occur within a specified time.</p> <p>To set the width of the double-click rectangle, call SystemParametersInfo with SPI_SETDOUBLECLKWIDTH.</p>
SM_CXDRAG 68	The number of pixels on either side of a mouse-down point that the mouse pointer can move before a drag operation begins. This allows the user to click and release the mouse button easily without unintentionally starting a drag operation. If this value is negative, it is subtracted from the left of the mouse-down point and added to the right of it.
SM_CXEDGE 45	The width of a 3-D border, in pixels. This metric is the 3-D counterpart of SM_CXBORDER.
SM_CXFIXEDFRAME 7	<p>The thickness of the frame around the perimeter of a window that has a caption but is not sizable, in pixels. SM_CXFIXEDFRAME is the height of the horizontal border, and SM_CYFIXEDFRAME is the width of the vertical border.</p> <p>This value is the same as SM_CXDLGFRAME.</p>
SM_CXFOCUSBORDER	The width of the left and right edges of the focus

83	rectangle that the DrawFocusRect draws. This value is in pixels.
SM_CXFRAME 32	This value is the same as SM_CXSIZEFRAME.
SM_CXFULLSCREEN 16	The width of the client area for a full-screen window on the primary display monitor, in pixels. To get the coordinates of the portion of the screen that is not obscured by the system taskbar or by application desktop toolbars, call the SystemParametersInfo function with the SPI_GETWORKAREA value.
SM_CXHSCROLL 21	The width of the arrow bitmap on a horizontal scroll bar, in pixels.
SM_CXHTHUMB 10	The width of the thumb box in a horizontal scroll bar, in pixels.
SM_CXICON 11	The system large width of an icon, in pixels. The LoadIcon function can load only icons with the dimensions that SM_CXICON and SM_CYICON specifies. See Icon Sizes for more info.
SM_CXICONSPACING 38	The width of a grid cell for items in large icon view, in pixels. Each item fits into a rectangle of size SM_CXICONSPACING by SM_CYICONSPACING when arranged. This value is always greater than or equal to SM_CXICON.
SM_CXMAXIMIZED 61	The default width, in pixels, of a maximized top-level window on the primary display monitor.
SM_CXMAXTRACK 59	The default maximum width of a window that has a caption and sizing borders, in pixels. This metric refers to the entire desktop. The user cannot drag the window frame to a size larger than these dimensions. A window can override this value by processing the WM_GETMINMAXINFO message.
SM_CXMENUCHECK 71	The width of the default menu check-mark bitmap, in pixels.
SM_CXMENUSIZE 54	The width of menu bar buttons, such as the child window close button that is used in the multiple document interface, in pixels.
SM_CXMIN 28	The minimum width of a window, in pixels.

SM_CXMINIMIZED	57	The width of a minimized window, in pixels.
SM_CXMINSPECING	47	The width of a grid cell for a minimized window, in pixels. Each minimized window fits into a rectangle this size when arranged. This value is always greater than or equal to SM_CXMINIMIZED.
SM_CXMINTRACK	34	The minimum tracking width of a window, in pixels. The user cannot drag the window frame to a size smaller than these dimensions. A window can override this value by processing the WM_GETMINMAXINFO message.
SM_CXPADDED BORDER	92	The amount of border padding for captioned windows, in pixels. Windows XP/2000: This value is not supported.
SM_CXSCREEN	0	The width of the screen of the primary display monitor, in pixels. This is the same value obtained by calling GetDeviceCaps as follows: <code>GetDeviceCaps(hdcPrimaryMonitor, HORZRES)</code> .
SM_CXSIZE	30	The width of a button in a window caption or title bar, in pixels.
SM_CXSIZEFRAME	32	The thickness of the sizing border around the perimeter of a window that can be resized, in pixels. SM_CXSIZEFRAME is the width of the horizontal border, and SM_CYSIZEFRAME is the height of the vertical border. This value is the same as SM_CXFRAME.
SM_CXSMICON	49	The system small width of an icon, in pixels. Small icons typically appear in window captions and in small icon view. See Icon Sizes for more info.
SM_CXSMSIZE	52	The width of small caption buttons, in pixels.
SM_CXVIRTUALSCREEN	78	The width of the virtual screen, in pixels. The virtual screen is the bounding rectangle of all display monitors. The SM_CXVIRTUALSCREEN metric is the coordinates for the left side of the virtual screen.
SM_CXVSCROLL	2	The width of a vertical scroll bar, in pixels.
SM_CYBORDER	6	The height of a window border, in pixels. This is equivalent to the SM_CYEDGE value for windows with the 3-D look.

SM_CYCAPTION	The height of a caption area, in pixels.
4	
SM_CYCURSOR	The nominal height of a cursor, in pixels.
14	
SM_CYDLGFRAME	This value is the same as SM_CYFIXEDFRAME.
8	
SM_CYDOUBLECLK	<p>The height of the rectangle around the location of a first click in a double-click sequence, in pixels. The second click must occur within the rectangle defined by SM_CXDOUBLECLK and SM_CYDOUBLECLK for the system to consider the two clicks a double-click. The two clicks must also occur within a specified time.</p> <p>To set the height of the double-click rectangle, call SystemParametersInfo with SPI_SETDOUBLECLKHEIGHT.</p>
37	
SM_CYDRAG	<p>The number of pixels above and below a mouse-down point that the mouse pointer can move before a drag operation begins. This allows the user to click and release the mouse button easily without unintentionally starting a drag operation. If this value is negative, it is subtracted from above the mouse-down point and added below it.</p>
69	
SM_CYEDGE	The height of a 3-D border, in pixels. This is the 3-D counterpart of SM_CYBORDER.
46	
SM_CYFIXEDFRAME	<p>The thickness of the frame around the perimeter of a window that has a caption but is not sizable, in pixels. SM_CXFIXEDFRAME is the height of the horizontal border, and SM_CYFIXEDFRAME is the width of the vertical border.</p> <p>This value is the same as SM_CYDLGFRAME.</p>
8	
SM_CYFOCUSBORDER	<p>The height of the top and bottom edges of the focus rectangle drawn by DrawFocusRect. This value is in pixels.</p> <p>Windows 2000: This value is not supported.</p>
84	
SM_CYFRAME	This value is the same as SM_CYSIZEFRAME.
33	
SM_CYFULLSCREEN	The height of the client area for a full-screen window on the primary display monitor, in pixels. To get the coordinates of the portion of the screen not obscured by the system taskbar or by application
17	

		desktop toolbars, call the SystemParametersInfo function with the SPI_GETWORKAREA value.
SM_CYHSCROLL		The height of a horizontal scroll bar, in pixels.
3		
SM_CYICON		The system large height of an icon, in pixels. The LoadIcon function can load only icons with the dimensions that SM_CXICON and SM_CYICON specifies. See Icon Sizes for more info.
12		
SM_CYICONSPACING		The height of a grid cell for items in large icon view, in pixels. Each item fits into a rectangle of size SM_CXICONSPACING by SM_CYICONSPACING when arranged. This value is always greater than or equal to SM_CYICON.
39		
SM_CYKANJIWINDOW		For double byte character set versions of the system, this is the height of the Kanji window at the bottom of the screen, in pixels.
18		
SM_CYMAXIMIZED		The default height, in pixels, of a maximized top-level window on the primary display monitor.
62		
SM_CYMAXTRACK		The default maximum height of a window that has a caption and sizing borders, in pixels. This metric refers to the entire desktop. The user cannot drag the window frame to a size larger than these dimensions. A window can override this value by processing the WM_GETMINMAXINFO message.
60		
SM_CYMENU		The height of a single-line menu bar, in pixels.
15		
SM_CYMENUCHECK		The height of the default menu check-mark bitmap, in pixels.
72		
SM_CYMENUSIZE		The height of menu bar buttons, such as the child window close button that is used in the multiple document interface, in pixels.
55		
SM_CYMIN		The minimum height of a window, in pixels.
29		
SM_CYMINIMIZED		The height of a minimized window, in pixels.
58		
SM_CYMINSPACING		The height of a grid cell for a minimized window, in pixels. Each minimized window fits into a rectangle this size when arranged. This value is always greater than or equal to SM_CYMINIMIZED.
48		

SM_CYMINTRACK		The minimum tracking height of a window, in pixels. The user cannot drag the window frame to a size smaller than these dimensions. A window can override this value by processing the WM_GETMINMAXINFO message.
SM_CYSCREEN	1	The height of the screen of the primary display monitor, in pixels. This is the same value obtained by calling GetDeviceCaps as follows: <code>GetDeviceCaps(hdcPrimaryMonitor, VERTRES)</code> .
SM_CYSIZE	31	The height of a button in a window caption or title bar, in pixels.
SM_CYSIZEFRAME	33	The thickness of the sizing border around the perimeter of a window that can be resized, in pixels. SM_CXSIZEFRAME is the width of the horizontal border, and SM_CYSIZEFRAME is the height of the vertical border. This value is the same as SM_CYFRAME .
SM_CYSMCAPTION	51	The height of a small caption, in pixels.
SM_CYSMICON	50	The system small height of an icon, in pixels. Small icons typically appear in window captions and in small icon view. See Icon Sizes for more info.
SM_CYSMSIZE	53	The height of small caption buttons, in pixels.
SM_CYVIRTUALSCREEN	79	The height of the virtual screen, in pixels. The virtual screen is the bounding rectangle of all display monitors. The SM_YVIRTUALSCREEN metric is the coordinates for the top of the virtual screen.
SM_CYVSCROLL	20	The height of the arrow bitmap on a vertical scroll bar, in pixels.
SM_CYVTHUMB	9	The height of the thumb box in a vertical scroll bar, in pixels.
SM_DBCSENABLED	42	Nonzero if User32.dll supports DBCS; otherwise, 0.
SM_DEBUG	22	Nonzero if the debug version of User.exe is installed; otherwise, 0.
SM_DIGITIZER	94	Nonzero if the current operating system is Windows 7 or Windows Server 2008 R2 and the Tablet PC Input service is started; otherwise, 0. The return value is a bitmask that specifies the type of

		digitizer input supported by the device. For more information, see Remarks.
		Windows Server 2008, Windows Vista and Windows XP/2000: This value is not supported.
SM_IMMENABLED 82		Nonzero if Input Method Manager/Input Method Editor features are enabled; otherwise, 0. SM_IMMENABLED indicates whether the system is ready to use a Unicode-based IME on a Unicode application. To ensure that a language-dependent IME works, check SM_DBCSENABLED and the system ANSI code page. Otherwise the ANSI-to-Unicode conversion may not be performed correctly, or some components like fonts or registry settings may not be present.
SM_MAXIMUMTOUCHES 95		Nonzero if there are digitizers in the system; otherwise, 0. SM_MAXIMUMTOUCHES returns the aggregate maximum of the maximum number of contacts supported by every digitizer in the system. If the system has only single-touch digitizers, the return value is 1. If the system has multi-touch digitizers, the return value is the number of simultaneous contacts the hardware can provide.
		Windows Server 2008, Windows Vista and Windows XP/2000: This value is not supported.
SM_MEDIACENTER 87		Nonzero if the current operating system is the Windows XP, Media Center Edition, 0 if not.
SM_MENUDROPALIGNMENT 40		Nonzero if drop-down menus are right-aligned with the corresponding menu-bar item; 0 if the menus are left-aligned.
SM_MIDEASTENABLED 74		Nonzero if the system is enabled for Hebrew and Arabic languages, 0 if not.
SM_MOUSEPRESENT 19		Nonzero if a mouse is installed; otherwise, 0. This value is rarely zero, because of support for virtual mice and because some systems detect the presence of the port instead of the presence of a mouse.
SM_MOUSEHORIZONTALWHEELPRESENT 91		Nonzero if a mouse with a horizontal scroll wheel is installed; otherwise 0.
SM_MOUSEWHEELPRESENT 75		Nonzero if a mouse with a vertical scroll wheel is installed; otherwise 0.
SM_NETWORK		The least significant bit is set if a network is present;

63	otherwise, it is cleared. The other bits are reserved for future use.
SM_PENWINDOWS 41	Nonzero if the Microsoft Windows for Pen computing extensions are installed; zero otherwise.
SM_REMOTECONTROL 0x2001	<p>This system metric is used in a Terminal Services environment to determine if the current Terminal Server session is being remotely controlled. Its value is nonzero if the current session is remotely controlled; otherwise, 0.</p> <p>You can use terminal services management tools such as Terminal Services Manager (tsadmin.msc) and shadow.exe to control a remote session. When a session is being remotely controlled, another user can view the contents of that session and potentially interact with it.</p>
SM_REMOTESESSION 0x1000	<p>This system metric is used in a Terminal Services environment. If the calling process is associated with a Terminal Services client session, the return value is nonzero. If the calling process is associated with the Terminal Services console session, the return value is 0.</p> <p>Windows Server 2003 and Windows XP: The console session is not necessarily the physical console. For more information, see WTSGetActiveConsoleSessionId.</p>
SM_SAMEDISPLAYFORMAT 81	Nonzero if all the display monitors have the same color format, otherwise, 0. Two displays can have the same bit depth, but different color formats. For example, the red, green, and blue pixels can be encoded with different numbers of bits, or those bits can be located in different places in a pixel color value.
SM_SECURE 44	This system metric should be ignored; it always returns 0.
SM_SERVERR2 89	The build number if the system is Windows Server 2003 R2; otherwise, 0.
SM_SHOWSOUNDS 70	Nonzero if the user requires an application to present information visually in situations where it would otherwise present the information only in audible form; otherwise, 0.
SM_SHUTTINGDOWN 0x2000	<p>Nonzero if the current session is shutting down; otherwise, 0.</p> <p>Windows 2000: This value is not supported.</p>

SM_SLOWMACHINE	Nonzero if the computer has a low-end (slow) processor; otherwise, 0.
SM_STARTER	Nonzero if the current operating system is Windows 7 Starter Edition, Windows Vista Starter, or Windows XP Starter Edition; otherwise, 0.
SM_SWAPBUTTON	Nonzero if the meanings of the left and right mouse buttons are swapped; otherwise, 0.
SM_SYSTEMDOCKED 0x2004	Reflects the state of the docking mode, 0 for Undocked Mode and non-zero otherwise. When this system metric changes, the system sends a broadcast message via WM_SETTINGCHANGE with "SystemDockMode" in the LPARAM.
SM_TABLETPC 86	Nonzero if the current operating system is the Windows XP Tablet PC edition or if the current operating system is Windows Vista or Windows 7 and the Tablet PC Input service is started; otherwise, 0. The SM_DIGITIZER setting indicates the type of digitizer input supported by a device running Windows 7 or Windows Server 2008 R2. For more information, see Remarks.
SM_XVIRTUALSCREEN 76	The coordinates for the left side of the virtual screen. The virtual screen is the bounding rectangle of all display monitors. The SM_CXVIRTUALSCREEN metric is the width of the virtual screen.
SM_YVIRTUALSCREEN 77	The coordinates for the top of the virtual screen. The virtual screen is the bounding rectangle of all display monitors. The SM_CYVIRTUALSCREEN metric is the height of the virtual screen.

Return value

Type: int

If the function succeeds, the return value is the requested system metric or configuration setting.

If the function fails, the return value is 0. [GetLastError](#) does not provide extended error information.

Remarks

System metrics can vary from display to display.

GetSystemMetrics(SM_CMONITORS) counts only visible display monitors. This is different from [EnumDisplayMonitors](#), which enumerates both visible display monitors and invisible pseudo-monitors that are associated with mirroring drivers. An invisible pseudo-monitor is associated with a pseudo-device used to mirror application drawing for remoting or other purposes.

The SM_ARRANGE setting specifies how the system arranges minimized windows, and consists of a starting position and a direction. The starting position can be one of the following values.

Value	Meaning
ARW_BOTTOMLEFT	Start at the lower-left corner of the screen. The default position.
ARW_BOTTOMRIGHT	Start at the lower-right corner of the screen. Equivalent to ARW_STARTRIGHT.
ARW_TOPLEFT	Start at the upper-left corner of the screen. Equivalent to ARW_STARTTOP.
ARW_TOPRIGHT	Start at the upper-right corner of the screen. Equivalent to ARW_STARTTOP SRW_STARTRIGHT.

The direction in which to arrange minimized windows can be one of the following values.

Value	Meaning
ARW_DOWN	Arrange vertically, top to bottom.
ARW_HIDE	Hide minimized windows by moving them off the visible area of the screen.
ARW_LEFT	Arrange horizontally, left to right.
ARW_RIGHT	Arrange horizontally, right to left.
ARW_UP	Arrange vertically, bottom to top.

The SM_DIGITIZER setting specifies the type of digitizers that are installed on a device running Windows 7 or Windows Server 2008 R2. The return value is a bitmask that specifies one or more of the following values.

Value	Meaning

NID_INTEGRATED_TOUCH 0x01	The device has an integrated touch digitizer.
NID_EXTERNAL_TOUCH 0x02	The device has an external touch digitizer.
NID_INTEGRATED_PEN 0x04	The device has an integrated pen digitizer.
NID_EXTERNAL_PEN 0x08	The device has an external pen digitizer.
NID_MULTI_INPUT 0x40	The device supports multiple sources of digitizer input.
NID_READY 0x80	The device is ready to receive digitizer input.

This API is not DPI aware, and should not be used if the calling thread is per-monitor DPI aware. For the DPI-aware version of this API, see [GetSystemMetricsForDPI](#). For more information on DPI awareness, see [the Windows High DPI documentation](#).

Examples

The following example uses the [GetSystemMetrics](#) function to determine whether a mouse is installed and whether the mouse buttons are swapped. The example also uses the [SystemParametersInfo](#) function to retrieve the mouse threshold and speed. It displays the information in the console.

syntax

```
#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "user32.lib")

void main()
{
    BOOL fResult;
    int aMouseInfo[3];

    fResult = GetSystemMetrics(SM_MOUSEPRESENT);

    if (fResult == 0)
        printf("No mouse installed.\n");
    else
    {
        printf("Mouse installed.\n");
    }
}
```

```

// Determine whether the buttons are swapped.

fResult = GetSystemMetrics(SM_SWAPBUTTON);

if (fResult == 0)
    printf("Buttons not swapped.\n");
else printf("Buttons swapped.\n");

// Get the mouse speed and the threshold values.

fResult = SystemParametersInfo(
    SPI_GETMOUSE, // get mouse information
    0,           // not used
    &aMouseInfo, // holds mouse information
    0);          // not used

if( fResult )
{
    printf("Speed: %d\n", aMouseInfo[2]);
    printf("Threshold (x,y): %d,%d\n",
        aMouseInfo[0], aMouseInfo[1]);
}
}

}

```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-sysparams-ext-l1-1-0 (introduced in Windows 8)

See also

[EnumDisplayMonitors](#)

[GetSystemMetricsForDPI](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetTitleBarInfo function (winuser.h)

Article 10/13/2021

Retrieves information about the specified title bar.

Syntax

C++

```
BOOL GetTitleBarInfo(
    [in]      HWND      hwnd,
    [in, out] PTITLEBARINFO pti
);
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the title bar whose information is to be retrieved.

[in, out] `pti`

Type: **PTITLEBARINFO**

A pointer to a **TITLEBARINFO** structure to receive the information. Note that you must set the **cbSize** member to `sizeof(TITLEBARINFO)` before calling this function.

Return value

Type: **BOOL**

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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[Reference](#)

[TITLEBARINFO](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetTopWindow function (winuser.h)

Article10/13/2021

Examines the Z order of the child windows associated with the specified parent window and retrieves a handle to the child window at the top of the Z order.

Syntax

C++

```
HWND GetTopWindow(  
    [in, optional] HWND hWnd  
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the parent window whose child windows are to be examined. If this parameter is **NULL**, the function returns a handle to the window at the top of the Z order.

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the child window at the top of the Z order. If the specified window has no child windows, the return value is **NULL**. To get extended error information, use the [GetLastError](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
--------------------------	---

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetNextWindow](#)

[GetWindow](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindow function (winuser.h)

Article 10/13/2021

Retrieves a handle to a window that has the specified relationship (Z-Order or owner) to the specified window.

Syntax

C++

```
HWND GetWindow(  
    [in] HWND hWnd,  
    [in] UINT uCmd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to a window. The window handle retrieved is relative to this window, based on the value of the *uCmd* parameter.

[in] uCmd

Type: **UINT**

The relationship between the specified window and the window whose handle is to be retrieved. This parameter can be one of the following values.

Value	Meaning
GW_CHILD 5	The retrieved handle identifies the child window at the top of the Z order, if the specified window is a parent window; otherwise, the retrieved handle is NULL . The function examines only child windows of the specified window. It does not examine descendant windows.
GW_ENABLEDPOPUP 6	The retrieved handle identifies the enabled popup window owned by the specified window (the search uses the first such window found using GW_HWNDNEXT); otherwise, if there are no enabled popup windows, the retrieved handle is that of the specified window.

GW_HWNDFIRST	The retrieved handle identifies the window of the same type that is highest in the Z order.
0	If the specified window is a topmost window, the handle identifies a topmost window. If the specified window is a top-level window, the handle identifies a top-level window. If the specified window is a child window, the handle identifies a sibling window.
GW_HWNDLAST	The retrieved handle identifies the window of the same type that is lowest in the Z order.
1	If the specified window is a topmost window, the handle identifies a topmost window. If the specified window is a top-level window, the handle identifies a top-level window. If the specified window is a child window, the handle identifies a sibling window.
GW_HWNDNEXT	The retrieved handle identifies the window below the specified window in the Z order.
2	If the specified window is a topmost window, the handle identifies a topmost window. If the specified window is a top-level window, the handle identifies a top-level window. If the specified window is a child window, the handle identifies a sibling window.
GW_HWNDPREV	The retrieved handle identifies the window above the specified window in the Z order.
3	If the specified window is a topmost window, the handle identifies a topmost window. If the specified window is a top-level window, the handle identifies a top-level window. If the specified window is a child window, the handle identifies a sibling window.
GW_OWNER	The retrieved handle identifies the specified window's owner window, if any. For more information, see Owned Windows .
4	

Return value

Type: **HWND**

If the function succeeds, the return value is a window handle. If no window exists with the specified relationship to the specified window, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The [EnumChildWindows](#) function is more reliable than calling [GetWindow](#) in a loop. An application that calls [GetWindow](#) to perform this task risks being caught in an infinite loop or referencing a handle to a window that has been destroyed.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[EnumChildWindows](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowDisplayAffinity function (winuser.h)

Article 10/13/2021

Retrieves the current display affinity setting, from any process, for a given window.

Syntax

C++

```
BOOL GetWindowDisplayAffinity(  
    [in]  HWND  hWnd,  
    [out] DWORD *pdwAffinity  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[out] pdwAffinity

Type: **DWORD***

A pointer to a variable that receives the display affinity setting. See [SetWindowDisplayAffinity](#) for a list of affinity settings and their meanings.

Return value

Type: **BOOL**

This function succeeds only when the window is layered and Desktop Windows Manager is composing the desktop. If this function succeeds, it returns **TRUE**; otherwise, it returns **FALSE**. To get extended error information, call [GetLastError](#).

Remarks

This function and [SetWindowDisplayAffinity](#) are designed to support the window content protection feature unique to Windows 7. This feature enables applications to protect their own onscreen window content from being captured or copied via a specific set of public operating system features and APIs. However, it works only when the Desktop Window Manager (DWM) is composing the desktop.

It is important to note that unlike a security feature or an implementation of Digital Rights Management (DRM), there is no guarantee that using [SetWindowDisplayAffinity](#) and [GetWindowDisplayAffinity](#), and other necessary functions such as [DwmIsCompositionEnabled](#), will strictly protect windowed content, as in the case where someone takes a photograph of the screen.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowInfo function (winuser.h)

Article07/27/2022

Retrieves information about the specified window.

Syntax

C++

```
BOOL GetWindowInfo(
    [in]      HWND      hwnd,
    [in, out] PWINDOWINFO pwi
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the window whose information is to be retrieved.

[in, out] pwi

Type: **PWINDOWINFO**

A pointer to a **WINDOWINFO** structure to receive the information. Note that you must set the **cbSize** member to `sizeof(WINDOWINFO)` before calling this function.

Return value

Type: **BOOL**

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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Reference](#)

[WINDOWINFO](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowLongA function (winuser.h)

Article02/09/2023

Retrieves information about the specified window. The function also retrieves the 32-bit (DWORD) value at the specified offset into the extra window memory.

Note If you are retrieving a pointer or a handle, this function has been superseded by the [GetWindowLongPtr](#) function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.) To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [GetWindowLongPtr](#).

Syntax

C++

```
LONG GetWindowLongA(
    [in] HWND hWnd,
    [in] int nIndex
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: **int**

The zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus four; for example, if you specified 12 or more bytes of extra memory, a value of 8 would be an index to the third 32-bit integer. To retrieve any other value, specify one of the following values.

Value	Meaning

GWL_EXSTYLE	Retrieves the extended window styles .
-20	
GWL_HINSTANCE	Retrieves a handle to the application instance.
-6	
GWL_HWNDPARENT	Retrieves a handle to the parent window, if any.
-8	
GWL_ID	Retrieves the identifier of the window.
-12	
GWL_STYLE	Retrieves the window styles .
-16	
GWL_USERDATA	Retrieves the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
-21	
GWL_WNDPROC	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.
-4	

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWL_DLGPROC	Retrieves the address of the dialog box procedure, or a handle representing the address of the dialog box procedure. You must use the CallWindowProc function to call the dialog box procedure.
DWLP_MSGRESULT + sizeof(LRESULT)	
0	Retrieves the return value of a message processed in the dialog box procedure.
DWL_USER	Retrieves extra information private to the application, such as handles or pointers.
DWLP_DLGPROC + sizeof(DLGPROC)	

Return value

Type: **LONG**

If the function succeeds, the return value is the requested value.

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

If [SetWindowLong](#) has not been called previously, [GetWindowLong](#) returns zero for values in the extra window or class memory.

Remarks

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Examples

For an example, see [Creating, Enumerating, and Sizing Child Windows](#).

Note

The winuser.h header defines [GetWindowLong](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[GetWindowLongPtr](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

[SetWindowLong](#)

[WNDCLASS](#)

[Window Classes](#)

Feedback

Was this page helpful?



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

GetWindowLongPtrA function (winuser.h)

Article02/09/2023

Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory.

Note To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **GetWindowLongPtr**. When compiling for 32-bit Windows, **GetWindowLongPtr** is defined as a call to the **GetWindowLong** function.

Syntax

C++

```
LONG_PTR GetWindowLongPtrA(  
    [in] HWND hWnd,  
    [in] int nIndex  
)
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: **int**

The zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of a **LONG_PTR**. To retrieve any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE	Retrieves the extended window styles .

GWLP_HINSTANCE	Retrieves a handle to the application instance.
-6	
GWLP_HWNDPARENT	Retrieves a handle to the parent window, if there is one.
-8	
GWLP_ID	Retrieves the identifier of the window.
-12	
GWL_STYLE	Retrieves the window styles .
-16	
GWLP_USERDATA	Retrieves the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
-21	
GWLP_WNDPROC	Retrieves the pointer to the window procedure, or a handle representing the pointer to the window procedure. You must use the CallWindowProc function to call the window procedure.
-4	

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWLP_DLGPROC DWLP_MSGRESULT + sizeof(LRESULT)	Retrieves the pointer to the dialog box procedure, or a handle representing the pointer to the dialog box procedure. You must use the CallWindowProc function to call the dialog box procedure.
DWLP_MSGRESULT 0	Retrieves the return value of a message processed in the dialog box procedure.
DWLP_USER DWLP_DLGPROC + sizeof(DLGPROC)	Retrieves extra information private to the application, such as handles or pointers.

Return value

Type: **LONG_PTR**

If the function succeeds, the return value is the requested value.

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

If [SetWindowLong](#) or [SetWindowLongPtr](#) has not been called previously, [GetWindowLongPtr](#) returns zero for values in the extra window or class memory.

Remarks

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Note

The winuser.h header defines [GetWindowLongPtr](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

[SetWindowLongPtr](#)

[WNDCLASSEX](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowLongPtrW function (winuser.h)

Article02/09/2023

Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory.

Note To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **GetWindowLongPtr**. When compiling for 32-bit Windows, **GetWindowLongPtr** is defined as a call to the **GetWindowLong** function.

Syntax

C++

```
LONG_PTR GetWindowLongPtrW(  
    [in] HWND hWnd,  
    [in] int nIndex  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: **int**

The zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of a **LONG_PTR**. To retrieve any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE	Retrieves the extended window styles .

GWLP_HINSTANCE	Retrieves a handle to the application instance.
-6	
GWLP_HWNDPARENT	Retrieves a handle to the parent window, if there is one.
-8	
GWLP_ID	Retrieves the identifier of the window.
-12	
GWL_STYLE	Retrieves the window styles .
-16	
GWLP_USERDATA	Retrieves the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
-21	
GWLP_WNDPROC	Retrieves the pointer to the window procedure, or a handle representing the pointer to the window procedure. You must use the CallWindowProc function to call the window procedure.
-4	

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWLP_DLGPROC DWLP_MSGRESULT + sizeof(LRESULT)	Retrieves the pointer to the dialog box procedure, or a handle representing the pointer to the dialog box procedure. You must use the CallWindowProc function to call the dialog box procedure.
DWLP_MSGRESULT 0	Retrieves the return value of a message processed in the dialog box procedure.
DWLP_USER DWLP_DLGPROC + sizeof(DLGPROC)	Retrieves extra information private to the application, such as handles or pointers.

Return value

Type: **LONG_PTR**

If the function succeeds, the return value is the requested value.

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

If [SetWindowLong](#) or [SetWindowLongPtr](#) has not been called previously, [GetWindowLongPtr](#) returns zero for values in the extra window or class memory.

Remarks

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Note

The winuser.h header defines [GetWindowLongPtr](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

[SetWindowLongPtr](#)

[WNDCLASSEX](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowLongW function (winuser.h)

Article02/09/2023

Retrieves information about the specified window. The function also retrieves the 32-bit (DWORD) value at the specified offset into the extra window memory.

Note If you are retrieving a pointer or a handle, this function has been superseded by the [GetWindowLongPtr](#) function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.) To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [GetWindowLongPtr](#).

Syntax

C++

```
LONG GetWindowLongW(
    [in] HWND hWnd,
    [in] int nIndex
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: **int**

The zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus four; for example, if you specified 12 or more bytes of extra memory, a value of 8 would be an index to the third 32-bit integer. To retrieve any other value, specify one of the following values.

Value	Meaning

GWL_EXSTYLE	Retrieves the extended window styles .
-20	
GWL_HINSTANCE	Retrieves a handle to the application instance.
-6	
GWL_HWNDPARENT	Retrieves a handle to the parent window, if any.
-8	
GWL_ID	Retrieves the identifier of the window.
-12	
GWL_STYLE	Retrieves the window styles .
-16	
GWL_USERDATA	Retrieves the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
-21	
GWL_WNDPROC	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.
-4	

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWL_DLGPROC	Retrieves the address of the dialog box procedure, or a handle representing the address of the dialog box procedure. You must use the CallWindowProc function to call the dialog box procedure.
DWLP_MSGRESULT + sizeof(LRESULT)	
0	Retrieves the return value of a message processed in the dialog box procedure.
DWL_USER	Retrieves extra information private to the application, such as handles or pointers.
DWLP_DLGPROC + sizeof(DLGPROC)	

Return value

Type: **LONG**

If the function succeeds, the return value is the requested value.

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

If [SetWindowLong](#) has not been called previously, [GetWindowLong](#) returns zero for values in the extra window or class memory.

Remarks

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Examples

For an example, see [Creating, Enumerating, and Sizing Child Windows](#).

Note

The winuser.h header defines [GetWindowLong](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[GetWindowLongPtr](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

[SetWindowLong](#)

[WNDCLASS](#)

[Window Classes](#)

Feedback

Was this page helpful?



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

GetWindowModuleFileNameA function (winuser.h)

Article 02/09/2023

Retrieves the full path and file name of the module associated with the specified window handle.

Syntax

C++

```
UINT GetWindowModuleFileNameA(
    [in]  HWND  hwnd,
    [out] LPSTR pszFileName,
    [in]  UINT   cchFileNameMax
);
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the window whose module file name is to be retrieved.

[out] `pszFileName`

Type: **LPTSTR**

The path and file name.

[in] `cchFileNameMax`

Type: **UINT**

The maximum number of characters that can be copied into the *lpszFileName* buffer.

Return value

Type: **UINT**

The return value is the total number of characters copied into the buffer.

Remarks

ⓘ Note

The winuser.h header defines GetWindowModuleFileName 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowModuleFileNameW function (winuser.h)

Article 02/09/2023

Retrieves the full path and file name of the module associated with the specified window handle.

Syntax

C++

```
UINT GetWindowModuleFileNameW(
    [in]  HWND    hwnd,
    [out] LPWSTR pszFileName,
    [in]  UINT    cchFileNameMax
);
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the window whose module file name is to be retrieved.

[out] `pszFileName`

Type: **LPTSTR**

The path and file name.

[in] `cchFileNameMax`

Type: **UINT**

The maximum number of characters that can be copied into the *lpszFileName* buffer.

Return value

Type: **UINT**

The return value is the total number of characters copied into the buffer.

Remarks

ⓘ Note

The winuser.h header defines GetWindowModuleFileName 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowPlacement function (winuser.h)

Article 10/13/2021

Retrieves the show state and the restored, minimized, and maximized positions of the specified window.

Syntax

C++

```
BOOL GetWindowPlacement(
    [in]      HWND      hWnd,
    [in, out] WINDOWPLACEMENT *lpwndpl
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[in, out] lpwndpl

Type: **WINDOWPLACEMENT***

A pointer to the **WINDOWPLACEMENT** structure that receives the show state and position information. Before calling **GetWindowPlacement**, set the **length** member to **sizeof(WINDOWPLACEMENT)**. **GetWindowPlacement** fails if *lpwndpl->length* is not set correctly.

Return value

Type: **BOOL**

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 **flags** member of [WINDOWPLACEMENT](#) retrieved by this function is always zero. If the window identified by the *hWnd* parameter is maximized, the **showCmd** member is **SW_SHOWMAXIMIZED**. If the window is minimized, **showCmd** is **SW_SHOWMINIMIZED**. Otherwise, it is **SW_SHOWNORMAL**.

The **length** member of [WINDOWPLACEMENT](#) must be set to `sizeof(WINDOWPLACEMENT)`. If this member is not set correctly, the function returns **FALSE**. For additional remarks on the proper use of window placement coordinates, see [WINDOWPLACEMENT](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Reference](#)

[SetWindowPlacement](#)

[WINDOWPLACEMENT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetWindowRect function (winuser.h)

Article 10/13/2021

Retrieves the dimensions of the bounding rectangle of the specified window. The dimensions are given in screen coordinates that are relative to the upper-left corner of the screen.

Syntax

C++

```
BOOL GetWindowRect(  
    [in]  HWND   hWnd,  
    [out] LPRECT lpRect  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[out] lpRect

Type: **LPRECT**

A pointer to a [RECT](#) structure that receives the screen coordinates of the upper-left and lower-right corners of the window.

Return value

Type: **BOOL**

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 conformance with conventions for the [RECT](#) structure, the bottom-right coordinates of the returned rectangle are exclusive. In other words, the pixel at (**right**, **bottom**) lies immediately outside the rectangle.

GetWindowRect is virtualized for DPI.

In Windows Vista and later, the Window Rect now includes the area occupied by the drop shadow.

Calling GetWindowRect will have different behavior depending on whether the window has ever been shown or not. If the window has not been shown before, GetWindowRect will not include the area of the drop shadow.

To get the window bounds excluding the drop shadow, use [DwmGetWindowAttribute](#), specifying **DWMWA_EXTENDED_FRAME_BOUNDS**. Note that unlike the Window Rect, the DWM Extended Frame Bounds are not adjusted for DPI. Getting the extended frame bounds can only be done after the window has been shown at least once.

Examples

For an example, see [Initializing a Dialog Box](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetClientRect](#)

Reference

[ScreenToClient](#)

[SetWindowPos](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowTextA function (winuser.h)

Article 02/09/2023

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, **GetWindowText** cannot retrieve the text of a control in another application.

Syntax

C++

```
int GetWindowTextA(
    [in] HWND hWnd,
    [out] LPSTR lpString,
    [in] int nMaxCount
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control containing the text.

[out] lpString

Type: **LPTSTR**

The buffer that will receive the text. If the string is as long or longer than the buffer, the string is truncated and terminated with a null character.

[in] nMaxCount

Type: **int**

The maximum number of characters to copy to the buffer, including the null character. If the text exceeds this limit, it is truncated.

Return value

Type: **int**

If the function succeeds, the return value is the length, in characters, of the copied string, not including the terminating null character. If the window has no title bar or text, if the title bar is empty, or if the window or control handle is invalid, the return value is zero. To get extended error information, call [GetLastError](#).

This function cannot retrieve the text of an edit control in another application.

Remarks

If the target window is owned by the current process, **GetWindowText** causes a [WM_GETTEXT](#) message to be sent to the specified window or control. If the target window is owned by another process and has a caption, **GetWindowText** retrieves the window caption text. If the window does not have a caption, the return value is a null string. This behavior is by design. It allows applications to call **GetWindowText** without becoming unresponsive if the process that owns the target window is not responding. However, if the target window is not responding and it belongs to the calling application, **GetWindowText** will cause the calling application to become unresponsive.

To retrieve the text of a control in another process, send a [WM_GETTEXT](#) message directly instead of calling **GetWindowText**.

Examples

The following example code demonstrates a call to **GetWindowTextA**.

C++

```
hwndCombo = GetDlgItem(hwndDlg, IDD_COMBO);
cTxtLen = GetWindowTextLength(hwndCombo);

// Allocate memory for the string and copy
// the string into the memory.

pszMem = (PSTR) VirtualAlloc((LPVOID) NULL,
    (DWORD) (cTxtLen + 1), MEM_COMMIT,
    PAGE_READWRITE);
GetWindowText(hwndCombo, pszMem,
    cTxtLen + 1);
```

To see this example in context, see [Sending a Message](#).

 **Note**

The winuser.h header defines GetWindowText 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[GetWindowTextLength](#)

[Reference](#)

[SetWindowText](#)

[WM_GETTEXT](#)

[Windows](#)

Feedback



Was this page helpful?  Yes  No

Get help at Microsoft Q&A

GetWindowTextLengthA function (winuser.h)

Article02/09/2023

Retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). If the specified window is a control, the function retrieves the length of the text within the control. However, **GetWindowTextLength** cannot retrieve the length of the text of an edit control in another application.

Syntax

C++

```
int GetWindowTextLengthA(  
    [in] HWND hWnd  
>;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control.

Return value

Type: **int**

If the function succeeds, the return value is the length, in characters, of the text. Under certain conditions, this value might be greater than the length of the text (see Remarks).

If the window has no text, the return value is zero.

Function failure is indicated by a return value of zero and a [GetLastError](#) result that is nonzero.

ⓘ Note

This function does not clear the most recent error information. To determine success or failure, clear the most recent error information by calling [SetLastError](#) with 0, then call [GetLastError](#).

Remarks

If the target window is owned by the current process, [GetWindowTextLength](#) causes a [WM_GETTEXTLENGTH](#) message to be sent to the specified window or control.

Under certain conditions, the [GetWindowTextLength](#) function may return a value that is larger than the actual length of the text. This occurs with certain mixtures of ANSI and Unicode, and is due to the system allowing for the possible existence of double-byte character set (DBCS) characters within the text. The return value, however, will always be at least as large as the actual length of the text; you can thus always use it to guide buffer allocation. This behavior can occur when an application uses both ANSI functions and common dialogs, which use Unicode. It can also occur when an application uses the ANSI version of [GetWindowTextLength](#) with a window whose window procedure is Unicode, or the Unicode version of [GetWindowTextLength](#) with a window whose window procedure is ANSI. For more information on ANSI and ANSI functions, see [Conventions for Function Prototypes](#).

To obtain the exact length of the text, use the [WM_GETTEXT](#), [LB_GETTEXT](#), or [CB_GETLBTEXT](#) messages, or the [GetWindowText](#) function.

Note

The winuser.h header defines [GetWindowTextLength](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[CB_GETLBTEXT](#)

[Conceptual](#)

[GetWindowText](#)

[LB_GETTEXT](#)

[Other Resources](#)

[Reference](#)

[SetWindowText](#)

[WM_GETTEXT](#)

[WM_GETTEXTLENGTH](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowTextLengthW function (winuser.h)

Article 02/09/2023

Retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). If the specified window is a control, the function retrieves the length of the text within the control. However, **GetWindowTextLength** cannot retrieve the length of the text of an edit control in another application.

Syntax

C++

```
int GetWindowTextLengthW(  
    [in] HWND hWnd  
>;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control.

Return value

Type: **int**

If the function succeeds, the return value is the length, in characters, of the text. Under certain conditions, this value might be greater than the length of the text (see Remarks).

If the window has no text, the return value is zero.

Function failure is indicated by a return value of zero and a [GetLastError](#) result that is nonzero.

ⓘ Note

This function does not clear the most recent error information. To determine success or failure, clear the most recent error information by calling [SetLastError](#) with 0, then call [GetLastError](#).

Remarks

If the target window is owned by the current process, [GetWindowTextLength](#) causes a [WM_GETTEXTLENGTH](#) message to be sent to the specified window or control.

Under certain conditions, the [GetWindowTextLength](#) function may return a value that is larger than the actual length of the text. This occurs with certain mixtures of ANSI and Unicode, and is due to the system allowing for the possible existence of double-byte character set (DBCS) characters within the text. The return value, however, will always be at least as large as the actual length of the text; you can thus always use it to guide buffer allocation. This behavior can occur when an application uses both ANSI functions and common dialogs, which use Unicode. It can also occur when an application uses the ANSI version of [GetWindowTextLength](#) with a window whose window procedure is Unicode, or the Unicode version of [GetWindowTextLength](#) with a window whose window procedure is ANSI. For more information on ANSI and ANSI functions, see [Conventions for Function Prototypes](#).

To obtain the exact length of the text, use the [WM_GETTEXT](#), [LB_GETTEXT](#), or [CB_GETLBTEXT](#) messages, or the [GetWindowText](#) function.

Note

The winuser.h header defines [GetWindowTextLength](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[CB_GETLBTEXT](#)

[Conceptual](#)

[GetWindowText](#)

[LB_GETTEXT](#)

[Other Resources](#)

[Reference](#)

[SetWindowText](#)

[WM_GETTEXT](#)

[WM_GETTEXTLENGTH](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowTextW function (winuser.h)

Article 02/09/2023

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, **GetWindowText** cannot retrieve the text of a control in another application.

Syntax

C++

```
int GetWindowTextW(
    [in] HWND hWnd,
    [out] LPWSTR lpString,
    [in] int nMaxCount
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control containing the text.

[out] lpString

Type: **LPTSTR**

The buffer that will receive the text. If the string is as long or longer than the buffer, the string is truncated and terminated with a null character.

[in] nMaxCount

Type: **int**

The maximum number of characters to copy to the buffer, including the null character. If the text exceeds this limit, it is truncated.

Return value

Type: **int**

If the function succeeds, the return value is the length, in characters, of the copied string, not including the terminating null character. If the window has no title bar or text, if the title bar is empty, or if the window or control handle is invalid, the return value is zero. To get extended error information, call [GetLastError](#).

This function cannot retrieve the text of an edit control in another application.

Remarks

If the target window is owned by the current process, **GetWindowText** causes a [WM_GETTEXT](#) message to be sent to the specified window or control. If the target window is owned by another process and has a caption, **GetWindowText** retrieves the window caption text. If the window does not have a caption, the return value is a null string. This behavior is by design. It allows applications to call **GetWindowText** without becoming unresponsive if the process that owns the target window is not responding. However, if the target window is not responding and it belongs to the calling application, **GetWindowText** will cause the calling application to become unresponsive.

To retrieve the text of a control in another process, send a [WM_GETTEXT](#) message directly instead of calling **GetWindowText**.

Examples

For an example, see [Sending a Message](#).

ⓘ Note

The winuser.h header defines **GetWindowText** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetWindowTextLength](#)

Reference

[SetWindowText](#)

[WM_GETTEXT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowThreadProcessId function (winuser.h)

Article 03/17/2023

Retrieves the identifier of the thread that created the specified window and, optionally, the identifier of the process that created the window.

Syntax

C++

```
DWORD GetWindowThreadProcessId(
    [in]             HWND     hWnd,
    [out, optional] LPDWORD lpdwProcessId
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[out, optional] lpdwProcessId

Type: **LPDWORD**

A pointer to a variable that receives the process identifier. If this parameter is not **NULL**, **GetWindowThreadProcessId** copies the identifier of the process to the variable; otherwise, it does not. If the function fails, the value of the variable is unchanged.

Return value

Type: **DWORD**

If the function succeeds, the return value is the identifier of the thread that created the window. If the window handle is invalid, 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

GetWindowWord function (winuser.h)

Article03/21/2023

Retrieves the 16-bit (DWORD) value at the specified offset into the extra window memory.

Syntax

C++

```
WORD GetWindowWord(
    HWND hWnd,
    int nIndex
);
```

Parameters

hWnd

A handle to the window and, indirectly, the class to which the window belongs.

nIndex

The zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus four; for example, if you specified 12 or more bytes of extra memory, a value of 8 would be an index to the third 32-bit integer. To retrieve any other value, specify one of the following values.

Constant	Value	Meaning
GWW_HINSTANCE	-6	Retrieves a handle to the application instance.
GWW_HWNDPARENT	-8	Retrieves a handle to the parent window, if any.
GWW_ID	-12	Retrieves the identifier of the window.

Return value

If the function succeeds, the return value is the requested value.

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

Remarks

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Requirements

Header	winuser.h
--------	-----------

Feedback

Was this page helpful?



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

GUITHREADINFO structure (winuser.h)

Article 04/02/2021

Contains information about a GUI thread.

Syntax

C++

```
typedef struct tagGUITHREADINFO {
    DWORD cbSize;
    DWORD flags;
    HWND hwndActive;
    HWND hwndFocus;
    HWND hwndCapture;
    HWND hwndMenuOwner;
    HWND hwndMoveSize;
    HWND hwndCaret;
    RECT rcCaret;
} GUITHREADINFO, *PGUITHREADINFO, *LPGUITHREADINFO;
```

Members

`cbSize`

Type: **DWORD**

The size of this structure, in bytes. The caller must set this member to `sizeof(GUITHREADINFO)`.

`flags`

Type: **DWORD**

The thread state. This member can be one or more of the following values.

Value	Meaning
<code>GUI_CARETBLINKING</code> 0x00000001	The caret's blink state. This bit is set if the caret is visible.
<code>GUI_INMENUMODE</code> 0x00000004	The thread's menu state. This bit is set if the thread is in menu mode.
<code>GUI_INMOVESIZE</code>	The thread's move state. This bit is set if the thread is in a

0x00000002	move or size loop.
GUI_POPUPMENUMODE 0x00000010	The thread's pop-up menu state. This bit is set if the thread has an active pop-up menu.
GUI_SYSTEMMENUMODE 0x00000008	The thread's system menu state. This bit is set if the thread is in a system menu mode.

`hwndActive`

Type: **HWND**

A handle to the active window within the thread.

`hwndFocus`

Type: **HWND**

A handle to the window that has the keyboard focus.

`hwndCapture`

Type: **HWND**

A handle to the window that has captured the mouse.

`hwndMenuOwner`

Type: **HWND**

A handle to the window that owns any active menus.

`hwndMoveSize`

Type: **HWND**

A handle to the window in a move or size loop.

`hwndCaret`

Type: **HWND**

A handle to the window that is displaying the caret.

`rcCaret`

Type: **RECT**

The caret's bounding rectangle, in client coordinates, relative to the window specified by the **hwndCaret** member.

Remarks

This structure is used with the [GetGUIThreadInfo](#) function to retrieve information about the active window or a specified GUI thread.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)
Redistributable	Service Pack 3

See also

Conceptual

[GetGUIThreadInfo](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

HOOKPROC callback function (winuser.h)

Article 06/30/2023

An application-defined or library-defined callback function used with the [SetWindowsHookEx](#) function. The system calls this function after the [SendMessage](#) function is called. The hook procedure can examine the message; it cannot modify it.

The **HOOKPROC** type defines a pointer to this callback function. *CallWndRetProc* is a placeholder for the application-defined or library-defined function name.

Syntax

C++

```
HOOKPROC Hookproc;

HRESULT Hookproc(
    int code,
    [in] WPARAM wParam,
    [in] LPARAM lParam
)
{...}
```

Parameters

code

[in] wParam

Type: **WPARAM**

Specifies whether the message is sent by the current process. If the message is sent by the current process, it is nonzero; otherwise, it is **NULL**.

[in] lParam

Type: **LPARAM**

A pointer to a [CWPRESTRUCT](#) structure that contains details about the message.

Return value

Type: LRESULT

If *nCode* is less than zero, the hook procedure must return the value returned by [CallNextHookEx function](#).

If *nCode* is greater than or equal to zero, it is highly recommended that you call [CallNextHookEx function](#) and return the value it returns; otherwise, other applications that have installed [WH_CALLWNDPROCRET](#) hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure does not call [CallNextHookEx](#), the return value should be zero.

Remarks

An application installs the hook procedure by specifying the [WH_CALLWNDPROCRET](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookEx](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

[CWPRETSTRUCT structure](#), [CallNextHookEx function](#), [CallWindowProcW function](#), [CallWindowProcA function](#), [SendMessage](#), [SetWindowsHookEx](#), [Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

InSendMessage function (winuser.h)

Article 06/29/2021

Determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process) by a call to the [SendMessage](#) function.

To obtain additional information about how the message was sent, use the [InSendMessageEx](#) function.

Syntax

C++

```
BOOL InSendMessage();
```

Return value

Type: **BOOL**

If the window procedure is processing a message sent to it from another thread using the [SendMessage](#) function, the return value is nonzero.

If the window procedure is not processing a message sent to it from another thread using the [SendMessage](#) function, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set

ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[InSendMessageEx](#)

[Messages and Message Queues](#)

Reference

[SendMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

InSendMessageEx function (winuser.h)

Article 04/02/2021

Determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process).

Syntax

C++

```
DWORD InSendMessageEx(
    LPVOID lpReserved
);
```

Parameters

lpReserved

Type: **LPVOID**

Reserved; must be **NULL**.

Return value

Type: **DWORD**

If the message was not sent, the return value is **ISMEX_NOSEND** (0x00000000).

Otherwise, the return value is one or more of the following values.

Return code/value	Description
ISMEX_CALLBACK 0x00000004	The message was sent using the SendMessageCallback function. The thread that sent the message is not blocked.
ISMEX_NOTIFY 0x00000002	The message was sent using the SendNotifyMessage function. The thread that sent the message is not blocked.
ISMEX_REPLIED 0x00000008	The window procedure has processed the message. The thread that sent the message is no longer blocked.
ISMEX_SEND	The message was sent using the SendMessage or

0x00000001

[SendMessageTimeout](#) function. If **ISMEX_REPLYED** is not set, the thread that sent the message is blocked.

Remarks

To determine if the sender is blocked, use the following test:

```
fBlocked = ( InSendMessageEx(NULL) & (ISMEX_REPLYED|ISMEX_SEND) ) == ISMEX_SEND;
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Messages and Message Queues](#)

[Reference](#)

[SendMessage](#)

[SendMessageCallback](#)

[SendMessageTimeout](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

InternalGetWindowText function (winuser.h)

Article 10/13/2021

[This function is not intended for general use. It may be altered or unavailable in subsequent versions of Windows.]

Copies the text of the specified window's title bar (if it has one) into a buffer.

This function is similar to the [GetWindowText](#) function. However, it obtains the window text directly from the window structure associated with the specified window's handle and then always provides the text as a Unicode string. This is unlike [GetWindowText](#) which obtains the text by sending the window a [WM_GETTEXT](#) message. If the specified window is a control, the text of the control is obtained.

Syntax

C++

```
int InternalGetWindowText(  
    [in] HWND hWnd,  
    [out] LPWSTR pString,  
    [in] int cchMaxCount  
>;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control containing the text.

[out] pString

Type: **LPWSTR**

The buffer that is to receive the text.

If the string is as long or longer than the buffer, the string is truncated and terminated with a null character.

[in] cchMaxCount

Type: int

The maximum number of characters to be copied to the buffer, including the null character. If the text exceeds this limit, it is truncated.

Return value

Type: int

If the function succeeds, the return value is the length, in characters, of the copied string, not including the terminating null character.

If the window has no title bar or text, if the title bar is empty, or if the window or control handle is invalid, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

This function was not included in the SDK headers and libraries until Windows XP with Service Pack 1 (SP1) and Windows Server 2003. If you do not have a header file and import library for this function, you can call the function using [LoadLibrary](#) and [GetProcAddress](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[GetWindowText](#)

[GetWindowTextLength](#)

[Reference](#)

[SetWindowText](#)

[Using Messages and Message Queues](#)

[WM_GETTEXT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsChild function (winuser.h)

Article10/13/2021

Determines whether a window is a child window or descendant window of a specified parent window. A child window is the direct descendant of a specified parent window if that parent window is in the chain of parent windows; the chain of parent windows leads from the original overlapped or pop-up window to the child window.

Syntax

C++

```
BOOL IsChild(
    [in] HWND hWndParent,
    [in] HWND hWnd
);
```

Parameters

[in] hWndParent

Type: **HWND**

A handle to the parent window.

[in] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the window is a child or descendant window of the specified parent window, the return value is nonzero.

If the window is not a child or descendant window of the specified parent window, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[IsWindow](#)

[Reference](#)

[SetParent](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsGUIThread function (winuser.h)

Article 10/13/2021

Determines whether the calling thread is already a GUI thread. It can also optionally convert the thread to a GUI thread.

Syntax

C++

```
BOOL IsGUIThread(  
    [in] BOOL bConvert  
);
```

Parameters

[in] bConvert

Type: **BOOL**

If **TRUE** and the thread is not a GUI thread, convert the thread to a GUI thread.

Return value

Type: **BOOL**

The function returns a nonzero value in the following situations:

- If the calling thread is already a GUI thread.
- If *bConvert* is **TRUE** and the function successfully converts the thread to a GUI thread.

Otherwise, the function returns zero.

If *bConvert* is **TRUE** and the function cannot successfully convert the thread to a GUI thread, **IsGUIThread** returns **ERROR_NOT_ENOUGH_MEMORY**.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsHungAppWindow function (winuser.h)

Article10/13/2021

[This function is not intended for general use. It may be altered or unavailable in subsequent versions of Windows.]

Determines whether the system considers that a specified application is not responding. An application is considered to be not responding if it is not waiting for input, is not in startup processing, and has not called [PeekMessage](#) within the internal timeout period of 5 seconds.

Syntax

C++

```
BOOL IsHungAppWindow(  
    [in] HWND hwnd  
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

The return value is **TRUE** if the window stops responding; otherwise, it is **FALSE**. Ghost windows always return **TRUE**.

Remarks

The Windows timeout criteria of 5 seconds is subject to change.

This function was not included in the SDK headers and libraries until Windows XP Service Pack 1 (SP1) and Windows Server 2003. If you do not have a header

file and import library for this function, you can call the function using [LoadLibrary](#) and [GetProcAddress](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[IsWindow](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsIconic function (winuser.h)

Article 10/13/2021

Determines whether the specified window is minimized (iconic).

Syntax

C++

```
BOOL IsIconic(  
    [in] HWND hWnd  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the window is iconic, the return value is nonzero.

If the window is not iconic, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib

DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[IsZoomed](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsProcessDPIAware function (winuser.h)

Article 06/29/2021

[`IsProcessDPIAware` is available for use in the operating systems specified in the Requirements section. It may be altered or unavailable in subsequent versions. Instead, use [GetProcessDPIAwareness](#).]

Determines whether the current process is dots per inch (dpi) aware such that it adjusts the sizes of UI elements to compensate for the dpi setting.

Syntax

C++

```
BOOL IsProcessDPIAware();
```

Return value

Type: **BOOL**

TRUE if the process is dpi aware; otherwise, FALSE.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

Feedback



Was this page helpful?  

Get help at Microsoft Q&A

IsWindow function (winuser.h)

Article 10/13/2021

Determines whether the specified window handle identifies an existing window.

Syntax

C++

```
BOOL IsWindow(  
    [in, optional] HWND hWnd  
) ;
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the window handle identifies an existing window, the return value is nonzero.

If the window handle does not identify an existing window, the return value is zero.

Remarks

A thread should not use **IsWindow** for a window that it did not create because the window could be destroyed after this function was called. Further, because window handles are recycled the handle could even point to a different window.

Examples

For an example, see [Creating a Modeless Dialog Box](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[IsWindowEnabled](#)

[IsWindowVisible](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsWindowArranged function (winuser.h)

Article06/27/2023

ⓘ Important

Some information relates to a prerelease product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

Determines whether a window is arranged.

Syntax

C++

```
BOOL IsWindowArranged(  
    HWND hwnd  
);
```

Parameters

hwnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

A nonzero value if the window is arranged; otherwise, zero.

Remarks

ⓘ Tip

At this time, this function does not have an associated header file or library file. Your application can call **LoadLibrary** with the DLL name (`User32.dll`) to obtain a module handle. It can then call **GetProcAddress** with the module handle and the name of this function to get the function address.

A snapped window (see [Snap your windows](#)) is considered to be *arranged*. You should treat *arranged* as a window state similar to *maximized*. Arranged, maximized, and minimized are mutually exclusive states. An arranged window can be restored to its original size and position. Restoring a window from minimized can make a window arranged if the window was arranged before it was minimized. When calling [GetWindowPlacement](#), keep in mind that the `showCmd` member on the returned [WINDOWPLACEMENT](#) can have a value of `SW_SHOWNORMAL` even if the window is arranged.

Example

C++

```
// Check whether the window is in the restored state.  
BOOL IsRestored(HWND hwnd)  
{  
    if (IsIconic(hwnd) || IsZoomed(hwnd) || IsWindowArranged(hwnd))  
    {  
        return false;  
    }  
    return true;  
}
```

Requirements

Minimum supported client	Windows 10, version 1903
Header	winuser.h
Library	User32.lib
DLL	User32.dll

See also

- [IsIconic](#)
 - [IsZoomed](#)
 - [Windows](#)
 - [Snap your windows ↗](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

IsWindowUnicode function (winuser.h)

Article 10/13/2021

Determines whether the specified window is a native Unicode window.

Syntax

C++

```
BOOL IsWindowUnicode(  
    [in] HWND hWnd  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the window is a native Unicode window, the return value is nonzero.

If the window is not a native Unicode window, the return value is zero. The window is a native ANSI window.

Remarks

The character set of a window is determined by the use of the [RegisterClass](#) function. If the window class was registered with the ANSI version of [RegisterClass](#) ([RegisterClassA](#)), the character set of the window is ANSI. If the window class was registered with the Unicode version of [RegisterClass](#) ([RegisterClassW](#)), the character set of the window is Unicode.

The system does automatic two-way translation (Unicode to ANSI) for window messages. For example, if an ANSI window message is sent to a window that uses the

Unicode character set, the system translates that message into a Unicode message before calling the window procedure. The system calls **IsWindowUnicode** to determine whether to translate the message.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsWindowVisible function (winuser.h)

Article 10/13/2021

Determines the visibility state of the specified window.

Syntax

C++

```
BOOL IsWindowVisible(  
    [in] HWND hWnd  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the specified window, its parent window, its parent's parent window, and so forth, have the **WS_VISIBLE** style, the return value is nonzero. Otherwise, the return value is zero.

Because the return value specifies whether the window has the **WS_VISIBLE** style, it may be nonzero even if the window is totally obscured by other windows.

Remarks

The visibility state of a window is indicated by the **WS_VISIBLE** style bit. When **WS_VISIBLE** is set, the window is displayed and subsequent drawing into it is displayed as long as the window has the **WS_VISIBLE** style.

Any drawing to a window with the **WS_VISIBLE** style will not be displayed if the window is obscured by other windows or is clipped by its parent window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[Conceptual](#)

[Reference](#)

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

IsZoomed function (winuser.h)

Article10/13/2021

Determines whether a window is maximized.

Syntax

C++

```
BOOL IsZoomed(  
    [in] HWND hWnd  
>;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the window is zoomed, the return value is nonzero.

If the window is not zoomed, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[Iconic](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

KBDLLHOOKSTRUCT structure (winuser.h)

Article 04/02/2021

Contains information about a low-level keyboard input event.

Syntax

C++

```
typedef struct tagKBDLLHOOKSTRUCT {
    DWORD      vkCode;
    DWORD      scanCode;
    DWORD      flags;
    DWORD      time;
    ULONG_PTR  dwExtraInfo;
} KBDLLHOOKSTRUCT, *LPKBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT;
```

Members

`vkCode`

Type: **DWORD**

A [virtual-key code](#). The code must be a value in the range 1 to 254.

`scanCode`

Type: **DWORD**

A hardware scan code for the key.

`flags`

Type: **DWORD**

The extended-key flag, event-injected flags, context code, and transition-state flag. This member is specified as follows. An application can use the following values to test the keystroke flags. Testing LLKHF_INJECTED (bit 4) will tell you whether the event was injected. If it was, then testing LLKHF_LOWER_IL_INJECTED (bit 1) will tell you whether or not the event was injected from a process running at lower integrity level.

Value	Meaning
LLKHF_EXTENDED (KF_EXTENDED >> 8)	Test the extended-key flag.
LLKHF_LOWER_IL_INJECTED 0x00000002	Test the event-injected (from a process running at lower integrity level) flag.
LLKHF_INJECTED 0x00000010	Test the event-injected (from any process) flag.
LLKHF_ALTDOWN (KF_ALTDOWN >> 8)	Test the context code.
LLKHF_UP (KF_UP >> 8)	Test the transition-state flag.

The following table describes the layout of this value.

Bits	Description
0	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if the key is an extended key; otherwise, it is 0.
1	Specifies whether the event was injected from a process running at lower integrity level. The value is 1 if that is the case; otherwise, it is 0. Note that bit 4 is also set whenever bit 1 is set.
2-3	Reserved.
4	Specifies whether the event was injected. The value is 1 if that is the case; otherwise, it is 0. Note that bit 1 is not necessarily set when bit 4 is set.
5	The context code. The value is 1 if the ALT key is pressed; otherwise, it is 0.
6	Reserved.
7	The transition state. The value is 0 if the key is pressed and 1 if it is being released.

time

Type: **DWORD**

The time stamp for this message, equivalent to what [GetMessageTime](#) would return for this message.

dwExtraInfo

Type: **ULONG_PTR**

Additional information associated with the message.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

Conceptual

[Hooks](#)

[LowLevelKeyboardProc](#)

Reference

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

KillTimer function (winuser.h)

Article10/13/2021

Destroys the specified timer.

Syntax

C++

```
BOOL KillTimer(
    [in, optional] HWND     hWnd,
    [in]          UINT_PTR uIDEvent
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window associated with the specified timer. This value must be the same as the *hWnd* value passed to the [SetTimer](#) function that created the timer.

[in] uIDEvent

Type: **UINT_PTR**

The timer to be destroyed. If the window handle passed to [SetTimer](#) is valid, this parameter must be the same as the *nIDEvent*

value passed to [SetTimer](#). If the application calls [SetTimer](#) with *hWnd* set to **NULL**, this parameter must be the timer identifier returned by [SetTimer](#).

Return value

Type: **BOOL**

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 `KillTimer` function does not remove `WM_TIMER` messages already posted to the message queue.

Examples

For an example, see [Destroying a Timer](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[Reference](#)

[SetTimer](#)

[Timers](#)

[WM_TIMER](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

LockSetForegroundWindow function (winuser.h)

Article 10/13/2021

The foreground process can call the **LockSetForegroundWindow** function to disable calls to the [SetForegroundWindow](#) function.

Syntax

C++

```
BOOL LockSetForegroundWindow(  
    [in] UINT uLockCode  
);
```

Parameters

[in] **uLockCode**

Type: **UINT**

Specifies whether to enable or disable calls to [SetForegroundWindow](#). This parameter can be one of the following values.

Value	Meaning
LSFW_LOCK 1	Disables calls to SetForegroundWindow .
LSFW_UNLOCK 2	Enables calls to SetForegroundWindow .

Return value

Type: **BOOL**

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 system automatically enables calls to [SetForegroundWindow](#) if the user presses the ALT key or takes some action that causes the system itself to change the foreground window (for example, clicking a background window).

This function is provided so applications can prevent other applications from making a foreground change that can interrupt its interaction with the user.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[AllowSetForegroundWindow](#)

[Conceptual](#)

[Reference](#)

[SetForegroundWindow](#)

[Windows](#)

Feedback



Was this page helpful?  Yes  No

Get help at Microsoft Q&A

LogicalToPhysicalPoint function (winuser.h)

Article11/19/2022

Converts the logical coordinates of a point in a window to physical coordinates.

Syntax

C++

```
BOOL LogicalToPhysicalPoint(
    [in]      HWND      hWnd,
    [in, out] LPPOINT  lpPoint
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose transform is used for the conversion. Top level windows are fully supported. In the case of child windows, only the area of overlap between the parent and the child window is converted.

[in, out] lpPoint

Type: **LPPOINT**

A pointer to a **POINT** structure that specifies the logical coordinates to be converted. The new physical coordinates are copied into this structure if the function succeeds.

Return value

None

Remarks

Windows Vista introduces the concept of physical coordinates. Desktop Window Manager (DWM) scales non-dots per inch (dpi) aware windows when the display is high

dpi. The window seen on the screen corresponds to the physical coordinates. The application continues to work in logical space. Therefore, the application's view of the window is different from that which appears on the screen. For scaled windows, logical and physical coordinates are different.

LogicalToPhysicalPoint is a transformation API that can be called by a process that declares itself as dpi aware. The function uses the window identified by the *hWnd* parameter and the logical coordinates given in the **POINT** structure to compute the physical coordinates.

The **LogicalToPhysicalPoint** function replaces the logical coordinates in the **POINT** structure with the physical coordinates. The physical coordinates are relative to the upper-left corner of the screen. The coordinates have to be inside the client area of *hWnd*.

On all platforms, **LogicalToPhysicalPoint** will fail on a window that has either 0 width or height; an application must first establish a non-0 width and height by calling, for example, [MoveWindow](#). On some versions of Windows (including Windows 7), **LogicalToPhysicalPoint** will still fail if [MoveWindow](#) has been called after a call to [ShowWindow](#) with **SH_HIDE** has hidden the window.

In Windows 8, system-DPI aware applications translate between physical and logical space using **PhysicalToLogicalPoint** and **LogicalToPhysicalPoint**. In Windows 8.1, the additional virtualization of the system and inter-process communications means that for the majority of applications, you do not need these APIs. As a result, in Windows 8.1, **PhysicalToLogicalPoint** and **LogicalToPhysicalPoint** no longer transform points. The system returns all points to an application in its own coordinate space. This behavior preserves functionality for the majority of applications, but there are some exceptions in which you must make changes to ensure that the application works as expected. In those cases, use [PhysicalToLogicalPointForPerMonitorDPI](#) and [LogicalToPhysicalPointForPerMonitorDPI](#).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

Feedback

Was this page helpful?

 Yes

 No

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

MAKELPARAM macro (winuser.h)

Article04/02/2021

Creates a value for use as an *lParam* parameter in a message. The macro concatenates the specified values.

Syntax

C++

```
void MAKELPARAM(  
    l,  
    h  
) ;
```

Parameters

l

The low-order word of the new value.

h

The high-order word of the new value.

Return value

None

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

Conceptual

[MAKELONG](#)

[MAKELRESULT](#)

[MAKEWPARAM](#)

Reference

[Windows Data Types](#)

Feedback

Was this page helpful?

 Yes

 No

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

MAKELRESULT macro (winuser.h)

Article04/02/2021

Creates a value for use as a return value from a window procedure. The macro concatenates the specified values.

Syntax

C++

```
void MAKELRESULT(  
    l,  
    h  
) ;
```

Parameters

l

The low-order word of the new value.

h

The high-order word of the new value.

Return value

None

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

Conceptual

[MAKELONG](#)

[MAKELPARAM](#)

[MAKEWPARAM](#)

Reference

[Windows Data Types](#)

Feedback

Was this page helpful?

 Yes

 No

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

MAKEWPARAM macro (winuser.h)

Article04/02/2021

Creates a value for use as a *wParam* parameter in a message. The macro concatenates the specified values.

Syntax

C++

```
void MAKEWPARAM(  
    l,  
    h  
)
```

Parameters

l

The low-order word of the new value.

h

The high-order word of the new value.

Return value

None

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

Conceptual

[MAKELONG](#)

[MAKELPARAM](#)

[MAKELRESULT](#)

Reference

[Windows Data Types](#)

Feedback

Was this page helpful?

 Yes

 No

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

MDICREATESTRUCTA structure (winuser.h)

Article 07/27/2022

Contains information about the class, title, owner, location, and size of a multiple-document interface (MDI) child window.

Syntax

C++

```
typedef struct tagMDICREATESTRUCTA {
    LPCSTR szClass;
    LPCSTR szTitle;
    HANDLE hOwner;
    int     x;
    int     y;
    int     cx;
    int     cy;
    DWORD   style;
    LPARAM lParam;
} MDICREATESTRUCTA, *LPMDICREATESTRUCTA;
```

Members

`szClass`

Type: [LPCTSTR](#)

The name of the window class of the MDI child window. The class name must have been registered by a previous call to the [RegisterClass](#) function.

`szTitle`

Type: [LPCTSTR](#)

The title of the MDI child window. The system displays the title in the child window's title bar.

`hOwner`

Type: [HANDLE](#)

A handle to the instance of the application creating the MDI client window.

x

Type: int

The initial horizontal position, in client coordinates, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default horizontal position.

y

Type: int

The initial vertical position, in client coordinates, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default vertical position.

cx

Type: int

The initial width, in device units, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default width.

cy

Type: int

The initial height, in device units, of the MDI child window. If this member is set to **CW_USEDEFAULT**, the MDI child window is assigned the default height.

style

Type: DWORD

The style of the MDI child window. If the MDI client window was created with the **MDIS_ALLCHILDSTYLES** window style, this member can be any combination of the window styles listed in the [Window Styles](#) page. Otherwise, this member can be one or more of the following values.

Value	Meaning
WS_MINIMIZE 0x20000000L	Creates an MDI child window that is initially minimized.
WS_MAXIMIZE	Creates an MDI child window that is initially maximized.

0x01000000L	
WS_HSCROLL 0x00100000L	Creates an MDI child window that has a horizontal scroll bar.
WS_VSCROLL 0x00200000L	Creates an MDI child window that has a vertical scroll bar.

lParam

Type: **LPARAM**

An application-defined value.

Remarks

When the MDI client window creates an MDI child window by calling [CreateWindow](#), the system sends a [WM_CREATE](#) message to the created window. The *lParam* member of the [WM_CREATE](#) message contains a pointer to a [CREATESTRUCT](#) structure. The *lpCreateParams* member of this structure contains a pointer to the [MDICREATESTRUCT](#) structure passed with the [WM_MDICREATE](#) message that created the MDI child window.

 **Note**

The winuser.h header defines [MDICREATESTRUCT](#) 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	winuser.h (include Windows.h)

See also

[CLIENTCREATESTRUCT](#)

[CREATESTRUCT](#)

[Conceptual](#)

[Multiple Document Interface](#)

[Reference](#)

[WM_CREATE](#)

Feedback

Was this page helpful?

 Yes

 No

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

MDICREATESTRUCTW structure (winuser.h)

Article 07/27/2022

Contains information about the class, title, owner, location, and size of a multiple-document interface (MDI) child window.

Syntax

C++

```
typedef struct tagMDICREATESTRUCTW {
    LPCWSTR szClass;
    LPCWSTR szTitle;
    HANDLE hOwner;
    int x;
    int y;
    int cx;
    int cy;
    DWORD style;
    LPARAM lParam;
} MDICREATESTRUCTW, *LPMDICREATESTRUCTW;
```

Members

`szClass`

Type: [LPCTSTR](#)

The name of the window class of the MDI child window. The class name must have been registered by a previous call to the [RegisterClass](#) function.

`szTitle`

Type: [LPCTSTR](#)

The title of the MDI child window. The system displays the title in the child window's title bar.

`hOwner`

Type: [HANDLE](#)

A handle to the instance of the application creating the MDI client window.

x

Type: int

The initial horizontal position, in client coordinates, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default horizontal position.

y

Type: int

The initial vertical position, in client coordinates, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default vertical position.

cx

Type: int

The initial width, in device units, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default width.

cy

Type: int

The initial height, in device units, of the MDI child window. If this member is set to **CW_USEDEFAULT**, the MDI child window is assigned the default height.

style

Type: DWORD

The style of the MDI child window. If the MDI client window was created with the **MDIS_ALLCHILDSTYLES** window style, this member can be any combination of the window styles listed in the [Window Styles](#) page. Otherwise, this member can be one or more of the following values.

Value	Meaning
WS_MINIMIZE 0x20000000L	Creates an MDI child window that is initially minimized.
WS_MAXIMIZE	Creates an MDI child window that is initially maximized.

0x01000000L	
WS_HSCROLL 0x00100000L	Creates an MDI child window that has a horizontal scroll bar.
WS_VSCROLL 0x00200000L	Creates an MDI child window that has a vertical scroll bar.

lParam

Type: **LPARAM**

An application-defined value.

Remarks

When the MDI client window creates an MDI child window by calling [CreateWindow](#), the system sends a [WM_CREATE](#) message to the created window. The *lParam* member of the [WM_CREATE](#) message contains a pointer to a [CREATESTRUCT](#) structure. The *lpCreateParams* member of this structure contains a pointer to the [MDICREATESTRUCT](#) structure passed with the [WM_MDICREATE](#) message that created the MDI child window.

(!) Note

The winuser.h header defines [MDICREATESTRUCT](#) 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	winuser.h (include Windows.h)

See also

[CLIENTCREATESTRUCT](#)

[CREATESTRUCT](#)

[Conceptual](#)

[Multiple Document Interface](#)

[Reference](#)

[WM_CREATE](#)

Feedback

Was this page helpful?

 Yes

 No

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

MINIMIZEDMETRICS structure (winuser.h)

Article 04/02/2021

Contains the scalable metrics associated with minimized windows. This structure is used with the [SystemParametersInfo](#) function when the SPI_GETMINIMIZEDMETRICS or SPI_SETMINIMIZEDMETRICS action value is specified.

Syntax

C++

```
typedef struct tagMINIMIZEDMETRICS {
    UINT cbSize;
    int iWidth;
    int iHorzGap;
    int iVertGap;
    int iArrange;
} MINIMIZEDMETRICS, *PMINIMIZEDMETRICS, *LPMINIMIZEDMETRICS;
```

Members

`cbSize`

The size of the structure, in bytes. The caller must set this to `sizeof(MINIMIZEDMETRICS)`.

`iWidth`

The width of minimized windows, in pixels.

`iHorzGap`

The horizontal space between arranged minimized windows, in pixels.

`iVertGap`

The vertical space between arranged minimized windows, in pixels.

`iArrange`

The starting position and direction used when arranging minimized windows. The starting position must be one of the following values.

Value	Meaning
ARW_BOTTOMLEFT 0x0000L	Start at the lower-left corner of the work area.
ARW_BOTTOMRIGHT 0x0001L	Start at the lower-right corner of the work area.
ARW_TOPLEFT 0x0002L	Start at the upper-left corner of the work area.
ARW_TOPRIGHT 0x0003L	Start at the upper-right corner of the work area.

The direction must be one of the following values.

Value	Meaning
ARW_LEFT 0x0000L	Arrange left (valid with ARW_BOTTOMRIGHT and ARW_TOPRIGHT only).
ARW_RIGHT 0x0000L	Arrange right (valid with ARW_BOTTOMLEFT and ARW_TOPLEFT only).
ARW_UP 0x0004L	Arrange up (valid with ARW_BOTTOMLEFT and ARW_BOTTOMRIGHT only).
ARW_DOWN 0x0004L	Arrange down (valid with ARW_TOPLEFT and ARW_TOPRIGHT only).
ARW_HIDE 0x0008L	Hide minimized windows by moving them off the visible area of the screen.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

MINMAXINFO structure (winuser.h)

Article11/19/2022

Contains information about a window's maximized size and position and its minimum and maximum tracking size.

Syntax

C++

```
typedef struct tagMINMAXINFO {
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO, *PMINMAXINFO, *LPMINMAXINFO;
```

Members

`ptReserved`

Type: [POINT](#)

Reserved; do not use.

`ptMaxSize`

Type: [POINT](#)

The maximized width (x member) and the maximized height (y member) of the window. For top-level windows, this value is based on the width of the primary monitor.

`ptMaxPosition`

Type: [POINT](#)

The position of the left side of the maximized window (x member) and the position of the top of the maximized window (y member). For top-level windows, this value is based on the position of the primary monitor.

`ptMinTrackSize`

Type: **POINT**

The minimum tracking width (x member) and the minimum tracking height (y member) of the window. This value can be obtained programmatically from the system metrics **SM_CXMINTRACK** and **SM_CYMINTRACK** (see the [GetSystemMetrics](#) function).

`ptMaxTrackSize`

Type: **POINT**

The maximum tracking width (x member) and the maximum tracking height (y member) of the window. This value is based on the size of the virtual screen and can be obtained programmatically from the system metrics **SM_CXMAXTRACK** and **SM_CYMAXTRACK** (see the [GetSystemMetrics](#) function).

Remarks

For systems with multiple monitors, the **ptMaxSize** and **ptMaxPosition** members describe the maximized size and position of the window on the primary monitor, even if the window ultimately maximizes onto a secondary monitor. In that case, the window manager adjusts these values to compensate for differences between the primary monitor and the monitor that displays the window. Thus, if the user leaves **ptMaxSize** untouched, a window on a monitor larger than the primary monitor maximizes to the size of the larger monitor.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[POINT](#)

[Reference](#)

WM_GETMINMAXINFO

Windows

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

MOUSEHOOKSTRUCT structure (winuser.h)

Article 11/19/2022

Contains information about a mouse event passed to a **WH_MOUSE** hook procedure, [MouseProc](#).

Syntax

C++

```
typedef struct tagMOUSEHOOKSTRUCT {
    POINT      pt;
    HWND       hwnd;
    UINT       wHitTestCode;
    ULONG_PTR dwExtraInfo;
} MOUSEHOOKSTRUCT, *LPMOUSEHOOKSTRUCT, *PMOUSEHOOKSTRUCT;
```

Members

pt

Type: [POINT](#)

The x- and y-coordinates of the cursor, in screen coordinates.

hwnd

Type: [HWND](#)

A handle to the window that will receive the mouse message corresponding to the mouse event.

wHitTestCode

Type: [UINT](#)

The hit-test value. For a list of hit-test values, see the description of the [WM_NCHITTEST](#) message.

dwExtraInfo

Type: **ULONG_PTR**

Additional information associated with the message.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Hooks](#)

[MouseProc](#)

[Reference](#)

[SetWindowsHookEx](#)

[WM_NCHITTEST](#)

Feedback

Was this page helpful?

 Yes

 No

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

MOUSEHOOKSTRUCTEX structure (winuser.h)

Article02/16/2023

Contains information about a mouse event passed to a [WH_MOUSE](#) hook procedure, [MouseProc](#).

This is an extension of the [MOUSEHOOKSTRUCT](#) structure that includes information about wheel movement or the use of the X button.

Syntax

C++

```
typedef struct tagMOUSEHOOKSTRUCTEX : tagMOUSEHOOKSTRUCT {  
    DWORD mouseData;  
} MOUSEHOOKSTRUCTEX, *LPMOUSEHOOKSTRUCTEX, *PMOUSEHOOKSTRUCTEX;
```

Inheritance

The [MOUSEHOOKSTRUCTEX](#) structure implements [tagMOUSEHOOKSTRUCT](#).

Members

`mouseData`

Type: [DWORD](#)

If the message is [WM_MOUSEWHEEL](#), the HIWORD of this member is the wheel delta. The LOWORD is undefined and reserved. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. One wheel click is defined as [WHEEL_DELTA](#), which is 120.

If the message is [WM_XBUTTONDOWN](#), [WM_XBUTTONUP](#), [WM_XBUTTONDOWNDBLCLK](#), [WM_NCXBUTTONDOWN](#), [WM_NCXBUTTONUP](#), or [WM_NCXBUTTONONDBLCLK](#), the HIWORD of `mouseData` specifies which X button was pressed or released, and the LOWORD is undefined and reserved. This member can be one or more of the following values. Otherwise, `mouseData` is not used.

Value	Meaning
XBUTTON1 0x0001	The first X button was pressed or released.
XBUTTON2 0x0002	The second X button was pressed or released.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Hooks](#)

[MOUSEHOOKSTRUCT](#)

[MouseProc](#)

Reference

[WM_MOUSEWHEEL](#)

[WM_NCXBUTTONDBLCLK](#)

[WM_NCXBUTTONDOWN](#)

[WM_NCXBUTTONUP](#)

[WM_XBUTTONDOWNDBLCLK](#)

[WM_XBUTTONDOWN](#)

[WM_XBUTTONUP](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

MoveWindow function (winuser.h)

Article 10/13/2021

Changes the position and dimensions of the specified window. For a top-level window, the position and dimensions are relative to the upper-left corner of the screen. For a child window, they are relative to the upper-left corner of the parent window's client area.

Syntax

C++

```
BOOL MoveWindow(
    [in] HWND hWnd,
    [in] int X,
    [in] int Y,
    [in] int nWidth,
    [in] int nHeight,
    [in] BOOL bRepaint
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[in] X

Type: **int**

The new position of the left side of the window.

[in] Y

Type: **int**

The new position of the top of the window.

[in] nWidth

Type: **int**

The new width of the window.

[in] nHeight

Type: int

The new height of the window.

[in] bRepaint

Type: BOOL

Indicates whether the window is to be repainted. If this parameter is TRUE, the window receives a message. If the parameter is FALSE, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of moving a child window.

Return value

Type: BOOL

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 *bRepaint* parameter is TRUE, the system sends the [WM_PAINT](#) message to the window procedure immediately after moving the window (that is, the [MoveWindow](#) function calls the [UpdateWindow](#) function). If *bRepaint* is FALSE, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.

[MoveWindow](#) sends the [WM_WINDOWPOSCHANGING](#), [WM_WINDOWPOSCHANGED](#), [WM_MOVE](#), [WM_SIZE](#), and [WM_NCCALCSIZE](#) messages to the window.

Examples

For an example, see [Creating, Enumerating, and Sizing Child Windows](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[Conceptual](#)

[Other Resources](#)

[Reference](#)

[SetWindowPos](#)

[UpdateWindow](#)

[WM_GETMINMAXINFO](#)

[WM_PAINT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

MSG structure (winuser.h)

Article 11/19/2022

Contains message information from a thread's message queue.

Syntax

C++

```
typedef struct tagMSG {
    HWND    hwnd;
    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT   pt;
    DWORD   lPrivate;
} MSG, *PMSG, *NPMSG, *LPMMSG;
```

Members

hwnd

Type: **HWND**

A handle to the window whose window procedure receives the message. This member is **NULL** when the message is a thread message.

message

Type: **UINT**

The message identifier. Applications can only use the low word; the high word is reserved by the system.

wParam

Type: **WPARAM**

Additional information about the message. The exact meaning depends on the value of the **message** member.

lParam

Type: **LPARAM**

Additional information about the message. The exact meaning depends on the value of the **message** member.

`time`

Type: **DWORD**

The time at which the message was posted.

`pt`

Type: **POINT**

The cursor position, in screen coordinates, when the message was posted.

`lPrivate`

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps UWP apps]
Minimum supported server	Windows 2000 Server [desktop apps UWP apps]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[GetMessage](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[PostThreadMessage](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

MSLLHOOKSTRUCT structure (winuser.h)

Article11/19/2022

Contains information about a low-level mouse input event.

Syntax

C++

```
typedef struct tagMSLLHOOKSTRUCT {
    POINT      pt;
    DWORD      mouseData;
    DWORD      flags;
    DWORD      time;
    ULONG_PTR  dwExtraInfo;
} MSLLHOOKSTRUCT, *LPMSSLHOOKSTRUCT, *PMSLLHOOKSTRUCT;
```

Members

pt

Type: **POINT**

The x- and y-coordinates of the cursor, in [per-monitor-aware](#) screen coordinates.

mouseData

Type: **DWORD**

If the message is [WM_MOUSEWHEEL](#), the high-order word of this member is the wheel delta. The low-order word is reserved. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. One wheel click is defined as [WHEEL_DELTA](#), which is 120.

If the message is [WM_XBUTTONDOWN](#), [WM_XBUTTONUP](#), [WM_XBUTTONDOWNDBLCLK](#), [WM_NCXBUTTONDOWN](#), [WM_NCXBUTTONUP](#), or [WM_NCXBUTTONONDBLCLK](#), the high-order word specifies which X button was pressed or released, and the low-order word is reserved. This value can be one or more of the following values. Otherwise, **mouseData** is not used.

Value	Meaning
XBUTTON1 0x0001	The first X button was pressed or released.
XBUTTON2 0x0002	The second X button was pressed or released.

flags

Type: **DWORD**

The event-injected flags. An application can use the following values to test the flags. Testing LLMHF_INJECTED (bit 0) will tell you whether the event was injected. If it was, then testing LLMHF_LOWER_IL_INJECTED (bit 1) will tell you whether or not the event was injected from a process running at lower integrity level.

Value	Meaning
LLMHF_INJECTED 0x00000001	Test the event-injected (from any process) flag.
LLMHF_LOWER_IL_INJECTED 0x00000002	Test the event-injected (from a process running at lower integrity level) flag.

time

Type: **DWORD**

The time stamp for this message.

dwExtraInfo

Type: **ULONG_PTR**

Additional information associated with the message.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Hooks](#)

[LowLevelMouseProc](#)

Other Resources

[POINT](#)

Reference

[SetWindowsHookEx](#)

[WM_MOUSEWHEEL](#)

[WM_NCXBUTTONDBLCLK](#)

[WM_NCXBUTTONDOWN](#)

[WM_NCXBUTTONUP](#)

[WM_XBUTTONDBLCLK](#)

[WM_XBUTTONDOWN](#)

[WM_XBUTTONUP](#)

Feedback

Was this page helpful?

 Yes

 No

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

NCCALCSIZE_PARAMS structure (winuser.h)

Article09/01/2022

Contains information that an application can use while processing the [WM_NCCALCSIZE](#) message to calculate the size, position, and valid contents of the client area of a window.

Syntax

C++

```
typedef struct tagNCCALCSIZE_PARAMS {
    RECT      rgrc[3];
    PWINDOWPOS lppos;
} NCCALCSIZE_PARAMS, *LPNCCALCSIZE_PARAMS;
```

Members

rgrc[3]

Type: [RECT\[3\]](#)

An array of rectangles. The meaning of the array of rectangles changes during the processing of the [WM_NCCALCSIZE](#) message.

When the window procedure receives the [WM_NCCALCSIZE](#) message, the first rectangle contains the new coordinates of a window that has been moved or resized, that is, it is the proposed new window coordinates. The second contains the coordinates of the window before it was moved or resized. The third contains the coordinates of the window's client area before the window was moved or resized. If the window is a child window, the coordinates are relative to the client area of the parent window. If the window is a top-level window, the coordinates are relative to the screen origin.

When the window procedure returns, the first rectangle contains the coordinates of the new client rectangle resulting from the move or resize. The second rectangle contains the valid destination rectangle, and the third rectangle contains the valid source rectangle. The last two rectangles are used in conjunction with the return value of the [WM_NCCALCSIZE](#) message to determine the area of the window to be preserved.

lppos

Type: PWINDOWPOS

A pointer to a [WINDOWPOS](#) structure that contains the size and position values specified in the operation that moved or resized the window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[MoveWindow](#)

[Other Resources](#)

[RECT](#)

[Reference](#)

[SetWindowPos](#)

[WINDOWPOS](#)

[WM_NCCALCSIZE](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

NONCLIENTMETRICS structure (winuser.h)

Article 07/27/2022

Contains the scalable metrics associated with the nonclient area of a nonminimized window. This structure is used by the [SPI_GETNONCLIENTMETRICS](#) and [SPI_SETNONCLIENTMETRICS](#) actions of the [SystemParametersInfo](#) function.

Syntax

C++

```
typedef struct tagNONCLIENTMETRICSA {
    UINT      cbSize;
    int       iBorderWidth;
    int       iScrollWidth;
    int       iScrollHeight;
    int       iCaptionWidth;
    int       iCaptionHeight;
    LOGFONTA lfCaptionFont;
    int       iSmCaptionWidth;
    int       iSmCaptionHeight;
    LOGFONTA lfSmCaptionFont;
    int       iMenuWidth;
    int       iMenuHeight;
    LOGFONTA lfMenuFont;
    LOGFONTA lfStatusFont;
    LOGFONTA lfMessageFont;
    int       iPaddedBorderWidth;
} NONCLIENTMETRICSA, *PNONCLIENTMETRICSA, *LPNONCLIENTMETRICSA;
```

Members

`cbSize`

The size of the structure, in bytes. The caller must set this to `sizeof(NONCLIENTMETRICS)`. For information about application compatibility, see Remarks.

`iBorderWidth`

The thickness of the sizing border, in pixels. The default is 1 pixel.

`iScrollWidth`

The width of a standard vertical scroll bar, in pixels.

`iScrollHeight`

The height of a standard horizontal scroll bar, in pixels.

`iCaptionWidth`

The width of caption buttons, in pixels.

`iCaptionHeight`

The height of caption buttons, in pixels.

`lfCaptionFont`

A [LOGFONT](#) structure that contains information about the caption font.

`iSmCaptionWidth`

The width of small caption buttons, in pixels.

`iSmCaptionHeight`

The height of small captions, in pixels.

`lfSmCaptionFont`

A [LOGFONT](#) structure that contains information about the small caption font.

`iMenuWidth`

The width of menu-bar buttons, in pixels.

`iMenuHeight`

The height of a menu bar, in pixels.

`lfMenuFont`

A [LOGFONT](#) structure that contains information about the font used in menu bars.

`lfStatusFont`

A [LOGFONT](#) structure that contains information about the font used in status bars and tooltips.

`lfMessageFont`

A [LOGFONT](#) structure that contains information about the font used in message boxes.

iPaddedBorderWidth

The thickness of the padded border, in pixels. The default value is 4 pixels. The **iPaddedBorderWidth** and **iBorderWidth** members are combined for both resizable and nonresizable windows in the Windows Aero desktop experience. To compile an application that uses this member, define [_WIN32_WINNT](#) as 0x0600 or later. For more information, see Remarks.

Windows Server 2003 and Windows XP/2000: This member is not supported.

Remarks

If the **iPaddedBorderWidth** member of the [NONCLIENTMETRICS](#) structure is present, this structure is 4 bytes larger than for an application that is compiled with [_WIN32_WINNT](#) less than or equal to 0x0502. For more information about conditional compilation, see [Using the Windows Headers](#).

Windows Server 2003 and Windows XP/2000: If an application that is compiled for Windows Server 2008 or Windows Vista must also run on Windows Server 2003 or Windows XP/2000, use the [GetVersionEx](#) function to check the operating system version at run time and, if the application is running on Windows Server 2003 or Windows XP/2000, subtract the size of the **iPaddedBorderWidth** member from the **cbSize** member of the [NONCLIENTMETRICS](#) structure before calling the [SystemParametersInfo](#) function.

ⓘ Note

The winuser.h header defines NONCLIENTMETRICS 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	winuser.h (include Windows.h)

See also

[LOGFONT](#)

[SystemParametersInfo](#)

Feedback

Was this page helpful?



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

NONCLIENTMETRICSW structure (winuser.h)

Article 07/27/2022

Contains the scalable metrics associated with the nonclient area of a nonminimized window. This structure is used by the [SPI_GETNONCLIENTMETRICS](#) and [SPI_SETNONCLIENTMETRICS](#) actions of the [SystemParametersInfo](#) function.

Syntax

C++

```
typedef struct tagNONCLIENTMETRICSW {
    UINT      cbSize;
    int       iBorderWidth;
    int       iScrollWidth;
    int       iScrollHeight;
    int       iCaptionWidth;
    int       iCaptionHeight;
    LOGFONTW lfCaptionFont;
    int       iSmCaptionWidth;
    int       iSmCaptionHeight;
    LOGFONTW lfSmCaptionFont;
    int       iMenuWidth;
    int       iMenuHeight;
    LOGFONTW lfMenuFont;
    LOGFONTW lfStatusFont;
    LOGFONTW lfMessageFont;
    int       iPaddedBorderWidth;
} NONCLIENTMETRICSW, *PNONCLIENTMETRICSW, *LPNONCLIENTMETRICSW;
```

Members

`cbSize`

The size of the structure, in bytes. The caller must set this to `sizeof(NONCLIENTMETRICS)`. For information about application compatibility, see Remarks.

`iBorderWidth`

The thickness of the sizing border, in pixels. The default is 1 pixel.

`iScrollWidth`

The width of a standard vertical scroll bar, in pixels.

`iScrollHeight`

The height of a standard horizontal scroll bar, in pixels.

`iCaptionWidth`

The width of caption buttons, in pixels.

`iCaptionHeight`

The height of caption buttons, in pixels.

`lfCaptionFont`

A [LOGFONT](#) structure that contains information about the caption font.

`iSmCaptionWidth`

The width of small caption buttons, in pixels.

`iSmCaptionHeight`

The height of small captions, in pixels.

`lfSmCaptionFont`

A [LOGFONT](#) structure that contains information about the small caption font.

`iMenuWidth`

The width of menu-bar buttons, in pixels.

`iMenuHeight`

The height of a menu bar, in pixels.

`lfMenuFont`

A [LOGFONT](#) structure that contains information about the font used in menu bars.

`lfStatusFont`

A [LOGFONT](#) structure that contains information about the font used in status bars and tooltips.

`lfMessageFont`

A [LOGFONT](#) structure that contains information about the font used in message boxes.

iPaddedBorderWidth

The thickness of the padded border, in pixels. The default value is 4 pixels. The **iPaddedBorderWidth** and **iBorderWidth** members are combined for both resizable and nonresizable windows in the Windows Aero desktop experience. To compile an application that uses this member, define [_WIN32_WINNT](#) as 0x0600 or later. For more information, see Remarks.

Windows Server 2003 and Windows XP/2000: This member is not supported.

Remarks

If the **iPaddedBorderWidth** member of the [NONCLIENTMETRICS](#) structure is present, this structure is 4 bytes larger than for an application that is compiled with [_WIN32_WINNT](#) less than or equal to 0x0502. For more information about conditional compilation, see [Using the Windows Headers](#).

Windows Server 2003 and Windows XP/2000: If an application that is compiled for Windows Server 2008 or Windows Vista must also run on Windows Server 2003 or Windows XP/2000, use the [GetVersionEx](#) function to check the operating system version at run time and, if the application is running on Windows Server 2003 or Windows XP/2000, subtract the size of the **iPaddedBorderWidth** member from the **cbSize** member of the [NONCLIENTMETRICS](#) structure before calling the [SystemParametersInfo](#) function.

ⓘ Note

The winuser.h header defines NONCLIENTMETRICS 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	winuser.h (include Windows.h)

See also

[LOGFONT](#)

[SystemParametersInfo](#)

Feedback

Was this page helpful?



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

OpenIcon function (winuser.h)

Article10/13/2021

Restores a minimized (iconic) window to its previous size and position; it then activates the window.

Syntax

C++

```
BOOL OpenIcon(  
    [in] HWND hWnd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be restored and activated.

Return value

Type: **BOOL**

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

OpenIcon sends a [WM_QUERYOPEN](#) message to the given window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[CloseWindow](#)

[Conceptual](#)

[Iconic](#)

[Reference](#)

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

PeekMessageA function (winuser.h)

Article 02/09/2023

Dispatches incoming nonqueued messages, checks the thread message queue for a posted message, and retrieves the message (if any exist).

Syntax

C++

```
BOOL PeekMessageA(
    [out]         LPMMSG lpMsg,
    [in, optional] HWND hWnd,
    [in]          UINT wMsgFilterMin,
    [in]          UINT wMsgFilterMax,
    [in]          UINT wRemoveMsg
);
```

Parameters

[out] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that receives message information.

[in, optional] hWnd

Type: [HWND](#)

A handle to the window whose messages are to be retrieved. The window must belong to the current thread.

If *hWnd* is **NULL**, **PeekMessage** retrieves messages for any window that belongs to the current thread, and any messages on the current thread's message queue whose *hwnd* value is **NULL** (see the [MSG](#) structure). Therefore if *hWnd* is **NULL**, both window messages and thread messages are processed.

If *hWnd* is -1, **PeekMessage** retrieves only messages on the current thread's message queue whose *hwnd* value is **NULL**, that is, thread messages as posted by [PostMessage](#) (when the *hWnd* parameter is **NULL**) or [PostThreadMessage](#).

[in] wMsgFilterMin

Type: **UINT**

The value of the first message in the range of messages to be examined. Use **WM_KEYFIRST** (0x0100) to specify the first keyboard message or **WM_MOUSEFIRST** (0x0200) to specify the first mouse message.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (that is, no range filtering is performed).

[in] *wMsgFilterMax*

Type: **UINT**

The value of the last message in the range of messages to be examined. Use **WM_KEYLAST** to specify the last keyboard message or **WM_MOUSELAST** to specify the last mouse message.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (that is, no range filtering is performed).

[in] *wRemoveMsg*

Type: **UINT**

Specifies how messages are to be handled. This parameter can be one or more of the following values.

Value	Meaning
PM_NOREMOVE 0x0000	Messages are not removed from the queue after processing by PeekMessage .
PM_REMOVE 0x0001	Messages are removed from the queue after processing by PeekMessage .
PM_NOYIELD 0x0002	Prevents the system from releasing any thread that is waiting for the caller to go idle (see WaitForInputIdle). Combine this value with either PM_NOREMOVE or PM_REMOVE .

By default, all message types are processed. To specify that only certain message should be processed, specify one or more of the following values.

Value	Meaning

PM_QS_INPUT (QS_INPUT << 16)	Process mouse and keyboard messages.
PM_QS_PAINT (QS_PAINT << 16)	Process paint messages.
PM_QS_POSTMESSAGE ((QS_POSTMESSAGE QS_HOTKEY QS_TIMER) << 16)	Process all posted messages, including timers and hotkeys.
PM_QS_SENDMESSAGE (QS_SENDMESSAGE << 16)	Process all sent messages.

Return value

Type: **BOOL**

If a message is available, the return value is nonzero.

If no messages are available, the return value is zero.

Remarks

PeekMessage retrieves messages associated with the window identified by the *hWnd* parameter or any of its children as specified by the [IsChild](#) function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. Note that an application can only use the low word in the *wMsgFilterMin* and *wMsgFilterMax* parameters; the high word is reserved for the system.

Note that **PeekMessage** always retrieves [WM_QUIT](#) messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

During this call, the system dispatches ([DispatchMessage](#)) pending, nonqueued messages, that is, messages sent to windows owned by the calling thread using the [SendMessage](#), [SendMessageCallback](#), [SendMessageTimeout](#), or [SendNotifyMessage](#) function. Then the first queued message that matches the specified filter is retrieved. The system may also process internal events. If no filter is specified, messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events
- Sent messages (again)
- [WM_PAINT](#) messages

- [WM_TIMER](#) messages

To retrieve input messages before posted messages, use the `wMsgFilterMin` and `wMsgFilterMax` parameters.

The [PeekMessage](#) function normally does not remove [WM_PAINT](#) messages from the queue. [WM_PAINT](#) messages remain in the queue until they are processed. However, if a [WM_PAINT](#) message has a **NULL** update region, [PeekMessage](#) does remove it from the queue.

If a top-level window stops responding to messages for more than several seconds, the system considers the window to be not responding and replaces it with a ghost window that has the same z-order, location, size, and visual attributes. This allows the user to move it, resize it, or even close the application. However, these are the only actions available because the application is actually not responding. When an application is being debugged, the system does not generate a ghost window.

DPI Virtualization

This API does not participate in DPI virtualization. The output is in the mode of the window that the message is targeting. The calling thread is not taken into consideration.

Examples

For an example, see [Examining a Message Queue](#).

Note

The winuser.h header defines PeekMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetMessage](#)

[IsChild](#)

[MSG](#)

[Messages and Message Queues](#)

[Other Resources](#)

[Reference](#)

[WaitForInputIdle](#)

[WaitMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

PeekMessageW function (winuser.h)

Article 02/09/2023

Dispatches incoming nonqueued messages, checks the thread message queue for a posted message, and retrieves the message (if any exist).

Syntax

C++

```
BOOL PeekMessageW(
    [out]         LPMMSG lpMsg,
    [in, optional] HWND hWnd,
    [in]          UINT wMsgFilterMin,
    [in]          UINT wMsgFilterMax,
    [in]          UINT wRemoveMsg
);
```

Parameters

[out] lpMsg

Type: [LPMMSG](#)

A pointer to an [MSG](#) structure that receives message information.

[in, optional] hWnd

Type: [HWND](#)

A handle to the window whose messages are to be retrieved. The window must belong to the current thread.

If *hWnd* is **NULL**, **PeekMessage** retrieves messages for any window that belongs to the current thread, and any messages on the current thread's message queue whose *hwnd* value is **NULL** (see the [MSG](#) structure). Therefore if *hWnd* is **NULL**, both window messages and thread messages are processed.

If *hWnd* is -1, **PeekMessage** retrieves only messages on the current thread's message queue whose *hwnd* value is **NULL**, that is, thread messages as posted by [PostMessage](#) (when the *hWnd* parameter is **NULL**) or [PostThreadMessage](#).

[in] wMsgFilterMin

Type: **UINT**

The value of the first message in the range of messages to be examined. Use **WM_KEYFIRST** (0x0100) to specify the first keyboard message or **WM_MOUSEFIRST** (0x0200) to specify the first mouse message.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (that is, no range filtering is performed).

[in] *wMsgFilterMax*

Type: **UINT**

The value of the last message in the range of messages to be examined. Use **WM_KEYLAST** to specify the last keyboard message or **WM_MOUSELAST** to specify the last mouse message.

If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (that is, no range filtering is performed).

[in] *wRemoveMsg*

Type: **UINT**

Specifies how messages are to be handled. This parameter can be one or more of the following values.

Value	Meaning
PM_NOREMOVE 0x0000	Messages are not removed from the queue after processing by PeekMessage .
PM_REMOVE 0x0001	Messages are removed from the queue after processing by PeekMessage .
PM_NOYIELD 0x0002	Prevents the system from releasing any thread that is waiting for the caller to go idle (see WaitForInputIdle). Combine this value with either PM_NOREMOVE or PM_REMOVE .

By default, all message types are processed. To specify that only certain message should be processed, specify one or more of the following values.

Value	Meaning

PM_QS_INPUT (QS_INPUT << 16)	Process mouse and keyboard messages.
PM_QS_PAINT (QS_PAINT << 16)	Process paint messages.
PM_QS_POSTMESSAGE ((QS_POSTMESSAGE QS_HOTKEY QS_TIMER) << 16)	Process all posted messages, including timers and hotkeys.
PM_QS_SENDMESSAGE (QS_SENDMESSAGE << 16)	Process all sent messages.

Return value

Type: **BOOL**

If a message is available, the return value is nonzero.

If no messages are available, the return value is zero.

Remarks

PeekMessage retrieves messages associated with the window identified by the *hWnd* parameter or any of its children as specified by the [IsChild](#) function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. Note that an application can only use the low word in the *wMsgFilterMin* and *wMsgFilterMax* parameters; the high word is reserved for the system.

Note that **PeekMessage** always retrieves [WM_QUIT](#) messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

During this call, the system dispatches ([DispatchMessage](#)) pending, nonqueued messages, that is, messages sent to windows owned by the calling thread using the [SendMessage](#), [SendMessageCallback](#), [SendMessageTimeout](#), or [SendNotifyMessage](#) function. Then the first queued message that matches the specified filter is retrieved. The system may also process internal events. If no filter is specified, messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events
- Sent messages (again)
- [WM_PAINT](#) messages

- [WM_TIMER](#) messages

To retrieve input messages before posted messages, use the `wMsgFilterMin` and `wMsgFilterMax` parameters.

The [PeekMessage](#) function normally does not remove [WM_PAINT](#) messages from the queue. [WM_PAINT](#) messages remain in the queue until they are processed. However, if a [WM_PAINT](#) message has a **NULL** update region, [PeekMessage](#) does remove it from the queue.

If a top-level window stops responding to messages for more than several seconds, the system considers the window to be not responding and replaces it with a ghost window that has the same z-order, location, size, and visual attributes. This allows the user to move it, resize it, or even close the application. However, these are the only actions available because the application is actually not responding. When an application is being debugged, the system does not generate a ghost window.

DPI Virtualization

This API does not participate in DPI virtualization. The output is in the mode of the window that the message is targeting. The calling thread is not taken into consideration.

Examples

For an example, see [Examining a Message Queue](#).

Note

The winuser.h header defines PeekMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetMessage](#)

[IsChild](#)

[MSG](#)

[Messages and Message Queues](#)

[Other Resources](#)

[Reference](#)

[WaitForInputIdle](#)

[WaitMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

PhysicalToLogicalPoint function (winuser.h)

Article11/19/2022

Converts the physical coordinates of a point in a window to logical coordinates.

Syntax

C++

```
BOOL PhysicalToLogicalPoint(
    [in]      HWND      hWnd,
    [in, out] LPPOINT  lpPoint
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose transform is used for the conversion. Top level windows are fully supported. In the case of child windows, only the area of overlap between the parent and the child window is converted.

[in, out] lpPoint

Type: **LPPOINT**

A pointer to a **POINT** structure that specifies the physical/screen coordinates to be converted. The new logical coordinates are copied into this structure if the function succeeds.

Return value

None

Remarks

Windows Vista introduces the concept of physical coordinates. Desktop Window Manager (DWM) scales non-dots per inch (dpi) aware windows when the display is high dpi. The window seen on the screen corresponds to the physical coordinates. The application continues to work in logical space. Therefore, the application's view of the window is different from that which appears on the screen. For scaled windows, logical and physical coordinates are different.

The function uses the window identified by the *hWnd* parameter and the physical coordinates given in the [POINT](#) structure to compute the logical coordinates. The logical coordinates are the *unscaled* coordinates that appear to the application in a programmatic way. In other words, the logical coordinates are the coordinates the application recognizes, which can be different from the physical coordinates. The API then replaces the physical coordinates with the logical coordinates. The new coordinates are in the *world* coordinates whose origin is (0, 0) on the desktop. The coordinates passed to the API have to be on the *hWnd*.

The source coordinates are in device units.

On all platforms, [PhysicalToLogicalPoint](#) will fail on a window that has either 0 width or height; an application must first establish a non-0 width and height by calling, for example, [MoveWindow](#). On some versions of Windows (including Windows 7), [PhysicalToLogicalPoint](#) will still fail if [MoveWindow](#) has been called after a call to [ShowWindow](#) with [SH_HIDE](#) has hidden the window.

In Windows 8, system-DPI aware applications translate between physical and logical space using [PhysicalToLogicalPoint](#) and [LogicalToPhysicalPoint](#). In Windows 8.1, the additional virtualization of the system and inter-process communications means that for the majority of applications, you do not need these APIs. As a result, in Windows 8.1, [PhysicalToLogicalPoint](#) and [LogicalToPhysicalPoint](#) no longer transform points. The system returns all points to an application in its own coordinate space. This behavior preserves functionality for the majority of applications, but there are some exceptions in which you must make changes to ensure that the application works as expected. In those cases, use [PhysicalToLogicalPointForPerMonitorDPI](#) and [LogicalToPhysicalPointForPerMonitorDPI](#).

Requirements

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

Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

Feedback

Was this page helpful?

 Yes

 No

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

PostMessageA function (winuser.h)

Article 02/09/2023

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message.

To post a message in the message queue associated with a thread, use the [PostThreadMessage](#) function.

Syntax

C++

```
BOOL PostMessageA(
    [in, optional] HWND hWnd,
    [in]           UINT Msg,
    [in]           WPARAM wParam,
    [in]           LPARAM lParam
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window whose window procedure is to receive the message. The following values have special meanings.

Value	Meaning
HWND_BROADCAST ((HWND)0xffff)	The message is posted to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows. The message is not posted to child windows.
NULL	The function behaves like a call to PostThreadMessage with the <i>dwThreadId</i> parameter set to the identifier of the current thread.

Starting with Windows Vista, message posting is subject to UIPI. The thread of a process can post messages only to message queues of threads in processes of lesser or equal

integrity level.

[in] `Msg`

Type: **UINT**

The message to be posted.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] `wParam`

Type: **WPARAM**

Additional message-specific information.

[in] `lParam`

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **BOOL**

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 a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

Messages in a message queue are retrieved by calls to the [GetMessage](#) or [PeekMessage](#) function.

Applications that need to communicate using **HWND_BROADCAST** should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= [WM_USER](#)) to another process, you

must do custom marshalling.

If you send a message in the range below [WM_USER](#) to the asynchronous message functions ([PostMessage](#), [SendNotifyMessage](#), and [SendMessageCallback](#)), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Do not post the [WM_QUIT](#) message using [PostMessage](#); use the [PostQuitMessage](#) function.

An accessibility application can use [PostMessage](#) to post [WM_APPCOMMAND](#) messages to the shell to launch applications. This functionality is not guaranteed to work for other types of applications.

A message queue can contain at most 10,000 messages. This limit should be sufficiently large. If your application exceeds the limit, it should be redesigned to avoid consuming so many system resources. To adjust this limit, modify the following registry key.

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          Windows  
            USERPostMessageLimit
```

If the function fails, call [GetLastError](#) to get extended error information. [GetLastError](#) returns [ERROR_NOT_ENOUGH_QUOTA](#) when the limit is hit.

The minimum acceptable value is 4000.

Examples

The following example shows how to post a private window message using the [PostMessage](#) function. Assume you defined a private window message called [WM_COMPLETE](#):

C++

```
#define WM_COMPLETE (WM_USER + 0)
```

You can post a message to the message queue associated with the thread that created the specified window as shown below:

C++

```
WaitForSingleObject (pparams->hEvent, INFINITE) ;  
lTime = GetCurrentTime () ;  
PostMessage (pparams->hwnd, WM_COMPLETE, 0, lTime);
```

For more examples, see [Initiating a Data Link](#).

 **Note**

The winuser.h header defines PostMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetMessage](#)

Messages and Message Queues

[PeekMessage](#)

[PostQuitMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

PostMessageW function (winuser.h)

Article 03/21/2023

Places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message.

To post a message in the message queue associated with a thread, use the [PostThreadMessage](#) function.

Syntax

C++

```
BOOL PostMessageW(
    [in, optional] HWND hWnd,
    [in]           UINT Msg,
    [in]           WPARAM wParam,
    [in]           LPARAM lParam
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window whose window procedure is to receive the message. The following values have special meanings.

Value	Meaning
HWND_BROADCAST ((HWND)0xffff)	The message is posted to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows. The message is not posted to child windows.
NULL	The function behaves like a call to PostThreadMessage with the <i>dwThreadId</i> parameter set to the identifier of the current thread.

Starting with Windows Vista, message posting is subject to UIPI. The thread of a process can post messages only to message queues of threads in processes of lesser or equal

integrity level.

[in] `Msg`

Type: **UINT**

The message to be posted.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] `wParam`

Type: **WPARAM**

Additional message-specific information.

[in] `lParam`

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **BOOL**

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 a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

Messages in a message queue are retrieved by calls to the [GetMessage](#) or [PeekMessage](#) function.

Applications that need to communicate using **HWND_BROADCAST** should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= [WM_USER](#)) to another process, you

must do custom marshalling.

If you send a message in the range below [WM_USER](#) to the asynchronous message functions ([PostMessage](#), [SendNotifyMessage](#), and [SendMessageCallback](#)), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Do not post the [WM_QUIT](#) message using [PostMessage](#); use the [PostQuitMessage](#) function.

An accessibility application can use [PostMessage](#) to post [WM_APPCOMMAND](#) messages to the shell to launch applications. This functionality is not guaranteed to work for other types of applications.

There is a limit of 10,000 posted messages per message queue. This limit should be sufficiently large. If your application exceeds the limit, it should be redesigned to avoid consuming so many system resources. To adjust this limit, modify the following registry key.

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          Windows  
            USERPostMessageLimit
```

If the function fails, call [GetLastError](#) to get extended error information. [GetLastError](#) returns [ERROR_NOT_ENOUGH_QUOTA](#) when the limit is hit.

The minimum acceptable value is 4000.

Examples

The following example shows how to post a private window message using the [PostMessage](#) function. Assume you defined a private window message called [WM_COMPLETE](#):

```
C++  
  
#define WM_COMPLETE (WM_USER + 0)
```

You can post a message to the message queue associated with the thread that created the specified window as shown below:

C++

```
WaitForSingleObject (pparams->hEvent, INFINITE) ;  
lTime = GetCurrentTime () ;  
PostMessage (pparams->hwnd, WM_COMPLETE, 0, lTime);
```

For more examples, see [Initiating a Data Link](#).

 **Note**

The winuser.h header defines PostMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetMessage](#)

Messages and Message Queues

[PeekMessage](#)

[PostQuitMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

PostQuitMessage function (winuser.h)

Article 10/13/2021

Indicates to the system that a thread has made a request to terminate (quit). It is typically used in response to a [WM_DESTROY](#) message.

Syntax

C++

```
void PostQuitMessage(  
    [in] int nExitCode  
);
```

Parameters

[in] nExitCode

Type: int

The application exit code. This value is used as the *wParam* parameter of the [WM_QUIT](#) message.

Return value

None

Remarks

The **PostQuitMessage** function posts a [WM_QUIT](#) message to the thread's message queue and returns immediately; the function simply indicates to the system that the thread is requesting to quit at some time in the future.

When the thread retrieves the [WM_QUIT](#) message from its message queue, it should exit its message loop and return control to the system. The exit value returned to the system must be the *wParam* parameter of the [WM_QUIT](#) message.

Examples

For an example, see [Posting a Message](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetMessage](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[PostMessage](#)

Reference

[WM_DESTROY](#)

[WM_QUIT](#)

Feedback

Was this page helpful?

 Yes

 No

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

PostThreadMessageA function (winuser.h)

Article02/09/2023

Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message.

Syntax

C++

```
BOOL PostThreadMessageA(
    [in] DWORD idThread,
    [in] UINT Msg,
    [in] WPARAM wParam,
    [in] LPARAM lParam
);
```

Parameters

[in] idThread

Type: **DWORD**

The identifier of the thread to which the message is to be posted.

The function fails if the specified thread does not have a message queue. The system creates a thread's message queue when the thread makes its first call to one of the User or GDI functions. For more information, see the Remarks section.

Message posting is subject to UIPI. The thread of a process can post messages only to posted-message queues of threads in processes of lesser or equal integrity level.

This thread must have the **SE_TCB_NAME** privilege to post a message to a thread that belongs to a process with the same locally unique identifier (LUID) but is in a different desktop. Otherwise, the function fails and returns **ERROR_INVALID_THREAD_ID**.

This thread must either belong to the same desktop as the calling thread or to a process with the same LUID. Otherwise, the function fails and returns **ERROR_INVALID_THREAD_ID**.

[in] Msg

Type: **UINT**

The type of message to be posted.

[in] **wParam**

Type: **WPARAM**

Additional message-specific information.

[in] **lParam**

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **BOOL**

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](#) returns **ERROR_INVALID_THREAD_ID** if *idThread* is not a valid thread identifier, or if the thread specified by *idThread* does not have a message queue. [GetLastError](#) returns **ERROR_NOT_ENOUGH_QUOTA** when the message limit is hit.

Remarks

When a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

The thread to which the message is posted must have created a message queue, or else the call to [PostThreadMessage](#) fails. Use the following method to handle this situation.

- Create an event object, then create the thread.
- Use the [WaitForSingleObject](#) function to wait for the event to be set to the signaled state before calling [PostThreadMessage](#).
- In the thread to which the message will be posted, call [PeekMessage](#) as shown here to force the system to create the message queue.

`PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE)`

- Set the event, to indicate that the thread is ready to receive posted messages.

The thread to which the message is posted retrieves the message by calling the [GetMessage](#) or [PeekMessage](#) function. The `hwnd` member of the returned [MSG](#) structure is `NULL`.

Messages posted by [PostThreadMessage](#) are not associated with a window. As a general rule, messages that are not associated with a window cannot be dispatched by the [DispatchMessage](#) function. Therefore, if the recipient thread is in a modal loop (as used by [MessageBox](#) or [DialogBox](#)), the messages will be lost. To intercept thread messages while in a modal loop, use a thread-specific hook.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

There is a limit of 10,000 posted messages per message queue. This limit should be sufficiently large. If your application exceeds the limit, it should be redesigned to avoid consuming so many system resources. To adjust this limit, modify the following registry key.

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          Windows  
            USERPostMessageLimit
```

The minimum acceptable value is 4000.

 **Note**

The `winuser.h` header defines `PostThreadMessage` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetCurrentThreadId](#)

[GetMessage](#)

[GetWindowThreadProcessId](#)

[MSG](#)

[Messages and Message Queues](#)

Other Resources

[PeekMessage](#)

[PostMessage](#)

Reference

[Sleep](#)

[WaitForSingleObject](#)

Feedback

Was this page helpful?

 Yes

 No

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

PostThreadMessageW function (winuser.h)

Article02/09/2023

Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message.

Syntax

C++

```
BOOL PostThreadMessageW(
    [in] DWORD idThread,
    [in] UINT Msg,
    [in] WPARAM wParam,
    [in] LPARAM lParam
);
```

Parameters

[in] idThread

Type: **DWORD**

The identifier of the thread to which the message is to be posted.

The function fails if the specified thread does not have a message queue. The system creates a thread's message queue when the thread makes its first call to one of the User or GDI functions. For more information, see the Remarks section.

Message posting is subject to UIPI. The thread of a process can post messages only to posted-message queues of threads in processes of lesser or equal integrity level.

This thread must have the **SE_TCB_NAME** privilege to post a message to a thread that belongs to a process with the same locally unique identifier (LUID) but is in a different desktop. Otherwise, the function fails and returns **ERROR_INVALID_THREAD_ID**.

This thread must either belong to the same desktop as the calling thread or to a process with the same LUID. Otherwise, the function fails and returns **ERROR_INVALID_THREAD_ID**.

[in] Msg

Type: **UINT**

The type of message to be posted.

[in] **wParam**

Type: **WPARAM**

Additional message-specific information.

[in] **lParam**

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **BOOL**

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](#) returns **ERROR_INVALID_THREAD_ID** if *idThread* is not a valid thread identifier, or if the thread specified by *idThread* does not have a message queue. [GetLastError](#) returns **ERROR_NOT_ENOUGH_QUOTA** when the message limit is hit.

Remarks

When a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

The thread to which the message is posted must have created a message queue, or else the call to [PostThreadMessage](#) fails. Use the following method to handle this situation.

- Create an event object, then create the thread.
- Use the [WaitForSingleObject](#) function to wait for the event to be set to the signaled state before calling [PostThreadMessage](#).
- In the thread to which the message will be posted, call [PeekMessage](#) as shown here to force the system to create the message queue.

`PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE)`

- Set the event, to indicate that the thread is ready to receive posted messages.

The thread to which the message is posted retrieves the message by calling the [GetMessage](#) or [PeekMessage](#) function. The `hwnd` member of the returned [MSG](#) structure is `NULL`.

Messages posted by [PostThreadMessage](#) are not associated with a window. As a general rule, messages that are not associated with a window cannot be dispatched by the [DispatchMessage](#) function. Therefore, if the recipient thread is in a modal loop (as used by [MessageBox](#) or [DialogBox](#)), the messages will be lost. To intercept thread messages while in a modal loop, use a thread-specific hook.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

There is a limit of 10,000 posted messages per message queue. This limit should be sufficiently large. If your application exceeds the limit, it should be redesigned to avoid consuming so many system resources. To adjust this limit, modify the following registry key.

```
HKEY_LOCAL_MACHINE  
  SOFTWARE  
    Microsoft  
      Windows NT  
        CurrentVersion  
          Windows  
            USERPostMessageLimit
```

The minimum acceptable value is 4000.

 **Note**

The `winuser.h` header defines `PostThreadMessage` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetCurrentThreadId](#)

[GetMessage](#)

[GetWindowThreadProcessId](#)

[MSG](#)

[Messages and Message Queues](#)

Other Resources

[PeekMessage](#)

[PostMessage](#)

Reference

[Sleep](#)

[WaitForSingleObject](#)

Feedback

Was this page helpful?

 Yes

 No

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

PROOPENUMPROCA callback function (winuser.h)

Article 07/27/2022

An application-defined callback function used with the [EnumProps](#) function. The function receives property entries from a window's property list. The **PROOPENUMPROC** type defines a pointer to this callback function. *PropEnumProc* is a placeholder for the application-defined function name.

Syntax

C++

```
PROOPENUMPROCA Propenumproca;

BOOL Propenumproca(
    HWND unnamedParam1,
    LPCSTR unnamedParam2,
    HANDLE unnamedParam3
)
{...}
```

Parameters

unnamedParam1

Type: **HWND**

A handle to the window whose property list is being enumerated.

unnamedParam2

Type: **LPCTSTR**

The string component of a property list entry. This is the string that was specified, along with a data handle, when the property was added to the window's property list via a call to the [SetProp](#) function.

unnamedParam3

Type: **HANDLE**

A handle to the data. This handle is the data component of a property list entry.

Return value

Type: **BOOL**

Return **TRUE** to continue the property list enumeration.

Return **FALSE** to stop the property list enumeration.

Remarks

The following restrictions apply to this callback function:

- The callback function can call the [RemoveProp](#) function. However, [RemoveProp](#) can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

ⓘ Note

The winuser.h header defines PROOPENUMPROC 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	winuser.h (include Windows.h)

See also

[Conceptual](#)

[EnumProps](#)

[Reference](#)

[RemoveProp](#)

[SetProp](#)

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

PROOPENUMPROCEXA callback function (winuser.h)

Article 07/27/2022

Application-defined callback function used with the [EnumPropsEx](#) function. The function receives property entries from a window's property list. The PROOPENUMPROCEX type defines a pointer to this callback function. **PropEnumProcEx** is a placeholder for the application-defined function name.

Syntax

C++

```
PROOPENUMPROCEXA Propenumprocex;  
  
BOOL Propenumprocex(  
    HWND unnamedParam1,  
    LPSTR unnamedParam2,  
    HANDLE unnamedParam3,  
    ULONG_PTR unnamedParam4  
)  
{...}
```

Parameters

unnamedParam1

Type: **HWND**

A handle to the window whose property list is being enumerated.

unnamedParam2

Type: **LPTSTR**

The string component of a property list entry. This is the string that was specified, along with a data handle, when the property was added to the window's property list via a call to the [SetProp](#) function.

unnamedParam3

Type: **HANDLE**

A handle to the data. This handle is the data component of a property list entry.

unnamedParam4

Type: **ULONG_PTR**

Application-defined data. This is the value that was specified as the *lParam* parameter of the call to [EnumPropsEx](#) that initiated the enumeration.

Return value

Type: **BOOL**

Return **TRUE** to continue the property list enumeration.

Return **FALSE** to stop the property list enumeration.

Remarks

The following restrictions apply to this callback function:

- The callback function can call the [RemoveProp](#) function. However, [RemoveProp](#) can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

Note

The winuser.h header defines PROOPENUMPROCEX 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	winuser.h (include Windows.h)

See also

Conceptual

[EnumPropsEx](#)

Reference

[RemoveProp](#)

[SetProp](#)

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

PROOPENUMPROCEXW callback function (winuser.h)

Article 07/27/2022

Application-defined callback function used with the [EnumPropsEx](#) function. The function receives property entries from a window's property list. The PROOPENUMPROCEX type defines a pointer to this callback function. **PropEnumProcEx** is a placeholder for the application-defined function name.

Syntax

C++

```
PROOPENUMPROCEXW Propenumprocexw;

BOOL Propenumprocexw(
    HWND unnamedParam1,
    LPWSTR unnamedParam2,
    HANDLE unnamedParam3,
    ULONG_PTR unnamedParam4
)
{...}
```

Parameters

unnamedParam1

Type: **HWND**

A handle to the window whose property list is being enumerated.

unnamedParam2

Type: **LPTSTR**

The string component of a property list entry. This is the string that was specified, along with a data handle, when the property was added to the window's property list via a call to the [SetProp](#) function.

unnamedParam3

Type: **HANDLE**

A handle to the data. This handle is the data component of a property list entry.

unnamedParam4

Type: **ULONG_PTR**

Application-defined data. This is the value that was specified as the *lParam* parameter of the call to [EnumPropsEx](#) that initiated the enumeration.

Return value

Type: **BOOL**

Return **TRUE** to continue the property list enumeration.

Return **FALSE** to stop the property list enumeration.

Remarks

The following restrictions apply to this callback function:

- The callback function can call the [RemoveProp](#) function. However, [RemoveProp](#) can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

Note

The winuser.h header defines PROOPENUMPROCEX 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	winuser.h (include Windows.h)

See also

Conceptual

[EnumPropsEx](#)

Reference

[RemoveProp](#)

[SetProp](#)

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

PROOPENUMPROCW callback function (winuser.h)

Article 07/27/2022

An application-defined callback function used with the [EnumProps](#) function. The function receives property entries from a window's property list. The **PROOPENUMPROC** type defines a pointer to this callback function. *PropEnumProc* is a placeholder for the application-defined function name.

Syntax

C++

```
PROOPENUMPROCW Propenumprocw;

BOOL Propenumprocw(
    HWND unnamedParam1,
    LPCWSTR unnamedParam2,
    HANDLE unnamedParam3
)
{...}
```

Parameters

unnamedParam1

Type: **HWND**

A handle to the window whose property list is being enumerated.

unnamedParam2

Type: **LPCTSTR**

The string component of a property list entry. This is the string that was specified, along with a data handle, when the property was added to the window's property list via a call to the [SetProp](#) function.

unnamedParam3

Type: **HANDLE**

A handle to the data. This handle is the data component of a property list entry.

Return value

Type: **BOOL**

Return **TRUE** to continue the property list enumeration.

Return **FALSE** to stop the property list enumeration.

Remarks

The following restrictions apply to this callback function:

- The callback function can call the [RemoveProp](#) function. However, [RemoveProp](#) can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

ⓘ Note

The winuser.h header defines PROOPENUMPROC 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	winuser.h (include Windows.h)

See also

[Conceptual](#)

[EnumProps](#)

[Reference](#)

[RemoveProp](#)

[SetProp](#)

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

RealChildWindowFromPoint function (winuser.h)

Article11/19/2022

Retrieves a handle to the child window at the specified point. The search is restricted to immediate child windows; grandchildren and deeper descendant windows are not searched.

Syntax

C++

```
HWND RealChildWindowFromPoint(
    [in] HWND hwndParent,
    [in] POINT ptParentClientCoords
);
```

Parameters

[in] hwndParent

Type: **HWND**

A handle to the window whose child is to be retrieved.

[in] ptParentClientCoords

Type: **POINT**

A **POINT** structure that defines the client coordinates of the point to be checked.

Return value

Type: **HWND**

The return value is a handle to the child window that contains the specified point.

Remarks

RealChildWindowFromPoint treats **HTTRANSPARENT** areas of a standard control differently from other areas of the control; it returns the child window behind a transparent part of a control. In contrast, [ChildWindowFromPoint](#) treats **HTTRANSPARENT** areas of a control the same as other areas. For example, if the point is in a transparent area of a groupbox, **RealChildWindowFromPoint** returns the child window behind a groupbox, whereas **ChildWindowFromPoint** returns the groupbox. However, both APIs return a static field, even though it, too, returns **HTTRANSPARENT**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[ChildWindowFromPoint](#)

[Conceptual](#)

[Other Resources](#)

[POINT](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

RealGetWindowClassA function (winuser.h)

Article 07/27/2022

Retrieves a string that specifies the window type.

Syntax

C++

```
UINT RealGetWindowClassA(
    [in]  HWND  hwnd,
    [out] LPSTR ptszClassName,
    [in]  UINT   cchClassNameMax
);
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the window whose type will be retrieved.

[out] `ptszClassName`

Type: **LPTSTR**

A pointer to a string that receives the window type.

[in] `cchClassNameMax`

Type: **UINT**

The length, in characters, of the buffer pointed to by the *pszType* parameter.

Return value

Type: **UINT**

If the function succeeds, the return value is the number of characters copied to the specified buffer.

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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

RealGetWindowClassW function (winuser.h)

Article 02/09/2023

Retrieves a string that specifies the window type.

Syntax

C++

```
UINT RealGetWindowClassW(
    [in] HWND     hwnd,
    [out] LPWSTR  ptszClassName,
    [in]  UINT    cchClassNameMax
);
```

Parameters

[in] `hwnd`

Type: **HWND**

A handle to the window whose type will be retrieved.

[out] `ptszClassName`

Type: **LPTSTR**

A pointer to a string that receives the window type.

[in] `cchClassNameMax`

Type: **UINT**

The length, in characters, of the buffer pointed to by the *pszType* parameter.

Return value

Type: **UINT**

If the function succeeds, the return value is the number of characters copied to the specified buffer.

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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-1 (introduced in Windows 8.1)

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterClassA function (winuser.h)

Article02/09/2023

Registers a window class for subsequent use in calls to the [CreateWindow](#) or [CreateWindowEx](#) function.

Note The [RegisterClass](#) function has been superseded by the [RegisterClassEx](#) function. You can still use [RegisterClass](#), however, if you do not need to set the class small icon.

Syntax

C++

```
ATOM RegisterClassA(  
    [in] const WNDCLASSA *lpWndClass  
)
```

Parameters

[in] lpWndClass

Type: **const WNDCLASS***

A pointer to a [WNDCLASS](#) structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Return value

Type: **ATOM**

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. This atom can only be used by the [CreateWindow](#), [CreateWindowEx](#), [GetClassInfo](#), [GetClassInfoEx](#), [FindWindow](#), [FindWindowEx](#), and [UnregisterClass](#) functions and the [IActiveIMMMap::FilterClientWindows](#) method.

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

Remarks

If you register the window class by using **RegisterClassA**, the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using **RegisterClassW**, the application requests that the system pass text parameters of messages as Unicode. The [IsWindowUnicode](#) function enables applications to query the nature of each window. For more information on ANSI and Unicode functions, see [Conventions for Function Prototypes](#).

All window classes that an application registers are unregistered when it terminates.

No window classes registered by a DLL are unregistered when the DLL is unloaded. A DLL must explicitly unregister its classes when it is unloaded.

Examples

For an example, see [Associating a Window Procedure with a Window Class](#).

ⓘ Note

The winuser.h header defines RegisterClass 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)
---------	--

See also

Conceptual

[CreateWindow](#)

[CreateWindowEx](#)

[FindWindow](#)

[FindWindowEx](#)

[GetClassInfo](#)

[GetClassInfoEx](#)

[GetClassName](#)

Reference

[RegisterClassEx](#)

[UnregisterClass](#)

[WNDCLASS](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterClassExA function (winuser.h)

Article 02/09/2023

Registers a window class for subsequent use in calls to the [CreateWindow](#) or [CreateWindowEx](#) function.

Syntax

C++

```
ATOM RegisterClassExA(  
    [in] const WNDCLASSEX *unnamedParam1  
);
```

Parameters

[in] unnamedParam1

Type: **const WNDCLASSEX***

A pointer to a [WNDCLASSEX](#) structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Return value

Type: **ATOM**

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. This atom can only be used by the [CreateWindow](#), [CreateWindowEx](#), [GetClassInfo](#), [GetClassInfoEx](#), [FindWindow](#), [FindWindowEx](#), and [UnregisterClass](#) functions and the [IActiveIMMMap::FilterClientWindows](#) method.

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

Remarks

If you register the window class by using [RegisterClassExA](#), the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using [RegisterClassExW](#),

the application requests that the system pass text parameters of messages as Unicode. The [IsWindowUnicode](#) function enables applications to query the nature of each window. For more information on ANSI and Unicode functions, see [Conventions for Function Prototypes](#).

All window classes that an application registers are unregistered when it terminates.

No window classes registered by a DLL are unregistered when the DLL is unloaded. A DLL must explicitly unregister its classes when it is unloaded.

Examples

For an example, see [Using Window Classes](#).

ⓘ Note

The winuser.h header defines RegisterClassEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[CreateWindow](#)

[CreateWindowEx](#)

[FindWindow](#)

[FindWindowEx](#)

[GetClassInfo](#)

[GetClassInfoEx](#)

[GetClassName](#)

Reference

[RegisterClass](#)

[UnregisterClass](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterClassExW function (winuser.h)

Article 02/09/2023

Registers a window class for subsequent use in calls to the [CreateWindow](#) or [CreateWindowEx](#) function.

Syntax

C++

```
ATOM RegisterClassExW(
    [in] const WNDCLASSEXW *unnamedParam1
);
```

Parameters

[in] unnamedParam1

Type: **const WNDCLASSEX***

A pointer to a [WNDCLASSEX](#) structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Return value

Type: **ATOM**

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. This atom can only be used by the [CreateWindow](#), [CreateWindowEx](#), [GetClassInfo](#), [GetClassInfoEx](#), [FindWindow](#), [FindWindowEx](#), and [UnregisterClass](#) functions and the [IActiveIMMMap::FilterClientWindows](#) method.

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

Remarks

If you register the window class by using [RegisterClassExA](#), the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using [RegisterClassExW](#),

the application requests that the system pass text parameters of messages as Unicode. The [IsWindowUnicode](#) function enables applications to query the nature of each window. For more information on ANSI and Unicode functions, see [Conventions for Function Prototypes](#).

All window classes that an application registers are unregistered when it terminates.

No window classes registered by a DLL are unregistered when the DLL is unloaded. A DLL must explicitly unregister its classes when it is unloaded.

Examples

For an example, see [Using Window Classes](#).

ⓘ Note

The winuser.h header defines RegisterClassEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[CreateWindow](#)

[CreateWindowEx](#)

[FindWindow](#)

[FindWindowEx](#)

[GetClassInfo](#)

[GetClassInfoEx](#)

[GetClassName](#)

Reference

[RegisterClass](#)

[UnregisterClass](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterClassW function (winuser.h)

Article 02/09/2023

Registers a window class for subsequent use in calls to the [CreateWindow](#) or [CreateWindowEx](#) function.

Note The [RegisterClass](#) function has been superseded by the [RegisterClassEx](#) function. You can still use [RegisterClass](#), however, if you do not need to set the class small icon.

Syntax

C++

```
ATOM RegisterClassW(
    [in] const WNDCLASSW *lpWndClass
);
```

Parameters

[in] lpWndClass

Type: **const WNDCLASS***

A pointer to a [WNDCLASS](#) structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Return value

Type: **ATOM**

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. This atom can only be used by the [CreateWindow](#), [CreateWindowEx](#), [GetClassInfo](#), [GetClassInfoEx](#), [FindWindow](#), [FindWindowEx](#), and [UnregisterClass](#) functions and the [IActiveIMMMap::FilterClientWindows](#) method.

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

Remarks

If you register the window class by using **RegisterClassA**, the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using **RegisterClassW**, the application requests that the system pass text parameters of messages as Unicode. The [IsWindowUnicode](#) function enables applications to query the nature of each window. For more information on ANSI and Unicode functions, see [Conventions for Function Prototypes](#).

All window classes that an application registers are unregistered when it terminates.

No window classes registered by a DLL are unregistered when the DLL is unloaded. A DLL must explicitly unregister its classes when it is unloaded.

Examples

For an example, see [Associating a Window Procedure with a Window Class](#).

ⓘ Note

The winuser.h header defines RegisterClass 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)
---------	--

See also

Conceptual

[CreateWindow](#)

[CreateWindowEx](#)

[FindWindow](#)

[FindWindowEx](#)

[GetClassInfo](#)

[GetClassInfoEx](#)

[GetClassName](#)

Reference

[RegisterClassEx](#)

[UnregisterClass](#)

[WNDCLASS](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterShellHookWindow function (winuser.h)

Article 10/13/2021

[This function is not intended for general use. It may be altered or unavailable in subsequent versions of Windows.]

Registers a specified Shell window to receive certain messages for events or notifications that are useful to Shell applications.

The event messages received are only those sent to the Shell window associated with the specified window's desktop. Many of the messages are the same as those that can be received after calling the [SetWindowsHookEx](#) function and specifying **WH_SHELL** for the hook type. The difference with **RegisterShellHookWindow** is that the messages are received through the specified window's [WindowProc](#) and not through a call back procedure.

Syntax

```
C++  
  
BOOL RegisterShellHookWindow(  
    [in] HWND hwnd  
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the window to register for Shell hook messages.

Return value

Type: **BOOL**

TRUE if the function succeeds; otherwise, **FALSE**.

Remarks

As with normal window messages, the second parameter of the window procedure identifies the message as a **WM_SHELLHOOKMESSAGE**. However, for these Shell hook messages, the message value is not a pre-defined constant like other message IDs such as [WM_COMMAND](#). The value must be obtained dynamically using a call to [RegisterWindowMessage](#) as shown here:

```
RegisterWindowMessage(TEXT("SHELLHOOK"));
```

This precludes handling these messages using a traditional switch statement which requires ID values that are known at compile time. For handling Shell hook messages, the normal practice is to code an If statement in the default section of your switch statement and then handle the message if the value of the message ID is the same as the value obtained from the [RegisterWindowMessage](#) call.

The following table describes the *wParam* and *lParam* parameter values passed to the window procedure for the Shell hook messages.

wParam	lParam
HSHELL_GETMINRECT	A pointer to a SHELLHOOKINFO structure.
HSHELL_WINDOWACTIVATED	A handle to the activated window.
HSHELL_RUDEAPPACTIVATED	A handle to the activated window.
HSHELL_WINDOWREPLACING	A handle to the window replacing the top-level window.
HSHELL_WINDOWREPLACED	A handle to the window being replaced.
HSHELL_WINDOWCREATED	A handle to the window being created.
HSHELL_WINDOWDESTROYED	A handle to the top-level window being destroyed.
HSHELL_ACTIVATESHELLWINDOW	Not used.
HSHELL_TASKMAN	Can be ignored.
HSHELL_REDRAW	A handle to the window that needs to be redrawn.
HSHELL_FLASH	A handle to the window that needs to be flashed.
HSHELL_ENDTASK	A handle to the window that should be forced to exit.
HSHELL_APPCOMMAND	The APPCOMMAND which has been unhandled by the application or other hooks. See WM_APPCOMMAND and use the GET_APPCOMMAND_LPARAM macro to retrieve this parameter.

HSHELL_MONITORCHANGED

A handle to the window that moved to a different monitor.

This function was not included in the SDK headers and libraries until Windows XP with Service Pack 1 (SP1) and Windows Server 2003. If you do not have a header file and import library for this function, you can call the function using [LoadLibrary](#) and [GetProcAddress](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[DeregisterShellHookWindow](#)

Other Resources

Reference

[SetWindowsHookEx](#)

[ShellProc](#)

[Using Messages and Message Queues](#)

[WinEvents](#)

[WindowProc](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterWindowMessageA function (winuser.h)

Article02/09/2023

Defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages.

Syntax

C++

```
UINT RegisterWindowMessageA(  
    [in] LPCSTR lpString  
);
```

Parameters

[in] lpString

Type: LPCTSTR

The message to be registered.

Return value

Type: UINT

If the message is successfully registered, the return value is a message identifier in the range 0xC000 through 0xFFFF.

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

Remarks

The **RegisterWindowMessage** function is typically used to register messages for communicating between two cooperating applications.

If two different applications register the same message string, the applications return the same message value. The message remains registered until the session ends.

Only use **RegisterWindowMessage** when more than one application must process the same message. For sending private messages within a window class, an application can use any integer in the range [WM_USER](#) through 0x7FFF. (Messages in this range are private to a window class, not to an application. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** may use values in this range.)

Examples

For an example, see [Finding Text](#).

ⓘ Note

The winuser.h header defines RegisterWindowMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

Messages and Message Queues

[PostMessage](#)

[Reference](#)

[SendMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

RegisterWindowMessageW function (winuser.h)

Article02/09/2023

Defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages.

Syntax

C++

```
UINT RegisterWindowMessageW(
    [in] LPCWSTR lpString
);
```

Parameters

[in] lpString

Type: LPCTSTR

The message to be registered.

Return value

Type: UINT

If the message is successfully registered, the return value is a message identifier in the range 0xC000 through 0xFFFF.

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

Remarks

The **RegisterWindowMessage** function is typically used to register messages for communicating between two cooperating applications.

If two different applications register the same message string, the applications return the same message value. The message remains registered until the session ends.

Only use **RegisterWindowMessage** when more than one application must process the same message. For sending private messages within a window class, an application can use any integer in the range [WM_USER](#) through 0x7FFF. (Messages in this range are private to a window class, not to an application. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** may use values in this range.)

Examples

For an example, see [Finding Text](#).

ⓘ Note

The winuser.h header defines RegisterWindowMessage 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

Messages and Message Queues

[PostMessage](#)

[Reference](#)

[SendMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

RemovePropA function (winuser.h)

Article 02/09/2023

Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed.

Syntax

C++

```
HANDLE RemovePropA(  
    [in] HWND     hWnd,  
    [in] LPCSTR  lpString  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be changed.

[in] lpString

Type: **LPCTSTR**

A null-terminated character string or an atom that identifies a string. If this parameter is an atom, it must have been created using the [GlobalAddAtom](#) function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

Return value

Type: **HANDLE**

The return value identifies the specified data. If the data cannot be found in the specified property list, the return value is **NULL**.

Remarks

The return value is the *hData* value that was passed to [SetProp](#); it is an application-defined value. Note, this function only destroys the association between the data and the window. If appropriate, the application must free the data handles associated with entries removed from a property list. The application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

The **RemoveProp** function returns the data handle associated with the string so that the application can free the data associated with the handle.

Starting with Windows Vista, **RemoveProp** is subject to the restrictions of User Interface Privilege Isolation (UIPI). A process can only call this function on a window belonging to a process of lesser or equal integrity level. When UIPI blocks property changes, [GetLastError](#) will return 5.

Examples

For an example, see [Deleting a Window Property](#).

 **Note**

The winuser.h header defines RemoveProp 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set

ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[AddAtom](#)

[Conceptual](#)

[GetProp](#)

[Reference](#)

[SetProp](#)

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

RemovePropW function (winuser.h)

Article 02/09/2023

Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed.

Syntax

C++

```
HANDLE RemovePropW(
    [in] HWND     hWnd,
    [in] LPCWSTR lpString
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list is to be changed.

[in] lpString

Type: **LPCTSTR**

A null-terminated character string or an atom that identifies a string. If this parameter is an atom, it must have been created using the [GlobalAddAtom](#) function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

Return value

Type: **HANDLE**

The return value identifies the specified data. If the data cannot be found in the specified property list, the return value is **NULL**.

Remarks

The return value is the *hData* value that was passed to [SetProp](#); it is an application-defined value. Note, this function only destroys the association between the data and the window. If appropriate, the application must free the data handles associated with entries removed from a property list. The application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

The **RemoveProp** function returns the data handle associated with the string so that the application can free the data associated with the handle.

Starting with Windows Vista, **RemoveProp** is subject to the restrictions of User Interface Privilege Isolation (UIPI). A process can only call this function on a window belonging to a process of lesser or equal integrity level. When UIPI blocks property changes, [GetLastError](#) will return 5.

Examples

For an example, see [Deleting a Window Property](#).

ⓘ Note

The winuser.h header defines RemoveProp 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set

ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[AddAtom](#)

[Conceptual](#)

[GetProp](#)

[Reference](#)

[SetProp](#)

[Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

ReplyMessage function (winuser.h)

Article 10/13/2021

Replies to a message sent from another thread by the [SendMessage](#) function.

Syntax

C++

```
BOOL ReplyMessage(
    [in] LRESULT lResult
);
```

Parameters

[in] lResult

Type: **LRESULT**

The result of the message processing. The possible values are based on the message sent.

Return value

Type: **BOOL**

If the calling thread was processing a message sent from another thread or process, the return value is nonzero.

If the calling thread was not processing a message sent from another thread or process, the return value is zero.

Remarks

By calling this function, the window procedure that receives the message allows the thread that called [SendMessage](#) to continue to run as though the thread receiving the message had returned control. The thread that calls the **ReplyMessage** function also continues to run.

If the message was not sent through [SendMessage](#) or if the message was sent by the same thread, [ReplyMessage](#) has no effect.

Examples

For an example, see [Sending a Message](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[InSendMessage](#)

[Messages and Message Queues](#)

Reference

[SendMessage](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

SENDASYNCPROC callback function (winuser.h)

Article04/02/2021

An application-defined callback function used with the [SendMessageCallback](#) function. The system passes the message to the callback function after passing the message to the destination window procedure. The **SENDASYNCPROC** type defines a pointer to this callback function. *SendAsyncProc* is a placeholder for the application-defined function name.

Syntax

C++

```
SENDASYNCPROC Sendsyncproc;

void Sendsyncproc(
    HWND unnamedParam1,
    UINT unnamedParam2,
    ULONG_PTR unnamedParam3,
    LRESULT unnamedParam4
)
{...}
```

Parameters

unnamedParam1

Type: **HWND**

A handle to the window whose window procedure received the message.

If the [SendMessageCallback](#) function was called with its *hwnd* parameter set to **HWND_BROADCAST**, the system calls the *SendAsyncProc* function once for each top-level window.

unnamedParam2

Type: **UINT**

The message.

unnamedParam3

Type: **ULONG_PTR**

An application-defined value sent from the [SendMessageCallback](#) function.

unnamedParam4

Type: **LRESULT**

The result of the message processing. This value depends on the message.

Return value

None

Remarks

You install a *SendAsyncProc* application-defined callback function by passing a **SENDASYNCPROC** pointer to the [SendMessageCallback](#) function.

The callback function is only called when the thread that called [SendMessageCallback](#) calls [GetMessage](#), [PeekMessage](#), or [WaitMessage](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[GetMessage](#)

[Messages and Message Queues](#)

[PeekMessage](#)

[Reference](#)

[SendMessageCallback](#)

[WaitMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessage function (winuser.h)

Article08/02/2022

Sends the specified message to a window or windows. The **SendMessage** function calls the window procedure for the specified window and does not return until the window procedure has processed the message.

To send a message and return immediately, use the [SendMessageCallback](#) or [SendNotifyMessage](#) function. To post a message to a thread's message queue and return immediately, use the [PostMessage](#) or [PostThreadMessage](#) function.

Syntax

C++

```
LRESULT SendMessage(
    [in] HWND     hWnd,
    [in] UINT     Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Message sending is subject to UIPI. The thread of a process can send messages only to message queues of threads in processes of lesser or equal integrity level.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **LRESULT**

The return value specifies the result of the message processing; it depends on the message sent.

Remarks

When a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

Applications that need to communicate using **HWND_BROADCAST** should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to [WM_USER-1](#)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

If the specified window was created by the calling thread, the window procedure is called immediately as a subroutine. If the specified window was created by a different thread, the system switches to that thread and calls the appropriate window procedure. Messages sent between threads are processed only when the receiving thread executes message retrieval code. The sending thread is blocked until the receiving thread processes the message. However, the sending thread will process incoming nonqueued messages while waiting for its message to be processed. To prevent this, use [SendMessageTimeout](#) with **SMT_BLOCK** set. For more information on nonqueued messages, see [Nonqueued Messages](#).

An accessibility application can use [SendMessage](#) to send [WM_APPCOMMAND](#) messages to the shell to launch applications. This functionality is not guaranteed to work for other types of applications.

Examples

For an example, see [Displaying Keyboard Input](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[InSendMessage](#)

[Messages and Message Queues](#)

[PostMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendDlgItemMessage](#)

[SendMessageCallback](#)

[SendMessageTimeout](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessageA function (winuser.h)

Article 02/09/2023

Sends the specified message to a window or windows. The **SendMessage** function calls the window procedure for the specified window and does not return until the window procedure has processed the message.

To send a message and return immediately, use the [SendMessageCallback](#) or [SendNotifyMessage](#) function. To post a message to a thread's message queue and return immediately, use the [PostMessage](#) or [PostThreadMessage](#) function.

Syntax

C++

```
LRESULT SendMessageA(
    [in] HWND     hWnd,
    [in] UINT     Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Message sending is subject to UIPI. The thread of a process can send messages only to message queues of threads in processes of lesser or equal integrity level.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **LRESULT**

The return value specifies the result of the message processing; it depends on the message sent.

Remarks

When a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

Applications that need to communicate using **HWND_BROADCAST** should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to [WM_USER-1](#)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

If the specified window was created by the calling thread, the window procedure is called immediately as a subroutine. If the specified window was created by a different thread, the system switches to that thread and calls the appropriate window procedure. Messages sent between threads are processed only when the receiving thread executes message retrieval code. The sending thread is blocked until the receiving thread processes the message. However, the sending thread will process incoming nonqueued messages while waiting for its message to be processed. To prevent this, use [SendMessageTimeout](#) with **SMT_BLOCK** set. For more information on nonqueued messages, see [Nonqueued Messages](#).

An accessibility application can use [SendMessage](#) to send [WM_APPCOMMAND](#) messages to the shell to launch applications. This functionality is not guaranteed to work for other types of applications.

Examples

For an example, see [Displaying Keyboard Input](#).

ⓘ Note

The winuser.h header defines `SendMessage` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[InSendMessage](#)

[Messages and Message Queues](#)

[PostMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendDlgItemMessage](#)

[SendMessageCallback](#)

[SendMessageTimeout](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessageCallbackA function (winuser.h)

Article02/09/2023

Sends the specified message to a window or windows. It calls the window procedure for the specified window and returns immediately if the window belongs to another thread. After the window procedure processes the message, the system calls the specified callback function, passing the result of the message processing and an application-defined value to the callback function.

Syntax

C++

```
BOOL SendMessageCallbackA(
    [in] HWND         hWnd,
    [in] UINT          Msg,
    [in] WPARAM        wParam,
    [in] LPARAM        lParam,
    [in] SENDASYNCPROC lpResultCallBack,
    [in] ULONG_PTR     dwData
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

[in] lpResultCallBack

Type: **SENDASYNCPROC**

A pointer to a callback function that the system calls after the window procedure processes the message. For more information, see [SendAsyncProc](#).

If *hWnd* is **HWND_BROADCAST** ((**HWND**)0xffff), the system calls the [SendAsyncProc](#) callback function once for each top-level window.

[in] dwData

Type: **ULONG_PTR**

An application-defined value to be sent to the callback function pointed to by the *lpCallBack* parameter.

Return value

Type: **BOOL**

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 target window belongs to the same thread as the caller, then the window procedure is called synchronously, and the callback function is called immediately after the window procedure returns. If the target window belongs to a different thread from the caller, then the callback function is called only when the thread that called **SendMessageCallback** also calls [GetMessage](#), [PeekMessage](#), or [WaitMessage](#).

If you send a message in the range below [WM_USER](#) to the asynchronous message functions ([PostMessage](#), [SendNotifyMessage](#), and [SendMessageCallback](#)), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Applications that need to communicate using [HWND_BROADCAST](#) should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= [WM_USER](#)) to another process, you must do custom marshalling.

 **Note**

The winuser.h header defines [SendMessageCallback](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

Messages and Message Queues

[PostMessage](#)

Reference

[RegisterWindowMessage](#)

[SendAsyncProc](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessageCallbackW function (winuser.h)

Article02/09/2023

Sends the specified message to a window or windows. It calls the window procedure for the specified window and returns immediately if the window belongs to another thread. After the window procedure processes the message, the system calls the specified callback function, passing the result of the message processing and an application-defined value to the callback function.

Syntax

C++

```
BOOL SendMessageCallbackW(
    [in] HWND         hWnd,
    [in] UINT          Msg,
    [in] WPARAM        wParam,
    [in] LPARAM        lParam,
    [in] SENDASYNCPROC lpResultCallBack,
    [in] ULONG_PTR     dwData
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

[in] lpResultCallBack

Type: **SENDASYNCPROC**

A pointer to a callback function that the system calls after the window procedure processes the message. For more information, see [SendAsyncProc](#).

If *hWnd* is **HWND_BROADCAST** ((**HWND**)0xffff), the system calls the [SendAsyncProc](#) callback function once for each top-level window.

[in] dwData

Type: **ULONG_PTR**

An application-defined value to be sent to the callback function pointed to by the *lpCallBack* parameter.

Return value

Type: **BOOL**

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 target window belongs to the same thread as the caller, then the window procedure is called synchronously, and the callback function is called immediately after the window procedure returns. If the target window belongs to a different thread from the caller, then the callback function is called only when the thread that called [SendMessageCallback](#) also calls [GetMessage](#), [PeekMessage](#), or [WaitMessage](#).

If you send a message in the range below [WM_USER](#) to the asynchronous message functions ([PostMessage](#), [SendNotifyMessage](#), and [SendMessageCallback](#)), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Applications that need to communicate using [HWND_BROADCAST](#) should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= [WM_USER](#)) to another process, you must do custom marshalling.

Note

The winuser.h header defines [SendMessageCallback](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

Messages and Message Queues

[PostMessage](#)

Reference

[RegisterWindowMessage](#)

[SendAsyncProc](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessageTimeoutA function (winuser.h)

Article02/09/2023

Sends the specified message to one or more windows.

Syntax

C++

```
LRESULT SendMessageTimeoutA(
    [in]             HWND      hWnd,
    [in]             UINT       Msg,
    [in]             WPARAM     wParam,
    [in]             LPARAM     lParam,
    [in]             UINT       fuFlags,
    [in]             UINT       uTimeout,
    [out, optional] PDWORD_PTR lpdwResult
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message.

If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows. The function does not return until each window has timed out. Therefore, the total wait time can be up to the value of *uTimeout* multiplied by the number of top-level windows.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Any additional message-specific information.

[in] **lParam**

Type: **LPARAM**

Any additional message-specific information.

[in] **fuFlags**

Type: **UINT**

The behavior of this function. This parameter can be one or more of the following values.

Value	Meaning
SMTO_ABORTIFHUNG 0x0002	The function returns without waiting for the time-out period to elapse if the receiving thread appears to not respond or "hangs."
SMTO_BLOCK 0x0001	Prevents the calling thread from processing any other requests until the function returns.
SMTO_NORMAL 0x0000	The calling thread is not prevented from processing other requests while waiting for the function to return.
SMTO_NOTIMEOUTIFNOTHUNG 0x0008	The function does not enforce the time-out period as long as the receiving thread is processing messages.
SMTO_ERRORONEXIT 0x0020	The function should return 0 if the receiving window is destroyed or its owning thread dies while the message is being processed.

[in] **uTimeout**

Type: **UINT**

The duration of the time-out period, in milliseconds. If the message is a broadcast message, each window can use the full time-out period. For example, if you specify a five second time-out period and there are three top-level windows that fail to process the message, you could have up to a 15 second delay.

[out, optional] **lpdwResult**

Type: **PDWORD_PTR**

The result of the message processing. The value of this parameter depends on the message that is specified.

Return value

Type: LRESULT

If the function succeeds, the return value is nonzero. [SendMessageTimeout](#) does not provide information about individual windows timing out if **HWND_BROADCAST** is used.

If the function fails or times out, the return value is 0. To get extended error information, call [GetLastError](#). If [GetLastError](#) returns **ERROR_TIMEOUT**, then the function timed out.

Windows 2000: If [GetLastError](#) returns 0, then the function timed out.

Remarks

The function calls the window procedure for the specified window and, if the specified window belongs to a different thread, does not return until the window procedure has processed the message or the specified time-out period has elapsed. If the window receiving the message belongs to the same queue as the current thread, the window procedure is called directly—the time-out value is ignored.

This function considers that a thread is not responding if it has not called [GetMessage](#) or a similar function within five seconds.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= [WM_USER](#)) to another process, you must do custom marshalling.

ⓘ Note

The winuser.h header defines [SendMessageTimeout](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetMessage](#)

[InSendMessage](#)

[Messages and Message Queues](#)

[PostMessage](#)

Reference

[SendDlgItemMessage](#)

[SendMessage](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessageTimeoutW function (winuser.h)

Article02/09/2023

Sends the specified message to one or more windows.

Syntax

C++

```
LRESULT SendMessageTimeoutW(
    [in]             HWND      hWnd,
    [in]             UINT       Msg,
    [in]             WPARAM     wParam,
    [in]             LPARAM     lParam,
    [in]             UINT       fuFlags,
    [in]             UINT       uTimeout,
    [out, optional] PDWORD_PTR lpdwResult
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message.

If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows. The function does not return until each window has timed out. Therefore, the total wait time can be up to the value of *uTimeout* multiplied by the number of top-level windows.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Any additional message-specific information.

[in] **lParam**

Type: **LPARAM**

Any additional message-specific information.

[in] **fuFlags**

Type: **UINT**

The behavior of this function. This parameter can be one or more of the following values.

Value	Meaning
SMTO_ABORTIFHUNG 0x0002	The function returns without waiting for the time-out period to elapse if the receiving thread appears to not respond or "hangs."
SMTO_BLOCK 0x0001	Prevents the calling thread from processing any other requests until the function returns.
SMTO_NORMAL 0x0000	The calling thread is not prevented from processing other requests while waiting for the function to return.
SMTO_NOTIMEOUTIFNOTHUNG 0x0008	The function does not enforce the time-out period as long as the receiving thread is processing messages.
SMTO_ERRORONEXIT 0x0020	The function should return 0 if the receiving window is destroyed or its owning thread dies while the message is being processed.

[in] **uTimeout**

Type: **UINT**

The duration of the time-out period, in milliseconds. If the message is a broadcast message, each window can use the full time-out period. For example, if you specify a five second time-out period and there are three top-level windows that fail to process the message, you could have up to a 15 second delay.

[out, optional] **lpdwResult**

Type: **PDWORD_PTR**

The result of the message processing. The value of this parameter depends on the message that is specified.

Return value

Type: LRESULT

If the function succeeds, the return value is nonzero. [SendMessageTimeout](#) does not provide information about individual windows timing out if **HWND_BROADCAST** is used.

If the function fails or times out, the return value is 0. To get extended error information, call [GetLastError](#). If [GetLastError](#) returns **ERROR_TIMEOUT**, then the function timed out.

Windows 2000: If [GetLastError](#) returns 0, then the function timed out.

Remarks

The function calls the window procedure for the specified window and, if the specified window belongs to a different thread, does not return until the window procedure has processed the message or the specified time-out period has elapsed. If the window receiving the message belongs to the same queue as the current thread, the window procedure is called directly—the time-out value is ignored.

This function considers that a thread is not responding if it has not called [GetMessage](#) or a similar function within five seconds.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those >= [WM_USER](#)) to another process, you must do custom marshalling.

ⓘ Note

The winuser.h header defines [SendMessageTimeout](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetMessage](#)

[InSendMessage](#)

[Messages and Message Queues](#)

[PostMessage](#)

Reference

[SendDlgItemMessage](#)

[SendMessage](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendMessageW function (winuser.h)

Article 02/09/2023

Sends the specified message to a window or windows. The **SendMessage** function calls the window procedure for the specified window and does not return until the window procedure has processed the message.

To send a message and return immediately, use the [SendMessageCallback](#) or [SendNotifyMessage](#) function. To post a message to a thread's message queue and return immediately, use the [PostMessage](#) or [PostThreadMessage](#) function.

Syntax

C++

```
LRESULT SendMessageW(
    [in] HWND     hWnd,
    [in] UINT     Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Message sending is subject to UIPI. The thread of a process can send messages only to message queues of threads in processes of lesser or equal integrity level.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **LRESULT**

The return value specifies the result of the message processing; it depends on the message sent.

Remarks

When a message is blocked by UIPI the last error, retrieved with [GetLastError](#), is set to 5 (access denied).

Applications that need to communicate using **HWND_BROADCAST** should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to [WM_USER-1](#)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

If the specified window was created by the calling thread, the window procedure is called immediately as a subroutine. If the specified window was created by a different thread, the system switches to that thread and calls the appropriate window procedure. Messages sent between threads are processed only when the receiving thread executes message retrieval code. The sending thread is blocked until the receiving thread processes the message. However, the sending thread will process incoming nonqueued messages while waiting for its message to be processed. To prevent this, use [SendMessageTimeout](#) with **SMT_BLOCK** set. For more information on nonqueued messages, see [Nonqueued Messages](#).

An accessibility application can use [SendMessage](#) to send [WM_APPCOMMAND](#) messages to the shell to launch applications. This functionality is not guaranteed to work for other types of applications.

Examples

For an example, see [Displaying Keyboard Input](#).

ⓘ Note

The winuser.h header defines `SendMessage` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[InSendMessage](#)

[Messages and Message Queues](#)

[PostMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendDlgItemMessage](#)

[SendMessageCallback](#)

[SendMessageTimeout](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendNotifyMessageA function (winuser.h)

Article02/09/2023

Sends the specified message to a window or windows. If the window was created by the calling thread, **SendNotifyMessage** calls the window procedure for the window and does not return until the window procedure has processed the message. If the window was created by a different thread, **SendNotifyMessage** passes the message to the window procedure and returns immediately; it does not wait for the window procedure to finish processing the message.

Syntax

C++

```
BOOL SendNotifyMessageA(
    [in] HWND hWnd,
    [in] UINT Msg,
    [in] WPARAM wParam,
    [in] LPARAM lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **BOOL**

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 you send a message in the range below [WM_USER](#) to the asynchronous message functions ([PostMessage](#), [SendNotifyMessage](#), and [SendMessageCallback](#)), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Applications that need to communicate using [HWND_BROADCAST](#) should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

Note

The winuser.h header defines [SendNotifyMessage](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-3 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[Messages and Message Queues](#)

[PostMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendMessage](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SendNotifyMessageW function (winuser.h)

Article02/09/2023

Sends the specified message to a window or windows. If the window was created by the calling thread, **SendNotifyMessage** calls the window procedure for the window and does not return until the window procedure has processed the message. If the window was created by a different thread, **SendNotifyMessage** passes the message to the window procedure and returns immediately; it does not wait for the window procedure to finish processing the message.

Syntax

C++

```
BOOL SendNotifyMessageW(
    [in] HWND     hWnd,
    [in] UINT     Msg,
    [in] WPARAM   wParam,
    [in] LPARAM   lParam
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose window procedure will receive the message. If this parameter is **HWND_BROADCAST** ((**HWND**)0xffff), the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

[in] Msg

Type: **UINT**

The message to be sent.

For lists of the system-provided messages, see [System-Defined Messages](#).

[in] wParam

Type: **WPARAM**

Additional message-specific information.

[in] lParam

Type: **LPARAM**

Additional message-specific information.

Return value

Type: **BOOL**

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 you send a message in the range below [WM_USER](#) to the asynchronous message functions ([PostMessage](#), [SendNotifyMessage](#), and [SendMessageCallback](#)), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Applications that need to communicate using [HWND_BROADCAST](#) should use the [RegisterWindowMessage](#) function to obtain a unique message for inter-application communication.

The system only does marshalling for system messages (those in the range 0 to ([WM_USER](#)-1)). To send other messages (those \geq [WM_USER](#)) to another process, you must do custom marshalling.

Note

The winuser.h header defines [SendNotifyMessage](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-3 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[Messages and Message Queues](#)

[PostMessage](#)

[PostThreadMessage](#)

Reference

[RegisterWindowMessage](#)

[SendMessage](#)

[SendMessageCallback](#)

[SendNotifyMessage](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetAdditionalForegroundBoostProcesses function (Winuser.h)

Article 05/24/2022

⚠️ Warning

SetAdditionalForegroundBoostProcesses is a **limited access feature**. Contact foregroundboostprocs@microsoft.com for more information.

SetAdditionalForegroundBoostProcesses is a performance assist API to help applications with a multi-process application model where multiple processes contribute to a foreground experience, either as data or rendering. Examples include browsers (with the browser manager or frame, tabs, plugins, etc. hosted in different processes) and IDEs (which spawn processes for compilation and other tasks).

Applications can use this API to provide a foreground priority boost to worker processes that help support the main application. Such applications can have a uniform priority boost applied to all of their constituent processes when the application's top level window is in the foreground.

Syntax

C++

```
BOOL SetAdditionalForegroundBoostProcesses(
    HWND topLevelWindow,
    DWORD processHandleCount,
    HANDLE *processHandleArray
);
```

Parameters

`topLevelWindow`

A handle to the top level window (HWND) of the application.

`processHandleCount`

The number of process handles in `processHandleArray`. This function can be called at a single time with a maximum of 32 handles. Set this parameter to **0** along with setting

processHandleArray to **NULL** to clear a prior boost configuration.

processHandleArray

A group of process handles to be foreground boosted or de-boosted. Set this parameter to **NULL** along with setting **processHandleCount** to **0** to clear a prior boost configuration.

Return value

Returns **TRUE** if the call succeeds in boosting the application, **FALSE** otherwise.

SetAdditionalForegroundBoostProcesses sets the last error code, so the application can call [GetLastError\(\)](#) to obtain extended information if the call failed (for example, **ERROR_INVALID_PARAMETER**, **ERROR_NOT_ENOUGH_MEMORY**, or **ERROR_ACCESS_DENIED**).

Remarks

This function takes a group of process handles that all get foreground boosted or de-boosted when the passed-in top level HWND moves to the foreground or background respectively. Whenever the passed-in top level HWND becomes the foreground window, a foreground boost will also be applied to the processes passed in the handle array. A similar de-boost happens when the top level HWND moves to the background.

The top level HWND passed to this function must be owned by the calling process. The calling process should have the **PROCESS_SET_INFORMATION** access right on the process handles in the **processHandleArray** - in other words, you must have full control of every window in your process. If some external component injects a window that takes foreground, or if a dialog box appears, then you lose your boost.

If you have two top level windows, you need to call this function for each one.

If the passed-in top level HWND is already in the foreground when **SetAdditionalForegroundBoostProcesses** is called, all of the processes in the **processHandleArray** are immediately boosted.

A process whose handle is in the **processHandleArray** will get a foreground boost only when the top level HWND becomes the foreground window.

Additional foreground boost is applied only when:

1. The foreground window changes, or

2. If this function is called while the window is in the foreground and the new list has the process handle, or the list does not include the process handle while it was previously included.

When the process owning the top level HWND exits or terminates, the additional boosting relationship is torn down and secondary processes do not receive any additional foreground boosting.

The primary process's top level HWND will continue to hold references to secondary processes until either the primary process's top level HWND clears its grouped boost state, or the HWND is destroyed.

Example

In this simple scenario, the application sets up its foreground process boost configuration when the top level window is created. When WM_CREATE is handled, the function is called with handles in the *lParam* and the count of handles in the *wParam*. These processes will get foreground or background priority boosted as *m_AppWindow* moves in and out of being the foreground window. If the *m_AppWindow* is the foreground window when the function is called, the processes will also get an immediate foreground priority boost.

C++

```
case WM_CREATE:  
    //  
    // Configure the passed in worker processes (handles) in lParam, to get  
    // foreground priority boost when m_AppWindow moves in and  
    // out of the foreground.  
    //  
    HANDLE *pMyHandles = reinterpret_cast<HANDLE*>(lParam);  
    DWORD cHandles = reinterpret_cast<DWORD>(wParam);  
  
    If (!SetAdditionalForegroundBoostProcesses(m_AppWindow, cHandles,  
    pMyHandles))  
    {  
        printf("SetAdditionalForegroundBoostProcesses() setup failed with  
error code : %d\n", GetLastError());  
    }  
  
    break;
```

Requirements

Minimum supported client	Windows 11 Build 22621
Header	Winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[SetForegroundWindow](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetClassLongA function (winuser.h)

Article02/09/2023

Replaces the specified 32-bit (`long`) value at the specified offset into the extra class memory or the [WNDCLASSEX](#) structure for the class to which the specified window belongs.

Note This function has been superseded by the [SetClassLongPtr](#) function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [SetClassLongPtr](#).

Syntax

C++

```
DWORD SetClassLongA(
    [in] HWND hWnd,
    [in] int nIndex,
    [in] LONG dwNewLong
);
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be replaced. To set a 32-bit value in the extra class memory, specify the positive, zero-based byte offset of the value to be set. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third 32-bit integer. To set any other value from the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning
GCL_CBCLSEXTRA -20	Sets the size, in bytes, of the extra memory associated with the class. Setting this value does not change the number of extra bytes already allocated.
GCL_CBWNDEXTRA -18	Sets the size, in bytes, of the extra window memory associated with each window in the class. Setting this value does not change the number of extra bytes already allocated. For information on how to access this memory, see SetWindowLong .
GCL_HBRBACKGROUND -10	Replaces a handle to the background brush associated with the class.
GCL_HCURSOR -12	Replaces a handle to the cursor associated with the class.
GCL_HICON -14	Replaces a handle to the icon associated with the class.
GCL_HICONSM -34	Replace a handle to the small icon associated with the class.
GCL_HMODULE -16	Replaces a handle to the module that registered the class.
GCL_MENUNAME -8	Replaces the address of the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Replaces the window-class style bits.
GCL_WNDPROC -24	Replaces the address of the window procedure associated with the class.

[in] dwNewLong

Type: **LONG**

The replacement value.

Return value

Type: **DWORD**

If the function succeeds, the return value is the previous value of the specified 32-bit integer. If the value was not previously set, the return value is zero.

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

Remarks

If you use the **SetClassLong** function and the **GCL_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

Calling **SetClassLong** with the **GCL_WNDPROC** index creates a subclass of the window class that affects all windows subsequently created with the class. An application can subclass a system class, but should not subclass a window class created by another process.

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Use the **SetClassLong** function with care. For example, it is possible to change the background color for a class by using **SetClassLong**, but this change does not immediately repaint all windows belonging to the class.

Examples

For an example, see [Displaying an Icon](#).

Note

The winuser.h header defines **SetClassLong** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[GetClassLong](#)

Reference

[RegisterClassEx](#)

[SetClassLongPtr](#)

[SetWindowLong](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetClassLongPtrA function (winuser.h)

Article02/09/2023

Replaces the specified value at the specified offset in the extra class memory or the [WNDCLASSEX](#) structure for the class to which the specified window belongs.

Note To write code that is compatible with both 32-bit and 64-bit Windows, use [SetClassLongPtr](#). When compiling for 32-bit Windows, [SetClassLongPtr](#) is defined as a call to the [SetClassLong](#) function

Syntax

C++

```
ULONG_PTR SetClassLongPtrA(  
    [in] HWND      hWnd,  
    [in] int       nIndex,  
    [in] LONG_PTR  dwNewLong  
)
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be replaced. To set a value in the extra class memory, specify the positive, zero-based byte offset of the value to be set. Valid values are in the range zero through the number of bytes of extra class memory, minus eight; for example, if you specified 24 or more bytes of extra class memory, a value of 16 would be an index to the third integer. To set a value other than the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning
GCL_CBCLSEXTRA -20	Sets the size, in bytes, of the extra memory associated with the class. Setting this value does not change the number of extra bytes already allocated.
GCL_CBWNDEXTRA -18	Sets the size, in bytes, of the extra window memory associated with each window in the class. Setting this value does not change the number of extra bytes already allocated. For information on how to access this memory, see SetWindowLongPtr .
GCLP_HBRBACKGROUND -10	Replaces a handle to the background brush associated with the class.
GCLP_HCURSOR -12	Replaces a handle to the cursor associated with the class.
GCLP_HICON -14	Replaces a handle to the icon associated with the class.
GCLP_HICONSM -34	Retrieves a handle to the small icon associated with the class.
GCLP_HMODULE -16	Replaces a handle to the module that registered the class.
GCLP_MENUNAME -8	Replaces the pointer to the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Replaces the window-class style bits.
GCLP_WNDPROC -24	Replaces the pointer to the window procedure associated with the class.

[in] dwNewLong

Type: **LONG_PTR**

The replacement value.

Return value

Type: **ULONG_PTR**

If the function succeeds, the return value is the previous value of the specified offset. If this was not previously set, the return value is zero.

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

Remarks

If you use the [SetClassLongPtr](#) function and the **GCLP_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

Calling [SetClassLongPtr](#) with the **GCLP_WNDPROC** index creates a subclass of the window class that affects all windows subsequently created with the class. An application can subclass a system class, but should not subclass a window class created by another process.

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Use the [SetClassLongPtr](#) function with care. For example, it is possible to change the background color for a class by using [SetClassLongPtr](#), but this change does not immediately repaint all windows belonging to the class.

 **Note**

The winuser.h header defines [SetClassLongPtr](#) 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	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[GetClassLongPtr](#)

Reference

[RegisterClassEx](#)

[SetWindowLongPtr](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetClassLongPtrW function (winuser.h)

Article02/09/2023

Replaces the specified value at the specified offset in the extra class memory or the [WNDCLASSEX](#) structure for the class to which the specified window belongs.

Note To write code that is compatible with both 32-bit and 64-bit Windows, use [SetClassLongPtr](#). When compiling for 32-bit Windows, [SetClassLongPtr](#) is defined as a call to the [SetClassLong](#) function

Syntax

C++

```
ULONG_PTR SetClassLongPtrW(  
    [in] HWND      hWnd,  
    [in] int       nIndex,  
    [in] LONG_PTR  dwNewLong  
)
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be replaced. To set a value in the extra class memory, specify the positive, zero-based byte offset of the value to be set. Valid values are in the range zero through the number of bytes of extra class memory, minus eight; for example, if you specified 24 or more bytes of extra class memory, a value of 16 would be an index to the third integer. To set a value other than the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning
GCL_CBCLSEXTRA -20	Sets the size, in bytes, of the extra memory associated with the class. Setting this value does not change the number of extra bytes already allocated.
GCL_CBWNDEXTRA -18	Sets the size, in bytes, of the extra window memory associated with each window in the class. Setting this value does not change the number of extra bytes already allocated. For information on how to access this memory, see SetWindowLongPtr .
GCLP_HBRBACKGROUND -10	Replaces a handle to the background brush associated with the class.
GCLP_HCURSOR -12	Replaces a handle to the cursor associated with the class.
GCLP_HICON -14	Replaces a handle to the icon associated with the class.
GCLP_HICONSM -34	Retrieves a handle to the small icon associated with the class.
GCLP_HMODULE -16	Replaces a handle to the module that registered the class.
GCLP_MENUNAME -8	Replaces the pointer to the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Replaces the window-class style bits.
GCLP_WNDPROC -24	Replaces the pointer to the window procedure associated with the class.

[in] dwNewLong

Type: **LONG_PTR**

The replacement value.

Return value

Type: **ULONG_PTR**

If the function succeeds, the return value is the previous value of the specified offset. If this was not previously set, the return value is zero.

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

Remarks

If you use the [SetClassLongPtr](#) function and the **GCLP_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

Calling [SetClassLongPtr](#) with the **GCLP_WNDPROC** index creates a subclass of the window class that affects all windows subsequently created with the class. An application can subclass a system class, but should not subclass a window class created by another process.

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Use the [SetClassLongPtr](#) function with care. For example, it is possible to change the background color for a class by using [SetClassLongPtr](#), but this change does not immediately repaint all windows belonging to the class.

 **Note**

The winuser.h header defines [SetClassLongPtr](#) 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	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[GetClassLongPtr](#)

Reference

[RegisterClassEx](#)

[SetWindowLongPtr](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetClassLongW function (winuser.h)

Article02/09/2023

Replaces the specified 32-bit (`long`) value at the specified offset into the extra class memory or the [WNDCLASSEX](#) structure for the class to which the specified window belongs.

Note This function has been superseded by the [SetClassLongPtr](#) function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use [SetClassLongPtr](#).

Syntax

C++

```
DWORD SetClassLongW(
    [in] HWND hWnd,
    [in] int nIndex,
    [in] LONG dwNewLong
);
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The value to be replaced. To set a 32-bit value in the extra class memory, specify the positive, zero-based byte offset of the value to be set. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third 32-bit integer. To set any other value from the [WNDCLASSEX](#) structure, specify one of the following values.

Value	Meaning
GCL_CBCLSEXTRA -20	Sets the size, in bytes, of the extra memory associated with the class. Setting this value does not change the number of extra bytes already allocated.
GCL_CBWNDEXTRA -18	Sets the size, in bytes, of the extra window memory associated with each window in the class. Setting this value does not change the number of extra bytes already allocated. For information on how to access this memory, see SetWindowLong .
GCL_HBRBACKGROUND -10	Replaces a handle to the background brush associated with the class.
GCL_HCURSOR -12	Replaces a handle to the cursor associated with the class.
GCL_HICON -14	Replaces a handle to the icon associated with the class.
GCL_HICONSM -34	Replace a handle to the small icon associated with the class.
GCL_HMODULE -16	Replaces a handle to the module that registered the class.
GCL_MENUNAME -8	Replaces the address of the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE -26	Replaces the window-class style bits.
GCL_WNDPROC -24	Replaces the address of the window procedure associated with the class.

[in] dwNewLong

Type: **LONG**

The replacement value.

Return value

Type: **DWORD**

If the function succeeds, the return value is the previous value of the specified 32-bit integer. If the value was not previously set, the return value is zero.

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

Remarks

If you use the **SetClassLong** function and the **GCL_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

Calling **SetClassLong** with the **GCL_WNDPROC** index creates a subclass of the window class that affects all windows subsequently created with the class. An application can subclass a system class, but should not subclass a window class created by another process.

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Use the **SetClassLong** function with care. For example, it is possible to change the background color for a class by using **SetClassLong**, but this change does not immediately repaint all windows belonging to the class.

Examples

For an example, see [Displaying an Icon](#).

Note

The winuser.h header defines **SetClassLong** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[GetClassLong](#)

Reference

[RegisterClassEx](#)

[SetClassLongPtr](#)

[SetWindowLong](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetClassWord function (winuser.h)

Article 10/13/2021

Replaces the 16-bit (WORD) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should use the **SetClassLong** function.

Syntax

C++

```
WORD SetClassWord(
    [in] HWND hWnd,
    [in] int nIndex,
    [in] WORD wNewWord
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: **int**

The zero-based byte offset of the value to be replaced. Valid values are in the range zero through the number of bytes of class memory minus two; for example, if you specified 10 or more bytes of extra class memory, a value of 8 would be an index to the fifth 16-bit integer.

[in] wNewWord

Type: **WORD**

The replacement value.

Return value

Type: WORD

If the function succeeds, the return value is the previous value of the specified 16-bit integer. If the value was not previously set, the return value is zero.

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

Remarks

Reserve extra class memory by specifying a nonzero value in the **cbClsExtra** member of the [WNDCLASS](#) structure used with the [RegisterClass](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[GetClassWord](#)

[Reference](#)

[RegisterClass](#)

[SetClassLong](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

SetCoalescableTimer function (winuser.h)

Article 10/13/2021

Creates a timer with the specified time-out value and coalescing tolerance delay.

Syntax

C++

```
UINT_PTR SetCoalescableTimer(
    [in, optional] HWND      hWnd,
    [in]          UINT_PTR   nIDEvent,
    [in]          UINT       uElapse,
    [in, optional] TIMERPROC lpTimerFunc,
    [in]          ULONG      uToleranceDelay
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window to be associated with the timer. This window must be owned by the calling thread. If a **NULL** value for *hWnd* is passed in along with an *nIDEvent* of an existing timer, that timer will be replaced in the same way that an existing non-**NULL** *hWnd* timer will be.

[in] nIDEvent

Type: **UINT_PTR**

A timer identifier. If the *hWnd* parameter is **NULL**, and the *nIDEvent* does not match an existing timer, then the *nIDEvent* is ignored and a new timer ID is generated. If the *hWnd* parameter is not **NULL** and the window specified by *hWnd* already has a timer with the value *nIDEvent*, then the existing timer is replaced by the new timer. When **SetCoalescableTimer** replaces a timer, the timer is reset. Therefore, a message will be sent after the current time-out value elapses, but the previously set time-out value is ignored. If the call is not intended to replace an existing timer, *nIDEvent* should be 0 if the *hWnd* is **NULL**.

[in] *uElapse*

Type: **UINT**

The time-out value, in milliseconds.

If *uElapse* is less than **USER_TIMER_MINIMUM** (0x0000000A), the timeout is set to **USER_TIMER_MINIMUM**. If *uElapse* is greater than **USER_TIMER_MAXIMUM** (0x7FFFFFFF), the timeout is set to **USER_TIMER_MAXIMUM**.

If the sum of *uElapse* and *uToleranceDelay* exceeds **USER_TIMER_MAXIMUM**, an **ERROR_INVALID_PARAMETER** exception occurs.

[in, optional] *lpTimerFunc*

Type: **TIMERPROC**

A pointer to the function to be notified when the time-out value elapses. For more information about the function, see [TimerProc](#). If *lpTimerFunc* is **NULL**, the system posts a [WM_TIMER](#) message to the application queue. The **hwnd** member of the message's [MSG](#) structure contains the value of the *hWnd* parameter.

[in] *uToleranceDelay*

Type: **ULONG**

It can be one of the following values:

Value	Meaning
TIMERV_DEFAULT_COALESCING 0x00000000	Uses the system default timer coalescing.
TIMERV_NO_COALESCING 0xFFFFFFFF	Uses no timer coalescing. When this value is used, the created timer is not coalesced, no matter what the system default timer coalescing is or the application compatibility flags are.
0x1 - 0x7FFFFFF5	<p>Note Do not use this value unless you are certain that the timer requires no coalescing.</p> <p>Specifies the coalescing tolerance delay, in milliseconds. Applications should set this value to the system default (TIMERV_DEFAULT_COALESCING) or the largest value possible.</p>

	If the sum of <i>uElapse</i> and <i>uToleranceDelay</i> exceeds USER_TIMER_MAXIMUM (0x7FFFFFFF), an ERROR_INVALID_PARAMETER exception occurs.
	See Windows Timer Coalescing for more details and best practices.
Any other value	An invalid value. If <i>uToleranceDelay</i> is set to an invalid value, the function fails and returns zero.

Return value

Type: **UINT_PTR**

If the function succeeds and the *hWnd* parameter is **NULL**, the return value is an integer identifying the new timer. An application can pass this value to the [KillTimer](#) function to destroy the timer.

If the function succeeds and the *hWnd* parameter is not **NULL**, then the return value is a nonzero integer. An application can pass the value of the *nIDEvent* parameter to the [KillTimer](#) function to destroy the timer.

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

Remarks

An application can process [WM_TIMER](#) messages by including a **WM_TIMER** case statement in the window procedure or by specifying a [TimerProc](#) callback function when creating the timer. When you specify a [TimerProc](#) callback function, the default window procedure calls the callback function when it processes **WM_TIMER**. Therefore, you need to dispatch messages in the calling thread, even when you use [TimerProc](#) instead of processing **WM_TIMER**.

The *wParam* parameter of the [WM_TIMER](#) message contains the value of the *nIDEvent* parameter.

The timer identifier, *nIDEvent*, is specific to the associated window. Another window can have its own timer which has the same identifier as a timer owned by another window. The timers are distinct.

[SetTimer](#) can reuse timer IDs in the case where *hWnd* is **NULL**.

When *uToleranceDelay* is set to 0, the system default timer coalescing is used and **SetCoalescableTimer** behaves the same as [SetTimer](#).

Before using **SetCoalescableTimer** or other timer-related functions, it is recommended to set the **UOI_TIMERPROC_EXCEPTION_SUPPRESSION** flag to **false** through the **SetUserObjectInformationW** function, otherwise the application could behave unpredictably and could be vulnerable to security exploits. For more info, see [SetUserObjectInformationW](#).

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

[Coalescing timers sample](#) ↗

Conceptual

[KeSetCoalescableTimer](#)

[KeSetTimer](#)

[KillTimer](#)

[MSG](#)

Reference

[Sample](#)

[SetTimer](#)

[TimerProc](#)

[Timers](#)

[Using Timers](#)

[WM_TIMER](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetForegroundWindow function (winuser.h)

Article03/11/2023

Brings the thread that created the specified window into the foreground and activates the window. Keyboard input is directed to the window, and various visual cues are changed for the user. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.

Syntax

C++

```
BOOL SetForegroundWindow(  
    [in] HWND hWnd  
)
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window that should be activated and brought to the foreground.

Return value

Type: **BOOL**

If the window was brought to the foreground, the return value is nonzero.

If the window was not brought to the foreground, the return value is zero.

Remarks

The system restricts which processes can set the foreground window. A process can set the foreground window by calling **SetForegroundWindow** only if:

- All of the following conditions are true:

- The calling process belongs to a desktop application, not a UWP app or a Windows Store app designed for Windows 8 or 8.1.
- The foreground process has not disabled calls to **SetForegroundWindow** by a previous call to the **LockSetForegroundWindow** function.
- The foreground lock time-out has expired (see **SPI_GETFOREGROUNDLOCKTIMEOUT** in **SystemParametersInfo**).
- No menus are active.
- Additionally, at least one of the following conditions is true:
 - The calling process is the foreground process.
 - The calling process was started by the foreground process.
 - There is currently no foreground window, and thus no foreground process.
 - The calling process received the last input event.
 - Either the foreground process or the calling process is being debugged.

It is possible for a process to be denied the right to set the foreground window even if it meets these conditions.

An application cannot force a window to the foreground while the user is working with another window. Instead, Windows flashes the taskbar button of the window to notify the user.

A process that can set the foreground window can enable another process to set the foreground window by calling the **AllowSetForegroundWindow** function. The process specified by the *dwProcessId* parameter to **AllowSetForegroundWindow** loses the ability to set the foreground window the next time that either the user generates input, unless the input is directed at that process, or the next time a process calls **AllowSetForegroundWindow**, unless the same process is specified as in the previous call to **AllowSetForegroundWindow**.

The foreground process can disable calls to **SetForegroundWindow** by calling the **LockSetForegroundWindow** function.

Example

The following code example demonstrates the use of **SetForegroundWindow**

C++

```
// If the window is invisible we will show it and make it topmost without
// the
// foreground focus. If the window is visible it will also be made the
// topmost window without the foreground focus. If wParam is TRUE then
// for both cases the window will be forced into the foreground focus
if (uMsg == m_ShowStageMessage) {
```

```

BOOL bVisible = IsWindowVisible(hwnd);
SetWindowPos(hwnd, HWND_TOP, 0, 0, 0, 0,
             SWP_NOMOVE | SWP_NOSIZE | SWP_SHOWWINDOW |
             (bVisible ? SWP_NOACTIVATE : 0));
// Should we bring the window to the foreground
if (wParam == TRUE) {
    SetForegroundWindow(hwnd);
}
return (LRESULT) 1;
}

```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[AllowSetForegroundWindow](#)

Conceptual

[FlashWindowEx](#)

[GetForegroundWindow](#)

[LockSetForegroundWindow](#)

Reference

[SetActiveWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetLayeredWindowAttributes function (winuser.h)

Article 10/13/2021

Sets the opacity and transparency color key of a layered window.

Syntax

C++

```
BOOL SetLayeredWindowAttributes(
    [in] HWND      hwnd,
    [in] COLORREF  crKey,
    [in] BYTE       bAlpha,
    [in] DWORD      dwFlags
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the layered window. A layered window is created by specifying **WS_EX_LAYERED** when creating the window with the [CreateWindowEx](#) function or by setting **WS_EX_LAYERED** via [SetWindowLong](#) after the window has been created.

Windows 8: The **WS_EX_LAYERED** style is supported for top-level windows and child windows. Previous Windows versions support **WS_EX_LAYERED** only for top-level windows.

[in] crKey

Type: **COLORREF**

A **COLORREF** structure that specifies the transparency color key to be used when composing the layered window. All pixels painted by the window in this color will be transparent. To generate a **COLORREF**, use the [RGB](#) macro.

[in] bAlpha

Type: **BYTE**

Alpha value used to describe the opacity of the layered window. Similar to the **SourceConstantAlpha** member of the [BLENDFUNCTION](#) structure. When *bAlpha* is 0, the window is completely transparent. When *bAlpha* is 255, the window is opaque.

[in] dwFlags

Type: **DWORD**

An action to be taken. This parameter can be one or more of the following values.

Value	Meaning
LWA_ALPHA 0x00000002	Use <i>bAlpha</i> to determine the opacity of the layered window.
LWA_COLORKEY 0x00000001	Use <i>crKey</i> as the transparency color.

Return value

Type: **BOOL**

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 that once [SetLayeredWindowAttributes](#) has been called for a layered window, subsequent [UpdateLayeredWindow](#) calls will fail until the layering style bit is cleared and set again.

For more information, see [Using Layered Windows](#).

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[AlphaBlend](#)

[COLORREF](#)

[Conceptual](#)

[CreateWindowEx](#)

[Other Resources](#)

[RGB](#)

[Reference](#)

[SetWindowLong](#)

[TransparentBlt](#)

[UpdateLayeredWindow](#)

[Using Windows](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetMessageExtraInfo function (winuser.h)

Article 10/13/2021

Sets the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue. An application can use the [GetMessageExtraInfo](#) function to retrieve a thread's extra message information.

Syntax

C++

```
LPARAM SetMessageExtraInfo(  
    [in] LPARAM lParam  
);
```

Parameters

[in] lParam

Type: **LPARAM**

The value to be associated with the current thread.

Return value

Type: **LPARAM**

The return value is the previous value associated with the current thread.

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetMessageExtraInfo](#)

[Messages and Message Queues](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

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

SetParent function (winuser.h)

Article 10/13/2021

Changes the parent window of the specified child window.

Syntax

C++

```
HWND SetParent(  
    [in]           HWND hWndChild,  
    [in, optional] HWND hWndNewParent  
);
```

Parameters

[in] hWndChild

Type: **HWND**

A handle to the child window.

[in, optional] hWndNewParent

Type: **HWND**

A handle to the new parent window. If this parameter is **NULL**, the desktop window becomes the new parent window. If this parameter is **HWND_MESSAGE**, the child window becomes a [message-only window](#).

Return value

Type: **HWND**

If the function succeeds, the return value is a handle to the previous parent window.

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

Remarks

An application can use the **SetParent** function to set the parent window of a pop-up, overlapped, or child window.

If the window identified by the *hWndChild* parameter is visible, the system performs the appropriate redrawing and repainting.

For compatibility reasons, **SetParent** does not modify the **WS_CHILD** or **WS_POPUP** window styles of the window whose parent is being changed. Therefore, if *hWndNewParent* is **NULL**, you should also clear the **WS_CHILD** bit and set the **WS_POPUP** style after calling **SetParent**. Conversely, if *hWndNewParent* is not **NULL** and the window was previously a child of the desktop, you should clear the **WS_POPUP** style and set the **WS_CHILD** style before calling **SetParent**.

When you change the parent of a window, you should synchronize the **UISTATE** of both windows. For more information, see [WM_CHANGEUISTATE](#) and [WM_UPDATEUISTATE](#).

Unexpected behavior or errors may occur if *hWndNewParent* and *hWndChild* are running in different DPI awareness modes. The table below outlines this behavior:

Operation	Windows 8.1	Windows 10 (1607 and earlier)	Windows 10 (1703 and later)
SetParent (In-Proc)	N/A	Forced reset (of current process)	Fail (ERROR_INVALID_STATE)
SetParent (Cross-Proc)	Forced reset (of child window's process)	Forced reset (of child window's process)	Forced reset (of child window's process)

For more information on DPI awareness, see [the Windows High DPI documentation](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set

ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetParent](#)

Reference

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetProcessDefaultLayout function (winuser.h)

Article 10/13/2021

Changes the default layout when windows are created with no parent or owner only for the currently running process.

Syntax

C++

```
BOOL SetProcessDefaultLayout(
    [in] DWORD dwDefaultLayout
);
```

Parameters

[in] dwDefaultLayout

Type: **DWORD**

The default process layout. This parameter can be 0 or the following value.

Value	Meaning
LAYOUTRTL 0x00000001	Sets the default horizontal layout to be right to left.

Return value

Type: **BOOL**

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 layout specifies how text and graphics are laid out; the default is left to right. The **SetProcessDefaultLayout** function changes layout to be right to left, which is the standard in Arabic and Hebrew cultures.

After the **LAYOUT_RTL** flag is selected, flags normally specifying right or left are reversed. To avoid confusion, consider defining alternate words for standard flags, such as those in the following table.

Standard flag	Suggested alternate name
WS_EX_RIGHT	WS_EX_TRAILING
WS_EX_RTLREADING	WS_EX_REVERSEREADING
WS_EX_LEFTSCROLLBAR	WS_EX_LEADSCROLLBAR
ES_LEFT	ES_LEAD
ES_RIGHT	ES_TRAIL
EC_LEFTMARGIN	EC_LEADMARGIN
EC_RIGHTMARGIN	EC_TRAILMARGIN

If using this function with a mirrored window, note that the **SetProcessDefaultLayout** function does not mirror the whole process and all the device contexts (DCs) created in it. It mirrors only the mirrored window's DCs. To mirror any DC, use the [SetLayout](#) function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)
---------	--

See also

Conceptual

[GetProcessDefaultLayout](#)

Other Resources

Reference

[SetLayout](#)

Windows

Feedback

Was this page helpful?

 Yes

 No

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

SetProcessDPIAware function (winuser.h)

Article 06/29/2021

Sets the process-default DPI awareness to system-DPI awareness. This is equivalent to calling [SetProcessDpiAwarenessContext](#) with a **DPI_AWARENESS_CONTEXT** value of **DPI_AWARENESS_CONTEXT_SYSTEM_AWARE**.

ⓘ Note

It is recommended that you set the process-default DPI awareness via application manifest, not an API call. See [Setting the default DPI awareness for a process](#) for more information. Setting the process-default DPI awareness via API call can lead to unexpected application behavior.

Syntax

C++

```
BOOL SetProcessDPIAware();
```

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero. Otherwise, the return value is zero.

Remarks

For more information, see [Setting the default DPI awareness for a process](#).

Requirements

Minimum supported client

Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Setting the default DPI awareness for a process](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetPropA function (winuser.h)

Article 02/09/2023

Adds a new entry or changes an existing entry in the property list of the specified window. The function adds a new entry to the list if the specified character string does not exist already in the list. The new entry contains the string and the handle. Otherwise, the function replaces the string's current handle with the specified handle.

Syntax

C++

```
BOOL SetPropA(
    [in]           HWND   hWnd,
    [in]           LPCSTR lpString,
    [in, optional] HANDLE hData
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list receives the new entry.

[in] lpString

Type: **LPCTSTR**

A null-terminated string or an atom that identifies a string. If this parameter is an atom, it must be a global atom created by a previous call to the [GlobalAddAtom](#) function. The atom must be placed in the low-order word of *lpString*; the high-order word must be zero.

[in, optional] hData

Type: **HANDLE**

A handle to the data to be copied to the property list. The data handle can identify any value useful to the application.

Return value

Type: **BOOL**

If the data handle and string are added to the property list, the return value is nonzero.

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

Remarks

Before a window is destroyed (that is, before it returns from processing the [WM_NCDESTROY](#) message), an application must remove all entries it has added to the property list. The application must use the [RemoveProp](#) function to remove the entries.

SetProp is subject to the restrictions of User Interface Privilege Isolation (UIPI). A process can only call this function on a window belonging to a process of lesser or equal integrity level. When UIPI blocks property changes, [GetLastError](#) will return 5.

Examples

For an example, see [Adding a Window Property](#).

Note

The winuser.h header defines SetProp 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GlobalAddAtom](#)

Reference

[RemoveProp](#)

[WM_NCDESTROY](#)

[Window Properties](#)

[ITaskbarList2::MarkFullscreenWindow](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetPropW function (winuser.h)

Article 02/09/2023

Adds a new entry or changes an existing entry in the property list of the specified window. The function adds a new entry to the list if the specified character string does not exist already in the list. The new entry contains the string and the handle. Otherwise, the function replaces the string's current handle with the specified handle.

Syntax

C++

```
BOOL SetPropW(
    [in]           HWND     hWnd,
    [in]           LPCWSTR  lpString,
    [in, optional] HANDLE   hData
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window whose property list receives the new entry.

[in] lpString

Type: **LPCTSTR**

A null-terminated string or an atom that identifies a string. If this parameter is an atom, it must be a global atom created by a previous call to the [GlobalAddAtom](#) function. The atom must be placed in the low-order word of *lpString*; the high-order word must be zero.

[in, optional] hData

Type: **HANDLE**

A handle to the data to be copied to the property list. The data handle can identify any value useful to the application.

Return value

Type: **BOOL**

If the data handle and string are added to the property list, the return value is nonzero.

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

Remarks

Before a window is destroyed (that is, before it returns from processing the [WM_NCDESTROY](#) message), an application must remove all entries it has added to the property list. The application must use the [RemoveProp](#) function to remove the entries.

SetProp is subject to the restrictions of User Interface Privilege Isolation (UIPI). A process can only call this function on a window belonging to a process of lesser or equal integrity level. When UIPI blocks property changes, [GetLastError](#) will return 5.

Examples

For an example, see [Adding a Window Property](#).

Note

The winuser.h header defines SetProp 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GlobalAddAtom](#)

Reference

[RemoveProp](#)

[WM_NCDESTROY](#)

[Window Properties](#)

[ITaskbarList2::MarkFullscreenWindow](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetSysColors function (winuser.h)

Article 10/13/2021

Sets the colors for the specified display elements. Display elements are the various parts of a window and the display that appear on the system display screen.

Syntax

C++

```
BOOL SetSysColors(
    [in] int             cElements,
    [in] const INT       *lpaElements,
    [in] const COLORREF *lpaRgbValues
);
```

Parameters

[in] `cElements`

Type: `int`

The number of display elements in the `lpaElements` array.

[in] `lpaElements`

Type: `const INT*`

An array of integers that specify the display elements to be changed. For a list of display elements, see [GetSysColor](#).

[in] `lpaRgbValues`

Type: `const COLORREF*`

An array of `COLORREF` values that contain the new red, green, blue (RGB) color values for the display elements in the array pointed to by the `lpaElements` parameter.

To generate a `COLORREF`, use the `RGB` macro.

Return value

Type: **BOOL**

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 **SetSysColors** function sends a [WM_SYSCOLORCHANGE](#) message to all windows to inform them of the change in color. It also directs the system to repaint the affected portions of all currently visible windows.

It is best to respect the color settings specified by the user. If you are writing an application to enable the user to change the colors, then it is appropriate to use this function. However, this function affects only the current session. The new colors are not saved when the system terminates.

Examples

The following example demonstrates the use of the [GetSysColor](#) and [SetSysColors](#) functions. First, the example uses [GetSysColor](#) to retrieve the colors of the window background and active caption and displays the red, green, blue (RGB) values in hexadecimal notation. Next, example uses [SetSysColors](#) to change the color of the window background to light gray and the active title bars to dark purple. After a 10-second delay, the example restores the previous colors for these elements using [SetSysColors](#).

```
#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "user32.lib")

void main()
{
    int aElements[2] = {COLOR_WINDOW, COLOR_ACTIVECAPTION};
    DWORD aOldColors[2];
    DWORD aNewColors[2];

    // Get the current color of the window background.

    aOldColors[0] = GetSysColor(aElements[0]);

    printf("Current window color: {0x%08x, 0x%08x, 0x%08x}\n",
        GetRValue(aOldColors[0]),
```

```

        GetGValue(aOldColors[0]),
        GetBValue(aOldColors[0]));

    // Get the current color of the active caption.

    aOldColors[1] = GetSysColor(aElements[1]);

    printf("Current active caption color: {0x%x, 0x%x, 0x%x}\n",
        GetRValue(aOldColors[1]),
        GetGValue(aOldColors[1]),
        GetBValue(aOldColors[1]));

    // Define new colors for the elements

    aNewColors[0] = RGB(0x80, 0x80, 0x80); // light gray
    aNewColors[1] = RGB(0x80, 0x00, 0x80); // dark purple

    printf("\nNew window color: {0x%x, 0x%x, 0x%x}\n",
        GetRValue(aNewColors[0]),
        GetGValue(aNewColors[0]),
        GetBValue(aNewColors[0]));

    printf("New active caption color: {0x%x, 0x%x, 0x%x}\n",
        GetRValue(aNewColors[1]),
        GetGValue(aNewColors[1]),
        GetBValue(aNewColors[1]));

    // Set the elements defined in aElements to the colors defined
    // in aNewColors

    SetSysColors(2, aElements, aNewColors);

    printf("\nWindow background and active border have been changed.\n");
    printf("Reverting to previous colors in 10 seconds...\n");

    Sleep(10000);

    // Restore the elements to their original colors

    SetSysColors(2, aElements, aOldColors);
}

```

Requirements

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

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[COLORREF](#)

[GetSysColor](#)

[RGB](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetTimer function (winuser.h)

Article 10/13/2021

Creates a timer with the specified time-out value.

Syntax

C++

```
UINT_PTR SetTimer(
    [in, optional] HWND      hWnd,
    [in]          UINT_PTR   nIDEvent,
    [in]          UINT        uElapse,
    [in, optional] TIMERPROC lpTimerFunc
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window to be associated with the timer. This window must be owned by the calling thread. If a **NULL** value for *hWnd* is passed in along with an *nIDEvent* of an existing timer, that timer will be replaced in the same way that an existing non-**NULL** *hWnd* timer will be.

[in] nIDEvent

Type: **UINT_PTR**

A nonzero timer identifier. If the *hWnd* parameter is **NULL**, and the *nIDEvent* does not match an existing timer then it is ignored and a new timer ID is generated. If the *hWnd* parameter is not **NULL** and the window specified by *hWnd* already has a timer with the value *nIDEvent*, then the existing timer is replaced by the new timer. When **SetTimer** replaces a timer, the timer is reset. Therefore, a message will be sent after the current time-out value elapses, but the previously set time-out value is ignored. If the call is not intended to replace an existing timer, *nIDEvent* should be 0 if the *hWnd* is **NULL**.

[in] uElapse

Type: **UINT**

The time-out value, in milliseconds.

If *uElapse* is less than **USER_TIMER_MINIMUM** (0x0000000A), the timeout is set to **USER_TIMER_MINIMUM**. If *uElapse* is greater than **USER_TIMER_MAXIMUM** (0x7FFFFFFF), the timeout is set to **USER_TIMER_MAXIMUM**.

[in, optional] *lpTimerFunc*

Type: **TIMERPROC**

A pointer to the function to be notified when the time-out value elapses. For more information about the function, see [TimerProc](#). If *lpTimerFunc* is **NULL**, the system posts a [WM_TIMER](#) message to the application queue. The **hwnd** member of the message's **MSG** structure contains the value of the *hWnd* parameter.

Return value

Type: **UINT_PTR**

If the function succeeds and the *hWnd* parameter is **NULL**, the return value is an integer identifying the new timer. An application can pass this value to the [KillTimer](#) function to destroy the timer.

If the function succeeds and the *hWnd* parameter is not **NULL**, then the return value is a nonzero integer. An application can pass the value of the *nIDEvent* parameter to the [KillTimer](#) function to destroy the timer.

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

Remarks

An application can process [WM_TIMER](#) messages by including a **WM_TIMER** case statement in the window procedure or by specifying a [TimerProc](#) callback function when creating the timer. When you specify a [TimerProc](#) callback function, the [DispatchMessage](#) function calls the callback function instead of calling the window procedure when it processes **WM_TIMER** with a non-NULL *IParam*. Therefore, you need to dispatch messages in the calling thread, even when you use [TimerProc](#) instead of processing **WM_TIMER**.

The *wParam* parameter of the [WM_TIMER](#) message contains the value of the *nIDEvent* parameter.

The timer identifier, *nIDEvent*, is specific to the associated window. Another window can have its own timer which has the same identifier as a timer owned by another window. The timers are distinct.

SetTimer can reuse timer IDs in the case where *hWnd* is **NULL**.

Before using **SetTimer** or other timer-related functions, it is recommended to set the **UOI_TIMERPROC_EXCEPTION_SUPPRESSION** flag to **false** through the **SetUserObjectInformationW** function, otherwise the application could behave unpredictably and could be vulnerable to security exploits. For more info, see [SetUserObjectInformationW](#).

Examples

For an example, see [Creating a Timer](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-2 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[KillTimer](#)

[MSG](#)

Reference

[SetWaitableTimer](#)

[TimerProc](#)

[Timers](#)

[WM_TIMER](#)

[SetCoalescableTimer](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowDisplayAffinity function (winuser.h)

Article 10/13/2021

Specifies where the content of the window can be displayed.

Syntax

C++

```
BOOL SetWindowDisplayAffinity(  
    [in] HWND hWnd,  
    [in] DWORD dwAffinity  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the top-level window. The window must belong to the current process.

[in] dwAffinity

Type: **DWORD**

The display affinity setting that specifies where the content of the window can be displayed.

This parameter can be one of the following values.

Value	Meaning
WDA_NONE 0x00000000	Imposes no restrictions on where the window can be displayed.
WDA_MONITOR 0x00000001	The window content is displayed only on a monitor. Everywhere else, the window appears with no content.
WDA_EXCLUDEFROMCAPTURE 0x00000011	The window is displayed only on a monitor. Everywhere else, the window does not appear at all.

One use for this affinity is for windows that show video recording controls, so that the controls are not included in the capture.

Introduced in Windows 10 Version 2004. See remarks about compatibility regarding previous versions of Windows.

Return value

Type: **BOOL**

If the function succeeds, it returns **TRUE**; otherwise, it returns **FALSE** when, for example, the function call is made on a non top-level window. To get extended error information, call [GetLastError](#).

Remarks

This function and [GetWindowDisplayAffinity](#) are designed to support the window content protection feature that is new to Windows 7. This feature enables applications to protect their own onscreen window content from being captured or copied through a specific set of public operating system features and APIs. However, it works only when the Desktop Window Manager(DWM) is composing the desktop.

It is important to note that unlike a security feature or an implementation of Digital Rights Management (DRM), there is no guarantee that using [SetWindowDisplayAffinity](#) and [GetWindowDisplayAffinity](#), and other necessary functions such as [DwmIsCompositionEnabled](#), will strictly protect windowed content, for example where someone takes a photograph of the screen.

Starting in Windows 10 Version 2004, WDA_EXCLUDEFROMCAPTURE is a supported value. Setting the display affinity to WDA_EXCLUDEFROMCAPTURE on previous version of Windows will behave as if WDA_MONITOR is applied.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[SetWindowDisplayAffinity](#), [Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowLongA function (winuser.h)

Article02/09/2023

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory.

Note This function has been superseded by the [SetWindowLongPtr](#) function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use the [SetWindowLongPtr](#) function.

Syntax

C++

```
LONG SetWindowLongA(
    [in] HWND hWnd,
    [in] int nIndex,
    [in] LONG dwNewLong
);
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of an integer. To set any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE	Sets a new extended window style .
-20	

GWL_HINSTANCE	Sets a new application instance handle.
-6	
GWL_ID	Sets a new identifier of the child window. The window cannot be a top-level window.
-12	
GWL_STYLE	Sets a new window style .
-16	
GWL_USERDATA	Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
-21	
GWL_WNDPROC	Sets a new address for the window procedure. You cannot change this attribute if the window does not belong to the same process as the calling thread.
-4	

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWL_DLGPROC DWLP_MSGRESULT + sizeof(LRESULT)	Sets the new address of the dialog box procedure.
DWL_MSGRESULT 0	Sets the return value of a message processed in the dialog box procedure.
DWL_USER DWLP_DLGPROC + sizeof(DLGPROC)	Sets new extra information that is private to the application, such as handles or pointers.

[in] `dwNewLong`

Type: **LONG**

The replacement value.

Return value

Type: **LONG**

If the function succeeds, the return value is the previous value of the specified 32-bit integer.

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

If the previous value of the specified 32-bit integer is zero, and the function succeeds, the return value is zero, but the function does not clear the last error information. This makes it difficult to determine success or failure. To deal with this, you should clear the last error information by calling [SetLastError](#) with 0 before calling [SetWindowLong](#). Then, function failure will be indicated by a return value of zero and a [GetLastError](#) result that is nonzero.

Remarks

Certain window data is cached, so changes you make using [SetWindowLong](#) will not take effect until you call the [SetWindowPos](#) function. Specifically, if you change any of the frame styles, you must call [SetWindowPos](#) with the **SWP_FRAMECHANGED** flag for the cache to be updated properly.

If you use [SetWindowLong](#) with the **GWL_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

If you use [SetWindowLong](#) with the **DWL_MSGRESULT** index to set the return value for a message processed by a dialog procedure, you should return **TRUE** directly afterward. Otherwise, if you call any function that results in your dialog procedure receiving a window message, the nested window message could overwrite the return value you set using **DWL_MSGRESULT**.

Calling [SetWindowLong](#) with the **GWL_WNDPROC** index creates a subclass of the window class used to create the window. An application can subclass a system class, but should not subclass a window class created by another process. The [SetWindowLong](#) function creates the window subclass by changing the window procedure associated with a particular window class, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling [CallWindowProc](#). This allows the application to create a chain of window procedures.

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

You must not call [SetWindowLong](#) with the **GWL_HWNDPARENT** index to change the parent of a child window. Instead, use the [SetParent](#) function.

If the window has a class style of **CS_CLASSDC** or **CS_OWNDC**, do not set the extended window styles **WS_EX_COMPOSITED** or **WS_EX_LAYERED**.

Calling [SetWindowLong](#) to set the style on a progressbar will reset its position.

Examples

For an example, see [Subclassing a Window](#).

Note

The winuser.h header defines SetWindowLong 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[GetWindowLong](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

[SetWindowLongPtr](#)

WNDCLASSEX

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowLongPtrA function (winuser.h)

Article02/09/2023

Changes an attribute of the specified window. The function also sets a value at the specified offset in the extra window memory.

Note To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **SetWindowLongPtr**. When compiling for 32-bit Windows, **SetWindowLongPtr** is defined as a call to the **SetWindowLong** function.

Syntax

C++

```
LONG_PTR SetWindowLongPtrA(  
    [in] HWND      hWnd,  
    [in] int       nIndex,  
    [in] LONG_PTR  dwNewLong  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs. The **SetWindowLongPtr** function fails if the process that owns the window specified by the *hWnd* parameter is at a higher process privilege in the UIPI hierarchy than the process the calling thread resides in.

Windows XP/2000: The **SetWindowLongPtr** function fails if the window specified by the *hWnd* parameter does not belong to the same process as the calling thread.

[in] nIndex

Type: **int**

The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of a **LONG_PTR**. To set any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE -20	Sets a new extended window style .
GWLP_HINSTANCE -6	Sets a new application instance handle.
GWLP_ID -12	Sets a new identifier of the child window. The window cannot be a top-level window.
GWL_STYLE -16	Sets a new window style .
GWLP_USERDATA -21	Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
GWLP_WNDPROC -4	Sets a new address for the window procedure.

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWLP_DLGPROC DWLP_MSGRESULT + sizeof(LRESULT)	Sets the new pointer to the dialog box procedure.
DWLP_MSGRESULT 0	Sets the return value of a message processed in the dialog box procedure.
DWLP_USER DWLP_DLGPROC + sizeof(DLGPROC)	Sets new extra information that is private to the application, such as handles or pointers.

[in] dwNewLong

Type: **LONG_PTR**

The replacement value.

Return value

Type: **LONG_PTR**

If the function succeeds, the return value is the previous value of the specified offset.

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

If the previous value is zero and the function succeeds, the return value is zero, but the function does not clear the last error information. To determine success or failure, clear the last error information by calling [SetLastError](#) with 0, then call [SetWindowLongPtr](#). Function failure will be indicated by a return value of zero and a [GetLastError](#) result that is nonzero.

Remarks

Certain window data is cached, so changes you make using [SetWindowLongPtr](#) will not take effect until you call the [SetWindowPos](#) function.

If you use [SetWindowLongPtr](#) with the **GWLP_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

If you use [SetWindowLongPtr](#) with the **DWLP_MSGRESULT** index to set the return value for a message processed by a dialog box procedure, the dialog box procedure should return **TRUE** directly afterward. Otherwise, if you call any function that results in your dialog box procedure receiving a window message, the nested window message could overwrite the return value you set by using **DWLP_MSGRESULT**.

Calling [SetWindowLongPtr](#) with the **GWLP_WNDPROC** index creates a subclass of the window class used to create the window. An application can subclass a system class, but should not subclass a window class created by another process. The [SetWindowLongPtr](#) function creates the window subclass by changing the window procedure associated with a particular window class, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling [CallWindowProc](#). This allows the application to create a chain of window procedures.

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Do not call [SetWindowLongPtr](#) with the **GWLP_HWNDPARENT** index to change the parent of a child window. Instead, use the [SetParent](#) function.

If the window has a class style of **CS_CLASSDC** or **CS_PARENTDC**, do not set the extended window styles **WS_EX_COMPOSITED** or **WS_EX_LAYERED**.

Calling **SetWindowLongPtr** to set the style on a progressbar will reset its position.

 **Note**

The winuser.h header defines **SetWindowLongPtr** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[GetWindowLongPtr](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

WNDCLASSEX

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowLongPtrW function (winuser.h)

Article02/09/2023

Changes an attribute of the specified window. The function also sets a value at the specified offset in the extra window memory.

Note To write code that is compatible with both 32-bit and 64-bit versions of Windows, use **SetWindowLongPtr**. When compiling for 32-bit Windows, **SetWindowLongPtr** is defined as a call to the **SetWindowLong** function.

Syntax

C++

```
LONG_PTR SetWindowLongPtrW(  
    [in] HWND      hWnd,  
    [in] int       nIndex,  
    [in] LONG_PTR  dwNewLong  
) ;
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window and, indirectly, the class to which the window belongs. The **SetWindowLongPtr** function fails if the process that owns the window specified by the *hWnd* parameter is at a higher process privilege in the UIPI hierarchy than the process the calling thread resides in.

Windows XP/2000: The **SetWindowLongPtr** function fails if the window specified by the *hWnd* parameter does not belong to the same process as the calling thread.

[in] nIndex

Type: **int**

The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of a **LONG_PTR**. To set any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE -20	Sets a new extended window style .
GWLP_HINSTANCE -6	Sets a new application instance handle.
GWLP_ID -12	Sets a new identifier of the child window. The window cannot be a top-level window.
GWL_STYLE -16	Sets a new window style .
GWLP_USERDATA -21	Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
GWLP_WNDPROC -4	Sets a new address for the window procedure.

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWLP_DLGPROC DWLP_MSGRESULT + sizeof(LRESULT)	Sets the new pointer to the dialog box procedure.
DWLP_MSGRESULT 0	Sets the return value of a message processed in the dialog box procedure.
DWLP_USER DWLP_DLGPROC + sizeof(DLGPROC)	Sets new extra information that is private to the application, such as handles or pointers.

[in] dwNewLong

Type: **LONG_PTR**

The replacement value.

Return value

Type: **LONG_PTR**

If the function succeeds, the return value is the previous value of the specified offset.

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

If the previous value is zero and the function succeeds, the return value is zero, but the function does not clear the last error information. To determine success or failure, clear the last error information by calling [SetLastError](#) with 0, then call [SetWindowLongPtr](#). Function failure will be indicated by a return value of zero and a [GetLastError](#) result that is nonzero.

Remarks

Certain window data is cached, so changes you make using [SetWindowLongPtr](#) will not take effect until you call the [SetWindowPos](#) function.

If you use [SetWindowLongPtr](#) with the **GWLP_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

If you use [SetWindowLongPtr](#) with the **DWLP_MSGRESULT** index to set the return value for a message processed by a dialog box procedure, the dialog box procedure should return **TRUE** directly afterward. Otherwise, if you call any function that results in your dialog box procedure receiving a window message, the nested window message could overwrite the return value you set by using **DWLP_MSGRESULT**.

Calling [SetWindowLongPtr](#) with the **GWLP_WNDPROC** index creates a subclass of the window class used to create the window. An application can subclass a system class, but should not subclass a window class created by another process. The [SetWindowLongPtr](#) function creates the window subclass by changing the window procedure associated with a particular window class, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling [CallWindowProc](#). This allows the application to create a chain of window procedures.

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

Do not call [SetWindowLongPtr](#) with the **GWLP_HWNDPARENT** index to change the parent of a child window. Instead, use the [SetParent](#) function.

If the window has a class style of **CS_CLASSDC** or **CS_PARENTDC**, do not set the extended window styles **WS_EX_COMPOSITED** or **WS_EX_LAYERED**.

Calling **SetWindowLongPtr** to set the style on a progressbar will reset its position.

 **Note**

The winuser.h header defines **SetWindowLongPtr** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[GetWindowLongPtr](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

WNDCLASSEX

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowLongW function (winuser.h)

Article 02/09/2023

Changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory.

Note This function has been superseded by the [SetWindowLongPtr](#) function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use the [SetWindowLongPtr](#) function.

Syntax

C++

```
LONG SetWindowLongW(
    [in] HWND hWnd,
    [in] int nIndex,
    [in] LONG dwNewLong
);
```

Parameters

[in] hWnd

Type: [HWND](#)

A handle to the window and, indirectly, the class to which the window belongs.

[in] nIndex

Type: [int](#)

The zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus the size of an integer. To set any other value, specify one of the following values.

Value	Meaning
GWL_EXSTYLE	Sets a new extended window style .
-20	

GWL_HINSTANCE	Sets a new application instance handle.
-6	
GWL_ID	Sets a new identifier of the child window. The window cannot be a top-level window.
-12	
GWL_STYLE	Sets a new window style .
-16	
GWL_USERDATA	Sets the user data associated with the window. This data is intended for use by the application that created the window. Its value is initially zero.
-21	
GWL_WNDPROC	Sets a new address for the window procedure. You cannot change this attribute if the window does not belong to the same process as the calling thread.
-4	

The following values are also available when the *hWnd* parameter identifies a dialog box.

Value	Meaning
DWL_DLGPROC DWLP_MSGRESULT + sizeof(LRESULT)	Sets the new address of the dialog box procedure.
DWL_MSGRESULT 0	Sets the return value of a message processed in the dialog box procedure.
DWL_USER DWLP_DLGPROC + sizeof(DLGPROC)	Sets new extra information that is private to the application, such as handles or pointers.

[in] `dwNewLong`

Type: **LONG**

The replacement value.

Return value

Type: **LONG**

If the function succeeds, the return value is the previous value of the specified 32-bit integer.

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

If the previous value of the specified 32-bit integer is zero, and the function succeeds, the return value is zero, but the function does not clear the last error information. This makes it difficult to determine success or failure. To deal with this, you should clear the last error information by calling [SetLastError](#) with 0 before calling [SetWindowLong](#). Then, function failure will be indicated by a return value of zero and a [GetLastError](#) result that is nonzero.

Remarks

Certain window data is cached, so changes you make using [SetWindowLong](#) will not take effect until you call the [SetWindowPos](#) function. Specifically, if you change any of the frame styles, you must call [SetWindowPos](#) with the **SWP_FRAMECHANGED** flag for the cache to be updated properly.

If you use [SetWindowLong](#) with the **GWL_WNDPROC** index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the [WindowProc](#) callback function.

If you use [SetWindowLong](#) with the **DWL_MSGRESULT** index to set the return value for a message processed by a dialog procedure, you should return **TRUE** directly afterward. Otherwise, if you call any function that results in your dialog procedure receiving a window message, the nested window message could overwrite the return value you set using **DWL_MSGRESULT**.

Calling [SetWindowLong](#) with the **GWL_WNDPROC** index creates a subclass of the window class used to create the window. An application can subclass a system class, but should not subclass a window class created by another process. The [SetWindowLong](#) function creates the window subclass by changing the window procedure associated with a particular window class, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling [CallWindowProc](#). This allows the application to create a chain of window procedures.

Reserve extra window memory by specifying a nonzero value in the **cbWndExtra** member of the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function.

You must not call [SetWindowLong](#) with the **GWL_HWNDPARENT** index to change the parent of a child window. Instead, use the [SetParent](#) function.

If the window has a class style of **CS_CLASSDC** or **CS_OWNDC**, do not set the extended window styles **WS_EX_COMPOSITED** or **WS_EX_LAYERED**.

Calling [SetWindowLong](#) to set the style on a progressbar will reset its position.

Examples

For an example, see [Subclassing a Window](#).

Note

The winuser.h header defines SetWindowLong 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[CallWindowProc](#)

[Conceptual](#)

[GetWindowLong](#)

[Reference](#)

[RegisterClassEx](#)

[SetParent](#)

[SetWindowLongPtr](#)

WNDCLASSEX

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowPlacement function (winuser.h)

Article 10/13/2021

Sets the show state and the restored, minimized, and maximized positions of the specified window.

Syntax

C++

```
BOOL SetWindowPlacement(
    [in] HWND                 hWnd,
    [in] const WINDOWPLACEMENT *lpwndpl
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[in] lpwndpl

Type: **const WINDOWPLACEMENT***

A pointer to a **WINDOWPLACEMENT** structure that specifies the new show state and window positions.

Before calling **SetWindowPlacement**, set the **length** member of the **WINDOWPLACEMENT** structure to **sizeof(WINDOWPLACEMENT)**.

SetWindowPlacement fails if the **length** member is not set correctly.

Return value

Type: **BOOL**

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 information specified in [WINDOWPLACEMENT](#) would result in a window that is completely off the screen, the system will automatically adjust the coordinates so that the window is visible, taking into account changes in screen resolution and multiple monitor configuration.

The **length** member of [WINDOWPLACEMENT](#) must be set to `sizeof(WINDOWPLACEMENT)`. If this member is not set correctly, the function returns **FALSE**. For additional remarks on the proper use of window placement coordinates, see [WINDOWPLACEMENT](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

[Conceptual](#)

[GetWindowPlacement](#)

[Reference](#)

[WINDOWPLACEMENT](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowPos function (winuser.h)

Article 10/13/2021

Changes the size, position, and Z order of a child, pop-up, or top-level window. These windows are ordered according to their appearance on the screen. The topmost window receives the highest rank and is the first window in the Z order.

Syntax

C++

```
BOOL SetWindowPos(
    [in]             HWND hWnd,
    [in, optional]   HWND hWndInsertAfter,
    [in]             int  X,
    [in]             int  Y,
    [in]             int  cx,
    [in]             int  cy,
    [in]             UINT uFlags
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[in, optional] hWndInsertAfter

Type: **HWND**

A handle to the window to precede the positioned window in the Z order. This parameter must be a window handle or one of the following values.

Value	Meaning
HWND_BOTTOM (HWND)1	Places the window at the bottom of the Z order. If the <i>hWnd</i> parameter identifies a topmost window, the window loses its topmost status and is placed at the bottom of all other windows.
HWND_NOTOPMOST	Places the window above all non-topmost windows (that

(HWND)-2	is, behind all topmost windows). This flag has no effect if the window is already a non-topmost window.
HWND_TOP (HWND)0	Places the window at the top of the Z order.
HWND_TOPMOST (HWND)-1	Places the window above all non-topmost windows. The window maintains its topmost position even when it is deactivated.

For more information about how this parameter is used, see the following Remarks section.

[in] `x`

Type: int

The new position of the left side of the window, in client coordinates.

[in] `y`

Type: int

The new position of the top of the window, in client coordinates.

[in] `cx`

Type: int

The new width of the window, in pixels.

[in] `cy`

Type: int

The new height of the window, in pixels.

[in] `uFlags`

Type: **UINT**

The window sizing and positioning flags. This parameter can be a combination of the following values.

Value	Meaning
SWP_ASYNCWINDOWPOS 0x4000	If the calling thread and the thread that owns the window are attached to different input queues, the system posts

	<p>the request to the thread that owns the window. This prevents the calling thread from blocking its execution while other threads process the request.</p>
SWP_DEFERERASE 0x2000	Prevents generation of the WM_SYNCPAINT message.
SWP_DRAWFRAME 0x0020	Draws a frame (defined in the window's class description) around the window.
SWP_FRAMECHANGED 0x0020	Applies new frame styles set using the SetWindowLong function. Sends a WM_NCCALCSIZE message to the window, even if the window's size is not being changed. If this flag is not specified, WM_NCCALCSIZE is sent only when the window's size is being changed.
SWP_HIDEWINDOW 0x0080	Hides the window.
SWP_NOACTIVATE 0x0010	Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the topmost or non-topmost group (depending on the setting of the <i>hWndInsertAfter</i> parameter).
SWP_NOCOPYBITS 0x0100	Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
SWP NOMOVE 0x0002	Retains the current position (ignores X and Y parameters).
SWP_NOOWNERZORDER 0x0200	Does not change the owner window's position in the Z order.
SWP_NOREDRAW 0x0008	Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of the window being moved. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.
SWP_NOREPOSITION 0x0200	Same as the SWP_NOOWNERZORDER flag.
SWP_NOSENDCHANGING 0x0400	Prevents the window from receiving the WM_WINDOWPOSCHANGING message.
SWP_NOSIZE 0x0001	Retains the current size (ignores the cx and cy parameters).

SWP_NOZORDER 0x0004	Retains the current Z order (ignores the <i>hWndInsertAfter</i> parameter).
SWP_SHOWWINDOW 0x0040	Displays the window.

Return value

Type: **BOOL**

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

As part of the Vista re-architecture, all services were moved off the interactive desktop into Session 0. *hwnd* and window manager operations are only effective inside a session and cross-session attempts to manipulate the *hwnd* will fail. For more information, see [The Windows Vista Developer Story: Application Compatibility Cookbook](#).

If you have changed certain window data using [SetWindowLong](#), you must call [SetWindowPos](#) for the changes to take effect. Use the following combination for *uFlags*:

`SWP_NOMOVE | SWP_NOSIZE | SWP_NOZORDER | SWP_FRAMECHANGED.`

A window can be made a topmost window either by setting the *hWndInsertAfter* parameter to **HWND_TOPMOST** and ensuring that the **SWP_NOZORDER** flag is not set, or by setting a window's position in the Z order so that it is above any existing topmost windows. When a non-topmost window is made topmost, its owned windows are also made topmost. Its owners, however, are not changed.

If neither the **SWP_NOACTIVATE** nor **SWP_NOZORDER** flag is specified (that is, when the application requests that a window be simultaneously activated and its position in the Z order changed), the value specified in *hWndInsertAfter* is used only in the following circumstances.

- Neither the **HWND_TOPMOST** nor **HWND_NOTOPMOST** flag is specified in *hWndInsertAfter*.
- The window identified by *hWnd* is not the active window.

An application cannot activate an inactive window without also bringing it to the top of the Z order. Applications can change an activated window's position in the Z order

without restrictions, or it can activate a window and then move it to the top of the topmost or non-topmost windows.

If a topmost window is repositioned to the bottom (**HWND_BOTTOM**) of the Z order or after any non-topmost window, it is no longer topmost. When a topmost window is made non-topmost, its owners and its owned windows are also made non-topmost windows.

A non-topmost window can own a topmost window, but the reverse cannot occur. Any window (for example, a dialog box) owned by a topmost window is itself made a topmost window, to ensure that all owned windows stay above their owner.

If an application is not in the foreground, and should be in the foreground, it must call the [SetForegroundWindow](#) function.

To use [SetWindowPos](#) to bring a window to the top, the process that owns the window must have [SetForegroundWindow](#) permission.

Examples

For an example, see [Initializing a Dialog Box](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[MoveWindow](#)

Reference

[SetActiveWindow](#)

[SetForegroundWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowsHookExA function (winuser.h)

Article02/09/2023

Installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.

Syntax

C++

```
HHOOK SetWindowsHookExA(
    [in] int      idHook,
    [in] HOOKPROC lpfn,
    [in] HINSTANCE hmod,
    [in] DWORD     dwThreadId
);
```

Parameters

[in] idHook

Type: int

The type of hook procedure to be installed. This parameter can be one of the following values.

Value	Meaning
WH_CALLWNDPROC 4	Installs a hook procedure that monitors messages before the system sends them to the destination window procedure. For more information, see the CallWindowProcW function/CallWindowProcA function hook procedure.
WH_CALLWNDPROCRET 12	Installs a hook procedure that monitors messages after they have been processed by the destination window procedure. For more information, see the [HOOKPROC callback function](nc-winuser-hookproc.md) hook procedure.

WH_CBT 5	Installs a hook procedure that receives notifications useful to a CBT application. For more information, see the CBTProc hook procedure.
WH_DEBUG 9	Installs a hook procedure useful for debugging other hook procedures. For more information, see the DebugProc hook procedure.
WH_FOREGROUNDIDLE 11	Installs a hook procedure that will be called when the application's foreground thread is about to become idle. This hook is useful for performing low priority tasks during idle time. For more information, see the ForegroundIdleProc hook procedure.
WH_GETMESSAGE 3	Installs a hook procedure that monitors messages posted to a message queue. For more information, see the GetMsgProc hook procedure.
WH_JOURNALPLAYBACK 1	<p>⚠ Warning</p> <p>Windows 11 and newer: Journaling hook APIs are not supported. We recommend using the SendInput TextInput API instead.</p> <p>Installs a hook procedure that posts messages previously recorded by a WH_JOURNALRECORD hook procedure. For more information, see the JournalPlaybackProc hook procedure.</p>
WH_JOURNALRECORD 0	<p>⚠ Warning</p> <p>Windows 11 and newer: Journaling hook APIs are not supported. We recommend using the SendInput TextInput API instead.</p> <p>Installs a hook procedure that records input messages posted to the system message queue. This hook is useful for recording macros. For more information, see the JournalRecordProc hook procedure.</p>
WH_KEYBOARD 2	Installs a hook procedure that monitors keystroke messages. For more information, see the KeyboardProc hook procedure.
WH_KEYBOARD_LL 13	Installs a hook procedure that monitors low-level keyboard input events. For more information, see the

[LowLevelKeyboardProc](#) hook procedure.

WH_MOUSE	Installs a hook procedure that monitors mouse messages.
7	For more information, see the MouseProc hook procedure.
WH_MOUSE_LL	Installs a hook procedure that monitors low-level mouse input events. For more information, see the LowLevelMouseProc hook procedure.
WH_MSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. For more information, see the MessageProc hook procedure.
WH_SHELL	Installs a hook procedure that receives notifications useful to shell applications. For more information, see the ShellProc hook procedure.
WH_SYSMSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. The hook procedure monitors these messages for all applications in the same desktop as the calling thread. For more information, see the SysMsgProc hook procedure.

[in] *lpfn*

Type: **HOOKPROC**

A pointer to the hook procedure. If the *dwThreadId* parameter is zero or specifies the identifier of a thread created by a different process, the *lpfn* parameter must point to a hook procedure in a DLL. Otherwise, *lpfn* can point to a hook procedure in the code associated with the current process.

[in] *hmod*

Type: **HINSTANCE**

A handle to the DLL containing the hook procedure pointed to by the *lpfn* parameter. The *hMod* parameter must be set to **NULL** if the *dwThreadId* parameter specifies a thread created by the current process and if the hook procedure is within the code associated with the current process.

[in] *dwThreadId*

Type: **DWORD**

The identifier of the thread with which the hook procedure is to be associated. For desktop apps, if this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread. For Windows Store apps, see the Remarks section.

Return value

Type: **HHOOK**

If the function succeeds, the return value is the handle to the hook procedure.

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

Remarks

SetWindowsHookEx can be used to inject a DLL into another process. A 32-bit DLL cannot be injected into a 64-bit process, and a 64-bit DLL cannot be injected into a 32-bit process. If an application requires the use of hooks in other processes, it is required that a 32-bit application call **SetWindowsHookEx** to inject a 32-bit DLL into 32-bit processes, and a 64-bit application call **SetWindowsHookEx** to inject a 64-bit DLL into 64-bit processes. The 32-bit and 64-bit DLLs must have different names.

Because hooks run in the context of an application, they must match the "bitness" of the application. If a 32-bit application installs a global hook on 64-bit Windows, the 32-bit hook is injected into each 32-bit process (the usual security boundaries apply). In a 64-bit process, the threads are still marked as "hooked." However, because a 32-bit application must run the hook code, the system executes the hook in the hooking app's context; specifically, on the thread that called **SetWindowsHookEx**. This means that the hooking application must continue to pump messages or it might block the normal functioning of the 64-bit processes.

If a 64-bit application installs a global hook on 64-bit Windows, the 64-bit hook is injected into each 64-bit process, while all 32-bit processes use a callback to the hooking application.

To hook all applications on the desktop of a 64-bit Windows installation, install a 32-bit global hook and a 64-bit global hook, each from appropriate processes, and be sure to keep pumping messages in the hooking application to avoid blocking normal functioning. If you already have a 32-bit global hooking application and it doesn't need to run in each application's context, you may not need to create a 64-bit version.

An error may occur if the *hMod* parameter is **NULL** and the *dwThreadId* parameter is zero or specifies the identifier of a thread created by another process.

Calling the [CallNextHookEx function](#) function to chain to the next hook procedure is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call [CallNextHookEx](#) unless you absolutely need to prevent the notification from being seen by other applications.

Before terminating, an application must call the [UnhookWindowsHookEx function](#) function to free system resources associated with the hook.

The scope of a hook depends on the hook type. Some hooks can be set only with global scope; others can also be set for only a specific thread, as shown in the following table.

Hook	Scope
WH_CALLWNDPROC	Thread or global
WH_CALLWNDPROCRET	Thread or global
WH_CBT	Thread or global
WH_DEBUG	Thread or global
WH_FOREGROUNDDIDLE	Thread or global
WH_GETMESSAGE	Thread or global
WH_JOURNALPLAYBACK	Global only
WH_JOURNALRECORD	Global only
WH_KEYBOARD	Thread or global
WH_KEYBOARD_LL	Global only
WH_MOUSE	Thread or global
WH_MOUSE_LL	Global only
WH_MSGFILTER	Thread or global
WH_SHELL	Thread or global
WH_SYSMSGFILTER	Global only

For a specified hook type, thread hooks are called first, then global hooks. Be aware that the WH_MOUSE, WH_KEYBOARD, WH_JOURNAL*, WH_SHELL, and low-level hooks can be called on the thread that installed the hook rather than the thread processing the

hook. For these hooks, it is possible that both the 32-bit and 64-bit hooks will be called if a 32-bit hook is ahead of a 64-bit hook in the hook chain.

The global hooks are a shared resource, and installing one affects all applications in the same desktop as the calling thread. All global hook functions must be in libraries. Global hooks should be restricted to special-purpose applications or to use as a development aid during application debugging. Libraries that no longer need a hook should remove its hook procedure.

Windows Store app development If dwThreadId is zero, then window hook DLLs are not loaded in-process for the Windows Store app processes and the Windows Runtime broker process unless they are installed by either UIAccess processes (accessibility tools). The notification is delivered on the installer's thread for these hooks:

- WH_JOURNALPLAYBACK
- WH_JOURNALRECORD
- WH_KEYBOARD
- WH_KEYBOARD_LL
- WH_MOUSE
- WH_MOUSE_LL

This behavior is similar to what happens when there is an architecture mismatch between the hook DLL and the target application process, for example, when the hook DLL is 32-bit and the application process 64-bit.

Examples

For an example, see [Installing and Releasing Hook Procedures](#).

Note

The winuser.h header defines SetWindowsHookEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[CBTProc](#)

[CallNextHookEx function](#)

[CallWindowProcW function](#)

[CallWindowProcA function](#)

[HOOKPROC callback function](#)

[*DebugProc*](#)

[ForegroundIdleProc](#)

[GetMsgProc](#)

[Hooks](#)

[JournalPlaybackProc](#)

[JournalRecordProc](#)

[KeyboardProc](#)

[LowLevelKeyboardProc](#)

[LowLevelMouseProc](#)

[MessageProc](#)

[MouseProc](#)

[ShellProc](#)

[SysMsgProc](#)

[UnhookWindowsHookEx function](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowsHookExW function (winuser.h)

Article 06/30/2023

Installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.

Syntax

C++

```
HHOOK SetWindowsHookExW(
    [in] int idHook,
    [in] HOOKPROC lpfn,
    [in] HINSTANCE hmod,
    [in] DWORD dwThreadId
);
```

Parameters

[in] idHook

Type: **int**

The type of hook procedure to be installed. This parameter can be one of the following values.

Value	Meaning
WH_CALLWNDPROC 4	Installs a hook procedure that monitors messages before the system sends them to the destination window procedure. For more information, see the CallWindowProcW function/CallWindowProcA function hook procedure.
WH_CALLWNDPROCRET 12	Installs a hook procedure that monitors messages after they have been processed by the destination window procedure. For more information, see the HOOKPROC callback function hook procedure.
WH_CBT	Installs a hook procedure that receives notifications

5	useful to a CBT application. For more information, see the CBTProc hook procedure.
WH_DEBUG 9	Installs a hook procedure useful for debugging other hook procedures. For more information, see the DebugProc hook procedure.
WH_FOREGROUNDDIDLE 11	Installs a hook procedure that will be called when the application's foreground thread is about to become idle. This hook is useful for performing low priority tasks during idle time. For more information, see the ForegroundIdleProc hook procedure.
WH_GETMESSAGE 3	Installs a hook procedure that monitors messages posted to a message queue. For more information, see the GetMsgProc hook procedure.
WH_JOURNALPLAYBACK 1	<p>⚠ Warning</p> <p>Journaling Hooks APIs are unsupported starting in Windows 11 and will be removed in a future release. Because of this, we highly recommend calling the SendInput TextInput API instead.</p>
WH_JOURNALRECORD 0	<p>⚠ Warning</p> <p>Journaling Hooks APIs are unsupported starting in Windows 11 and will be removed in a future release. Because of this, we highly recommend calling the SendInput TextInput API instead.</p>
WH_KEYBOARD 2	Installs a hook procedure that monitors keystroke messages. For more information, see the KeyboardProc

	hook procedure.
WH_KEYBOARD_LL 13	Installs a hook procedure that monitors low-level keyboard input events. For more information, see the LowLevelKeyboardProc hook procedure.
WH_MOUSE 7	Installs a hook procedure that monitors mouse messages. For more information, see the MouseProc hook procedure.
WH_MOUSE_LL 14	Installs a hook procedure that monitors low-level mouse input events. For more information, see the LowLevelMouseProc hook procedure.
WH_MSGFILTER -1	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. For more information, see the MessageProc hook procedure.
WH_SHELL 10	Installs a hook procedure that receives notifications useful to shell applications. For more information, see the ShellProc hook procedure.
WH_SYSMSGFILTER 6	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. The hook procedure monitors these messages for all applications in the same desktop as the calling thread. For more information, see the SysMsgProc hook procedure.

[in] *lpfn*

Type: **HOOKPROC**

A pointer to the hook procedure. If the *dwThreadId* parameter is zero or specifies the identifier of a thread created by a different process, the *lpfn* parameter must point to a hook procedure in a DLL. Otherwise, *lpfn* can point to a hook procedure in the code associated with the current process.

[in] *hmod*

Type: **HINSTANCE**

A handle to the DLL containing the hook procedure pointed to by the *lpfn* parameter. The *hMod* parameter must be set to **NULL** if the *dwThreadId* parameter specifies a thread created by the current process and if the hook procedure is within the code associated with the current process.

[in] *dwThreadId*

Type: **DWORD**

The identifier of the thread with which the hook procedure is to be associated. For desktop apps, if this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread. For Windows Store apps, see the Remarks section.

Return value

Type: **HHOOK**

If the function succeeds, the return value is the handle to the hook procedure.

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

Remarks

SetWindowsHookEx can be used to inject a DLL into another process. A 32-bit DLL cannot be injected into a 64-bit process, and a 64-bit DLL cannot be injected into a 32-bit process. If an application requires the use of hooks in other processes, it is required that a 32-bit application call **SetWindowsHookEx** to inject a 32-bit DLL into 32-bit processes, and a 64-bit application call **SetWindowsHookEx** to inject a 64-bit DLL into 64-bit processes. The 32-bit and 64-bit DLLs must have different names.

Because hooks run in the context of an application, they must match the "bitness" of the application. If a 32-bit application installs a global hook on 64-bit Windows, the 32-bit hook is injected into each 32-bit process (the usual security boundaries apply). In a 64-bit process, the threads are still marked as "hooked." However, because a 32-bit application must run the hook code, the system executes the hook in the hooking app's context; specifically, on the thread that called **SetWindowsHookEx**. This means that the hooking application must continue to pump messages or it might block the normal functioning of the 64-bit processes.

If a 64-bit application installs a global hook on 64-bit Windows, the 64-bit hook is injected into each 64-bit process, while all 32-bit processes use a callback to the hooking application.

To hook all applications on the desktop of a 64-bit Windows installation, install a 32-bit global hook and a 64-bit global hook, each from appropriate processes, and be sure to keep pumping messages in the hooking application to avoid blocking normal

functioning. If you already have a 32-bit global hooking application and it doesn't need to run in each application's context, you may not need to create a 64-bit version.

An error may occur if the *hMod* parameter is **NULL** and the *dwThreadId* parameter is zero or specifies the identifier of a thread created by another process.

Calling the [CallNextHookEx function](#) function to chain to the next hook procedure is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call **CallNextHookEx** unless you absolutely need to prevent the notification from being seen by other applications.

Before terminating, an application must call the [UnhookWindowsHookEx function](#) function to free system resources associated with the hook.

The scope of a hook depends on the hook type. Some hooks can be set only with global scope; others can also be set for only a specific thread, as shown in the following table.

Hook	Scope
WH_CALLWNDPROC	Thread or global
WH_CALLWNDPROCRET	Thread or global
WH_CBT	Thread or global
WH_DEBUG	Thread or global
WH_FOREGROUNDIDLE	Thread or global
WH_GETMESSAGE	Thread or global
WH_JOURNALPLAYBACK	Global only
WH_JOURNALRECORD	Global only
WH_KEYBOARD	Thread or global
WH_KEYBOARD_LL	Global only
WH_MOUSE	Thread or global
WH_MOUSE_LL	Global only
WH_MSGFILTER	Thread or global
WH_SHELL	Thread or global
WH_SYSMSGFILTER	Global only

For a specified hook type, thread hooks are called first, then global hooks. Be aware that the WH_MOUSE, WH_KEYBOARD, WH_JOURNAL*, WH_SHELL, and low-level hooks can be called on the thread that installed the hook rather than the thread processing the hook. For these hooks, it is possible that both the 32-bit and 64-bit hooks will be called if a 32-bit hook is ahead of a 64-bit hook in the hook chain.

The global hooks are a shared resource, and installing one affects all applications in the same desktop as the calling thread. All global hook functions must be in libraries. Global hooks should be restricted to special-purpose applications or to use as a development aid during application debugging. Libraries that no longer need a hook should remove its hook procedure.

Windows Store apps: If dwThreadId is zero, then window hook DLLs are not loaded in-process for the Windows Store app processes and the Windows Runtime broker process unless they are installed by either UIAccess processes (accessibility tools). The notification is delivered on the installer's thread for these hooks:

- WH_JOURNALPLAYBACK
- WH_JOURNALRECORD
- WH_KEYBOARD
- WH_KEYBOARD_LL
- WH_MOUSE
- WH_MOUSE_LL

This behavior is similar to what happens when there is an architecture mismatch between the hook DLL and the target application process, for example, when the hook DLL is 32-bit and the application process 64-bit.

Examples

For an example, see [Installing and Releasing Hook Procedures](#).

Note

The winuser.h header defines SetWindowsHookEx 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[CBTProc](#)

[CallNextHookEx function](#)

[CallWindowProcW function](#)

[CallWindowProcA function](#)

[HOOKPROC callback function](#)

Conceptual

[*DebugProc*](#)

[ForegroundIdleProc](#)

[GetMsgProc](#)

[Hooks](#)

[JournalPlaybackProc](#)

[JournalRecordProc](#)

[KeyboardProc](#)

[LowLevelKeyboardProc](#)

[LowLevelMouseProc](#)

[MessageProc](#)

[MouseProc](#)

[Reference](#)

[ShellProc](#)

[SysMsgProc](#)

[UnhookWindowsHookEx function](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowTextA function (winuser.h)

Article 02/09/2023

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, **SetWindowText** cannot change the text of a control in another application.

Syntax

C++

```
BOOL SetWindowTextA(
    [in]           HWND   hWnd,
    [in, optional] LPCSTR lpString
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control whose text is to be changed.

[in, optional] lpString

Type: **LPCTSTR**

The new title or control text.

Return value

Type: **BOOL**

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 target window is owned by the current process, **SetWindowText** causes a [WM_SETTEXT](#) message to be sent to the specified window or control. If the control is a list box control created with the **WS_CAPTION** style, however, **SetWindowText** sets the text for the control, not for the list box entries.

To set the text of a control in another process, send the [WM_SETTEXT](#) message directly instead of calling **SetWindowText**.

The **SetWindowText** function does not expand tab characters (ASCII code 0x09). Tab characters are displayed as vertical bar (|) characters.

Examples

For an example, see [Sending a Message](#).

ⓘ Note

The winuser.h header defines **SetWindowText** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[GetWindowText](#)

Reference

[WM_SETTEXT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SetWindowTextW function (winuser.h)

Article 03/11/2023

Changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, **SetWindowText** cannot change the text of a control in another application.

Syntax

C++

```
BOOL SetWindowTextW(
    [in]           HWND      hWnd,
    [in, optional] LPCWSTR  lpString
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window or control whose text is to be changed.

[in, optional] lpString

Type: **LPCWSTR**

The new title or control text.

Return value

Type: **BOOL**

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 target window is owned by the current process, **SetWindowText** causes a [WM_SETTEXT](#) message to be sent to the specified window or control. If the control is a list box control created with the **WS_CAPTION** style, however, **SetWindowText** sets the text for the control, not for the list box entries.

To set the text of a control in another process, send the [WM_SETTEXT](#) message directly instead of calling **SetWindowText**.

The **SetWindowText** function does not expand tab characters (ASCII code 0x09). Tab characters are displayed as vertical bar (|) characters.

Examples

For an example, see [Sending a Message](#).

ⓘ Note

The winuser.h header defines **SetWindowText** 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[GetWindowText](#)

Reference

[WM_SETTEXT](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ShowOwnedPopups function (winuser.h)

Article 10/13/2021

Shows or hides all pop-up windows owned by the specified window.

Syntax

C++

```
BOOL ShowOwnedPopups(
    [in] HWND hWnd,
    [in] BOOL fShow
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window that owns the pop-up windows to be shown or hidden.

[in] fShow

Type: **BOOL**

If this parameter is **TRUE**, all hidden pop-up windows are shown. If this parameter is **FALSE**, all visible pop-up windows are hidden.

Return value

Type: **BOOL**

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

`ShowOwnedPopups` shows only windows hidden by a previous call to `ShowOwnedPopups`. For example, if a pop-up window is hidden by using the `ShowWindow` function, subsequently calling `ShowOwnedPopups` with the *fShow* parameter set to `TRUE` does not cause the window to be shown.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-3 (introduced in Windows 10, version 10.0.10240)

See also

Conceptual

[IsWindowVisible](#)

Reference

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ShowWindow function (winuser.h)

Article 06/06/2023

Sets the specified window's show state.

Syntax

C++

```
BOOL ShowWindow(
    [in] HWND hWnd,
    [in] int nCmdShow
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[in] nCmdShow

Type: **int**

Controls how the window is to be shown. This parameter is ignored the first time an application calls **ShowWindow**, if the program that launched the application provides a **STARTUPINFO** structure. Otherwise, the first time **ShowWindow** is called, the value should be the value obtained by the **WinMain** function in its *nCmdShow* parameter. In subsequent calls, this parameter can be one of the following values.

Value	Meaning
SW_HIDE 0	Hides the window and activates another window.
SW_SHOWNORMAL SW_NORMAL 1	Activates and displays a window. If the window is minimized, maximized, or arranged, the system restores it to its original size and position. An application should specify this flag when displaying the window for the first time.

Value	Meaning
SW_SHOWMINIMIZED 2	Activates the window and displays it as a minimized window.
SW_SHOWMAXIMIZED SW_MAXIMIZE 3	Activates the window and displays it as a maximized window.
SW_SHOWNOACTIVATE 4	Displays a window in its most recent size and position. This value is similar to SW_SHOWNORMAL , except that the window is not activated.
SW_SHOW 5	Activates the window and displays it in its current size and position.
SW_MINIMIZE 6	Minimizes the specified window and activates the next top-level window in the Z order.
SW_SHOWMINNOACTIVE 7	Displays the window as a minimized window. This value is similar to SW_SHOWMINIMIZED , except the window is not activated.
SW_SHOWNA 8	Displays the window in its current size and position. This value is similar to SW_SHOW , except that the window is not activated.
SW_RESTORE 9	Activates and displays the window. If the window is minimized, maximized, or arranged, the system restores it to its original size and position. An application should specify this flag when restoring a minimized window.
SW_SHOWDEFAULT 10	Sets the show state based on the SW_ value specified in the STARTUPINFO structure passed to the CreateProcess function by the program that started the application.
SW_FORCEMINIMIZE 11	Minimizes a window, even if the thread that owns the window is not responding. This flag should only be used when minimizing windows from a different thread.

Return value

Type: **BOOL**

If the window was previously visible, the return value is nonzero.

If the window was previously hidden, the return value is zero.

Remarks

To perform certain special effects when showing or hiding a window, use [AnimateWindow](#).

The first time an application calls **ShowWindow**, it should use the [WinMain](#) function's *nCmdShow* parameter as its *nCmdShow* parameter. Subsequent calls to **ShowWindow** must use one of the values in the given list, instead of the one specified by the [WinMain](#) function's *nCmdShow* parameter.

As noted in the discussion of the *nCmdShow* parameter, the *nCmdShow* value is ignored in the first call to **ShowWindow** if the program that launched the application specifies startup information in the structure. In this case, **ShowWindow** uses the information specified in the [STARTUPINFO](#) structure to show the window. On subsequent calls, the application must call **ShowWindow** with *nCmdShow* set to **SW_SHOWDEFAULT** to use the startup information provided by the program that launched the application. This behavior is designed for the following situations:

- Applications create their main window by calling [CreateWindow](#) with the **WS_VISIBLE** flag set.
- Applications create their main window by calling [CreateWindow](#) with the **WS_VISIBLE** flag cleared, and later call **ShowWindow** with the **SW_SHOW** flag set to make it visible.

Examples

For an example, see [Creating a Main Window](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[AnimateWindow](#)

[Conceptual](#)

[CreateProcess](#)

[CreateWindow](#)

Other Resources

Reference

[STARTUPINFO](#)

[ShowOwnedPopups](#)

[ShowWindowAsync](#)

[WinMain](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

ShowWindowAsync function (winuser.h)

Article10/13/2021

Sets the show state of a window without waiting for the operation to complete.

Syntax

C++

```
BOOL ShowWindowAsync(
    [in] HWND hWnd,
    [in] int nCmdShow
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window.

[in] nCmdShow

Type: **int**

Controls how the window is to be shown. For a list of possible values, see the description of the [ShowWindow](#) function.

Return value

Type: **BOOL**

If the operation was successfully started, the return value is nonzero.

Remarks

This function posts a show-window event to the message queue of the given window. An application can use this function to avoid becoming nonresponsive while waiting for a nonresponsive application to finish processing a show-window event.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[Reference](#)

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SoundSentry function (winuser.h)

Article 06/29/2021

Triggers a visual signal to indicate that a sound is playing.

Syntax

C++

```
BOOL SoundSentry();
```

Return value

Type: **BOOL**

This function returns one of the following values.

Return code	Description
TRUE	The visual signal was or will be displayed correctly.
FALSE	An error prevented the signal from being displayed.

Remarks

Set the notification behavior by calling [SystemParametersInfo](#) with the **SPI_SETSOUNDSENTRY** value.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib

DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

Reference

[SOUNDSENTRY](#)

[SoundSentryProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

STYLESTRUCT structure (winuser.h)

Article04/02/2021

Contains the styles for a window.

Syntax

C++

```
typedef struct tagSTYLESTRUCT {
    DWORD styleOld;
    DWORD styleNew;
} STYLESTRUCT, *LPSTYLESTRUCT;
```

Members

`styleOld`

Type: **DWORD**

The previous styles for a window. For more information, see the Remarks.

`styleNew`

Type: **DWORD**

The new styles for a window. For more information, see the Remarks.

Remarks

The styles in `styleOld` and `styleNew` can be either the window styles (**WS_**) or the extended window styles (**WS_EX_**), depending on the `wParam` of the message that includes **STYLESTRUCT**.

The `styleOld` and `styleNew` members indicate the styles through their bit pattern. Note that several styles are equal to zero; to detect these styles, test for the negation of their inverse style. For example, to see if **WS_EX_LEFT** is set, you test for `~WS_EX_RIGHT`.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Reference](#)

[WM_STYLECHANGED](#)

[WM_STYLECHANGING](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SwitchToThisWindow function (winuser.h)

Article 10/13/2021

[This function is not intended for general use. It may be altered or unavailable in subsequent versions of Windows.]

Switches focus to the specified window and brings it to the foreground.

Syntax

C++

```
void SwitchToThisWindow(
    [in] HWND hwnd,
    [in] BOOL fUnknown
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the window.

[in] fUnknown

Type: **BOOL**

A **TRUE** for this parameter indicates that the window is being switched to using the Alt/Ctrl+Tab key sequence. This parameter should be **FALSE** otherwise.

Return value

None

Remarks

This function is typically called to maintain window z-ordering.

This function was not included in the SDK headers and libraries until Windows XP with Service Pack 1 (SP1) and Windows Server 2003. If you do not have a header file and import library for this function, you can call the function using [LoadLibrary](#) and [GetProcAddress](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[IsWindowVisible](#)

[Reference](#)

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

SystemParametersInfoA function (winuser.h)

Article02/09/2023

Retrieves or sets the value of one of the system-wide parameters. This function can also update the user profile while setting a parameter.

Syntax

C++

```
BOOL SystemParametersInfoA(
    [in]      UINT  uiAction,
    [in]      UINT  uiParam,
    [in, out] PVOID pvParam,
    [in]      UINT  fWinIni
);
```

Parameters

[in] uiAction

Type: **UINT**

The system-wide parameter to be retrieved or set. The possible values are organized in the following tables of related parameters:

- Accessibility parameters
- Desktop parameters
- Icon parameters
- Input parameters
- Menu parameters
- Power parameters
- Screen saver parameters
- Time-out parameters
- UI effect parameters
- Window parameters

The following are the accessibility parameters.

Accessibility parameter	Meaning
-------------------------	---------

SPI_GETACCESSTIMEOUT 0x003C	<p>Retrieves information about the time-out period associated with the accessibility features. The <i>pvParam</i> parameter must point to an ACCESSTIMEOUT structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ACCESSTIMEOUT)</code>.</p>
SPI_GETAUDIODESCRIPTION 0x0074	<p>Determines whether audio descriptions are enabled or disabled. The <i>pvParam</i> parameter is a pointer to an AUDIODESCRIPTION structure. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(AUDIODESCRIPTION)</code>.</p> <p>While it is possible for users who have visual impairments to hear the audio in video content, there is a lot of action in video that does not have corresponding audio. Specific audio description of what is happening in a video helps these users understand the content better. This flag enables you to determine whether audio descriptions have been enabled and in which language.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETCLIENTAREAANIMATION 0x1042	<p>Determines whether animations are enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if animations are enabled, or FALSE otherwise.</p> <p>Display features such as flashing, blinking, flickering, and moving content can cause seizures in users with photo-sensitive epilepsy. This flag enables you to determine whether such animations have been disabled in the client area.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETDISABLEOVERLAPPEDCONTENT 0x1040	<p>Determines whether overlapped content is enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Display features such as background images, textured backgrounds, water marks on documents, alpha blending, and transparency can reduce the contrast between the foreground and background, making it harder for users with low vision to see objects on the screen. This flag enables you to determine whether such overlapped content has been disabled.</p>

		Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_GETFILTERKEYS 0x0032		Retrieves information about the FilterKeys accessibility feature. The <i>pvParam</i> parameter must point to a FILTERKEYS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(FILTERKEYS)</code> .
SPI_GETFOCUSBORDERHEIGHT 0x2010		Retrieves the height, in pixels, of the top and bottom edges of the focus rectangle drawn with DrawFocusRect . The <i>pvParam</i> parameter must point to a UINT value. Windows 2000: This parameter is not supported.
SPI_GETFOCUSBORDERWIDTH 0x200E		Retrieves the width, in pixels, of the left and right edges of the focus rectangle drawn with DrawFocusRect . The <i>pvParam</i> parameter must point to a UINT . Windows 2000: This parameter is not supported.
SPI_GETHIGHCONTRAST 0x0042		Retrieves information about the HighContrast accessibility feature. The <i>pvParam</i> parameter must point to a HIGHCONTRAST structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(HIGHCONTRAST)</code> . For a general discussion, see Remarks.
SPI_GETLOGICALDPIOVERRIDE 0x009E		Retrieves a value that determines whether Windows 8 is displaying apps using the default scaling plateau for the hardware or going to the next higher plateau. This value is based on the current "Make everything on your screen bigger" setting, found in the Ease of Access section of PC settings : 1 is on, 0 is off. Apps can provide text and image resources for each of several scaling plateaus: 100%, 140%, and 180%. Providing separate resources optimized for a particular scale avoids distortion due to resizing. Windows 8 determines the appropriate scaling plateau based on a number of factors, including screen size and pixel density. When "Make everything on your screen bigger" is selected (SPI_GETLOGICALDPIOVERRIDE returns a value of 1), Windows uses resources from the next higher plateau. For example, in the case of hardware that Windows determines should use a scale of SCALE_100_PERCENT , this override causes Windows to

use the [SCALE_140_PERCENT](#) scale value, assuming that it does not violate other constraints.

Note You should not use this value. It might be altered or unavailable in subsequent versions of Windows. Instead, use the [GetScaleFactorForDevice](#) function or the [DisplayProperties](#) class to retrieve the preferred scaling factor. Desktop applications should use desktop logical DPI rather than scale factor. Desktop logical DPI can be retrieved through the [GetDeviceCaps](#) function.

SPI_GETMESSAGEDURATION 0x2016	<p>Retrieves the time that notification pop-ups should be displayed, in seconds. The <i>pvParam</i> parameter must point to a ULONG that receives the message duration.</p> <p>Users with visual impairments or cognitive conditions such as ADHD and dyslexia might need a longer time to read the text in notification messages. This flag enables you to retrieve the message duration.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETMOUSECLICKLOCK 0x101E	<p>Retrieves the state of the Mouse ClickLock feature. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_GETMOUSECLICKLOCKTIME 0x2008	<p>Retrieves the time delay before the primary mouse button is locked. The <i>pvParam</i> parameter must point to DWORD that receives the time delay, in milliseconds. This is only enabled if SPI_SETMOUSECLICKLOCK is set to TRUE. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_GETMOUSEKEYS 0x0036	<p>Retrieves information about the MouseKeys accessibility feature. The <i>pvParam</i> parameter must point to a MOUSEKEYS structure that receives the</p>

information. Set the **cbSize** member of this structure and the *uiParam* parameter to `sizeof(MOUSEKEYS)`.

SPI_GETMOUSESONAR 0x101C	Retrieves the state of the Mouse Sonar feature. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE otherwise. For more information, see Mouse Input Overview . Windows 2000: This parameter is not supported.
SPI_GETMOUSEVANISH 0x1020	Retrieves the state of the Mouse Vanish feature. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE otherwise. For more information, see Mouse Input Overview . Windows 2000: This parameter is not supported.
SPI_GETSCREENREADER 0x0046	Determines whether a screen reviewer utility is running. A screen reviewer utility directs textual information to an output device, such as a speech synthesizer or Braille display. When this flag is set, an application should provide textual information in situations where it would otherwise present the information graphically. The <i>pvParam</i> parameter is a pointer to a BOOL variable that receives TRUE if a screen reviewer utility is running, or FALSE otherwise. <div style="border: 1px solid #ccc; padding: 10px; border-radius: 10px;"><p>Note Narrator, the screen reader that is included with Windows, does not set the SPI_SETSCREENREADER or SPI_GETSCREENREADER flags.</p></div>
SPI_GETSERIALKEYS 0x003E	This parameter is not supported. Windows Server 2003 and Windows XP/2000: The user should control this setting through the Control Panel.
SPI_GETSHOWSOUNDS 0x0038	Determines whether the Show Sounds accessibility flag is on or off. If it is on, the user requires an application to present information visually in situations where it would otherwise present the information only in audible form. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the feature is on, or FALSE if it is off. Using this value is equivalent to calling GetSystemMetrics with SM_SHOWSOUNDS . That is

the recommended call.

SPI_GETSOUNDSENTRY 0x0040	Retrieves information about the SoundSentry accessibility feature. The <i>pvParam</i> parameter must point to a SOUNDSENTRY structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(SOUNDSENTRY)</code> .
SPI_GETSTICKYKEYS 0x003A	Retrieves information about the StickyKeys accessibility feature. The <i>pvParam</i> parameter must point to a STICKYKEYS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(STICKYKEYS)</code> .
SPI_GETTOGGLEKEYS 0x0034	Retrieves information about the ToggleKeys accessibility feature. The <i>pvParam</i> parameter must point to a TOGGLEKEYS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(TOGGLEKEYS)</code> .
SPI_SETACCESSTIMEOUT 0x003D	Sets the time-out period associated with the accessibility features. The <i>pvParam</i> parameter must point to an ACCESSTIMEOUT structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ACCESSTIMEOUT)</code> .
SPI_SETAUDIODESCRIPTION 0x0075	Turns the audio descriptions feature on or off. The <i>pvParam</i> parameter is a pointer to an AUDIODESCRIPTION structure. While it is possible for users who are visually impaired to hear the audio in video content, there is a lot of action in video that does not have corresponding audio. Specific audio description of what is happening in a video helps these users understand the content better. This flag enables you to enable or disable audio descriptions in the languages they are provided in. Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETCLIENTAREAANIMATION 0x1043	Turns client area animations on or off. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to enable animations and other transient effects in the client area, or FALSE to disable them. Display features such as flashing, blinking, flickering, and moving content can cause seizures in users with

	<p>photo-sensitive epilepsy. This flag enables you to enable or disable all such animations.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETDISABLEOVERLAPPEDCONTENT 0x1041	<p>Turns overlapped content (such as background images and watermarks) on or off. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to disable overlapped content, or FALSE to enable overlapped content.</p> <p>Display features such as background images, textured backgrounds, water marks on documents, alpha blending, and transparency can reduce the contrast between the foreground and background, making it harder for users with low vision to see objects on the screen. This flag enables you to enable or disable all such overlapped content.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETFILTERKEYS 0x0033	<p>Sets the parameters of the FilterKeys accessibility feature. The <i>pvParam</i> parameter must point to a FILTERKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(FILTERKEYS)</code>.</p>
SPI_SETFOCUSBORDERHEIGHT 0x2011	<p>Sets the height of the top and bottom edges of the focus rectangle drawn with DrawFocusRect to the value of the <i>pvParam</i> parameter.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETFOCUSBORDERWIDTH 0x200F	<p>Sets the height of the left and right edges of the focus rectangle drawn with DrawFocusRect to the value of the <i>pvParam</i> parameter.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETHIGHCONTRAST 0x0043	<p>Sets the parameters of the HighContrast accessibility feature. The <i>pvParam</i> parameter must point to a HIGHCONTRAST structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(HIGHCONTRAST)</code>.</p>
SPI_SETLOGICALDPIOVERRIDE 0x009F	Do not use.
SPI_SETMESSAGEDURATION 0x2017	Sets the time that notification pop-ups should be displayed, in seconds. The <i>pvParam</i> parameter

	<p>specifies the message duration.</p> <p>Users with visual impairments or cognitive conditions such as ADHD and dyslexia might need a longer time to read the text in notification messages. This flag enables you to set the message duration.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETMOUSECLICKLOCK 0x101F	<p>Turns the Mouse ClickLock accessibility feature on or off. This feature temporarily locks down the primary mouse button when that button is clicked and held down for the time specified by SPI_SETMOUSECLICKLOCKTIME. The <i>pvParam</i> parameter specifies TRUE for on, or FALSE for off. The default is off. For more information, see Remarks and AboutMouse Input.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSECLICKLOCKTIME 0x2009	<p>Adjusts the time delay before the primary mouse button is locked. The <i>uiParam</i> parameter should be set to 0. The <i>pvParam</i> parameter points to a DWORD that specifies the time delay in milliseconds. For example, specify 1000 for a 1 second delay. The default is 1200. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSEKEYS 0x0037	<p>Sets the parameters of the MouseKeys accessibility feature. The <i>pvParam</i> parameter must point to a MOUSEKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(MOUSEKEYS)</code>.</p>
SPI_SETMOUSESONAR 0x101D	<p>Turns the Sonar accessibility feature on or off. This feature briefly shows several concentric circles around the mouse pointer when the user presses and releases the CTRL key. The <i>pvParam</i> parameter specifies TRUE for on and FALSE for off. The default is off. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSEVANISH 0x1021	<p>Turns the Vanish feature on or off. This feature hides the mouse pointer when the user types; the pointer reappears when the user moves the mouse. The <i>pvParam</i> parameter specifies TRUE for on and FALSE for off. The default is off. For more information, see Mouse Input Overview.</p>

Windows 2000: This parameter is not supported.

SPI_SETSCREENREADER 0x0047	Determines whether a screen review utility is running. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
	<p>Note Narrator, the screen reader that is included with Windows, does not set the SPI_SETSCREENREADER or SPI_GETSCREENREADER flags.</p>
SPI_SETSERIALKEYS 0x003F	This parameter is not supported. Windows Server 2003 and Windows XP/2000: The user should control this setting through the Control Panel.
SPI_SETSHOWSOUNDS 0x0039	Turns the ShowSounds accessibility feature on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
SPI_SETSOUNDSENTRY 0x0041	Sets the parameters of the SoundSentry accessibility feature. The <i>pvParam</i> parameter must point to a SOUNDSENTRY structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(SOUNDSENTRY)</code> .
SPI_SETSTICKYKEYS 0x003B	Sets the parameters of the StickyKeys accessibility feature. The <i>pvParam</i> parameter must point to a STICKYKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(STICKYKEYS)</code> .
SPI_SETTOGGLEKEYS 0x0035	Sets the parameters of the ToggleKeys accessibility feature. The <i>pvParam</i> parameter must point to a TOGGLEKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(TOGGLEKEYS)</code> .

The following are the desktop parameters.

Desktop parameter	Meaning
SPI_GETCLEARATYPE 0x1048	Determines whether ClearType is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if ClearType is enabled, or FALSE otherwise.

		ClearType is a software technology that improves the readability of text on liquid crystal display (LCD) monitors.
		Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_GETDESKWALLPAPER 0x0073		Retrieves the full path of the bitmap file for the desktop wallpaper. The <i>pvParam</i> parameter must point to a buffer to receive the null-terminated path string. Set the <i>uiParam</i> parameter to the size, in characters, of the <i>pvParam</i> buffer. The returned string will not exceed MAX_PATH characters. If there is no desktop wallpaper, the returned string is empty.
SPI_GETDROPSHADOW 0x1024		Determines whether the drop shadow effect is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that returns TRUE if enabled or FALSE if disabled.
		Windows 2000: This parameter is not supported.
SPI_GETFLATMENU 0x1022		Determines whether native User menus have flat menu appearance. The <i>pvParam</i> parameter must point to a BOOL variable that returns TRUE if the flat menu appearance is set, or FALSE otherwise.
		Windows 2000: This parameter is not supported.
SPI_GETFONTSMOOTHING 0x004A		Determines whether the font smoothing feature is enabled. This feature uses font antialiasing to make font curves appear smoother by painting pixels at different gray levels. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the feature is enabled, or FALSE if it is not.
SPI_GETFONTSMOOTHINGCONTRAST 0x200C		Retrieves a contrast value that is used in ClearType smoothing. The <i>pvParam</i> parameter must point to a UINT that receives the information. Valid contrast values are from 1000 to 2200. The default value is 1400.
		Windows 2000: This parameter is not supported.
SPI_GETFONTSMOOTHINGORIENTATION 0x2012		Retrieves the font smoothing orientation. The <i>pvParam</i> parameter must point to a UINT that receives the information. The possible values are FE_FONTSMOOTHINGORIENTATIONBGR (blue-green-red) and FE_FONTSMOOTHINGORIENTATIONRGB (red-green-blue).

		Windows XP/2000: This parameter is not supported until Windows XP with SP2.
SPI_GETFONTSMOOTHINGTYPE 0x200A	Retrieves the type of font smoothing. The <i>pvParam</i> parameter must point to a UINT that receives the information. The possible values are FE_FONTSMOOTHINGSTANDARD and FE_FONTSMOOTHINGCLEARATYPE .	Windows 2000: This parameter is not supported.
SPI_GETWORKAREA 0x0030	Retrieves the size of the work area on the primary display monitor. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars. The <i>pvParam</i> parameter must point to a RECT structure that receives the coordinates of the work area, expressed in physical pixel size. Any DPI virtualization mode of the caller has no effect on this output.	To get the work area of a monitor other than the primary display monitor, call the GetMonitorInfo function.
SPI_SETCLEARATYPE 0x1049	Turns ClearType on or off. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to enable ClearType, or FALSE to disable it.	ClearType is a software technology that improves the readability of text on LCD monitors.
		Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETCURSORS 0x0057	Reloads the system cursors. Set the <i>uiParam</i> parameter to zero and the <i>pvParam</i> parameter to NULL .	
SPI_SETDESKPATTERN 0x0015	Sets the current desktop pattern by causing Windows to read the Pattern= setting from the WIN.INI file.	
SPI_SETDESKWALLPAPER 0x0014	<p>Note When the SPI_SETDESKWALLPAPER flag is used, SystemParametersInfo returns TRUE unless there is an error (like when the specified file doesn't exist).</p>	
SPI_SETDROPSHADOW	Enables or disables the drop shadow effect. Set	

0x1025	<p><i>pvParam</i> to TRUE to enable the drop shadow effect or FALSE to disable it. You must also have CS_DROPSHADOW in the window class style.</p>
	<p>Windows 2000: This parameter is not supported.</p>
SPI_SETFLATMENU 0x1023	<p>Enables or disables flat menu appearance for native User menus. Set <i>pvParam</i> to TRUE to enable flat menu appearance or FALSE to disable it.</p> <p>When enabled, the menu bar uses COLOR_MENUBAR for the menubar background, COLOR_MENU for the menu-popup background, COLOR_MENUHIGHLIGHT for the fill of the current menu selection, and COLOR_HIGHLIGHT for the outline of the current menu selection. If disabled, menus are drawn using the same metrics and colors as in Windows 2000.</p>
	<p>Windows 2000: This parameter is not supported.</p>
SPI_SETFONTSMOOTHING 0x004B	<p>Enables or disables the font smoothing feature, which uses font antialiasing to make font curves appear smoother by painting pixels at different gray levels.</p> <p>To enable the feature, set the <i>uiParam</i> parameter to TRUE. To disable the feature, set <i>uiParam</i> to FALSE.</p>
SPI_SETFONTSMOOTHINGCONTRAST 0x200D	<p>Sets the contrast value used in ClearType smoothing. The <i>pvParam</i> parameter is the contrast value. Valid contrast values are from 1000 to 2200. The default value is 1400.</p> <p>SPI_SETFONTSMOOTHINGTYPE must also be set to FE_FONTSMOOTHINGCLEARTEXT.</p>
	<p>Windows 2000: This parameter is not supported.</p>
SPI_SETFONTSMOOTHINGORIENTATION 0x2013	<p>Sets the font smoothing orientation. The <i>pvParam</i> parameter is either FE_FONTSMOOTHINGORIENTATIONBGR (blue-green-red) or FE_FONTSMOOTHINGORIENTATIONRGB (red-green-blue).</p>
	<p>Windows XP/2000: This parameter is not supported until Windows XP with SP2.</p>
SPI_SETFONTSMOOTHINGTYPE 0x200B	<p>Sets the font smoothing type. The <i>pvParam</i> parameter is either FE_FONTSMOOTHINGSTANDARD, if standard anti-aliasing is used, or</p>

`FE_FONTSMOOTHINGCLEARTEXT`, if [ClearType](#) is used. The default is `FE_FONTSMOOTHINGSTANDARD`.

`SPI_SETFONTSMOOTHING` must also be set.

Windows 2000: This parameter is not supported.

SPI_SETWORKAREA

0x002F

Sets the size of the work area. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars. The *pvParam* parameter is a pointer to a [RECT](#) structure that specifies the new work area rectangle, expressed in virtual screen coordinates. In a system with multiple display monitors, the function sets the work area of the monitor that contains the specified rectangle.

The following are the icon parameters.

Icon parameter	Meaning
SPI_GETICONMETRICS 0x002D	Retrieves the metrics associated with icons. The <i>pvParam</i> parameter must point to an ICONMETRICS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ICONMETRICS)</code> .
SPI_GETicontitlelogfont 0x001F	Retrieves the logical font information for the current icon-title font. The <i>uiParam</i> parameter specifies the size of a LOGFONT structure, and the <i>pvParam</i> parameter must point to the LOGFONT structure to fill in.
SPI_GETicontitlewrap 0x0019	Determines whether icon-title wrapping is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.
SPI_ICONHORIZONTALSPACING 0x000D	Sets or retrieves the width, in pixels, of an icon cell. The system uses this rectangle to arrange icons in large icon view. To set this value, set <i>uiParam</i> to the new value and set <i>pvParam</i> to NULL . You cannot set this value to less than SM_CXICON . To retrieve this value, <i>pvParam</i> must point to an integer that receives the current value.
SPI_ICONVERTICALSPACING 0x0018	Sets or retrieves the height, in pixels, of an icon cell.

	To set this value, set <i>uiParam</i> to the new value and set <i>pvParam</i> to NULL . You cannot set this value to less than SM_CYICON .
	To retrieve this value, <i>pvParam</i> must point to an integer that receives the current value.
SPI_SETICONMETRICS 0x002E	Sets the metrics associated with icons. The <i>pvParam</i> parameter must point to an ICONMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ICONMETRICS)</code> .
SPI_SETICONS 0x0058	Reloads the system icons. Set the <i>uiParam</i> parameter to zero and the <i>pvParam</i> parameter to NULL .
SPI_SETicontitlelogfont 0x0022	Sets the font that is used for icon titles. The <i>uiParam</i> parameter specifies the size of a LOGFONT structure, and the <i>pvParam</i> parameter must point to a LOGFONT structure.
SPI_Seticontitlewrap 0x001A	Turns icon-title wrapping on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.

The following are the input parameters. They include parameters related to the keyboard, mouse, pen, input language, and the warning beeper.

Input parameter	Meaning
SPI_GETBEEP 0x0001	Determines whether the warning beeper is on. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the beeper is on, or FALSE if it is off.
SPI_GETBLOCKSENDINPUTRESETS 0x1026	Retrieves a BOOL indicating whether an application can reset the screensaver's timer by calling the SendInput function to simulate keyboard or mouse input. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the simulated input will be blocked, or FALSE otherwise.
SPI_GETCONTACTVISUALIZATION 0x2018	Retrieves the current contact visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that receives the setting. For more information, see Contact Visualization .
SPI_GETDEFAULTINPUTLANG 0x0059	Retrieves the input locale identifier for the system default input language. The <i>pvParam</i> parameter must point to an HKL variable that receives this value. For

	<p>more information, see Languages, Locales, and Keyboard Layouts.</p>
SPI_GETGESTUREVISUALIZATION 0x201A	Retrieves the current gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that receives the setting. For more information, see Gesture Visualization .
SPI_GETKEYBOARDCUES 0x100A	Determines whether menu access keys are always underlined. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if menu access keys are always underlined, and FALSE if they are underlined only when the menu is activated by the keyboard.
SPI_GETKEYBOARDDELAY 0x0016	Retrieves the keyboard repeat-delay setting, which is a value in the range from 0 (approximately 250 ms delay) through 3 (approximately 1 second delay). The actual delay associated with each value may vary depending on the hardware. The <i>pvParam</i> parameter must point to an integer variable that receives the setting.
SPI_GETKEYBOARDPREF 0x0044	Determines whether the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the user relies on the keyboard; or FALSE otherwise.
SPI_GETKEYBOARDSPEED 0x000A	Retrieves the keyboard repeat-speed setting, which is a value in the range from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second). The actual repeat rates are hardware-dependent and may vary from a linear scale by as much as 20%. The <i>pvParam</i> parameter must point to a DWORD variable that receives the setting.
SPI_GETMOUSE 0x0003	Retrieves the two mouse threshold values and the mouse acceleration. The <i>pvParam</i> parameter must point to an array of three integers that receives these values. See mouse_event for further information.
SPI_GETMOUSEHOVERHEIGHT 0x0064	Retrieves the height, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEOVER message. The <i>pvParam</i> parameter must point to a UINT variable that receives the height.
SPI_GETMOUSEHOVERTIME 0x0066	Retrieves the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle for

	<p>TrackMouseEvent to generate a WM_MOUSEHOVER message. The <i>pvParam</i> parameter must point to a UINT variable that receives the time.</p>
SPI_GETMOUSEHOVERWIDTH 0x0062	<p>Retrieves the width, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. The <i>pvParam</i> parameter must point to a UINT variable that receives the width.</p>
SPI_GETMOUSESPEED 0x0070	<p>Retrieves the current mouse speed. The mouse speed determines how far the pointer will move based on the distance the mouse moves. The <i>pvParam</i> parameter must point to an integer that receives a value which ranges between 1 (slowest) and 20 (fastest). A value of 10 is the default. The value can be set by an end-user using the mouse control panel application or by an application using SPI_SETMOUSESPEED.</p>
SPI_GETMOUSETRAILS 0x005E	<p>Determines whether the Mouse Trails feature is enabled. This feature improves the visibility of mouse cursor movements by briefly showing a trail of cursors and quickly erasing them.</p> <p>The <i>pvParam</i> parameter must point to an integer variable that receives a value. If the value is zero or 1, the feature is disabled. If the value is greater than 1, the feature is enabled and the value indicates the number of cursors drawn in the trail. The <i>uiParam</i> parameter is not used.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_GETMOUSEWHEELROUTING 0x201C	<p>Retrieves the routing setting for mouse wheel input. The routing setting determines whether mouse wheel input is sent to the app with focus (foreground) or the app under the mouse cursor.</p> <p>The <i>pvParam</i> parameter must point to a DWORD variable that receives the routing option. The <i>uiParam</i> parameter is not used.</p> <p>If the value is zero (MOUSEWHEEL_ROUTING_FOCUS), mouse wheel input is delivered to the app with focus. If the value is 1 (MOUSEWHEEL_ROUTING_HYBRID), mouse wheel input is delivered to the app with focus (desktop apps) or the app under the mouse pointer (Windows Store apps).</p> <p>Starting with Windows 10: If the value is 2 (MOUSEWHEEL_ROUTING_MOUSE_POS), mouse wheel input is delivered to the app under the mouse pointer. This is the new default, and</p>

		MOUSEWHEEL_ROUTING_HYBRID is no longer available in Settings.
SPI_GETPENVISUALIZATION 0x201E		Retrieves the current pen gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that receives the setting. For more information, see Pen Visualization .
SPI_GETSNAPTODEFBUTTON 0x005F		Determines whether the snap-to-default-button feature is enabled. If enabled, the mouse cursor automatically moves to the default button, such as OK or Apply , of a dialog box. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the feature is on, or FALSE if it is off.
SPI_GETSYSTEMLANGUAGEBAR 0x1050		Starting with Windows 8: Determines whether the system language bar is enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the language bar is enabled, or FALSE otherwise.
SPI_GETTHREADLOCALINPUTSETTINGS 0x104E		Starting with Windows 8: Determines whether the active input settings have Local (per-thread, TRUE) or Global (session, FALSE) scope. The <i>pvParam</i> parameter must point to a BOOL variable.
SPI_GETWHEELSCROLLCHARS 0x006C		Retrieves the number of characters to scroll when the horizontal mouse wheel is moved. The <i>pvParam</i> parameter must point to a UINT variable that receives the number of lines. The default value is 3.
SPI_GETWHEELSCROLLLINES 0x0068		Retrieves the number of lines to scroll when the vertical mouse wheel is moved. The <i>pvParam</i> parameter must point to a UINT variable that receives the number of lines. The default value is 3.
SPI_SETBEEP 0x0002		Turns the warning beeper on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
SPI_SETBLOCKSENDINPUTRESETS 0x1027		Determines whether an application can reset the screensaver's timer by calling the SendInput function to simulate keyboard or mouse input. The <i>uiParam</i> parameter specifies TRUE if the screensaver will not be deactivated by simulated input, or FALSE if the screensaver will be deactivated by simulated input.
SPI_SETCONTACTVISUALIZATION 0x2019		Sets the current contact visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that identifies the setting. For more information, see Contact Visualization .

Note If contact visualizations are disabled, gesture visualizations cannot be enabled.

SPI_SETDEFAULTINPUTLANG 0x005A	Sets the default input language for the system shell and applications. The specified language must be displayable using the current system character set. The <i>pvParam</i> parameter must point to an HKL variable that contains the input locale identifier for the default language. For more information, see Languages, Locales, and Keyboard Layouts .
SPI_SETDOUBLECLKTIME 0x0020	<p>Sets the double-click time for the mouse to the value of the <i>uiParam</i> parameter. If the <i>uiParam</i> value is greater than 5000 milliseconds, the system sets the double-click time to 5000 milliseconds.</p> <p>The double-click time is the maximum number of milliseconds that can occur between the first and second clicks of a double-click. You can also call the SetDoubleClickTime function to set the double-click time. To get the current double-click time, call the GetDoubleClickTime function.</p>
SPI_SETDOUBLECLKHEIGHT 0x001E	<p>Sets the height of the double-click rectangle to the value of the <i>uiParam</i> parameter.</p> <p>The double-click rectangle is the rectangle within which the second click of a double-click must fall for it to be registered as a double-click.</p> <p>To retrieve the height of the double-click rectangle, call GetSystemMetrics with the SM_CYDOUBLECLK flag.</p>
SPI_SETDOUBLECLKWIDTH 0x001D	<p>Sets the width of the double-click rectangle to the value of the <i>uiParam</i> parameter.</p> <p>The double-click rectangle is the rectangle within which the second click of a double-click must fall for it to be registered as a double-click.</p> <p>To retrieve the width of the double-click rectangle, call GetSystemMetrics with the SM_CXDOUBLECLK flag.</p>
SPI_SETGESTUREVISUALIZATION 0x201B	Sets the current gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that identifies the setting. For more information, see Gesture Visualization .

Note If contact visualizations are disabled, gesture visualizations cannot be enabled.

SPI_SETKEYBOARDCUES 0x100B	Sets the underlining of menu access key letters. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to always underline menu access keys, or FALSE to underline menu access keys only when the menu is activated from the keyboard.
SPI_SETKEYBOARDDELAY 0x0017	Sets the keyboard repeat-delay setting. The <i>uiParam</i> parameter must specify 0, 1, 2, or 3, where zero sets the shortest delay (approximately 250 ms) and 3 sets the longest delay (approximately 1 second). The actual delay associated with each value may vary depending on the hardware.
SPI_SETKEYBOARDPREF 0x0045	Sets the keyboard preference. The <i>uiParam</i> parameter specifies TRUE if the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden; <i>uiParam</i> is FALSE otherwise.
SPI_SETKEYBOARDSPEED 0x000B	Sets the keyboard repeat-speed setting. The <i>uiParam</i> parameter must specify a value in the range from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second). The actual repeat rates are hardware-dependent and may vary from a linear scale by as much as 20%. If <i>uiParam</i> is greater than 31, the parameter is set to 31.
SPI_SETLANGTOGGLE 0x005B	Sets the hot key set for switching between input languages. The <i>uiParam</i> and <i>pvParam</i> parameters are not used. The value sets the shortcut keys in the keyboard property sheets by reading the registry again. The registry must be set before this flag is used. the path in the registry is HKEY_CURRENT_USER\Keyboard Layout\Toggle . Valid values are "1" = ALT+SHIFT, "2" = CTRL+SHIFT, and "3" = none.
SPI_SETMOUSE 0x0004	Sets the two mouse threshold values and the mouse acceleration. The <i>pvParam</i> parameter must point to an array of three integers that specifies these values. See mouse_event for further information.
SPI_SETMOUSEBUTTONSWAP 0x0021	Swaps or restores the meaning of the left and right mouse buttons. The <i>uiParam</i> parameter specifies TRUE

	<p>to swap the meanings of the buttons, or FALSE to restore their original meanings.</p> <p>To retrieve the current setting, call GetSystemMetrics with the SM_SWAPBUTTON flag.</p>
SPI_SETMOUSEOVERHEIGHT 0x0065	Sets the height, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. Set the <i>uiParam</i> parameter to the new height.
SPI_SETMOUSEHOVERTIME 0x0067	<p>Sets the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle for TrackMouseEvent to generate a WM_MOUSEHOVER message. This is used only if you pass HOVER_DEFAULT in the <i>dwHoverTime</i> parameter in the call to TrackMouseEvent. Set the <i>uiParam</i> parameter to the new time.</p> <p>The time specified should be between USER_TIMER_MAXIMUM and USER_TIMER_MINIMUM. If <i>uiParam</i> is less than USER_TIMER_MINIMUM, the function will use USER_TIMER_MINIMUM. If <i>uiParam</i> is greater than USER_TIMER_MAXIMUM, the function will be USER_TIMER_MAXIMUM.</p> <p>Windows Server 2003 and Windows XP: The operating system does not enforce the use of USER_TIMER_MAXIMUM and USER_TIMER_MINIMUM until Windows Server 2003 with SP1 and Windows XP with SP2.</p>
SPI_SETMOUSEOVERWIDTH 0x0063	Sets the width, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. Set the <i>uiParam</i> parameter to the new width.
SPI_SETMOUSESPEED 0x0071	Sets the current mouse speed. The <i>pvParam</i> parameter is an integer between 1 (slowest) and 20 (fastest). A value of 10 is the default. This value is typically set using the mouse control panel application.
SPI_SETMOUSETRAILS 0x005D	<p>Enables or disables the Mouse Trails feature, which improves the visibility of mouse cursor movements by briefly showing a trail of cursors and quickly erasing them.</p> <p>To disable the feature, set the <i>uiParam</i> parameter to zero or 1. To enable the feature, set <i>uiParam</i> to a value greater than 1 to indicate the number of cursors drawn in the trail.</p>

Windows 2000: This parameter is not supported.

SPI_SETMOUSEWHEELROUTING 0x201D	<p>Sets the routing setting for mouse wheel input. The routing setting determines whether mouse wheel input is sent to the app with focus (foreground) or the app under the mouse cursor.</p> <p>The <i>pvParam</i> parameter must point to a DWORD variable that receives the routing option. Set the <i>uiParam</i> parameter to zero.</p> <p>If the value is zero (MOUSEWHEEL_ROUTING_FOCUS), mouse wheel input is delivered to the app with focus. If the value is 1 (MOUSEWHEEL_ROUTING_HYBRID), mouse wheel input is delivered to the app with focus (desktop apps) or the app under the mouse pointer (Windows Store apps).</p> <p>Starting with Windows 10: If the value is 2 (MOUSEWHEEL_ROUTING_MOUSE_POS), mouse wheel input is delivered to the app under the mouse pointer. This is the new default, and MOUSEWHEEL_ROUTING_HYBRID is no longer available in Settings.</p>
SPI_SETPENVISUALIZATION 0x201F	Sets the current pen gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that identifies the setting. For more information, see Pen Visualization .
SPI_SETSNAPTODEFBUTTON 0x0060	Enables or disables the snap-to-default-button feature. If enabled, the mouse cursor automatically moves to the default button, such as OK or Apply , of a dialog box. Set the <i>uiParam</i> parameter to TRUE to enable the feature, or FALSE to disable it. Applications should use the ShowWindow function when displaying a dialog box so the dialog manager can position the mouse cursor.
SPI_SETSYSTEMLANGUAGEBAR 0x1051	<p>Starting with Windows 8: Turns the legacy language bar feature on or off. The <i>pvParam</i> parameter is a pointer to a BOOL variable. Set <i>pvParam</i> to TRUE to enable the legacy language bar, or FALSE to disable it. The flag is supported on Windows 8 where the legacy language bar is replaced by Input Switcher and therefore turned off by default. Turning the legacy language bar on is provided for compatibility reasons and has no effect on the Input Switcher.</p>
SPI_SETTHREADLOCALINPUTSETTINGS 0x104F	<p>Starting with Windows 8: Determines whether the active input settings have Local (per-thread, TRUE) or</p>

	Global (session, FALSE) scope. The <i>pvParam</i> parameter must be a BOOL variable, casted by PVOID.
SPI_SETWHEELSCROLLCHARS 0x006D	Sets the number of characters to scroll when the horizontal mouse wheel is moved. The number of characters is set from the <i>uiParam</i> parameter.
SPI_SETWHEELSCROLLLINES 0x0069	Sets the number of lines to scroll when the vertical mouse wheel is moved. The number of lines is set from the <i>uiParam</i> parameter. The number of lines is the suggested number of lines to scroll when the mouse wheel is rolled without using modifier keys. If the number is 0, then no scrolling should occur. If the number of lines to scroll is greater than the number of lines viewable, and in particular if it is WHEEL_PAGESCROLL (#defined as UINT_MAX), the scroll operation should be interpreted as clicking once in the page down or page up regions of the scroll bar.

The following are the menu parameters.

Menu parameter	Meaning
SPI_GETMENUDROPALIGNMENT 0x001B	Determines whether pop-up menus are left-aligned or right-aligned, relative to the corresponding menu-bar item. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if right-aligned, or FALSE otherwise.
SPI_GETMENUFADE 0x1012	Determines whether menu fade animation is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE when fade animation is enabled and FALSE when it is disabled. If fade animation is disabled, menus use slide animation. This flag is ignored unless menu animation is enabled, which you can do using the SPI_SETMENUANIMATION flag. For more information, see AnimateWindow .
SPI_GETMENUSHOWDELAY 0x006A	Retrieves the time, in milliseconds, that the system waits before displaying a shortcut menu when the mouse cursor is over a submenu item. The <i>pvParam</i> parameter must point to a DWORD variable that receives the time of the delay.
SPI_SETMENUDROPALIGNMENT 0x001C	Sets the alignment value of pop-up menus. The <i>uiParam</i> parameter specifies TRUE for right alignment, or FALSE for left alignment.

SPI_SETMENUFADE 0x1013	Enables or disables menu fade animation. Set <i>pvParam</i> to TRUE to enable the menu fade effect or FALSE to disable it. If fade animation is disabled, menus use slide animation. The menu fade effect is possible only if the system has a color depth of more than 256 colors. This flag is ignored unless SPI_MENUANIMATION is also set. For more information, see AnimateWindow .
SPI_SETMENUSHOWDELAY 0x006B	Sets <i>uiParam</i> to the time, in milliseconds, that the system waits before displaying a shortcut menu when the mouse cursor is over a submenu item.

The following are the power parameters.

Beginning with Windows Server 2008 and Windows Vista, these power parameters are not supported. Instead, to determine the current display power state, an application should register for **GUID_MONITOR_POWER_STATE** notifications. To determine the current display power down time-out, an application should register for notification of changes to the **GUID_VIDEO_POWERDOWN_TIMEOUT** power setting. For more information, see [Registering for Power Events](#).

Windows Server 2003 and Windows XP/2000: To determine the current display power state, use the following power parameters.

Power parameter	Meaning
SPI_GETLOWPOWERACTIVE 0x0053	This parameter is not supported. Windows Server 2003 and Windows XP/2000: Determines whether the low-power phase of screen saving is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE if disabled. This flag is supported for 32-bit applications only.
SPI_GETLOWPOWERTIMEOUT 0x004F	This parameter is not supported. Windows Server 2003 and Windows XP/2000: Retrieves the time-out value for the low-power phase of screen saving. The <i>pvParam</i> parameter must point to an integer variable that receives the value. This flag is supported for 32-bit applications only.
SPI_GETPOWEROFFACTIVE 0x0054	This parameter is not supported. When the power-off phase of screen saving is enabled, the GUID_VIDEO_POWERDOWN_TIMEOUT power setting is greater than zero.

	Windows Server 2003 and Windows XP/2000: Determines whether the power-off phase of screen saving is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE if disabled. This flag is supported for 32-bit applications only.
SPI_GETPOWEROFFTIMEOUT 0x0050	This parameter is not supported. Instead, check the GUID_VIDEO_POWERDOWN_TIMEOUT power setting. Windows Server 2003 and Windows XP/2000: Retrieves the time-out value for the power-off phase of screen saving. The <i>pvParam</i> parameter must point to an integer variable that receives the value. This flag is supported for 32-bit applications only.
SPI_SETLOWPOWERACTIVE 0x0055	This parameter is not supported. Windows Server 2003 and Windows XP/2000: Activates or deactivates the low-power phase of screen saving. Set <i>uiParam</i> to 1 to activate, or zero to deactivate. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.
SPI_SETLOWPOWERTIMEOUT 0x0051	This parameter is not supported. Windows Server 2003 and Windows XP/2000: Sets the time-out value, in seconds, for the low-power phase of screen saving. The <i>uiParam</i> parameter specifies the new value. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.
SPI_SETPOWEROFFACTIVE 0x0056	This parameter is not supported. Instead, set the GUID_VIDEO_POWERDOWN_TIMEOUT power setting. Windows Server 2003 and Windows XP/2000: Activates or deactivates the power-off phase of screen saving. Set <i>uiParam</i> to 1 to activate, or zero to deactivate. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.
SPI_SETPOWEROFFTIMEOUT 0x0052	This parameter is not supported. Instead, set the GUID_VIDEO_POWERDOWN_TIMEOUT power setting to a time-out value. Windows Server 2003 and Windows XP/2000: Sets the time-out value, in seconds, for the power-off phase of screen saving. The <i>uiParam</i> parameter specifies the new value. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.

The following are the screen saver parameters.

Screen saver parameter	Meaning
SPI_GETSCREENSAVEACTIVE 0x0010	Determines whether screen saving is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if screen saving is enabled, or FALSE otherwise. Windows 7, Windows Server 2008 R2 and Windows 2000: The function returns TRUE even when screen saving is not enabled.
SPI_GETSCREENSAVERRUNNING 0x0072	Determines whether a screen saver is currently running on the window station of the calling process. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if a screen saver is currently running, or FALSE otherwise. Note that only the interactive window station, WinSta0, can have a screen saver running.
SPI_GETSCREENSAVESECURE 0x0076	Determines whether the screen saver requires a password to display the Windows desktop. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the screen saver requires a password, or FALSE otherwise. The <i>uiParam</i> parameter is ignored. Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_GETSCREENSAVETIMEOUT 0x000E	Retrieves the screen saver time-out value, in seconds. The <i>pvParam</i> parameter must point to an integer variable that receives the value.
SPI_SETSCREENSAVEACTIVE 0x0011	Sets the state of the screen saver. The <i>uiParam</i> parameter specifies TRUE to activate screen saving, or FALSE to deactivate it. If the machine has entered power saving mode or system lock state, an ERROR_OPERATION_IN_PROGRESS exception occurs.
SPI_SETSCREENSAVESECURE 0x0077	Sets whether the screen saver requires the user to enter a password to display the Windows desktop. The <i>uiParam</i> parameter is a BOOL variable. The <i>pvParam</i> parameter is ignored. Set <i>uiParam</i> to TRUE to require a password, or FALSE to not require a password. If the machine has entered power saving mode or system lock state, an ERROR_OPERATION_IN_PROGRESS exception occurs.

	Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETSCREENSAVETIMEOUT 0x000F	<p>Sets the screen saver time-out value to the value of the <i>uiParam</i> parameter. This value is the amount of time, in seconds, that the system must be idle before the screen saver activates.</p> <p>If the machine has entered power saving mode or system lock state, an ERROR_OPERATION_IN_PROGRESS exception occurs.</p>
The following are the time-out parameters for applications and services.	
Time-out parameter	Meaning
SPI_GETHUNGAPPTIMEOUT 0x0078	<p>Retrieves the number of milliseconds that a thread can go without dispatching a message before the system considers it unresponsive. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETWAITTOKILLTIMEOUT 0x007A	<p>Retrieves the number of milliseconds that the system waits before terminating an application that does not respond to a shutdown request. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETWAITTOKILLSERVICETIMEOUT 0x007C	<p>Retrieves the number of milliseconds that the service control manager waits before terminating a service that does not respond to a shutdown request. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETHUNGAPPTIMEOUT 0x0079	<p>Sets the hung application time-out to the value of the <i>uiParam</i> parameter. This value is the number of milliseconds that a thread can go without dispatching a message before the system considers it unresponsive.</p>

	Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETWAITTOKILLTIMEOUT 0x007B	Sets the application shutdown request time-out to the value of the <i>uiParam</i> parameter. This value is the number of milliseconds that the system waits before terminating an application that does not respond to a shutdown request.
	Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETWAITTOKILLSERVICETIMEOUT 0x007D	Sets the service shutdown request time-out to the value of the <i>uiParam</i> parameter. This value is the number of milliseconds that the system waits before terminating a service that does not respond to a shutdown request.
	Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.

The following are the UI effects. The **SPI_SETUIEFFECTS** value is used to enable or disable all UI effects at once. This table contains the complete list of UI effect values.

UI effects parameter	Meaning
SPI_GETCOMBOBOXANIMATION 0x1004	Determines whether the slide-open effect for combo boxes is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled.
SPI_GETCURSORSHADOW 0x101A	Determines whether the cursor has a shadow around it. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the shadow is enabled, FALSE if it is disabled. This effect appears only if the system has a color depth of more than 256 colors.
SPI_GETGRADIENTCAPTIONS 0x1008	Determines whether the gradient effect for window title bars is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled. For more information about the gradient effect, see the GetSysColor function.
SPI_GETHOTTRACKING 0x100E	Determines whether hot tracking of user-interface elements, such as menu names on menu bars, is enabled. The <i>pvParam</i> parameter must point to a BOOL

	<p>variable that receives TRUE for enabled, or FALSE for disabled.</p> <p>Hot tracking means that when the cursor moves over an item, it is highlighted but not selected. You can query this value to decide whether to use hot tracking in the user interface of your application.</p>
SPI_GETLISTBOXSMOOTHSCROLLING 0x1006	Determines whether the smooth-scrolling effect for list boxes is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled.
SPI_GETMENUANIMATION 0x1002	<p>Determines whether the menu animation feature is enabled. This master switch must be on to enable menu animation effects. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if animation is enabled and FALSE if it is disabled.</p> <p>If animation is enabled, SPI_GETMENUADE indicates whether menus use fade or slide animation.</p>
SPI_GETMENUUNDERLINES 0x100A	Same as SPI_GETKEYBOARDCUES .
SPI_GETSELECTIONFADE 0x1014	<p>Determines whether the selection fade effect is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE if disabled.</p> <p>The selection fade effect causes the menu item selected by the user to remain on the screen briefly while fading out after the menu is dismissed.</p>
SPI_GETTOOLTIPANIMATION 0x1016	<p>Determines whether ToolTip animation is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE if disabled. If ToolTip animation is enabled, SPI_GETTOOLTIPFADE indicates whether ToolTips use fade or slide animation.</p>
SPI_GETTOOLTIPFADE 0x1018	<p>If SPI_SETTOOLTIPANIMATION is enabled, SPI_GETTOOLTIPFADE indicates whether ToolTip animation uses a fade effect or a slide effect. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for fade animation or FALSE for slide animation. For more information on slide and fade effects, see AnimateWindow.</p>
SPI_GETUIEFFECTS 0x103E	Determines whether UI effects are enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if all UI effects are enabled, or FALSE if they are disabled.

SPI_SETCOMBOBOXANIMATION 0x1005	Enables or disables the slide-open effect for combo boxes. Set the <i>pvParam</i> parameter to TRUE to enable the gradient effect, or FALSE to disable it.
SPI_SETCURSORSHADOW 0x101B	Enables or disables a shadow around the cursor. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to enable the shadow or FALSE to disable the shadow. This effect appears only if the system has a color depth of more than 256 colors.
SPI_SETGRADIENTCAPTIONS 0x1009	Enables or disables the gradient effect for window title bars. Set the <i>pvParam</i> parameter to TRUE to enable it, or FALSE to disable it. The gradient effect is possible only if the system has a color depth of more than 256 colors. For more information about the gradient effect, see the GetSysColor function.
SPI_SETHOTTRACKING 0x100F	Enables or disables hot tracking of user-interface elements such as menu names on menu bars. Set the <i>pvParam</i> parameter to TRUE to enable it, or FALSE to disable it. Hot-tracking means that when the cursor moves over an item, it is highlighted but not selected.
SPI_SETLISTBOXSMOOTHSCROLLING 0x1007	Enables or disables the smooth-scrolling effect for list boxes. Set the <i>pvParam</i> parameter to TRUE to enable the smooth-scrolling effect, or FALSE to disable it.
SPI_SETMENUANIMATION 0x1003	Enables or disables menu animation. This master switch must be on for any menu animation to occur. The <i>pvParam</i> parameter is a BOOL variable; set <i>pvParam</i> to TRUE to enable animation and FALSE to disable animation. If animation is enabled, SPI_GETMENUFADE indicates whether menus use fade or slide animation.
SPI_SETMENUUNDERLINES 0x100B	Same as SPI_SETKEYBOARDCUES .
SPI_SETSELECTIONFADE 0x1015	Set <i>pvParam</i> to TRUE to enable the selection fade effect or FALSE to disable it. The selection fade effect causes the menu item selected by the user to remain on the screen briefly while fading out after the menu is dismissed. The selection fade effect is possible only if the system has a color depth of more than 256 colors.
SPI_SETTOOLTIPANIMATION 0x1017	Set <i>pvParam</i> to TRUE to enable ToolTip animation or FALSE to disable it. If enabled, you can use

SPI_SETTOOLTIPFADE to specify fade or slide animation.

SPI_SETTOOLTIPFADE 0x1019	If the SPI_SETTOOLTIPANIMATION flag is enabled, use SPI_SETTOOLTIPFADE to indicate whether ToolTip animation uses a fade effect or a slide effect. Set <i>pvParam</i> to TRUE for fade animation or FALSE for slide animation. The tooltip fade effect is possible only if the system has a color depth of more than 256 colors. For more information on the slide and fade effects, see the AnimateWindow function.
SPI_SETUIEFFECTS 0x103F	Enables or disables UI effects. Set the <i>pvParam</i> parameter to TRUE to enable all UI effects or FALSE to disable all UI effects.

The following are the window parameters.

Window parameter	Meaning
SPI_GETACTIVEWINDOWTRACKING 0x1000	Determines whether active window tracking (activating the window the mouse is on) is on or off. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for on, or FALSE for off.
SPI_GETACTIVEWNDTRKZORDER 0x100C	Determines whether windows activated through active window tracking will be brought to the top. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for on, or FALSE for off.
SPI_GETACTIVEWNDTRKTIMEOUT 0x2002	Retrieves the active window tracking delay, in milliseconds. The <i>pvParam</i> parameter must point to a DWORD variable that receives the time.
SPI_GETANIMATION 0x0048	Retrieves the animation effects associated with user actions. The <i>pvParam</i> parameter must point to an ANIMATIONINFO structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ANIMATIONINFO)</code> .
SPI_GETBORDER 0x0005	Retrieves the border multiplier factor that determines the width of a window's sizing border. The <i>pvParam</i> parameter must point to an integer variable that receives this value.
SPI_GETCARETWIDTH 0x2006	Retrieves the caret width in edit controls, in pixels. The <i>pvParam</i> parameter must point to a DWORD variable that receives this value.
SPI_GETDOCKMOVING	Determines whether a window is docked when it is

0x0090	<p>moved to the top, left, or right edges of a monitor or monitor array. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETDRAGFROMMAXIMIZE 0x008C	<p>Determines whether a maximized window is restored when its caption bar is dragged. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETDRAGFULLWINDOWS 0x0026	<p>Determines whether dragging of full windows is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p>
SPI_GETFOREGROUNDFLASHCOUNT 0x2004	<p>Retrieves the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request. The <i>pvParam</i> parameter must point to a DWORD variable that receives the value.</p>
SPI_GETFOREGROUNDLOCKTIMEOUT 0x2000	<p>Retrieves the amount of time following user input, in milliseconds, during which the system will not allow applications to force themselves into the foreground. The <i>pvParam</i> parameter must point to a DWORD variable that receives the time.</p>
SPI_GETMINIMIZEDMETRICS 0x002B	<p>Retrieves the metrics associated with minimized windows. The <i>pvParam</i> parameter must point to a MINIMIZEDMETRICS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(MINIMIZEDMETRICS)</code>.</p>
SPI_GETMOUSEDOCKTHRESHOLD 0x007E	<p>Retrieves the threshold in pixels where docking behavior is triggered by using a mouse to drag a window to the edge of a monitor or monitor array. The</p>

	<p>default threshold is 1. The <i>pvParam</i> parameter must point to a DWORD variable that receives the value.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETMOUSEDRAGOUTTHRESHOLD 0x0084	<p>Retrieves the threshold in pixels where undocking behavior is triggered by using a mouse to drag a window from the edge of a monitor or a monitor array toward the center. The default threshold is 20.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETMOUSESIDEMOVETHRESHOLD 0x0088	<p>Retrieves the threshold in pixels from the top of a monitor or a monitor array where a vertically maximized window is restored when dragged with the mouse. The default threshold is 50.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETNONCLIENTMETRICS 0x0029	<p>Retrieves the metrics associated with the nonclient area of nonminimized windows. The <i>pvParam</i> parameter must point to a NONCLIENTMETRICS structure that receives the information. Set the <i>cbSize</i> member of this structure and the <i>uiParam</i> parameter to <code>sizeof(NONCLIENTMETRICS)</code>.</p> <p>Windows Server 2003 and Windows XP/2000: See Remarks for NONCLIENTMETRICS.</p>
SPI_GETPENDOCKTHRESHOLD 0x0080	<p>Retrieves the threshold in pixels where docking behavior is triggered by using a pen to drag a window to the edge of a monitor or monitor array. The default is 30.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is</p>

	not supported.
SPI_GETPENDRAGOUTTHRESHOLD 0x0086	<p>Retrieves the threshold in pixels where undocking behavior is triggered by using a pen to drag a window from the edge of a monitor or monitor array toward its center. The default threshold is 30.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETPENSIDEMOVETHRESHOLD 0x008A	<p>Retrieves the threshold in pixels from the top of a monitor or monitor array where a vertically maximized window is restored when dragged with the mouse. The default threshold is 50.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETSHOWIMEUI 0x006E	Determines whether the IME status window is visible (on a per-user basis). The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the status window is visible, or FALSE if it is not.
SPI_GETSNAPSIZING 0x008E	<p>Determines whether a window is vertically maximized when it is sized to the top or bottom of a monitor or monitor array. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETWINARRANGING 0x0082	<p>Determines whether window arrangement is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Window arrangement reduces the number of mouse, pen, or touch interactions needed to move and size top-level windows by simplifying the default behavior of a window when it is dragged or sized.</p>

	<p>The following parameters retrieve individual window arrangement settings:</p> <p>SPI_GETDOCKMOVING SPI_GETMOUSEDOCKTHRESHOLD SPI_GETMOUSEDRAGOUTTHRESHOLD SPI_GETMOUSESIDEMOVETHRESHOLD SPI_GETPENDOCKTHRESHOLD SPI_GETPENDRAGOUTTHRESHOLD SPI_GETPENSIDEMOVETHRESHOLD SPI_GETSNAPSIZING</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETACTIVEWINDOWTRACKING 0x1001	Sets active window tracking (activating the window the mouse is on) either on or off. Set <i>pvParam</i> to TRUE for on or FALSE for off.
SPI_SETACTIVEWNDTRKZORDER 0x100D	Determines whether or not windows activated through active window tracking should be brought to the top. Set <i>pvParam</i> to TRUE for on or FALSE for off.
SPI_SETACTIVEWNDTRKTIMEOUT 0x2003	Sets the active window tracking delay. Set <i>pvParam</i> to the number of milliseconds to delay before activating the window under the mouse pointer.
SPI_SETANIMATION 0x0049	Sets the animation effects associated with user actions. The <i>pvParam</i> parameter must point to an ANIMATIONINFO structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ANIMATIONINFO)</code> .
SPI_SETBORDER 0x0006	Sets the border multiplier factor that determines the width of a window's sizing border. The <i>uiParam</i> parameter specifies the new value.
SPI_SETCARETWIDTH 0x2007	Sets the caret width in edit controls. Set <i>pvParam</i> to the desired width, in pixels. The default and minimum value is 1.
SPI_SETDOCKMOVING 0x0091	Sets whether a window is docked when it is moved to the top, left, or right docking targets on a monitor or monitor array. Set <i>pvParam</i> to TRUE for on or FALSE for off.
	SPI_GETWINARRANGING must be TRUE to enable this behavior.
	Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.

SPI_SETDRAGFROMMAXIMIZE 0x008D	Sets whether a maximized window is restored when its caption bar is dragged. Set <i>pvParam</i> to TRUE for on or FALSE for off.
	SPI_GETWINARRANGING must be TRUE to enable this behavior.
	Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETDRAGFULLWINDOWS 0x0025	Sets dragging of full windows either on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
SPI_SETDRAGHEIGHT 0x004D	Sets the height, in pixels, of the rectangle used to detect the start of a drag operation. Set <i>uiParam</i> to the new value. To retrieve the drag height, call GetSystemMetrics with the SM_CYDRAG flag.
SPI_SETDRAGWIDTH 0x004C	Sets the width, in pixels, of the rectangle used to detect the start of a drag operation. Set <i>uiParam</i> to the new value. To retrieve the drag width, call GetSystemMetrics with the SM_CXDRAG flag.
SPI_SETFOREGROUNDFLASHCOUNT 0x2005	Sets the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request. Set <i>pvParam</i> to the number of times to flash.
SPI_SETFOREGROUNDLOCKTIMEOUT 0x2001	Sets the amount of time following user input, in milliseconds, during which the system does not allow applications to force themselves into the foreground. Set <i>pvParam</i> to the new time-out value. The calling thread must be able to change the foreground window, otherwise the call fails.
SPI_SETMINIMIZEDMETRICS 0x002C	Sets the metrics associated with minimized windows. The <i>pvParam</i> parameter must point to a MINIMIZEDMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(MINIMIZEDMETRICS)</code> .
SPI_SETMOUSEDOCKTHRESHOLD 0x007F	Sets the threshold in pixels where docking behavior is triggered by using a mouse to drag a window to the edge of a monitor or monitor array. The default threshold is 1. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.

	<p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETMOUSEDRAGOUTTHRESHOLD 0x0085	<p>Sets the threshold in pixels where undocking behavior is triggered by using a mouse to drag a window from the edge of a monitor or monitor array to its center. The default threshold is 20. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETMOUSESIDEMOVETHRESHOLD 0x0089	<p>Sets the threshold in pixels from the top of the monitor where a vertically maximized window is restored when dragged with the mouse. The default threshold is 50. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETNONCLIENTMETRICS 0x002A	<p>Sets the metrics associated with the nonclient area of nonminimized windows. The <i>pvParam</i> parameter must point to a NONCLIENTMETRICS structure that contains the new parameters. Set the <i>cbSize</i> member of this structure and the <i>uiParam</i> parameter to <i>sizeof(NONCLIENTMETRICS)</i>. Also, the <i>IfHeight</i> member of the LOGFONT structure must be a negative value.</p>
SPI_SETPENDOCKTHRESHOLD 0x0081	<p>Sets the threshold in pixels where docking behavior is triggered by using a pen to drag a window to the edge of a monitor or monitor array. The default threshold is 30. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p>

	<p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETPENDRAGOUTTHRESHOLD 0x0087	<p>Sets the threshold in pixels where undocking behavior is triggered by using a pen to drag a window from the edge of a monitor or monitor array to its center. The default threshold is 30. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETPENSIDEMOVEThreshold 0x008B	<p>Sets the threshold in pixels from the top of the monitor where a vertically maximized window is restored when dragged with a pen. The default threshold is 50. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETSHOWIMEUI 0x006F	<p>Sets whether the IME status window is visible or not on a per-user basis. The <i>uiParam</i> parameter specifies TRUE for on or FALSE for off.</p>
SPI_SETSNAPSIZING 0x008F	<p>Sets whether a window is vertically maximized when it is sized to the top or bottom of the monitor. Set <i>pvParam</i> to TRUE for on or FALSE for off.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETWINARRANGING 0x0083	<p>Sets whether window arrangement is enabled. Set <i>pvParam</i> to TRUE for on or FALSE for off.</p> <p>Window arrangement reduces the number of mouse, pen, or touch interactions needed to move and size</p>

top-level windows by simplifying the default behavior of a window when it is dragged or sized.

The following parameters set individual window arrangement settings:

SPI_SETDOCKMOVING

SPI_SETMOUSEDOCKTHRESHOLD

SPI_SETMOUSEDRAGOUTTHRESHOLD

SPI_SETMOUSESIDEMOVETHRESHOLD

SPI_SETPENDOCKTHRESHOLD

SPI_SETPENDRAGOUTTHRESHOLD

SPI_SETPENSIDEMOVETHRESHOLD

SPI_SETSNAPSIZING

Windows Server 2008, Windows Vista, Windows

Server 2003 and Windows XP/2000: This parameter is not supported.

[in] uiParam

Type: **UINT**

A parameter whose usage and format depends on the system parameter being queried or set. For more information about system-wide parameters, see the *uiAction* parameter. If not otherwise indicated, you must specify zero for this parameter.

[in, out] pvParam

Type: **PVOID**

A parameter whose usage and format depends on the system parameter being queried or set. For more information about system-wide parameters, see the *uiAction* parameter. If not otherwise indicated, you must specify **NULL** for this parameter. For information on the **PVOID** datatype, see [Windows Data Types](#).

[in] fWinIni

Type: **UINT**

If a system parameter is being set, specifies whether the user profile is to be updated, and if so, whether the [WM_SETTINGCHANGE](#) message is to be broadcast to all top-level windows to notify them of the change.

This parameter can be zero if you do not want to update the user profile or broadcast the [WM_SETTINGCHANGE](#) message, or it can be one or more of the following values.

Value	Meaning

SPIF_UPDATEINIFILE	Writes the new system-wide parameter setting to the user profile.
SPIF_SENDCHANGE	Broadcasts the WM_SETTINGCHANGE message after updating the user profile.
SPIF_SENDWININICHANGE	Same as SPIF_SENDCHANGE .

Return value

Type: **BOOL**

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

This function is intended for use with applications that allow the user to customize the environment.

A keyboard layout name should be derived from the hexadecimal value of the language identifier corresponding to the layout. For example, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named "00000409". Variants of U.S. English layout, such as the Dvorak layout, are named "00010409", "00020409" and so on. For a list of the primary language identifiers and sublanguage identifiers that make up a language identifier, see the [MAKELANGID](#) macro.

There is a difference between the High Contrast color scheme and the High Contrast Mode. The High Contrast color scheme changes the system colors to colors that have obvious contrast; you switch to this color scheme by using the Display Options in the control panel. The High Contrast Mode, which uses [SPI_GETHIGHCONTRAST](#) and [SPI_SETHIGHCONTRAST](#), advises applications to modify their appearance for visually-impaired users. It involves such things as audible warning to users and customized color scheme (using the Accessibility Options in the control panel). For more information, see [HIGHCONTRAST](#). For more information on general accessibility features, see [Accessibility](#).

During the time that the primary button is held down to activate the Mouse ClickLock feature, the user can move the mouse. After the primary button is locked down, releasing the primary button does not result in a [WM_LBUTTONUP](#) message. Thus, it will appear to an application that the primary button is still down. Any subsequent

button message releases the primary button, sending a **WM_LBUTTONUP** message to the application, thus the button can be unlocked programmatically or through the user clicking any button.

This API is not DPI aware, and should not be used if the calling thread is per-monitor DPI aware. For the DPI-aware version of this API, see [SystemParametersInfoForDPI](#). For more information on DPI awareness, see [the Windows High DPI documentation](#).

Examples

The following example uses [SystemParametersInfo](#) to double the mouse speed.

C++

```
#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "user32.lib")

void main()
{
    BOOL fResult;
    int aMouseInfo[3];      // Array for mouse information

    // Get the current mouse speed.
    fResult = SystemParametersInfo(SPI_GETMOUSE,           // Get mouse information
                                   0,                  // Not used
                                   &aMouseInfo,        // Holds mouse
                                   information          // Not used
                                   0);                // Not used

    // Double it.
    if( fResult )
    {
        aMouseInfo[2] = 2 * aMouseInfo[2];

        // Change the mouse speed to the new value.
        SystemParametersInfo(SPI_SETMOUSE,           // Set mouse information
                           0,                  // Not used
                           aMouseInfo,         // Mouse information
                           SPIF_SENDCHANGE); // Update Win.ini
    }
}
```

ⓘ Note

The winuser.h header defines [SystemParametersInfo](#) 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-sysparams-ext-l1-1-0 (introduced in Windows 8)

See also

[ACCESSTIMEOUT](#)

[ANIMATIONINFO](#)

[AUDIODESCRIPTION](#)

[FILTERKEYS](#)

[HIGHCONTRAST](#)

[ICONMETRICS](#)

[LOGFONT](#)

[MAKELANGID](#)

[MINIMIZEDMETRICS](#)

[MOUSEKEYS](#)

[NONCLIENTMETRICS](#)

[RECT](#)

[SERIALKEYS](#)

[SOUNDSENTRY](#)

[STICKYKEYS](#)

[SystemParametersInfoForDPI](#)

[TOGGLEKEYS](#)

[WM_SETTINGCHANGE](#)

[Windows Data Types](#)

Feedback

Was this page helpful?

 Yes

 No

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

SystemParametersInfoW function (winuser.h)

Article02/09/2023

Retrieves or sets the value of one of the system-wide parameters. This function can also update the user profile while setting a parameter.

Syntax

C++

```
BOOL SystemParametersInfoW(
    [in]      UINT  uiAction,
    [in]      UINT  uiParam,
    [in, out] PVOID  pvParam,
    [in]      UINT  fWinIni
);
```

Parameters

[in] uiAction

Type: **UINT**

The system-wide parameter to be retrieved or set. The possible values are organized in the following tables of related parameters:

- Accessibility parameters
- Desktop parameters
- Icon parameters
- Input parameters
- Menu parameters
- Power parameters
- Screen saver parameters
- Time-out parameters
- UI effect parameters
- Window parameters

The following are the accessibility parameters.

Accessibility parameter	Meaning
-------------------------	---------

SPI_GETACCESSTIMEOUT 0x003C	<p>Retrieves information about the time-out period associated with the accessibility features. The <i>pvParam</i> parameter must point to an ACCESSTIMEOUT structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ACCESSTIMEOUT)</code>.</p>
SPI_GETAUDIODESCRIPTION 0x0074	<p>Determines whether audio descriptions are enabled or disabled. The <i>pvParam</i> parameter is a pointer to an AUDIODESCRIPTION structure. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(AUDIODESCRIPTION)</code>.</p> <p>While it is possible for users who have visual impairments to hear the audio in video content, there is a lot of action in video that does not have corresponding audio. Specific audio description of what is happening in a video helps these users understand the content better. This flag enables you to determine whether audio descriptions have been enabled and in which language.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETCLIENTAREAANIMATION 0x1042	<p>Determines whether animations are enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if animations are enabled, or FALSE otherwise.</p> <p>Display features such as flashing, blinking, flickering, and moving content can cause seizures in users with photo-sensitive epilepsy. This flag enables you to determine whether such animations have been disabled in the client area.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETDISABLEOVERLAPPEDCONTENT 0x1040	<p>Determines whether overlapped content is enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Display features such as background images, textured backgrounds, water marks on documents, alpha blending, and transparency can reduce the contrast between the foreground and background, making it harder for users with low vision to see objects on the screen. This flag enables you to determine whether such overlapped content has been disabled.</p>

		Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_GETFILTERKEYS 0x0032		Retrieves information about the FilterKeys accessibility feature. The <i>pvParam</i> parameter must point to a FILTERKEYS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(FILTERKEYS)</code> .
SPI_GETFOCUSBORDERHEIGHT 0x2010		Retrieves the height, in pixels, of the top and bottom edges of the focus rectangle drawn with DrawFocusRect . The <i>pvParam</i> parameter must point to a UINT value. Windows 2000: This parameter is not supported.
SPI_GETFOCUSBORDERWIDTH 0x200E		Retrieves the width, in pixels, of the left and right edges of the focus rectangle drawn with DrawFocusRect . The <i>pvParam</i> parameter must point to a UINT . Windows 2000: This parameter is not supported.
SPI_GETHIGHCONTRAST 0x0042		Retrieves information about the HighContrast accessibility feature. The <i>pvParam</i> parameter must point to a HIGHCONTRAST structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(HIGHCONTRAST)</code> . For a general discussion, see Remarks.
SPI_GETLOGICALDPIOVERRIDE 0x009E		Retrieves a value that determines whether Windows 8 is displaying apps using the default scaling plateau for the hardware or going to the next higher plateau. This value is based on the current "Make everything on your screen bigger" setting, found in the Ease of Access section of PC settings : 1 is on, 0 is off. Apps can provide text and image resources for each of several scaling plateaus: 100%, 140%, and 180%. Providing separate resources optimized for a particular scale avoids distortion due to resizing. Windows 8 determines the appropriate scaling plateau based on a number of factors, including screen size and pixel density. When "Make everything on your screen bigger" is selected (SPI_GETLOGICALDPIOVERRIDE returns a value of 1), Windows uses resources from the next higher plateau. For example, in the case of hardware that Windows determines should use a scale of SCALE_100_PERCENT , this override causes Windows to

use the [SCALE_140_PERCENT](#) scale value, assuming that it does not violate other constraints.

Note You should not use this value. It might be altered or unavailable in subsequent versions of Windows. Instead, use the [GetScaleFactorForDevice](#) function or the [DisplayProperties](#) class to retrieve the preferred scaling factor. Desktop applications should use desktop logical DPI rather than scale factor. Desktop logical DPI can be retrieved through the [GetDeviceCaps](#) function.

SPI_GETMESSAGEDURATION 0x2016	<p>Retrieves the time that notification pop-ups should be displayed, in seconds. The <i>pvParam</i> parameter must point to a ULONG that receives the message duration.</p> <p>Users with visual impairments or cognitive conditions such as ADHD and dyslexia might need a longer time to read the text in notification messages. This flag enables you to retrieve the message duration.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETMOUSECLICKLOCK 0x101E	<p>Retrieves the state of the Mouse ClickLock feature. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_GETMOUSECLICKLOCKTIME 0x2008	<p>Retrieves the time delay before the primary mouse button is locked. The <i>pvParam</i> parameter must point to DWORD that receives the time delay, in milliseconds. This is only enabled if SPI_SETMOUSECLICKLOCK is set to TRUE. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_GETMOUSEKEYS 0x0036	<p>Retrieves information about the MouseKeys accessibility feature. The <i>pvParam</i> parameter must point to a MOUSEKEYS structure that receives the</p>

information. Set the **cbSize** member of this structure and the *uiParam* parameter to `sizeof(MOUSEKEYS)`.

SPI_GETMOUSESONAR 0x101C	Retrieves the state of the Mouse Sonar feature. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE otherwise. For more information, see Mouse Input Overview . Windows 2000: This parameter is not supported.
SPI_GETMOUSEVANISH 0x1020	Retrieves the state of the Mouse Vanish feature. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE otherwise. For more information, see Mouse Input Overview . Windows 2000: This parameter is not supported.
SPI_GETSCREENREADER 0x0046	Determines whether a screen reviewer utility is running. A screen reviewer utility directs textual information to an output device, such as a speech synthesizer or Braille display. When this flag is set, an application should provide textual information in situations where it would otherwise present the information graphically. The <i>pvParam</i> parameter is a pointer to a BOOL variable that receives TRUE if a screen reviewer utility is running, or FALSE otherwise. <div style="border: 1px solid #ccc; padding: 10px; border-radius: 10px;"><p>Note Narrator, the screen reader that is included with Windows, does not set the SPI_SETSCREENREADER or SPI_GETSCREENREADER flags.</p></div>
SPI_GETSERIALKEYS 0x003E	This parameter is not supported. Windows Server 2003 and Windows XP/2000: The user should control this setting through the Control Panel.
SPI_GETSHOWSOUNDS 0x0038	Determines whether the Show Sounds accessibility flag is on or off. If it is on, the user requires an application to present information visually in situations where it would otherwise present the information only in audible form. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the feature is on, or FALSE if it is off. Using this value is equivalent to calling GetSystemMetrics with SM_SHOWSOUNDS . That is

the recommended call.

SPI_GETSOUNDSENTRY 0x0040	Retrieves information about the SoundSentry accessibility feature. The <i>pvParam</i> parameter must point to a SOUNDSENTRY structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(SOUNDSENTRY)</code> .
SPI_GETSTICKYKEYS 0x003A	Retrieves information about the StickyKeys accessibility feature. The <i>pvParam</i> parameter must point to a STICKYKEYS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(STICKYKEYS)</code> .
SPI_GETTOGGLEKEYS 0x0034	Retrieves information about the ToggleKeys accessibility feature. The <i>pvParam</i> parameter must point to a TOGGLEKEYS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(TOGGLEKEYS)</code> .
SPI_SETACCESSTIMEOUT 0x003D	Sets the time-out period associated with the accessibility features. The <i>pvParam</i> parameter must point to an ACCESSTIMEOUT structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ACCESSTIMEOUT)</code> .
SPI_SETAUDIODESCRIPTION 0x0075	Turns the audio descriptions feature on or off. The <i>pvParam</i> parameter is a pointer to an AUDIODESCRIPTION structure. While it is possible for users who are visually impaired to hear the audio in video content, there is a lot of action in video that does not have corresponding audio. Specific audio description of what is happening in a video helps these users understand the content better. This flag enables you to enable or disable audio descriptions in the languages they are provided in. Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETCLIENTAREAANIMATION 0x1043	Turns client area animations on or off. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to enable animations and other transient effects in the client area, or FALSE to disable them. Display features such as flashing, blinking, flickering, and moving content can cause seizures in users with

	<p>photo-sensitive epilepsy. This flag enables you to enable or disable all such animations.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETDISABLEOVERLAPPEDCONTENT 0x1041	<p>Turns overlapped content (such as background images and watermarks) on or off. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to disable overlapped content, or FALSE to enable overlapped content.</p> <p>Display features such as background images, textured backgrounds, water marks on documents, alpha blending, and transparency can reduce the contrast between the foreground and background, making it harder for users with low vision to see objects on the screen. This flag enables you to enable or disable all such overlapped content.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETFILTERKEYS 0x0033	<p>Sets the parameters of the FilterKeys accessibility feature. The <i>pvParam</i> parameter must point to a FILTERKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(FILTERKEYS)</code>.</p>
SPI_SETFOCUSBORDERHEIGHT 0x2011	<p>Sets the height of the top and bottom edges of the focus rectangle drawn with DrawFocusRect to the value of the <i>pvParam</i> parameter.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETFOCUSBORDERWIDTH 0x200F	<p>Sets the height of the left and right edges of the focus rectangle drawn with DrawFocusRect to the value of the <i>pvParam</i> parameter.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETHIGHCONTRAST 0x0043	<p>Sets the parameters of the HighContrast accessibility feature. The <i>pvParam</i> parameter must point to a HIGHCONTRAST structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(HIGHCONTRAST)</code>.</p>
SPI_SETLOGICALDPIOVERRIDE 0x009F	Do not use.
SPI_SETMESSAGEDURATION 0x2017	Sets the time that notification pop-ups should be displayed, in seconds. The <i>pvParam</i> parameter

	<p>specifies the message duration.</p> <p>Users with visual impairments or cognitive conditions such as ADHD and dyslexia might need a longer time to read the text in notification messages. This flag enables you to set the message duration.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETMOUSECLICKLOCK 0x101F	<p>Turns the Mouse ClickLock accessibility feature on or off. This feature temporarily locks down the primary mouse button when that button is clicked and held down for the time specified by SPI_SETMOUSECLICKLOCKTIME. The <i>pvParam</i> parameter specifies TRUE for on, or FALSE for off. The default is off. For more information, see Remarks and AboutMouse Input.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSECLICKLOCKTIME 0x2009	<p>Adjusts the time delay before the primary mouse button is locked. The <i>uiParam</i> parameter should be set to 0. The <i>pvParam</i> parameter points to a DWORD that specifies the time delay in milliseconds. For example, specify 1000 for a 1 second delay. The default is 1200. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSEKEYS 0x0037	<p>Sets the parameters of the MouseKeys accessibility feature. The <i>pvParam</i> parameter must point to a MOUSEKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(MOUSEKEYS)</code>.</p>
SPI_SETMOUSESONAR 0x101D	<p>Turns the Sonar accessibility feature on or off. This feature briefly shows several concentric circles around the mouse pointer when the user presses and releases the CTRL key. The <i>pvParam</i> parameter specifies TRUE for on and FALSE for off. The default is off. For more information, see Mouse Input Overview.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSEVANISH 0x1021	<p>Turns the Vanish feature on or off. This feature hides the mouse pointer when the user types; the pointer reappears when the user moves the mouse. The <i>pvParam</i> parameter specifies TRUE for on and FALSE for off. The default is off. For more information, see Mouse Input Overview.</p>

Windows 2000: This parameter is not supported.

SPI_SETSCREENREADER 0x0047	Determines whether a screen review utility is running. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
	<p>Note Narrator, the screen reader that is included with Windows, does not set the SPI_SETSCREENREADER or SPI_GETSCREENREADER flags.</p>
SPI_SETSERIALKEYS 0x003F	This parameter is not supported. Windows Server 2003 and Windows XP/2000: The user should control this setting through the Control Panel.
SPI_SETSHOWSOUNDS 0x0039	Turns the ShowSounds accessibility feature on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
SPI_SETSOUNDSENTRY 0x0041	Sets the parameters of the SoundSentry accessibility feature. The <i>pvParam</i> parameter must point to a SOUNDSENTRY structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(SOUNDSENTRY)</code> .
SPI_SETSTICKYKEYS 0x003B	Sets the parameters of the StickyKeys accessibility feature. The <i>pvParam</i> parameter must point to a STICKYKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(STICKYKEYS)</code> .
SPI_SETTOGGLEKEYS 0x0035	Sets the parameters of the ToggleKeys accessibility feature. The <i>pvParam</i> parameter must point to a TOGGLEKEYS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(TOGGLEKEYS)</code> .

The following are the desktop parameters.

Desktop parameter	Meaning
SPI_GETCLEARATYPE 0x1048	Determines whether ClearType is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if ClearType is enabled, or FALSE otherwise.

		ClearType is a software technology that improves the readability of text on liquid crystal display (LCD) monitors.
		Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_GETDESKWALLPAPER 0x0073		Retrieves the full path of the bitmap file for the desktop wallpaper. The <i>pvParam</i> parameter must point to a buffer to receive the null-terminated path string. Set the <i>uiParam</i> parameter to the size, in characters, of the <i>pvParam</i> buffer. The returned string will not exceed MAX_PATH characters. If there is no desktop wallpaper, the returned string is empty.
SPI_GETDROPSHADOW 0x1024		Determines whether the drop shadow effect is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that returns TRUE if enabled or FALSE if disabled.
		Windows 2000: This parameter is not supported.
SPI_GETFLATMENU 0x1022		Determines whether native User menus have flat menu appearance. The <i>pvParam</i> parameter must point to a BOOL variable that returns TRUE if the flat menu appearance is set, or FALSE otherwise.
		Windows 2000: This parameter is not supported.
SPI_GETFONTSMOOTHING 0x004A		Determines whether the font smoothing feature is enabled. This feature uses font antialiasing to make font curves appear smoother by painting pixels at different gray levels. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the feature is enabled, or FALSE if it is not.
SPI_GETFONTSMOOTHINGCONTRAST 0x200C		Retrieves a contrast value that is used in ClearType smoothing. The <i>pvParam</i> parameter must point to a UINT that receives the information. Valid contrast values are from 1000 to 2200. The default value is 1400.
		Windows 2000: This parameter is not supported.
SPI_GETFONTSMOOTHINGORIENTATION 0x2012		Retrieves the font smoothing orientation. The <i>pvParam</i> parameter must point to a UINT that receives the information. The possible values are FE_FONTSMOOTHINGORIENTATIONBGR (blue-green-red) and FE_FONTSMOOTHINGORIENTATIONRGB (red-green-blue).

		Windows XP/2000: This parameter is not supported until Windows XP with SP2.
SPI_GETFONTSMOOTHINGTYPE 0x200A	Retrieves the type of font smoothing. The <i>pvParam</i> parameter must point to a UINT that receives the information. The possible values are FE_FONTSMOOTHINGSTANDARD and FE_FONTSMOOTHINGCLEARATYPE .	Windows 2000: This parameter is not supported.
SPI_GETWORKAREA 0x0030	Retrieves the size of the work area on the primary display monitor. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars. The <i>pvParam</i> parameter must point to a RECT structure that receives the coordinates of the work area, expressed in physical pixel size. Any DPI virtualization mode of the caller has no effect on this output.	To get the work area of a monitor other than the primary display monitor, call the GetMonitorInfo function.
SPI_SETCLEARATYPE 0x1049	Turns ClearType on or off. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to enable ClearType, or FALSE to disable it.	ClearType is a software technology that improves the readability of text on LCD monitors.
		Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETCURSORS 0x0057	Reloads the system cursors. Set the <i>uiParam</i> parameter to zero and the <i>pvParam</i> parameter to NULL .	
SPI_SETDESKPATTERN 0x0015	Sets the current desktop pattern by causing Windows to read the Pattern= setting from the WIN.INI file.	
SPI_SETDESKWALLPAPER 0x0014	<p>Note When the SPI_SETDESKWALLPAPER flag is used, SystemParametersInfo returns TRUE unless there is an error (like when the specified file doesn't exist).</p>	
SPI_SETDROPSHADOW	Enables or disables the drop shadow effect. Set	

0x1025	<p><i>pvParam</i> to TRUE to enable the drop shadow effect or FALSE to disable it. You must also have CS_DROPSHADOW in the window class style.</p>
	<p>Windows 2000: This parameter is not supported.</p>
SPI_SETFLATMENU 0x1023	<p>Enables or disables flat menu appearance for native User menus. Set <i>pvParam</i> to TRUE to enable flat menu appearance or FALSE to disable it.</p> <p>When enabled, the menu bar uses COLOR_MENUBAR for the menubar background, COLOR_MENU for the menu-popup background, COLOR_MENUHIGHLIGHT for the fill of the current menu selection, and COLOR_HIGHLIGHT for the outline of the current menu selection. If disabled, menus are drawn using the same metrics and colors as in Windows 2000.</p>
	<p>Windows 2000: This parameter is not supported.</p>
SPI_SETFONTSMOOTHING 0x004B	<p>Enables or disables the font smoothing feature, which uses font antialiasing to make font curves appear smoother by painting pixels at different gray levels.</p> <p>To enable the feature, set the <i>uiParam</i> parameter to TRUE. To disable the feature, set <i>uiParam</i> to FALSE.</p>
SPI_SETFONTSMOOTHINGCONTRAST 0x200D	<p>Sets the contrast value used in ClearType smoothing. The <i>pvParam</i> parameter is the contrast value. Valid contrast values are from 1000 to 2200. The default value is 1400.</p> <p>SPI_SETFONTSMOOTHINGTYPE must also be set to FE_FONTSMOOTHINGCLEARTEXT.</p>
	<p>Windows 2000: This parameter is not supported.</p>
SPI_SETFONTSMOOTHINGORIENTATION 0x2013	<p>Sets the font smoothing orientation. The <i>pvParam</i> parameter is either FE_FONTSMOOTHINGORIENTATIONBGR (blue-green-red) or FE_FONTSMOOTHINGORIENTATIONRGB (red-green-blue).</p>
	<p>Windows XP/2000: This parameter is not supported until Windows XP with SP2.</p>
SPI_SETFONTSMOOTHINGTYPE 0x200B	<p>Sets the font smoothing type. The <i>pvParam</i> parameter is either FE_FONTSMOOTHINGSTANDARD, if standard anti-aliasing is used, or</p>

`FE_FONTSMOOTHINGCLEARTEXT`, if [ClearType](#) is used. The default is `FE_FONTSMOOTHINGSTANDARD`.

`SPI_SETFONTSMOOTHING` must also be set.

Windows 2000: This parameter is not supported.

SPI_SETWORKAREA

0x002F

Sets the size of the work area. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars. The *pvParam* parameter is a pointer to a [RECT](#) structure that specifies the new work area rectangle, expressed in virtual screen coordinates. In a system with multiple display monitors, the function sets the work area of the monitor that contains the specified rectangle.

The following are the icon parameters.

Icon parameter	Meaning
SPI_GETICONMETRICS 0x002D	Retrieves the metrics associated with icons. The <i>pvParam</i> parameter must point to an ICONMETRICS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ICONMETRICS)</code> .
SPI_GETicontitleLOGFONT 0x001F	Retrieves the logical font information for the current icon-title font. The <i>uiParam</i> parameter specifies the size of a LOGFONT structure, and the <i>pvParam</i> parameter must point to the LOGFONT structure to fill in.
SPI_GETicontitleWRAP 0x0019	Determines whether icon-title wrapping is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.
SPI_ICONHORIZONTALSPACING 0x000D	Sets or retrieves the width, in pixels, of an icon cell. The system uses this rectangle to arrange icons in large icon view. To set this value, set <i>uiParam</i> to the new value and set <i>pvParam</i> to NULL . You cannot set this value to less than SM_CXICON . To retrieve this value, <i>pvParam</i> must point to an integer that receives the current value.
SPI_ICONVERTICALSPACING 0x0018	Sets or retrieves the height, in pixels, of an icon cell.

	To set this value, set <i>uiParam</i> to the new value and set <i>pvParam</i> to NULL . You cannot set this value to less than SM_CYICON .
	To retrieve this value, <i>pvParam</i> must point to an integer that receives the current value.
SPI_SETICONMETRICS 0x002E	Sets the metrics associated with icons. The <i>pvParam</i> parameter must point to an ICONMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ICONMETRICS)</code> .
SPI_SETICONS 0x0058	Reloads the system icons. Set the <i>uiParam</i> parameter to zero and the <i>pvParam</i> parameter to NULL .
SPI_SETicontitlelogfont 0x0022	Sets the font that is used for icon titles. The <i>uiParam</i> parameter specifies the size of a LOGFONT structure, and the <i>pvParam</i> parameter must point to a LOGFONT structure.
SPI_Seticontitlewrap 0x001A	Turns icon-title wrapping on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.

The following are the input parameters. They include parameters related to the keyboard, mouse, pen, input language, and the warning beeper.

Input parameter	Meaning
SPI_GETBEEP 0x0001	Determines whether the warning beeper is on. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the beeper is on, or FALSE if it is off.
SPI_GETBLOCKSENDINPUTRESETS 0x1026	Retrieves a BOOL indicating whether an application can reset the screensaver's timer by calling the SendInput function to simulate keyboard or mouse input. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the simulated input will be blocked, or FALSE otherwise.
SPI_GETCONTACTVISUALIZATION 0x2018	Retrieves the current contact visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that receives the setting. For more information, see Contact Visualization .
SPI_GETDEFAULTINPUTLANG 0x0059	Retrieves the input locale identifier for the system default input language. The <i>pvParam</i> parameter must point to an HKL variable that receives this value. For

	<p>more information, see Languages, Locales, and Keyboard Layouts.</p>
SPI_GETGESTUREVISUALIZATION 0x201A	Retrieves the current gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that receives the setting. For more information, see Gesture Visualization .
SPI_GETKEYBOARDCUES 0x100A	Determines whether menu access keys are always underlined. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if menu access keys are always underlined, and FALSE if they are underlined only when the menu is activated by the keyboard.
SPI_GETKEYBOARDDELAY 0x0016	Retrieves the keyboard repeat-delay setting, which is a value in the range from 0 (approximately 250 ms delay) through 3 (approximately 1 second delay). The actual delay associated with each value may vary depending on the hardware. The <i>pvParam</i> parameter must point to an integer variable that receives the setting.
SPI_GETKEYBOARDPREF 0x0044	Determines whether the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the user relies on the keyboard; or FALSE otherwise.
SPI_GETKEYBOARDSPEED 0x000A	Retrieves the keyboard repeat-speed setting, which is a value in the range from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second). The actual repeat rates are hardware-dependent and may vary from a linear scale by as much as 20%. The <i>pvParam</i> parameter must point to a DWORD variable that receives the setting.
SPI_GETMOUSE 0x0003	Retrieves the two mouse threshold values and the mouse acceleration. The <i>pvParam</i> parameter must point to an array of three integers that receives these values. See mouse_event for further information.
SPI_GETMOUSEHOVERHEIGHT 0x0064	Retrieves the height, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEOVER message. The <i>pvParam</i> parameter must point to a UINT variable that receives the height.
SPI_GETMOUSEHOVERTIME 0x0066	Retrieves the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle for

	<p>TrackMouseEvent to generate a WM_MOUSEHOVER message. The <i>pvParam</i> parameter must point to a UINT variable that receives the time.</p>
SPI_GETMOUSEHOVERWIDTH 0x0062	<p>Retrieves the width, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. The <i>pvParam</i> parameter must point to a UINT variable that receives the width.</p>
SPI_GETMOUSESPEED 0x0070	<p>Retrieves the current mouse speed. The mouse speed determines how far the pointer will move based on the distance the mouse moves. The <i>pvParam</i> parameter must point to an integer that receives a value which ranges between 1 (slowest) and 20 (fastest). A value of 10 is the default. The value can be set by an end-user using the mouse control panel application or by an application using SPI_SETMOUSESPEED.</p>
SPI_GETMOUSETRAILS 0x005E	<p>Determines whether the Mouse Trails feature is enabled. This feature improves the visibility of mouse cursor movements by briefly showing a trail of cursors and quickly erasing them.</p> <p>The <i>pvParam</i> parameter must point to an integer variable that receives a value. If the value is zero or 1, the feature is disabled. If the value is greater than 1, the feature is enabled and the value indicates the number of cursors drawn in the trail. The <i>uiParam</i> parameter is not used.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_GETMOUSEWHEELROUTING 0x201C	<p>Retrieves the routing setting for wheel button input. The routing setting determines whether wheel button input is sent to the app with focus (foreground) or the app under the mouse cursor.</p> <p>The <i>pvParam</i> parameter must point to a DWORD variable that receives the routing option. If the value is zero or MOUSEWHEEL_ROUTING_FOCUS, mouse wheel input is delivered to the app with focus. If the value is 1 or MOUSEWHEEL_ROUTING_HYBRID (default), mouse wheel input is delivered to the app with focus (desktop apps) or the app under the mouse cursor (Windows Store apps). The <i>uiParam</i> parameter is not used.</p>
SPI_GETPENVISUALIZATION 0x201E	<p>Retrieves the current pen gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that receives the setting. For more information, see Pen Visualization.</p>

SPI_GETSNAPTODEFBUTTON 0x005F	Determines whether the snap-to-default-button feature is enabled. If enabled, the mouse cursor automatically moves to the default button, such as OK or Apply , of a dialog box. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the feature is on, or FALSE if it is off.
SPI_GETSYSTEMLANGUAGEBAR 0x1050	Starting with Windows 8: Determines whether the system language bar is enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the language bar is enabled, or FALSE otherwise.
SPI_GETTHREADLOCALINPUTSETTINGS 0x104E	Starting with Windows 8: Determines whether the active input settings have Local (per-thread, TRUE) or Global (session, FALSE) scope. The <i>pvParam</i> parameter must point to a BOOL variable.
SPI_GETWHEELSCROLLCHARS 0x006C	Retrieves the number of characters to scroll when the horizontal mouse wheel is moved. The <i>pvParam</i> parameter must point to a UINT variable that receives the number of lines. The default value is 3.
SPI_GETWHEELSCROLLLINES 0x0068	Retrieves the number of lines to scroll when the vertical mouse wheel is moved. The <i>pvParam</i> parameter must point to a UINT variable that receives the number of lines. The default value is 3.
SPI_SETBEEP 0x0002	Turns the warning beeper on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.
SPI_SETBLOCKSENDINPUTRESETS 0x1027	Determines whether an application can reset the screensaver's timer by calling the SendInput function to simulate keyboard or mouse input. The <i>uiParam</i> parameter specifies TRUE if the screensaver will not be deactivated by simulated input, or FALSE if the screensaver will be deactivated by simulated input.
SPI_SETCONTACTVISUALIZATION 0x2019	Sets the current contact visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that identifies the setting. For more information, see Contact Visualization .
Note If contact visualizations are disabled, gesture visualizations cannot be enabled.	
SPI_SETDEFAULTINPUTLANG 0x005A	Sets the default input language for the system shell and applications. The specified language must be

	<p>displayable using the current system character set. The <i>pvParam</i> parameter must point to an HKL variable that contains the input locale identifier for the default language. For more information, see Languages, Locales, and Keyboard Layouts.</p>
SPI_SETDOUBLECLICKTIME 0x0020	<p>Sets the double-click time for the mouse to the value of the <i>uiParam</i> parameter. If the <i>uiParam</i> value is greater than 5000 milliseconds, the system sets the double-click time to 5000 milliseconds.</p> <p>The double-click time is the maximum number of milliseconds that can occur between the first and second clicks of a double-click. You can also call the SetDoubleClickTime function to set the double-click time. To get the current double-click time, call the GetDoubleClickTime function.</p>
SPI_SETDOUBLECLKHEIGHT 0x001E	<p>Sets the height of the double-click rectangle to the value of the <i>uiParam</i> parameter.</p> <p>The double-click rectangle is the rectangle within which the second click of a double-click must fall for it to be registered as a double-click.</p> <p>To retrieve the height of the double-click rectangle, call GetSystemMetrics with the SM_CYDOUBLECLK flag.</p>
SPI_SETDOUBLECLKWIDTH 0x001D	<p>Sets the width of the double-click rectangle to the value of the <i>uiParam</i> parameter.</p> <p>The double-click rectangle is the rectangle within which the second click of a double-click must fall for it to be registered as a double-click.</p> <p>To retrieve the width of the double-click rectangle, call GetSystemMetrics with the SM_CXDOUBLECLK flag.</p>
SPI_SETGESTUREVISUALIZATION 0x201B	<p>Sets the current gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that identifies the setting. For more information, see Gesture Visualization.</p> <p>Note If contact visualizations are disabled, gesture visualizations cannot be enabled.</p>
SPI_SETKEYBOARDCUES 0x100B	<p>Sets the underlining of menu access key letters. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to always underline menu access keys, or FALSE</p>

		to underline menu access keys only when the menu is activated from the keyboard.
SPI_SETKEYBOARDDELAY 0x0017	Sets the keyboard repeat-delay setting. The <i>uiParam</i> parameter must specify 0, 1, 2, or 3, where zero sets the shortest delay approximately 250 ms) and 3 sets the longest delay (approximately 1 second). The actual delay associated with each value may vary depending on the hardware.	
SPI_SETKEYBOARDPREF 0x0045	Sets the keyboard preference. The <i>uiParam</i> parameter specifies TRUE if the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden; <i>uiParam</i> is FALSE otherwise.	
SPI_SETKEYBOARDSPEED 0x000B	Sets the keyboard repeat-speed setting. The <i>uiParam</i> parameter must specify a value in the range from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second). The actual repeat rates are hardware-dependent and may vary from a linear scale by as much as 20%. If <i>uiParam</i> is greater than 31, the parameter is set to 31.	
SPI_SETLANGTOGGLE 0x005B	Sets the hot key set for switching between input languages. The <i>uiParam</i> and <i>pvParam</i> parameters are not used. The value sets the shortcut keys in the keyboard property sheets by reading the registry again. The registry must be set before this flag is used. the path in the registry is HKEY_CURRENT_USER\Keyboard Layout\Toggle . Valid values are "1" = ALT+SHIFT, "2" = CTRL+SHIFT, and "3" = none.	
SPI_SETMOUSE 0x0004	Sets the two mouse threshold values and the mouse acceleration. The <i>pvParam</i> parameter must point to an array of three integers that specifies these values. See mouse_event for further information.	
SPI_SETMOUSEBUTTONSWAP 0x0021	Swaps or restores the meaning of the left and right mouse buttons. The <i>uiParam</i> parameter specifies TRUE to swap the meanings of the buttons, or FALSE to restore their original meanings. To retrieve the current setting, call GetSystemMetrics with the SM_SWAPBUTTON flag.	
SPI_SETMOUSEOVERHEIGHT 0x0065	Sets the height, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to	

	<p>generate a WM_MOUSEHOVER message. Set the <i>uiParam</i> parameter to the new height.</p>
SPI_SETMOUSEHOVERTIME 0x0067	<p>Sets the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle for TrackMouseEvent to generate a WM_MOUSEHOVER message. This is used only if you pass HOVER_DEFAULT in the <i>dwHoverTime</i> parameter in the call to TrackMouseEvent. Set the <i>uiParam</i> parameter to the new time.</p> <p>The time specified should be between USER_TIMER_MAXIMUM and USER_TIMER_MINIMUM. If <i>uiParam</i> is less than USER_TIMER_MINIMUM, the function will use USER_TIMER_MINIMUM. If <i>uiParam</i> is greater than USER_TIMER_MAXIMUM, the function will be USER_TIMER_MAXIMUM.</p> <p>Windows Server 2003 and Windows XP: The operating system does not enforce the use of USER_TIMER_MAXIMUM and USER_TIMER_MINIMUM until Windows Server 2003 with SP1 and Windows XP with SP2.</p>
SPI_SETMOUSEHOVERWIDTH 0x0063	Sets the width, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. Set the <i>uiParam</i> parameter to the new width.
SPI_SETMOUSESPEED 0x0071	Sets the current mouse speed. The <i>pvParam</i> parameter is an integer between 1 (slowest) and 20 (fastest). A value of 10 is the default. This value is typically set using the mouse control panel application.
SPI_SETMOUSETRAILS 0x005D	<p>Enables or disables the Mouse Trails feature, which improves the visibility of mouse cursor movements by briefly showing a trail of cursors and quickly erasing them.</p> <p>To disable the feature, set the <i>uiParam</i> parameter to zero or 1. To enable the feature, set <i>uiParam</i> to a value greater than 1 to indicate the number of cursors drawn in the trail.</p> <p>Windows 2000: This parameter is not supported.</p>
SPI_SETMOUSEWHEELROUTING 0x201D	Sets the routing setting for wheel button input. The routing setting determines whether wheel button input is sent to the app with focus (foreground) or the app under the mouse cursor.

	<p>The <i>pvParam</i> parameter must point to a DWORD variable that receives the routing option. If the value is zero or MOUSEWHEEL_ROUTING_FOCUS, mouse wheel input is delivered to the app with focus. If the value is 1 or MOUSEWHEEL_ROUTING_HYBRID (default), mouse wheel input is delivered to the app with focus (desktop apps) or the app under the mouse cursor (Windows Store apps). Set the <i>uiParam</i> parameter to zero.</p>
SPI_SETPENVISUALIZATION 0x201F	Sets the current pen gesture visualization setting. The <i>pvParam</i> parameter must point to a ULONG variable that identifies the setting. For more information, see Pen Visualization .
SPI_SETSNAPTODEFBUTTON 0x0060	Enables or disables the snap-to-default-button feature. If enabled, the mouse cursor automatically moves to the default button, such as OK or Apply , of a dialog box. Set the <i>uiParam</i> parameter to TRUE to enable the feature, or FALSE to disable it. Applications should use the ShowWindow function when displaying a dialog box so the dialog manager can position the mouse cursor.
SPI_SETSYSTEMLANGUAGEBAR 0x1051	Starting with Windows 8: Turns the legacy language bar feature on or off. The <i>pvParam</i> parameter is a pointer to a BOOL variable. Set <i>pvParam</i> to TRUE to enable the legacy language bar, or FALSE to disable it. The flag is supported on Windows 8 where the legacy language bar is replaced by Input Switcher and therefore turned off by default. Turning the legacy language bar on is provided for compatibility reasons and has no effect on the Input Switcher.
SPI_SETTHREADLOCALINPUTSETTINGS 0x104F	Starting with Windows 8: Determines whether the active input settings have Local (per-thread, TRUE) or Global (session, FALSE) scope. The <i>pvParam</i> parameter must be a BOOL variable, casted by PVOID .
SPI_SETWHEELSCROLLCHARS 0x006D	Sets the number of characters to scroll when the horizontal mouse wheel is moved. The number of characters is set from the <i>uiParam</i> parameter.
SPI_SETWHEELSCROLLLINES 0x0069	Sets the number of lines to scroll when the vertical mouse wheel is moved. The number of lines is set from the <i>uiParam</i> parameter.
	The number of lines is the suggested number of lines to scroll when the mouse wheel is rolled without using modifier keys. If the number is 0, then no scrolling should occur. If the number of lines to scroll is greater

than the number of lines viewable, and in particular if it is **WHEEL_PAGESCROLL** (#defined as **UINT_MAX**), the scroll operation should be interpreted as clicking once in the page down or page up regions of the scroll bar.

The following are the menu parameters.

Menu parameter	Meaning
SPI_GETMENUDROPALIGNMENT 0x001B	Determines whether pop-up menus are left-aligned or right-aligned, relative to the corresponding menu-bar item. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if right-aligned, or FALSE otherwise.
SPI_GETMENUFADE 0x1012	Determines whether menu fade animation is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE when fade animation is enabled and FALSE when it is disabled. If fade animation is disabled, menus use slide animation. This flag is ignored unless menu animation is enabled, which you can do using the SPI_SETMENUANIMATION flag. For more information, see AnimateWindow .
SPI_GETMENUSHOWDELAY 0x006A	Retrieves the time, in milliseconds, that the system waits before displaying a shortcut menu when the mouse cursor is over a submenu item. The <i>pvParam</i> parameter must point to a DWORD variable that receives the time of the delay.
SPI_SETMENUDROPALIGNMENT 0x001C	Sets the alignment value of pop-up menus. The <i>uiParam</i> parameter specifies TRUE for right alignment, or FALSE for left alignment.
SPI_SETMENUFADE 0x1013	Enables or disables menu fade animation. Set <i>pvParam</i> to TRUE to enable the menu fade effect or FALSE to disable it. If fade animation is disabled, menus use slide animation. The menu fade effect is possible only if the system has a color depth of more than 256 colors. This flag is ignored unless SPI_MENUANIMATION is also set. For more information, see AnimateWindow .
SPI_SETMENUSHOWDELAY 0x006B	Sets <i>uiParam</i> to the time, in milliseconds, that the system waits before displaying a shortcut menu when the mouse cursor is over a submenu item.

The following are the power parameters.

Beginning with Windows Server 2008 and Windows Vista, these power parameters are not supported. Instead, to determine the current display power state, an application should register for **GUID_MONITOR_POWER_STATE** notifications. To determine the current display power down time-out, an application should register for notification of changes to the **GUID_VIDEO_POWERDOWN_TIMEOUT** power setting. For more information, see [Registering for Power Events](#).

Windows Server 2003 and Windows XP/2000: To determine the current display power state, use the following power parameters.

Power parameter	Meaning
SPI_GETLOWPOWERACTIVE 0x0053	<p>This parameter is not supported.</p> <p>Windows Server 2003 and Windows XP/2000: Determines whether the low-power phase of screen saving is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE if disabled. This flag is supported for 32-bit applications only.</p>
SPI_GETLOWPOWERTIMEOUT 0x004F	<p>This parameter is not supported.</p> <p>Windows Server 2003 and Windows XP/2000: Retrieves the time-out value for the low-power phase of screen saving. The <i>pvParam</i> parameter must point to an integer variable that receives the value. This flag is supported for 32-bit applications only.</p>
SPI_GETPOWEROFFACTIVE 0x0054	<p>This parameter is not supported. When the power-off phase of screen saving is enabled, the GUID_VIDEO_POWERDOWN_TIMEOUT power setting is greater than zero.</p> <p>Windows Server 2003 and Windows XP/2000: Determines whether the power-off phase of screen saving is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE if disabled. This flag is supported for 32-bit applications only.</p>
SPI_GETPOWEROFFTIMEOUT 0x0050	<p>This parameter is not supported. Instead, check the GUID_VIDEO_POWERDOWN_TIMEOUT power setting.</p> <p>Windows Server 2003 and Windows XP/2000: Retrieves the time-out value for the power-off phase of screen saving. The <i>pvParam</i> parameter must point to an integer variable that receives the value. This flag is supported for 32-bit applications only.</p>

SPI_SETLOWPOWERACTIVE 0x0055	This parameter is not supported. Windows Server 2003 and Windows XP/2000: Activates or deactivates the low-power phase of screen saving. Set <i>uiParam</i> to 1 to activate, or zero to deactivate. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.
SPI_SETLOWPOWERTIMEOUT 0x0051	This parameter is not supported. Windows Server 2003 and Windows XP/2000: Sets the time-out value, in seconds, for the low-power phase of screen saving. The <i>uiParam</i> parameter specifies the new value. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.
SPI_SETPOWEROFFACTIVE 0x0056	This parameter is not supported. Instead, set the GUID_VIDEO_POWERDOWN_TIMEOUT power setting. Windows Server 2003 and Windows XP/2000: Activates or deactivates the power-off phase of screen saving. Set <i>uiParam</i> to 1 to activate, or zero to deactivate. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.
SPI_SETPOWEROFFTIMEOUT 0x0052	This parameter is not supported. Instead, set the GUID_VIDEO_POWERDOWN_TIMEOUT power setting to a time-out value. Windows Server 2003 and Windows XP/2000: Sets the time-out value, in seconds, for the power-off phase of screen saving. The <i>uiParam</i> parameter specifies the new value. The <i>pvParam</i> parameter must be NULL . This flag is supported for 32-bit applications only.

The following are the screen saver parameters.

Screen saver parameter	Meaning
SPI_GETSCREENSAVEACTIVE 0x0010	Determines whether screen saving is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if screen saving is enabled, or FALSE otherwise. Windows 7, Windows Server 2008 R2 and Windows 2000: The function returns TRUE even when screen saving is not enabled.
SPI_GETSCREENSAVERRUNNING 0x0072	Determines whether a screen saver is currently running on the window station of the calling process. The <i>pvParam</i> parameter must point to a BOOL variable that

	<p>receives TRUE if a screen saver is currently running, or FALSE otherwise. Note that only the interactive window station, WinSta0, can have a screen saver running.</p>
SPI_GETSCREENSAVESECURE 0x0076	<p>Determines whether the screen saver requires a password to display the Windows desktop. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the screen saver requires a password, or FALSE otherwise. The <i>uiParam</i> parameter is ignored.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETSCREENSAVETIMEOUT 0x000E	<p>Retrieves the screen saver time-out value, in seconds. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p>
SPI_SETSCREENSAVEACTIVE 0x0011	<p>Sets the state of the screen saver. The <i>uiParam</i> parameter specifies TRUE to activate screen saving, or FALSE to deactivate it.</p> <p>If the machine has entered power saving mode or system lock state, an ERROR_OPERATION_IN_PROGRESS exception occurs.</p>
SPI_SETSCREENSAVESECURE 0x0077	<p>Sets whether the screen saver requires the user to enter a password to display the Windows desktop. The <i>uiParam</i> parameter is a BOOL variable. The <i>pvParam</i> parameter is ignored. Set <i>uiParam</i> to TRUE to require a password, or FALSE to not require a password.</p> <p>If the machine has entered power saving mode or system lock state, an ERROR_OPERATION_IN_PROGRESS exception occurs.</p> <p>Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETSCREENSAVETIMEOUT 0x000F	<p>Sets the screen saver time-out value to the value of the <i>uiParam</i> parameter. This value is the amount of time, in seconds, that the system must be idle before the screen saver activates.</p> <p>If the machine has entered power saving mode or system lock state, an ERROR_OPERATION_IN_PROGRESS exception occurs.</p>

The following are the time-out parameters for applications and services.

Time-out parameter	Meaning
--------------------	---------

SPI_GETHUNGAPPTIMEOUT 0x0078	<p>Retrieves the number of milliseconds that a thread can go without dispatching a message before the system considers it unresponsive. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETWAITTOKILLTIMEOUT 0x007A	<p>Retrieves the number of milliseconds that the system waits before terminating an application that does not respond to a shutdown request. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETWAITTOKILLSERVICETIMEOUT 0x007C	<p>Retrieves the number of milliseconds that the service control manager waits before terminating a service that does not respond to a shutdown request. The <i>pvParam</i> parameter must point to an integer variable that receives the value.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETHUNGAPPTIMEOUT 0x0079	<p>Sets the hung application time-out to the value of the <i>uiParam</i> parameter. This value is the number of milliseconds that a thread can go without dispatching a message before the system considers it unresponsive.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETWAITTOKILLTIMEOUT 0x007B	<p>Sets the application shutdown request time-out to the value of the <i>uiParam</i> parameter. This value is the number of milliseconds that the system waits before terminating an application that does not respond to a shutdown request.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETWAITTOKILLSERVICETIMEOUT 0x007D	<p>Sets the service shutdown request time-out to the value of the <i>uiParam</i> parameter. This value is the number of</p>

milliseconds that the system waits before terminating a service that does not respond to a shutdown request.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.

The following are the UI effects. The **SPI_SETUIEFFECTS** value is used to enable or disable all UI effects at once. This table contains the complete list of UI effect values.

UI effects parameter	Meaning
SPI_GETCOMBOBOXANIMATION 0x1004	Determines whether the slide-open effect for combo boxes is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled.
SPI_GETCURSORSHADOW 0x101A	Determines whether the cursor has a shadow around it. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the shadow is enabled, FALSE if it is disabled. This effect appears only if the system has a color depth of more than 256 colors.
SPI_GETGRADIENTCAPTIONS 0x1008	Determines whether the gradient effect for window title bars is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled. For more information about the gradient effect, see the GetSysColor function.
SPI_GETHOTTRACKING 0x100E	Determines whether hot tracking of user-interface elements, such as menu names on menu bars, is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled. Hot tracking means that when the cursor moves over an item, it is highlighted but not selected. You can query this value to decide whether to use hot tracking in the user interface of your application.
SPI_GETLISTBOXSMOOTHSCROLLING 0x1006	Determines whether the smooth-scrolling effect for list boxes is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled.
SPI_GETMENUANIMATION 0x1002	Determines whether the menu animation feature is enabled. This master switch must be on to enable menu animation effects. The <i>pvParam</i> parameter must point to

	<p>a BOOL variable that receives TRUE if animation is enabled and FALSE if it is disabled.</p> <p>If animation is enabled, SPI_GETMENUFADE indicates whether menus use fade or slide animation.</p>
SPI_GETMENUUNDERLINES 0x100A	Same as SPI_GETKEYBOARDDCUES .
SPI_GETSELECTIONFADE 0x1014	<p>Determines whether the selection fade effect is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE if disabled.</p> <p>The selection fade effect causes the menu item selected by the user to remain on the screen briefly while fading out after the menu is dismissed.</p>
SPI_GETTOOLTIPANIMATION 0x1016	<p>Determines whether ToolTip animation is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled or FALSE if disabled. If ToolTip animation is enabled, SPI_GETTOOLTIPFADE indicates whether ToolTips use fade or slide animation.</p>
SPI_GETTOOLTIPFADE 0x1018	<p>If SPI_SETTOOLTIPANIMATION is enabled, SPI_GETTOOLTIPFADE indicates whether ToolTip animation uses a fade effect or a slide effect. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for fade animation or FALSE for slide animation. For more information on slide and fade effects, see AnimateWindow.</p>
SPI_GETUIEFFECTS 0x103E	<p>Determines whether UI effects are enabled or disabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if all UI effects are enabled, or FALSE if they are disabled.</p>
SPI_SETCOMBOBOXANIMATION 0x1005	Enables or disables the slide-open effect for combo boxes. Set the <i>pvParam</i> parameter to TRUE to enable the gradient effect, or FALSE to disable it.
SPI_SETCURSORSHADOW 0x101B	Enables or disables a shadow around the cursor. The <i>pvParam</i> parameter is a BOOL variable. Set <i>pvParam</i> to TRUE to enable the shadow or FALSE to disable the shadow. This effect appears only if the system has a color depth of more than 256 colors.
SPI_SETGRADIENTCAPTIONS 0x1009	Enables or disables the gradient effect for window title bars. Set the <i>pvParam</i> parameter to TRUE to enable it, or FALSE to disable it. The gradient effect is possible only if the system has a color depth of more than 256 colors.

		For more information about the gradient effect, see the GetSysColor function.
SPI_SETHOTTRACKING 0x100F	Enables or disables hot tracking of user-interface elements such as menu names on menu bars. Set the <i>pvParam</i> parameter to TRUE to enable it, or FALSE to disable it.	Hot-tracking means that when the cursor moves over an item, it is highlighted but not selected.
SPI_SETLISTBOXSMOOTHSCROLLING 0x1007	Enables or disables the smooth-scrolling effect for list boxes. Set the <i>pvParam</i> parameter to TRUE to enable the smooth-scrolling effect, or FALSE to disable it.	
SPI_SETMENUANIMATION 0x1003	Enables or disables menu animation. This master switch must be on for any menu animation to occur. The <i>pvParam</i> parameter is a BOOL variable; set <i>pvParam</i> to TRUE to enable animation and FALSE to disable animation.	If animation is enabled, SPI_GETMENUFADE indicates whether menus use fade or slide animation.
SPI_SETMENUUNDERLINES 0x100B	Same as SPI_SETKEYBOARDCUES .	
SPI_SETSELECTIONFADE 0x1015	Set <i>pvParam</i> to TRUE to enable the selection fade effect or FALSE to disable it.	The selection fade effect causes the menu item selected by the user to remain on the screen briefly while fading out after the menu is dismissed. The selection fade effect is possible only if the system has a color depth of more than 256 colors.
SPI_SETOOLTIPANIMATION 0x1017	Set <i>pvParam</i> to TRUE to enable ToolTip animation or FALSE to disable it. If enabled, you can use SPI_SETOOLTIPFADE to specify fade or slide animation.	
SPI_SETOOLTIPFADE 0x1019	If the SPI_SETOOLTIPANIMATION flag is enabled, use SPI_SETOOLTIPFADE to indicate whether ToolTip animation uses a fade effect or a slide effect. Set <i>pvParam</i> to TRUE for fade animation or FALSE for slide animation. The tooltip fade effect is possible only if the system has a color depth of more than 256 colors. For more information on the slide and fade effects, see the AnimateWindow function.	
SPI_SETUIEFFECTS 0x103F	Enables or disables UI effects. Set the <i>pvParam</i> parameter to TRUE to enable all UI effects or FALSE to disable all UI effects.	

The following are the window parameters.

Window parameter	Meaning
SPI_GETACTIVEWINDOWTRACKING 0x1000	Determines whether active window tracking (activating the window the mouse is on) is on or off. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for on, or FALSE for off.
SPI_GETACTIVEWNDTRKZORDER 0x100C	Determines whether windows activated through active window tracking will be brought to the top. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE for on, or FALSE for off.
SPI_GETACTIVEWNDTRKTIMEOUT 0x2002	Retrieves the active window tracking delay, in milliseconds. The <i>pvParam</i> parameter must point to a DWORD variable that receives the time.
SPI_GETANIMATION 0x0048	Retrieves the animation effects associated with user actions. The <i>pvParam</i> parameter must point to an ANIMATIONINFO structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ANIMATIONINFO)</code> .
SPI_GETBORDER 0x0005	Retrieves the border multiplier factor that determines the width of a window's sizing border. The <i>pvParam</i> parameter must point to an integer variable that receives this value.
SPI_GETCARETWIDTH 0x2006	Retrieves the caret width in edit controls, in pixels. The <i>pvParam</i> parameter must point to a DWORD variable that receives this value.
SPI_GETDOCKMOVING 0x0090	Determines whether a window is docked when it is moved to the top, left, or right edges of a monitor or monitor array. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise. Use SPI_GETWINARRANGING to determine whether this behavior is enabled. Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_GETDRAGFROMMAXIMIZE 0x008C	Determines whether a maximized window is restored when its caption bar is dragged. The <i>pvParam</i>

	<p>parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETDRAGFULLWINDOWS 0x0026	Determines whether dragging of full windows is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.
SPI_GETFOREGROUNDFLASHCOUNT 0x2004	Retrieves the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request. The <i>pvParam</i> parameter must point to a DWORD variable that receives the value.
SPI_GETFOREGROUNDLOCKTIMEOUT 0x2000	Retrieves the amount of time following user input, in milliseconds, during which the system will not allow applications to force themselves into the foreground. The <i>pvParam</i> parameter must point to a DWORD variable that receives the time.
SPI_GETMINIMIZEDMETRICS 0x002B	Retrieves the metrics associated with minimized windows. The <i>pvParam</i> parameter must point to a MINIMIZEDMETRICS structure that receives the information. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(MINIMIZEDMETRICS)</code> .
SPI_GETMOUSEDOCKTHRESHOLD 0x007E	<p>Retrieves the threshold in pixels where docking behavior is triggered by using a mouse to drag a window to the edge of a monitor or monitor array. The default threshold is 1. The <i>pvParam</i> parameter must point to a DWORD variable that receives the value.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETMOUSEDRAGOUTTHRESHOLD 0x0084	Retrieves the threshold in pixels where undocking behavior is triggered by using a mouse to drag a window from the edge of a monitor or a monitor array toward the center. The default threshold is 20.

	<p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETMOUSESIDEMOVETHRESHOLD 0x0088	<p>Retrieves the threshold in pixels from the top of a monitor or a monitor array where a vertically maximized window is restored when dragged with the mouse. The default threshold is 50.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETNONCLIENTMETRICS 0x0029	<p>Retrieves the metrics associated with the nonclient area of nonminimized windows. The <i>pvParam</i> parameter must point to a NONCLIENTMETRICS structure that receives the information. Set the <i>cbSize</i> member of this structure and the <i>uiParam</i> parameter to <code>sizeof(NONCLIENTMETRICS)</code>.</p> <p>Windows Server 2003 and Windows XP/2000: See Remarks for NONCLIENTMETRICS.</p>
SPI_GETPENDOCKTHRESHOLD 0x0080	<p>Retrieves the threshold in pixels where docking behavior is triggered by using a pen to drag a window to the edge of a monitor or monitor array. The default is 30.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETPENDRAGOUTTHRESHOLD 0x0086	<p>Retrieves the threshold in pixels where undocking behavior is triggered by using a pen to drag a window from the edge of a monitor or monitor array toward its center. The default threshold is 30.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETPENSIDEMOVETHRESHOLD	Retrieves the threshold in pixels from the top of a

0x008A	<p>monitor or monitor array where a vertically maximized window is restored when dragged with the mouse. The default threshold is 50.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETSHOWIMEUI 0x006E	Determines whether the IME status window is visible (on a per-user basis). The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if the status window is visible, or FALSE if it is not.
SPI_GETSNAPSIZING 0x008E	<p>Determines whether a window is vertically maximized when it is sized to the top or bottom of a monitor or monitor array. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Use SPI_GETWINARRANGING to determine whether this behavior is enabled.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_GETWINARRANGING 0x0082	<p>Determines whether window arrangement is enabled. The <i>pvParam</i> parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.</p> <p>Window arrangement reduces the number of mouse, pen, or touch interactions needed to move and size top-level windows by simplifying the default behavior of a window when it is dragged or sized.</p> <p>The following parameters retrieve individual window arrangement settings:</p> <ul style="list-style-type: none"> SPI_GETDOCKMOVING SPI_GETMOUSEDOCKTHRESHOLD SPI_GETMOUSEDRAGOUTTHRESHOLD SPI_GETMOUSESIDEMOVEVETHRESHOLD SPI_GETPENDOCKTHRESHOLD SPI_GETPENDRAGOUTTHRESHOLD SPI_GETPENSIDEMOVEVETHRESHOLD SPI_GETSNAPSIZING <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>

SPI_SETACTIVEWINDOWTRACKING 0x1001	Sets active window tracking (activating the window the mouse is on) either on or off. Set <i>pvParam</i> to TRUE for on or FALSE for off.
SPI_SETACTIVEWNDTRKZORDER 0x100D	Determines whether or not windows activated through active window tracking should be brought to the top. Set <i>pvParam</i> to TRUE for on or FALSE for off.
SPI_SETACTIVEWNDTRKTIMEOUT 0x2003	Sets the active window tracking delay. Set <i>pvParam</i> to the number of milliseconds to delay before activating the window under the mouse pointer.
SPI_SETANIMATION 0x0049	Sets the animation effects associated with user actions. The <i>pvParam</i> parameter must point to an ANIMATIONINFO structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(ANIMATIONINFO)</code> .
SPI_SETBORDER 0x0006	Sets the border multiplier factor that determines the width of a window's sizing border. The <i>uiParam</i> parameter specifies the new value.
SPI_SETCARETWIDTH 0x2007	Sets the caret width in edit controls. Set <i>pvParam</i> to the desired width, in pixels. The default and minimum value is 1.
SPI_SETDOCKMOVING 0x0091	Sets whether a window is docked when it is moved to the top, left, or right docking targets on a monitor or monitor array. Set <i>pvParam</i> to TRUE for on or FALSE for off. SPI_GETWINARRANGING must be TRUE to enable this behavior. Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETDRAGFROMMAXIMIZE 0x008D	Sets whether a maximized window is restored when its caption bar is dragged. Set <i>pvParam</i> to TRUE for on or FALSE for off. SPI_GETWINARRANGING must be TRUE to enable this behavior. Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETDRAGFULLWINDOWS 0x0025	Sets dragging of full windows either on or off. The <i>uiParam</i> parameter specifies TRUE for on, or FALSE for off.

SPI_SETDRAGHEIGHT 0x004D	Sets the height, in pixels, of the rectangle used to detect the start of a drag operation. Set <i>uiParam</i> to the new value. To retrieve the drag height, call GetSystemMetrics with the SM_CYDRAG flag.
SPI_SETDRAGWIDTH 0x004C	Sets the width, in pixels, of the rectangle used to detect the start of a drag operation. Set <i>uiParam</i> to the new value. To retrieve the drag width, call GetSystemMetrics with the SM_CXDRAG flag.
SPI_SETFOREGROUNDFLASHCOUNT 0x2005	Sets the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request. Set <i>pvParam</i> to the number of times to flash.
SPI_SETFOREGROUNDLOCKTIMEOUT 0x2001	Sets the amount of time following user input, in milliseconds, during which the system does not allow applications to force themselves into the foreground. Set <i>pvParam</i> to the new time-out value. The calling thread must be able to change the foreground window, otherwise the call fails.
SPI_SETMINIMIZEDMETRICS 0x002C	Sets the metrics associated with minimized windows. The <i>pvParam</i> parameter must point to a MINIMIZEDMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to <code>sizeof(MINIMIZEDMETRICS)</code> .
SPI_SETMOUSEDOCKTHRESHOLD 0x007F	Sets the threshold in pixels where docking behavior is triggered by using a mouse to drag a window to the edge of a monitor or monitor array. The default threshold is 1. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value. SPI_GETWINARRANGING must be TRUE to enable this behavior. Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETMOUSEDRAGOUTTHRESHOLD 0x0085	Sets the threshold in pixels where undocking behavior is triggered by using a mouse to drag a window from the edge of a monitor or monitor array to its center. The default threshold is 20. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.

	<p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETMOUSESIDEMOVETHRESHOLD 0x0089	<p>Sets the threshold in pixels from the top of the monitor where a vertically maximized window is restored when dragged with the mouse. The default threshold is 50. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETNONCLIENTMETRICS 0x002A	<p>Sets the metrics associated with the nonclient area of nonminimized windows. The <i>pvParam</i> parameter must point to a NONCLIENTMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the <i>uiParam</i> parameter to sizeof(NONCLIENTMETRICS). Also, the IfHeight member of the LOGFONT structure must be a negative value.</p>
SPI_SETPENDOCKTHRESHOLD 0x0081	<p>Sets the threshold in pixels where docking behavior is triggered by using a pen to drag a window to the edge of a monitor or monitor array. The default threshold is 30. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETPENDRAGOUTTHRESHOLD 0x0087	<p>Sets the threshold in pixels where undocking behavior is triggered by using a pen to drag a window from the edge of a monitor or monitor array to its center. The default threshold is 30. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p>

	<p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETPENSIDEMOVETHRESHOLD 0x008B	<p>Sets the threshold in pixels from the top of the monitor where a vertically maximized window is restored when dragged with a pen. The default threshold is 50. The <i>pvParam</i> parameter must point to a DWORD variable that contains the new value.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETSHOWIMEUI 0x006F	<p>Sets whether the IME status window is visible or not on a per-user basis. The <i>uiParam</i> parameter specifies TRUE for on or FALSE for off.</p>
SPI_SETSNAPSIZING 0x008F	<p>Sets whether a window is vertically maximized when it is sized to the top or bottom of the monitor. Set <i>pvParam</i> to TRUE for on or FALSE for off.</p> <p>SPI_GETWINARRANGING must be TRUE to enable this behavior.</p> <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.</p>
SPI_SETWINARRANGING 0x0083	<p>Sets whether window arrangement is enabled. Set <i>pvParam</i> to TRUE for on or FALSE for off.</p> <p>Window arrangement reduces the number of mouse, pen, or touch interactions needed to move and size top-level windows by simplifying the default behavior of a window when it is dragged or sized.</p> <p>The following parameters set individual window arrangement settings:</p> <ul style="list-style-type: none"> SPI_SETDOCKMOVING SPI_SETMOUSEDOCKTHRESHOLD SPI_SETMOUSEDRAゴOUTTHRESHOLD SPI_SETMOUSESIDEMOVETHRESHOLD SPI_SETPENDOCKTHRESHOLD SPI_SETPENDRAゴOUTTHRESHOLD SPI_SETPENSIDEMOVETHRESHOLD SPI_SETSNAPSIZING <p>Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is</p>

not supported.

[in] uiParam

Type: **UINT**

A parameter whose usage and format depends on the system parameter being queried or set. For more information about system-wide parameters, see the *uiAction* parameter. If not otherwise indicated, you must specify zero for this parameter.

[in, out] pvParam

Type: **PVOID**

A parameter whose usage and format depends on the system parameter being queried or set. For more information about system-wide parameters, see the *uiAction* parameter. If not otherwise indicated, you must specify **NULL** for this parameter. For information on the **PVOID** datatype, see [Windows Data Types](#).

[in] fWinIni

Type: **UINT**

If a system parameter is being set, specifies whether the user profile is to be updated, and if so, whether the [WM_SETTINGCHANGE](#) message is to be broadcast to all top-level windows to notify them of the change.

This parameter can be zero if you do not want to update the user profile or broadcast the [WM_SETTINGCHANGE](#) message, or it can be one or more of the following values.

Value	Meaning
SPIF_UPDATEINIFILE	Writes the new system-wide parameter setting to the user profile.
SPIF_SENDCHANGE	Broadcasts the WM_SETTINGCHANGE message after updating the user profile.
SPIF_SENDWININICHANGE	Same as SPIF_SENDCHANGE .

Return value

Type: **BOOL**

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

This function is intended for use with applications that allow the user to customize the environment.

A keyboard layout name should be derived from the hexadecimal value of the language identifier corresponding to the layout. For example, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named "00000409". Variants of U.S. English layout, such as the Dvorak layout, are named "00010409", "00020409" and so on. For a list of the primary language identifiers and sublanguage identifiers that make up a language identifier, see the **MAKELANGID** macro.

There is a difference between the High Contrast color scheme and the High Contrast Mode. The High Contrast color scheme changes the system colors to colors that have obvious contrast; you switch to this color scheme by using the Display Options in the control panel. The High Contrast Mode, which uses **SPI_GETHIGHCONTRAST** and **SPI_SETHIGHCONTRAST**, advises applications to modify their appearance for visually-impaired users. It involves such things as audible warning to users and customized color scheme (using the Accessibility Options in the control panel). For more information, see [HIGHCONTRAST](#). For more information on general accessibility features, see [Accessibility](#).

During the time that the primary button is held down to activate the Mouse ClickLock feature, the user can move the mouse. After the primary button is locked down, releasing the primary button does not result in a **WM_LBUTTONUP** message. Thus, it will appear to an application that the primary button is still down. Any subsequent button message releases the primary button, sending a **WM_LBUTTONUP** message to the application, thus the button can be unlocked programmatically or through the user clicking any button.

This API is not DPI aware, and should not be used if the calling thread is per-monitor DPI aware. For the DPI-aware version of this API, see [SystemParametersInfoForDPI](#). For more information on DPI awareness, see [the Windows High DPI documentation](#).

Examples

The following example uses **SystemParametersInfo** to double the mouse speed.

C++

```

#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "user32.lib")

void main()
{
    BOOL fResult;
    int aMouseInfo[3];      // Array for mouse information

    // Get the current mouse speed.
    fResult = SystemParametersInfo(SPI_GETMOUSE,           // Get mouse information
                                   0,                      // Not used
                                   &aMouseInfo,            // Holds mouse
information
                                   0);                    // Not used

    // Double it.
    if( fResult )
    {
        aMouseInfo[2] = 2 * aMouseInfo[2];

        // Change the mouse speed to the new value.
        SystemParametersInfo(SPI_SETMOUSE,           // Set mouse information
                           0,                      // Not used
                           aMouseInfo,             // Mouse information
                           SPIF_SENDCHANGE);     // Update Win.ini
    }
}

```

Note

The winuser.h header defines `SystemParametersInfo` 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-sysparams-ext-l1-1-0 (introduced in Windows 8)

See also

[ACCESSTIMEOUT](#)

[ANIMATIONINFO](#)

[AUDIODESCRIPTION](#)

[FILTERKEYS](#)

[HIGHCONTRAST](#)

[ICONMETRICS](#)

[LOGFONT](#)

[MAKELANGID](#)

[MINIMIZEDMETRICS](#)

[MOUSEKEYS](#)

[NONCLIENTMETRICS](#)

[RECT](#)

[SERIALKEYS](#)

[SOUNDSENTRY](#)

[STICKYKEYS](#)

[SystemParametersInfoForDPI](#)

[TOGGLEKEYS](#)

[WM_SETTINGCHANGE](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

TileWindows function (winuser.h)

Article 10/13/2021

Tiles the specified child windows of the specified parent window.

Syntax

C++

```
WORD TileWindows(
    [in, optional] HWND      hwndParent,
    [in]          UINT       wHow,
    [in, optional] const RECT *lpRect,
    [in]          UINT       cKids,
    [in, optional] const HWND *lpKids
);
```

Parameters

[in, optional] hwndParent

Type: **HWND**

A handle to the parent window. If this parameter is **NULL**, the desktop window is assumed.

[in] wHow

Type: **UINT**

The tiling flags. This parameter can be one of the following values—optionally combined with **MDITILE_SKIPDISABLED** to prevent disabled MDI child windows from being tiled.

Value	Meaning
MDITILE_HORIZONTAL 0x0001	Tiles windows horizontally.
MDITILE_VERTICAL 0x0000	Tiles windows vertically.

[in, optional] lpRect

Type: **const RECT***

A pointer to a structure that specifies the rectangular area, in client coordinates, within which the windows are arranged. If this parameter is **NULL**, the client area of the parent window is used.

[in] cKids

Type: **UINT**

The number of elements in the array specified by the *lpKids* parameter. This parameter is ignored if *lpKids* is **NULL**.

[in, optional] lpKids

Type: **const HWND***

An array of handles to the child windows to arrange. If a specified child window is a top-level window with the style **WS_EX_TOPMOST** or **WS_EX_TOOLWINDOW**, the child window is not arranged. If this parameter is **NULL**, all child windows of the specified parent window (or of the desktop window) are arranged.

Return value

Type: **WORD**

If the function succeeds, the return value is the number of windows arranged.

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

Remarks

Calling **TileWindows** causes all maximized windows to be restored to their previous size.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Library	User32.lib
DLL	User32.dll

See also

[CascadeWindows](#)

[Conceptual](#)

[Other Resources](#)

[RECT](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

TIMERPROC callback function (winuser.h)

Article05/03/2021

An application-defined callback function that processes [WM_TIMER](#) messages. The **TIMERPROC** type defines a pointer to this callback function. *TimerProc* is a placeholder for the application-defined function name.

Syntax

C++

```
TIMERPROC Timerproc;

void Timerproc(
    HWND unnamedParam1,
    UINT unnamedParam2,
    UINT_PTR unnamedParam3,
    DWORD unnamedParam4
)
{...}
```

Parameters

unnamedParam1

unnamedParam2

unnamedParam3

unnamedParam4

Return value

None

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

Conceptual

[KillTimer](#)

Reference

[SetTimer](#)

[Timers](#)

[WM_TIMER](#)

Feedback

Was this page helpful?

 Yes

 No

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

TITLEBARINFO structure (winuser.h)

Article 09/01/2022

Contains title bar information.

Syntax

C++

```
typedef struct tagTITLEBARINFO {
    DWORD cbSize;
    RECT rcTitleBar;
    DWORD rgstate[CCHILDRN_TITLEBAR + 1];
} TITLEBARINFO, *PTITLEBARINFO, *LPTITLEBARINFO;
```

Members

`cbSize`

Type: **DWORD**

The size, in bytes, of the structure. The caller must set this member to `sizeof(TITLEBARINFO)`.

`rcTitleBar`

Type: **RECT**

The coordinates of the title bar. These coordinates include all title-bar elements except the window menu.

`rgstate[CCHILDRN_TITLEBAR + 1]`

Type: **DWORD[CCHILDRN_TITLEBAR+1]**

An array that receives a value for each element of the title bar. The following are the title bar elements represented by the array.

Index	Title Bar Element
0	The title bar itself.
1	Reserved.

2	Minimize button.
3	Maximize button.
4	Help button.
5	Close button.

Each array element is a combination of one or more of the following values.

Value	Meaning
STATE_SYSTEM_FOCUSABLE 0x00100000	The element can accept the focus.
STATE_SYSTEM_INVISIBLE 0x00008000	The element is invisible.
STATE_SYSTEM_OFSSCREEN 0x00010000	The element has no visible representation.
STATE_SYSTEM_UNAVAILABLE 0x00000001	The element is unavailable.
STATE_SYSTEM_PRESSED 0x00000008	The element is in the pressed state.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[GetTitleBarInfo](#)

[Reference](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

TITLEBARINFOEX structure (winuser.h)

Article 09/01/2022

Expands on the information described in the [TITLEBARINFO](#) structure by including the coordinates of each element of the title bar.

This structure is sent with the [WM_GETTITLEBARINFOEX](#) message.

Syntax

C++

```
typedef struct tagTITLEBARINFOEX {
    DWORD cbSize;
    RECT rcTitleBar;
    DWORD rgstate[CCHILDREN_TITLEBAR + 1];
    RECT rgrect[CCHILDREN_TITLEBAR + 1];
} TITLEBARINFOEX, *PTITLEBARINFOEX, *LPTITLEBARINFOEX;
```

Members

`cbSize`

Type: [DWORD](#)

The size of the structure, in bytes. Set this member to `sizeof(TITLEBARINFOEX)` before sending with the [WM_GETTITLEBARINFOEX](#) message.

`rcTitleBar`

Type: [RECT](#)

The bounding rectangle of the title bar. The rectangle is expressed in screen coordinates and includes all titlebar elements except the window menu.

`rgstate[CCHILDREN_TITLEBAR + 1]`

Type: [DWORD\[CCHILDREN_TITLEBAR+1\]](#)

An array that receives a [DWORD](#) value for each element of the title bar. The following are the title bar elements represented by the array.

[Index](#)

[Title Bar Element](#)

0	The title bar itself.
1	Reserved.
2	Minimize button.
3	Maximize button.
4	Help button.
5	Close button.

Each array element is a combination of one or more of the following values.

Value	Meaning
STATE_SYSTEM_FOCUSABLE 0x00100000	The element can accept the focus.
STATE_SYSTEM_INVISIBLE 0x00008000	The element is invisible.
STATE_SYSTEM_OFFSCREEN 0x00010000	The element has no visible representation.
STATE_SYSTEM_UNAVAILABLE 0x00000001	The element is unavailable.
STATE_SYSTEM_PRESSED 0x00000008	The element is in the pressed state.

`rgrect[CCHILDREN_TITLEBAR + 1]`

Type: **RECT[CCHILDREN_TITLEBAR+1]**

An array that receives a structure for each element of the title bar. The structures are expressed in screen coordinates. The following are the title bar elements represented by the array.

Index	Title Bar Element
0	Reserved.
1	Reserved.
2	Minimize button.
3	Maximize button.

4

Help button.

5

Close button.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Reference](#)

[WM_GETTITLEBARINFOEX](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

TranslateMDISysAccel function (winuser.h)

Article08/04/2022

Processes accelerator keystrokes for window menu commands of the multiple-document interface (MDI) child windows associated with the specified MDI client window. The function translates [WM_KEYUP](#) and [WM_KEYDOWN](#) messages to [WM_SYSCOMMAND](#) messages and sends them to the appropriate MDI child windows.

Syntax

C++

```
BOOL TranslateMDISysAccel(
    [in] HWND hWndClient,
    [in] LPMMSG lpMsg
);
```

Parameters

[in] hWndClient

Type: **HWND**

A handle to the MDI client window.

[in] lpMsg

Type: **LPMMSG**

A pointer to a message retrieved by using the [GetMessage](#) or [PeekMessage](#) function. The message must be an [MSG](#) structure and contain message information from the application's message queue.

Return value

Type: **BOOL**

If the message is translated into a system command, the return value is nonzero.

If the message is not translated into a system command, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetMessage](#)
- [MSG](#)
- [Multiple Document Interface](#)
- [PeekMessage](#)
- [TranslateAccelerator](#)
- [WM_KEYDOWN](#)
- [WM_KEYUP](#)
- [WM_SYSCOMMAND](#)
- [Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

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

TranslateMessage function (winuser.h)

Article 08/04/2022

Translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the [GetMessage](#) or [PeekMessage](#) function.

Syntax

C++

```
BOOL TranslateMessage(  
    [in] const MSG *lpMsg  
)
```

Parameters

[in] lpMsg

Type: **const MSG***

A pointer to an [MSG](#) structure that contains message information retrieved from the calling thread's message queue by using the [GetMessage](#) or [PeekMessage](#) function.

Return value

Type: **BOOL**

If the message is translated (that is, a character message is posted to the thread's message queue), the return value is nonzero.

If the message is [WM_KEYDOWN](#), [WM_KEYUP](#), [WM_SYSKEYDOWN](#), or [WM_SYSKEYUP](#), the return value is nonzero, regardless of the translation.

If the message is not translated (that is, a character message is not posted to the thread's message queue), the return value is zero.

Remarks

The **TranslateMessage** function does not modify the message pointed to by the *lpMsg* parameter.

[WM_KEYDOWN](#) and [WM_KEYUP](#) combinations produce a [WM_CHAR](#) or [WM_DEADCHAR](#) message. [WM_SYSKEYDOWN](#) and [WM_SYSKEYUP](#) combinations produce a [WM_SYSCHAR](#) or [WM_SYSDEADCHAR](#) message.

TranslateMessage produces [WM_CHAR](#) messages only for keys that are mapped to ASCII characters by the keyboard driver.

If applications process virtual-key messages for some other purpose, they should not call **TranslateMessage**. For instance, an application should not call **TranslateMessage** if the [TranslateAccelerator](#) function returns a nonzero value. Note that the application is responsible for retrieving and dispatching input messages to the dialog box. Most applications use the main message loop for this. However, to permit the user to move to and to select controls by using the keyboard, the application must call [IsDialogMessage](#). For more information, see [Dialog Box Keyboard Interface](#).

Examples

For an example, see [Creating a Message Loop](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-message-l1-1-0 (introduced in Windows 8)

See also

- [GetMessage](#)
- [IsDialogMessage](#)

- [Messages and Message Queues](#)
 - [PeekMessage](#)
 - [TranslateAccelerator](#)
 - [WM_CHAR](#)
 - [WM_DEADCHAR](#)
 - [WM_KEYDOWN](#)
 - [WM_KEYUP](#)
 - [WM_SYSCHAR](#)
 - [WM_SYSDEADCHAR](#)
 - [WM_SYSKEYDOWN](#)
 - [WM_SYSKEYUP](#)
 - [Keyboard Input](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

UnhookWindowsHookEx function (winuser.h)

Article 10/13/2021

Removes a hook procedure installed in a hook chain by the [SetWindowsHookEx](#) function.

Syntax

C++

```
BOOL UnhookWindowsHookEx(  
    [in] HHOOK hhk  
);
```

Parameters

[in] hhk

Type: **HHOOK**

A handle to the hook to be removed. This parameter is a hook handle obtained by a previous call to [SetWindowsHookEx](#).

Return value

Type: **BOOL**

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 hook procedure can be in the state of being called by another thread even after **UnhookWindowsHookEx** returns. If the hook procedure is not being called concurrently, the hook procedure is removed immediately before **UnhookWindowsHookEx** returns.

Examples

For an example, see [Monitoring System Events](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Hooks](#)

[Reference](#)

[SetWindowsHookEx](#)

Feedback

Was this page helpful?

 Yes

 No

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

UnregisterClassA function (winuser.h)

Article02/09/2023

Unregisters a window class, freeing the memory required for the class.

Syntax

C++

```
BOOL UnregisterClassA(
    [in]          LPCSTR    lpClassName,
    [in, optional] HINSTANCE hInstance
);
```

Parameters

[in] *lpClassName*

Type: **LPCTSTR**

A null-terminated string or a class atom. If *lpClassName* is a string, it specifies the window class name. This class name must have been registered by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. System classes, such as dialog box controls, cannot be unregistered. If this parameter is an atom, it must be a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

[in, optional] *hInstance*

Type: **HINSTANCE**

A handle to the instance of the module that created the class.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

If the class could not be found or if a window still exists that was created with the class, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Before calling this function, an application must destroy all windows created with the specified class.

All window classes that an application registers are unregistered when it terminates.

Class atoms are special atoms returned only by [RegisterClass](#) and [RegisterClassEx](#).

No window classes registered by a DLL are unregistered when the .dll is unloaded.

Note

The winuser.h header defines UnregisterClass 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Reference](#)

[RegisterClass](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

UnregisterClassW function (winuser.h)

Article 02/09/2023

Unregisters a window class, freeing the memory required for the class.

Syntax

C++

```
BOOL UnregisterClassW(
    [in]          LPCWSTR lpClassName,
    [in, optional] HINSTANCE hInstance
);
```

Parameters

[in] *lpClassName*

Type: **LPCTSTR**

A null-terminated string or a class atom. If *lpClassName* is a string, it specifies the window class name. This class name must have been registered by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. System classes, such as dialog box controls, cannot be unregistered. If this parameter is an atom, it must be a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of *lpClassName*; the high-order word must be zero.

[in, optional] *hInstance*

Type: **HINSTANCE**

A handle to the instance of the module that created the class.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

If the class could not be found or if a window still exists that was created with the class, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Before calling this function, an application must destroy all windows created with the specified class.

All window classes that an application registers are unregistered when it terminates.

Class atoms are special atoms returned only by [RegisterClass](#) and [RegisterClassEx](#).

No window classes registered by a DLL are unregistered when the .dll is unloaded.

Note

The winuser.h header defines UnregisterClass 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	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-windowclass-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Reference](#)

[RegisterClass](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

UpdateLayeredWindow function (winuser.h)

Article11/19/2022

Updates the position, size, shape, content, and translucency of a layered window.

Syntax

C++

```
BOOL UpdateLayeredWindow(
    [in]           HWND      hWnd,
    [in, optional] HDC       hdcDst,
    [in, optional] POINT    *pptDst,
    [in, optional] SIZE     *psize,
    [in, optional] HDC       hdcSrc,
    [in, optional] POINT    *pptSrc,
    [in]           COLORREF crKey,
    [in, optional] BLENDFUNCTION *pblend,
    [in]           DWORD      dwFlags
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to a layered window. A layered window is created by specifying **WS_EX_LAYERED** when creating the window with the [CreateWindowEx](#) function.

Windows 8: The **WS_EX_LAYERED** style is supported for top-level windows and child windows. Previous Windows versions support **WS_EX_LAYERED** only for top-level windows.

[in, optional] hdcDst

Type: **HDC**

A handle to a DC for the screen. This handle is obtained by specifying **NULL** when calling the [GetDC](#) function. It is used for palette color matching when the window contents are updated. If *hdcDst* is **NULL**, the default palette will be used.

If *hdcSrc* is **NULL**, *hdcDst* must be **NULL**.

[in, optional] *pptDst*

Type: **POINT***

A pointer to a structure that specifies the new screen position of the layered window. If the current position is not changing, *pptDst* can be **NULL**.

[in, optional] *psize*

Type: **SIZE***

A pointer to a structure that specifies the new size of the layered window. If the size of the window is not changing, *psize* can be **NULL**. If *hdcSrc* is **NULL**, *psize* must be **NULL**.

[in, optional] *hdcSrc*

Type: **HDC**

A handle to a DC for the surface that defines the layered window. This handle can be obtained by calling the [CreateCompatibleDC](#) function. If the shape and visual context of the window are not changing, *hdcSrc* can be **NULL**.

[in, optional] *pptSrc*

Type: **POINT***

A pointer to a structure that specifies the location of the layer in the device context. If *hdcSrc* is **NULL**, *pptSrc* should be **NULL**.

[in] *crKey*

Type: **COLORREF**

A structure that specifies the color key to be used when composing the layered window. To generate a **COLORREF**, use the [RGB](#) macro.

[in, optional] *pblend*

Type: **BLENDFUNCTION***

A pointer to a structure that specifies the transparency value to be used when composing the layered window.

[in] *dwFlags*

Type: **DWORD**

This parameter can be one of the following values.

Value	Meaning
ULW_ALPHA 0x00000002	Use <i>pblend</i> as the blend function. If the display mode is 256 colors or less, the effect of this value is the same as the effect of ULW_OPAQUE .
ULW_COLORKEY 0x00000001	Use <i>crKey</i> as the transparency color.
ULW_OPAQUE 0x00000004	Draw an opaque layered window.
ULW_EX_NORESIZE 0x00000008	Force the UpdateLayeredWindowIndirect function to fail if the current window size does not match the size specified in the <i>psize</i> .

If *hdcSrc* is **NULL**, *dwFlags* should be zero.

Return value

Type: **BOOL**

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 source DC should contain the surface that defines the visible contents of the layered window. For example, you can select a bitmap into a device context obtained by calling the [CreateCompatibleDC](#) function.

An application should call [SetLayout](#) on the *hdcSrc* device context to properly set the mirroring mode. [SetLayout](#) will properly mirror all drawing into an **HDC** while properly preserving text glyph and (optionally) bitmap direction order. It cannot modify drawing directly into the bits of a device-independent bitmap (DIB). For more information, see [Window Layout and Mirroring](#).

The **UpdateLayeredWindow** function maintains the window's appearance on the screen. The windows underneath a layered window do not need to be repainted when they are uncovered due to a call to **UpdateLayeredWindow**, because the system will automatically repaint them. This permits seamless animation of the layered window.

UpdateLayeredWindow always updates the entire window. To update part of a window, use the traditional [WM_PAINT](#) and set the blend value using [SetLayeredWindowAttributes](#).

For best drawing performance by the layered window and any underlying windows, the layered window should be as small as possible. An application should also process the message and re-create its layered windows when the display's color depth changes.

For more information, see [Layered Windows](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[AlphaBlend](#)

[Conceptual](#)

[CreateCompatibleBitmap](#)

[Other Resources](#)

[Reference](#)

[SetWindowLong](#)

[SetWindowPos](#)

[TransparentBlt](#)

[UpdateLayeredWindowIndirect](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

UPDATELAYEREDWINDOWINFO structure (winuser.h)

Article 11/19/2022

Used by [UpdateLayeredWindowIndirect](#) to provide position, size, shape, content, and translucency information for a layered window.

Syntax

C++

```
typedef struct tagUPDATELAYEREDWINDOWINFO {
    DWORD          cbSize;
    HDC            hdcDst;
    const POINT    *pptDst;
    const SIZE     *psize;
    HDC            hdcSrc;
    const POINT    *pptSrc;
    COLORREF       crKey;
    const BLENDFUNCTION *pblend;
    DWORD          dwFlags;
    const RECT     *prcDirty;
} UPDATELAYEREDWINDOWINFO, *PUPDATELAYEREDWINDOWINFO;
```

Members

`cbSize`

Type: **DWORD**

The size, in bytes, of this structure.

`hdcDst`

Type: **HDC**

A handle to a DC for the screen. This handle is obtained by specifying **NULL** in this member when calling [UpdateLayeredWindowIndirect](#). The handle is used for palette color matching when the window contents are updated. If `hdcDst` is **NULL**, the default palette is used.

If `hdcSrc` is **NULL**, `hdcDst` must be **NULL**.

`pptDst`

Type: **const POINT***

The new screen position of the layered window. If the new position is unchanged from the current position, `pptDst` can be **NULL**.

`psize`

Type: **const SIZE***

The new size of the layered window. If the size of the window will not change, this parameter can be **NULL**. If `hdcSrc` is **NULL**, `psize` must be **NULL**.

`hdcSrc`

Type: **HDC**

A handle to the DC for the surface that defines the layered window. This handle can be obtained by calling the [CreateCompatibleDC](#) function. If the shape and visual context of the window will not change, `hdcSrc` can be **NULL**.

`pptSrc`

Type: **const POINT***

The location of the layer in the device context. If `hdcSrc` is **NULL**, `pptSrc` should be **NULL**.

`crKey`

Type: **COLORREF**

The color key to be used when composing the layered window. To generate a **COLORREF**, use the [RGB](#) macro.

`pblend`

Type: **const BLENDFUNCTION***

The transparency value to be used when composing the layered window.

`dwFlags`

Type: **DWORD**

This parameter can be one of the following values.

Value	Meaning
ULW_ALPHA 0x00000002	Use <i>pblend</i> as the blend function. If the display mode is 256 colors or less, the effect of this value is the same as the effect of ULW_OPAQUE .
ULW_COLORKEY 0x00000001	Use <i>crKey</i> as the transparency color.
ULW_OPAQUE 0x00000004	Draw an opaque layered window.
ULW_EX_NORESIZE 0x00000008	Force the UpdateLayeredWindowIndirect function to fail if the current window size does not match the size specified in the <i>psize</i> .

If **hdcSrc** is **NULL**, **dwFlags** should be zero.

prcDirty

Type: **const RECT***

The area to be updated. This parameter can be **NULL**. If it is non-**NULL**, only the area in this rectangle is updated from the source DC.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[Reference](#)

[UpdateLayeredWindow](#)

[Window Features](#)

Feedback

Was this page helpful?

 Yes

 No

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

WaitMessage function (winuser.h)

Article 06/29/2021

Yields control to other threads when a thread has no other messages in its message queue. The **WaitMessage** function suspends the thread and does not return until a new message is placed in the thread's message queue.

Syntax

C++

```
BOOL WaitMessage();
```

Return value

Type: **BOOL**

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 that **WaitMessage** does not return if there is unread input in the message queue after the thread has called a function to check the queue. This is because functions such as [PeekMessage](#), [GetMessage](#), [GetQueueStatus](#), [WaitMessage](#), [MsgWaitForMultipleObjects](#), and [MsgWaitForMultipleObjectsEx](#) check the queue and then change the state information for the queue so that the input is no longer considered new. A subsequent call to **WaitMessage** will not return until new input of the specified type arrives. The existing unread input (received prior to the last time the thread checked the queue) is ignored.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
--------------------------	---

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetMessage](#)

[Messages and Message Queues](#)

[PeekMessage](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

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

WindowFromPhysicalPoint function (winuser.h)

Article11/19/2022

Retrieves a handle to the window that contains the specified physical point.

Syntax

C++

```
HWND WindowFromPhysicalPoint(  
    [in] POINT Point  
);
```

Parameters

[in] Point

Type: [POINT](#)

The physical coordinates of the point.

Return value

Type: [HWND](#)

A handle to the window that contains the given physical point. If no window exists at the point, this value is [NULL](#).

Remarks

The [WindowFromPhysicalPoint](#) function does not retrieve a handle to a hidden or disabled window, even if the point is within the window.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[ChildWindowFromPoint](#)

[Conceptual](#)

[Other Resources](#)

[POINT](#)

[Reference](#)

[WindowFromDC](#)

[WindowFromPoint](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WindowFromPoint function (winuser.h)

Article11/19/2022

Retrieves a handle to the window that contains the specified point.

Syntax

C++

```
HWND WindowFromPoint(  
    [in] POINT Point  
) ;
```

Parameters

[in] Point

Type: [POINT](#)

The point to be checked.

Return value

Type: [HWND](#)

The return value is a handle to the window that contains the point. If no window exists at the given point, the return value is [NULL](#). If the point is over a static text control, the return value is a handle to the window under the static text control.

Remarks

The [WindowFromPoint](#) function does not retrieve a handle to a hidden or disabled window, even if the point is within the window. An application should use the [ChildWindowFromPoint](#) function for a nonrestrictive search.

Examples

For an example, see "Interface from Running Object Table" in [About Text Object Model](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-1 (introduced in Windows 8.1)

See also

[ChildWindowFromPoint](#)

[Conceptual](#)

[Other Resources](#)

[POINT](#)

[Reference](#)

[WindowFromDC](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WINDOWINFO structure (winuser.h)

Article 04/02/2021

Contains window information.

Syntax

C++

```
typedef struct tagWINDOWINFO {
    DWORD cbSize;
    RECT rcWindow;
    RECT rcClient;
    DWORD dwStyle;
    DWORD dwExStyle;
    DWORD dwWindowStatus;
    UINT cxWindowBorders;
    UINT cyWindowBorders;
    ATOM atomWindowType;
    WORD wCreatorVersion;
} WINDOWINFO, *PWINDOWINFO, *LPWINDOWINFO;
```

Members

`cbSize`

Type: **DWORD**

The size of the structure, in bytes. The caller must set this member to `sizeof(WINDOWINFO)`.

`rcWindow`

Type: **RECT**

The coordinates of the window.

`rcClient`

Type: **RECT**

The coordinates of the client area.

`dwStyle`

Type: **DWORD**

The window styles. For a table of window styles, see [Window Styles](#).

`dwExStyle`

Type: **DWORD**

The extended window styles. For a table of extended window styles, see [Extended Window Styles](#).

`dwWindowStatus`

Type: **DWORD**

The window status. If this member is **WS_ACTIVECAPTION** (0x0001), the window is active. Otherwise, this member is zero.

`cxWindowBorders`

Type: **UINT**

The width of the window border, in pixels.

`cyWindowBorders`

Type: **UINT**

The height of the window border, in pixels.

`atomWindowType`

Type: **ATOM**

The window class atom (see [RegisterClass](#)).

`wCreatorVersion`

Type: **WORD**

The Windows version of the application that created the window.

Requirements

Minimum supported client

Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

Conceptual

[CreateWindowEx](#)

[GetWindowInfo](#)

Reference

[RegisterClass](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WINDOWPLACEMENT structure (winuser.h)

Article11/19/2022

Contains information about the placement of a window on the screen.

Syntax

C++

```
typedef struct tagWINDOWPLACEMENT {  
    UINT    length;  
    UINT    flags;  
    UINT    showCmd;  
    POINT   ptMinPosition;  
    POINT   ptMaxPosition;  
    RECT    rcNormalPosition;  
    RECT    rcDevice;  
} WINDOWPLACEMENT;
```

Members

`length`

Type: **UINT**

The length of the structure, in bytes. Before calling the [GetWindowPlacement](#) or [SetWindowPlacement](#) functions, set this member to `sizeof(WINDOWPLACEMENT)`.

[GetWindowPlacement](#) and [SetWindowPlacement](#) fail if this member is not set correctly.

`flags`

Type: **UINT**

The flags that control the position of the minimized window and the method by which the window is restored. This member can be one or more of the following values.

Value	Meaning
WPF_ASYNCWINDOWPLACEMENT 0x0004	If the calling thread and the thread that owns the window are attached to different input queues, the system posts the request to the thread that owns the window. This

	prevents the calling thread from blocking its execution while other threads process the request.
WPF_RESTORETOMAXIMIZED 0x0002	The restored window will be maximized, regardless of whether it was maximized before it was minimized. This setting is only valid the next time the window is restored. It does not change the default restoration behavior. This flag is only valid when the SW_SHOWMINIMIZED value is specified for the showCmd member.
WPF_SETMINPOSITION 0x0001	The coordinates of the minimized window may be specified. This flag must be specified if the coordinates are set in the ptMinPosition member.

showCmd

Type: **UINT**

The current show state of the window. It can be any of the values that can be specified in the *nCmdShow* parameter for the [ShowWindow](#) function.

ptMinPosition

Type: **POINT**

The coordinates of the window's upper-left corner when the window is minimized.

ptMaxPosition

Type: **POINT**

The coordinates of the window's upper-left corner when the window is maximized.

rcNormalPosition

Type: **RECT**

The window's coordinates when the window is in the restored position.

rcDevice

Remarks

If the window is a top-level window that does not have the **WS_EX_TOOLWINDOW** window style, then the coordinates represented by the following members are in

workspace coordinates: **ptMinPosition**, **ptMaxPosition**, and **rcNormalPosition**.

Otherwise, these members are in screen coordinates.

Workspace coordinates differ from screen coordinates in that they take the locations and sizes of application toolbars (including the taskbar) into account. Workspace coordinate (0,0) is the upper-left corner of the workspace area, the area of the screen not being used by application toolbars.

The coordinates used in a **WINDOWPLACEMENT** structure should be used only by the [GetWindowPlacement](#) and [SetWindowPlacement](#) functions. Passing workspace coordinates to functions which expect screen coordinates (such as [SetWindowPos](#)) will result in the window appearing in the wrong location. For example, if the taskbar is at the top of the screen, saving window coordinates using [GetWindowPlacement](#) and restoring them using [SetWindowPos](#) causes the window to appear to "creep" up the screen.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[GetWindowPlacement](#)

[POINT](#)

[RECT](#)

[Reference](#)

[SetWindowPlacement](#)

[SetWindowPos](#)

[ShowWindow](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WINDOWPOS structure (winuser.h)

Article 04/02/2021

Contains information about the size and position of a window.

Syntax

C++

```
typedef struct tagWINDOWPOS {  
    HWND hwnd;  
    HWND hwndInsertAfter;  
    int x;  
    int y;  
    int cx;  
    int cy;  
    UINT flags;  
} WINDOWPOS, *LPWINDOWPOS, *PWINDOWPOS;
```

Members

hwnd

Type: **HWND**

A handle to the window.

hwndInsertAfter

Type: **HWND**

The position of the window in Z order (front-to-back position). This member can be a handle to the window behind which this window is placed, or can be one of the special values listed with the [SetWindowPos](#) function.

x

Type: **int**

The position of the left edge of the window.

y

Type: **int**

The position of the top edge of the window.

`cx`

Type: `int`

The window width, in pixels.

`cy`

Type: `int`

The window height, in pixels.

`flags`

Type: `UINT`

The window position. This member can be one or more of the following values.

Value	Meaning
<code>SWP_DRAWFRAME</code> 0x0020	Draws a frame (defined in the window's class description) around the window. Same as the <code>SWP_FRAMECHANGED</code> flag.
<code>SWP_FRAMECHANGED</code> 0x0020	Sends a <code>WM_NCCALCSIZE</code> message to the window, even if the window's size is not being changed. If this flag is not specified, <code>WM_NCCALCSIZE</code> is sent only when the window's size is being changed.
<code>SWP_HIDEWINDOW</code> 0x0080	Hides the window.
<code>SWP_NOACTIVATE</code> 0x0010	Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the topmost or non-topmost group (depending on the setting of the <code>hwndInsertAfter</code> member).
<code>SWP_NOCOPYBITS</code> 0x0100	Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
<code>SWP NOMOVE</code> 0x0002	Retains the current position (ignores the <code>x</code> and <code>y</code> members).
<code>SWP_NOOWNERZORDER</code> 0x0200	Does not change the owner window's position in the Z order.

SWP_NOREDRAW 0x0008	Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of the window being moved. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.
SWP_NOREPOSITION 0x0200	Does not change the owner window's position in the Z order. Same as the SWP_NOOWNERZORDER flag.
SWP_NOSENDCHANGING 0x0400	Prevents the window from receiving the WM_WINDOWPOSCHANGING message.
SWP_NOSIZE 0x0001	Retains the current size (ignores the cx and cy members).
SWP_NOZORDER 0x0004	Retains the current Z order (ignores the hwndInsertAfter member).
SWP_SHOWWINDOW 0x0040	Displays the window.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include Windows.h)

See also

[Conceptual](#)

[EndDeferWindowPos](#)

[Reference](#)

[SetWindowPos](#)

[WM_NCCALCSIZE](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WNDCLASSA structure (winuser.h)

Article 07/27/2022

Contains the window class attributes that are registered by the [RegisterClass](#) function.

This structure has been superseded by the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function. You can still use [WNDCLASS](#) and [RegisterClass](#) if you do not need to set the small icon associated with the window class.

Syntax

C++

```
typedef struct tagWNDCLASSA {
    UINT      style;
    WNDPROC   lpfnWndProc;
    int       cbClsExtra;
    int       cbWndExtra;
    HINSTANCE hInstance;
    HICON     hIcon;
    HCURSOR   hCursor;
    HBRUSH    hbrBackground;
    LPCSTR    lpszMenuName;
    LPCSTR    lpszClassName;
} WNDCLASSA, *PWNDCLASSA, *NPWNDCLASSA, *LPWNDCLASSA;
```

Members

`style`

Type: [UINT](#)

The class style(s). This member can be any combination of the [Class Styles](#).

`lpfnWndProc`

Type: [WNDPROC](#)

A pointer to the window procedure. You must use the [CallWindowProc](#) function to call the window procedure. For more information, see [WindowProc](#).

`cbClsExtra`

Type: [int](#)

The number of extra bytes to allocate following the window-class structure. The system initializes the bytes to zero.

`cbWndExtra`

Type: **int**

The number of extra bytes to allocate following the window instance. The system initializes the bytes to zero. If an application uses **WNDCLASS** to register a dialog box created by using the **CLASS** directive in the resource file, it must set this member to **DLGWINDOWEXTRA**.

`hInstance`

Type: **HINSTANCE**

A handle to the instance that contains the window procedure for the class.

`hIcon`

Type: **HICON**

A handle to the class icon. This member must be a handle to an icon resource. If this member is **NULL**, the system provides a default icon.

`hCursor`

Type: **HCURSOR**

A handle to the class cursor. This member must be a handle to a cursor resource. If this member is **NULL**, an application must explicitly set the cursor shape whenever the mouse moves into the application's window.

`hbrBackground`

Type: **HBRUSH**

A handle to the class background brush. This member can be a handle to the physical brush to be used for painting the background, or it can be a color value. A color value must be one of the following standard system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert it to one of the following **HBRUSH** types:

- **COLOR_ACTIVEBORDER**
- **COLOR_ACTIVECAPTION**
- **COLOR_APPWORKSPACE**

- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVEBORDER
- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

The system automatically deletes class background brushes when the class is unregistered by using [UnregisterClass](#). An application should not delete these brushes.

When this member is **NULL**, an application must paint its own background whenever it is requested to paint in its client area. To determine whether the background must be painted, an application can either process the [WM_ERASEBKGND](#) message or test the **fErase** member of the [PAINTSTRUCT](#) structure filled by the [BeginPaint](#) function.

`lpszMenuName`

Type: **LPCTSTR**

The resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the [MAKEINTRESOURCE](#) macro. If this member is **NULL**, windows belonging to this class have no default menu.

`lpszClassName`

Type: **LPCTSTR**

A pointer to a null-terminated string or is an atom. If this parameter is an atom, it must be a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of **lpszClassName**; the high-order word must be zero.

If **lpszClassName** is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined

control-class names.

The maximum length for `IpszClassName` is 256. If `IpszClassName` is greater than the maximum length, the [RegisterClass](#) function will fail.

Remarks

ⓘ Note

The `winuser.h` header defines `WNDCLASS` 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	<code>winuser.h</code> (include <code>Windows.h</code>)

See also

[BeginPaint](#)

[Conceptual](#)

[CreateWindow](#)

[CreateWindowEx](#)

[GetDC](#)

[MAKEINTRESOURCE](#)

[Other Resources](#)

[PAINTSTRUCT](#)

Reference

[RegisterClass](#)

[UnregisterClass](#)

[WM_PAINT](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

WNDCLASSEX structure (winuser.h)

Article 07/27/2022

Contains window class information. It is used with the [RegisterClassEx](#) and [GetClassInfoEx](#) functions.

The **WNDCLASSEX** structure is similar to the [WNDCLASS](#) structure. There are two differences. **WNDCLASSEX** includes the **cbSize** member, which specifies the size of the structure, and the **hIconSm** member, which contains a handle to a small icon associated with the window class.

Syntax

C++

```
typedef struct tagWNDCLASSEX {  
    UINT      cbSize;  
    UINT      style;  
    WNDPROC   lpfnWndProc;  
    int       cbClsExtra;  
    int       cbWndExtra;  
    HINSTANCE hInstance;  
    HICON     hIcon;  
    HCURSOR   hCursor;  
    HBRUSH    hbrBackground;  
    LPCSTR    lpszMenuName;  
    LPCSTR    lpszClassName;  
    HICON     hIconSm;  
} WNDCLASSEX, *PWNDCLASSEX, *NPWNDCLASSEX, *LPWNDCLASSEX;
```

Members

cbSize

Type: **UINT**

The size, in bytes, of this structure. Set this member to `sizeof(WNDCLASSEX)`. Be sure to set this member before calling the [GetClassInfoEx](#) function.

style

Type: **UINT**

The class style(s). This member can be any combination of the [Class Styles](#).

`lpfnWndProc`

Type: **WNDPROC**

A pointer to the window procedure. You must use the [CallWindowProc](#) function to call the window procedure. For more information, see [WindowProc](#).

`cbClsExtra`

Type: **int**

The number of extra bytes to allocate following the window-class structure. The system initializes the bytes to zero.

`cbWndExtra`

Type: **int**

The number of extra bytes to allocate following the window instance. The system initializes the bytes to zero. If an application uses **WNDCLASSEX** to register a dialog box created by using the **CLASS** directive in the resource file, it must set this member to **DLGWINDOWEXTRA**.

`hInstance`

Type: **HINSTANCE**

A handle to the instance that contains the window procedure for the class.

`hIcon`

Type: **HICON**

A handle to the class icon. This member must be a handle to an icon resource. If this member is **NULL**, the system provides a default icon.

`hCursor`

Type: **HCURSOR**

A handle to the class cursor. This member must be a handle to a cursor resource. If this member is **NULL**, an application must explicitly set the cursor shape whenever the mouse moves into the application's window.

`hbrBackground`

Type: **HBRUSH**

A handle to the class background brush. This member can be a handle to the brush to be used for painting the background, or it can be a color value. A color value must be one of the following standard system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert it to one of the following **HBRUSH** types:

- COLOR_ACTIVEBORDER
- COLOR_ACTIVECAPTION
- COLOR_APPWORKSPACE
- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVEBORDER
- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

The system automatically deletes class background brushes when the class is unregistered by using [UnregisterClass](#). An application should not delete these brushes.

When this member is **NULL**, an application must paint its own background whenever it is requested to paint in its client area. To determine whether the background must be painted, an application can either process the [WM_ERASEBKGND](#) message or test the **fErase** member of the [PAINTSTRUCT](#) structure filled by the [BeginPaint](#) function.

`lpszMenuName`

Type: **LPCTSTR**

Pointer to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the [MAKEINTRESOURCE](#) macro. If this member is **NULL**, windows belonging to this class have no default menu.

`lpszClassName`

Type: **LPCTSTR**

A pointer to a null-terminated string or is an atom. If this parameter is an atom, it must be a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of `lpszClassName`; the high-order word must be zero.

If `lpszClassName` is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined control-class names.

The maximum length for `lpszClassName` is 256. If `lpszClassName` is greater than the maximum length, the [RegisterClassEx](#) function will fail.

`hIconSm`

Type: **HICON**

A handle to a small icon that is associated with the window class. If this member is **NULL**, the system searches the icon resource specified by the `hIcon` member for an icon of the appropriate size to use as the small icon.

Remarks

Note

The winuser.h header defines `WNDCLASSEX` 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

winuser.h (include Windows.h)

See also

Conceptual

[GetClassInfoEx](#)

Reference

[RegisterClassEx](#)

[UnregisterClass](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

WNDCLASSEXW structure (winuser.h)

Article03/11/2023

Contains window class information. It is used with the [RegisterClassEx](#) and [GetClassInfoEx](#) functions.

The **WNDCLASSEX** structure is similar to the [WNDCLASS](#) structure. There are two differences. **WNDCLASSEX** includes the **cbSize** member, which specifies the size of the structure, and the **hIconSm** member, which contains a handle to a small icon associated with the window class.

Syntax

C++

```
typedef struct tagWNDCLASSEXW {
    UINT      cbSize;
    UINT      style;
    WNDPROC   lpfnWndProc;
    int       cbClsExtra;
    int       cbWndExtra;
    HINSTANCE hInstance;
    HICON     hIcon;
    HCURSOR   hCursor;
    HBRUSH    hbrBackground;
    LPCWSTR   lpszMenuName;
    LPCWSTR   lpszClassName;
    HICON     hIconSm;
} WNDCLASSEXW, *PWNDCLASSEXW, *NPWNDCLASSEXW, *LPWNDCLASSEXW;
```

Members

cbSize

Type: **UINT**

The size, in bytes, of this structure. Set this member to `sizeof(WNDCLASSEX)`. Be sure to set this member before calling the [GetClassInfoEx](#) function.

style

Type: **UINT**

The class style(s). This member can be any combination of the [Class Styles](#).

`lpfnWndProc`

Type: **WNDPROC**

A pointer to the window procedure. You must use the [CallWindowProc](#) function to call the window procedure. For more information, see [WindowProc](#).

`cbClsExtra`

Type: **int**

The number of extra bytes to allocate following the window-class structure. The system initializes the bytes to zero.

`cbWndExtra`

Type: **int**

The number of extra bytes to allocate following the window instance. The system initializes the bytes to zero. If an application uses **WNDCLASSEX** to register a dialog box created by using the **CLASS** directive in the resource file, it must set this member to **DLGWINDOWEXTRA**.

`hInstance`

Type: **HINSTANCE**

A handle to the instance that contains the window procedure for the class.

`hIcon`

Type: **HICON**

A handle to the class icon. This member must be a handle to an icon resource. If this member is **NULL**, the system provides a default icon.

`hCursor`

Type: **HCURSOR**

A handle to the class cursor. This member must be a handle to a cursor resource. If this member is **NULL**, an application must explicitly set the cursor shape whenever the mouse moves into the application's window.

`hbrBackground`

Type: **HBRUSH**

A handle to the class background brush. This member can be a handle to the brush to be used for painting the background, or it can be a color value. A color value must be one of the following standard system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert it to one of the following **HBRUSH** types:

- COLOR_ACTIVEBORDER
- COLOR_ACTIVECAPTION
- COLOR_APPWORKSPACE
- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVEBORDER
- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

The system automatically deletes class background brushes when the class is unregistered by using [UnregisterClass](#). An application should not delete these brushes.

When this member is **NULL**, an application must paint its own background whenever it is requested to paint in its client area. To determine whether the background must be painted, an application can either process the [WM_ERASEBKGND](#) message or test the **fErase** member of the [PAINTSTRUCT](#) structure filled by the [BeginPaint](#) function.

`lpszMenuName`

Type: **LPCTSTR**

Pointer to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the [MAKEINTRESOURCE](#) macro. If this member is **NULL**, windows belonging to this class have no default menu.

`lpszClassName`

Type: **LPCTSTR**

A pointer to a null-terminated string or is an atom. If this parameter is an atom, it must be a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of `lpszClassName`; the high-order word must be zero.

If `lpszClassName` is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined control-class names.

The maximum length for `lpszClassName` is 256. If `lpszClassName` is greater than the maximum length, the [RegisterClassEx](#) function will fail.

`hIconSm`

Type: **HICON**

A handle to a small icon that is associated with the window class. If this member is **NULL**, the system searches the icon resource specified by the `hIcon` member for an icon of the appropriate size to use as the small icon.

Remarks

Note

The winuser.h header defines `WNDCLASSEX` 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

winuser.h (include Windows.h)

See also

Conceptual

[GetClassInfoEx](#)

Reference

[RegisterClassEx](#)

[UnregisterClass](#)

[Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

WNDCLASSW structure (winuser.h)

Article 07/27/2022

Contains the window class attributes that are registered by the [RegisterClass](#) function.

This structure has been superseded by the [WNDCLASSEX](#) structure used with the [RegisterClassEx](#) function. You can still use [WNDCLASS](#) and [RegisterClass](#) if you do not need to set the small icon associated with the window class.

Syntax

C++

```
typedef struct tagWNDCLASSW {
    UINT      style;
    WNDPROC   lpfnWndProc;
    int       cbClsExtra;
    int       cbWndExtra;
    HINSTANCE hInstance;
    HICON     hIcon;
    HCURSOR   hCursor;
    HBRUSH    hbrBackground;
    LPCWSTR   lpszMenuName;
    LPCWSTR   lpszClassName;
} WNDCLASSW, *PWNDCLASSW, *NPWNDCLASSW, *LPWNDCLASSW;
```

Members

`style`

Type: [UINT](#)

The class style(s). This member can be any combination of the [Class Styles](#).

`lpfnWndProc`

Type: [WNDPROC](#)

A pointer to the window procedure. You must use the [CallWindowProc](#) function to call the window procedure. For more information, see [WindowProc](#).

`cbClsExtra`

Type: [int](#)

The number of extra bytes to allocate following the window-class structure. The system initializes the bytes to zero.

`cbWndExtra`

Type: **int**

The number of extra bytes to allocate following the window instance. The system initializes the bytes to zero. If an application uses **WNDCLASS** to register a dialog box created by using the **CLASS** directive in the resource file, it must set this member to **DLGWINDOWEXTRA**.

`hInstance`

Type: **HINSTANCE**

A handle to the instance that contains the window procedure for the class.

`hIcon`

Type: **HICON**

A handle to the class icon. This member must be a handle to an icon resource. If this member is **NULL**, the system provides a default icon.

`hCursor`

Type: **HCURSOR**

A handle to the class cursor. This member must be a handle to a cursor resource. If this member is **NULL**, an application must explicitly set the cursor shape whenever the mouse moves into the application's window.

`hbrBackground`

Type: **HBRUSH**

A handle to the class background brush. This member can be a handle to the physical brush to be used for painting the background, or it can be a color value. A color value must be one of the following standard system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert it to one of the following **HBRUSH** types:

- **COLOR_ACTIVEBORDER**
- **COLOR_ACTIVECAPTION**
- **COLOR_APPWORKSPACE**

- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVEBORDER
- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

The system automatically deletes class background brushes when the class is unregistered by using [UnregisterClass](#). An application should not delete these brushes.

When this member is **NULL**, an application must paint its own background whenever it is requested to paint in its client area. To determine whether the background must be painted, an application can either process the [WM_ERASEBKGND](#) message or test the **fErase** member of the [PAINTSTRUCT](#) structure filled by the [BeginPaint](#) function.

`lpszMenuName`

Type: **LPCTSTR**

The resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the [MAKEINTRESOURCE](#) macro. If this member is **NULL**, windows belonging to this class have no default menu.

`lpszClassName`

Type: **LPCTSTR**

A pointer to a null-terminated string or is an atom. If this parameter is an atom, it must be a class atom created by a previous call to the [RegisterClass](#) or [RegisterClassEx](#) function. The atom must be in the low-order word of **lpszClassName**; the high-order word must be zero.

If **lpszClassName** is a string, it specifies the window class name. The class name can be any name registered with [RegisterClass](#) or [RegisterClassEx](#), or any of the predefined

control-class names.

The maximum length for `IpszClassName` is 256. If `IpszClassName` is greater than the maximum length, the [RegisterClass](#) function will fail.

Remarks

ⓘ Note

The `winuser.h` header defines `WNDCLASS` 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	<code>winuser.h</code> (include <code>Windows.h</code>)

See also

[BeginPaint](#)

[Conceptual](#)

[CreateWindow](#)

[CreateWindowEx](#)

[GetDC](#)

[MAKEINTRESOURCE](#)

[Other Resources](#)

[PAINTSTRUCT](#)

Reference

[RegisterClass](#)

[UnregisterClass](#)

[WM_PAINT](#)

[WNDCLASSEX](#)

[Window Classes](#)

[WindowProc](#)

Feedback

Was this page helpful?

 Yes

 No

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

WNDPROC callback function (winuser.h)

Article05/03/2021

A callback function, which you define in your application, that processes messages sent to a window. The **WNDPROC** type defines a pointer to this callback function. The *WndProc* name is a placeholder for the name of the function that you define in your application.

Syntax

C++

```
WNDPROC Wndproc;

LRESULT Wndproc(
    HWND unnamedParam1,
    UINT unnamedParam2,
    WPARAM unnamedParam3,
    LPARAM unnamedParam4
)
{...}
```

Parameters

unnamedParam1

Type: [HWND](#)

A handle to the window. This parameter is typically named *hWnd*.

unnamedParam2

Type: [UINT](#)

The message. This parameter is typically named *uMsg*.

For lists of the system-provided messages, see [System-defined messages](#).

unnamedParam3

Type: [WPARAM](#)

Additional message information. This parameter is typically named *wParam*.

The contents of the *wParam* parameter depend on the value of the *uMsg* parameter.

unnamedParam4

Type: [LPARAM](#)

Additional message information. This parameter is typically named *lParam*.

The contents of the *lParam* parameter depend on the value of the *uMsg* parameter.

Return value

Type: [LRESULT](#)

The return value is the result of the message processing, and depends on the message sent.

Remarks

If your application runs on a 32-bit version of Windows operating system, uncaught exceptions from the callback will be passed onto higher-level exception handlers of your application when available. The system then calls the unhandled exception filter to handle the exception prior to terminating the process. If the PCA is enabled, it will offer to fix the problem the next time you run the application.

However, if your application runs on a 64-bit version of Windows operating system or WOW64, you should be aware that a 64-bit operating system handles uncaught exceptions differently based on its 64-bit processor architecture, exception architecture, and calling convention. The following table summarizes all possible ways that a 64-bit Windows operating system or WOW64 handles uncaught exceptions.

Behavior type	How the system handles uncaught exceptions
1	The system suppresses any uncaught exceptions.
2	The system first terminates the process, and then the Program Compatibility Assistant (PCA) offers to fix it the next time you run the application. You can disable the PCA mitigation by adding a Compatibility section to the application manifest .
3	The system calls the exception filters but suppresses any uncaught exceptions when it leaves the callback scope, without invoking the associated handlers.

The following table shows how a 64-bit version of the Windows operating system, and WOW64, handles uncaught exceptions. Notice that behavior type 2 applies only to the 64-bit version of the Windows 7 operating system and later.

Operating system	WOW64	64-bit Windows
Windows XP	3	1
Windows Server 2003	3	1
Windows Vista	3	1
Windows Vista SP1	1	1
Windows 7 and later	1	2

Note

On Windows 7 with SP1 (32-bit, 64-bit, or WOW64), the system calls the unhandled exception filter to handle the exception prior to terminating the process. If the Program Compatibility Assistant (PCA) is enabled, then it will offer to fix the problem the next time you run the application.

If you need to handle exceptions in your application, you can use structured exception handling to do so. For more information on how to use structured exception handling, see [Structured exception handling](#).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winuser.h (include windows.h)

See also

- [CallWindowProcW](#)
- [DefWindowProcW](#)
- [RegisterClassExW](#)
- [Window procedures](#)

Feedback

Was this page helpful?

 Yes

 No

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