

Windows and Messages

Article • 01/07/2021

The following sections describe the elements of an application with a Windows-based graphical user interface.

In This Section

Name	Description
Windows	Discusses windows in general.
Window Classes	Describes the types of window classes, how the system locates them, and the elements that define the default behavior of windows that belong to them.
Window Procedures	Discusses window procedures. Every window has an associated window procedure that processes all messages sent or posted to all windows of the class.
Messages and Message Queues	Describes messages and message queues and how to use them in your applications.
Timers	Discusses timers. A timer is an internal routine that repeatedly measures a specified interval, in milliseconds.
Window Properties	Discusses window properties. A window property is any data assigned to a window.
Configuration	Describes the functions that can be used to control the configuration of system metrics and various system attributes such as double-click time, screen saver time-out, window border width, and desktop pattern.
Hooks	Discusses hooks. A hook is a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic.
Multiple Document Interface	Discusses the Multiple Document Interface which is a specification that defines a user interface for applications that enable the user to work with more than one document at the same time.

Feedback

Was this page helpful?

 Yes

 No

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

Windows (Windows and Messages)

Article • 01/07/2021

In a graphical Windows-based application, a window is a rectangular area of the screen where the application displays output and receives input from the user. Therefore, one of the first tasks of a graphical Windows-based application is to create a window.

A window shares the screen with other windows, including those from other applications. Only one window at a time can receive input from the user. The user can use the mouse, keyboard, or other input device to interact with this window and the application that owns it.

In This Section

Name	Description
About Windows	Describes the programming elements that applications use to create and use windows; manage relationships between windows; and size, move, and display windows.
Using Windows	Contains examples that perform tasks associated with using windows.
Window Features	Discusses features of windows such as window types, states, size, and position.
Window Reference	Contains the API reference.

Window Functions

Name	Description
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.

Name	Description
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.
CalculatePopupWindowPosition	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.
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 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.

Name	Description
ChildWindowFromPointEx	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.
CloseWindow	Minimizes (but does not destroy) the specified window.
CreateWindow	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.
CreateWindowEx	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 .
DeferWindowPos	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.
DeregisterShellHookWindow	Unregisters a specified Shell window that is registered to receive Shell hook messages. It unregisters windows that are registered with a call to the RegisterShellHookWindow function.
DestroyWindow	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).
EndDeferWindowPos	Simultaneously updates the position and size of one or more windows in a single screen-refreshing cycle.
EndTask	Forcibly closes a specified window.

Name	Description
EnumChildProc	Application-defined callback function used with the EnumChildWindows function. It receives the child window handles. The WNDENUMPROC type defines a pointer to this callback function. EnumChildProc is a placeholder for the application-defined function name.
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. EnumChildWindows continues until the last child window is enumerated or the callback function returns FALSE .
EnumThreadWindows	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.
EnumThreadWndProc	An application-defined callback function used with the EnumThreadWindows function. It receives the window handles associated with a thread. The WNDENUMPROC type defines a pointer to this callback function. EnumThreadWndProc is a placeholder for the application-defined function name.
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 .
EnumWindowsProc	An application-defined callback function used with the EnumWindows or EnumDesktopWindows function. It receives top-level window handles. The WNDENUMPROC type defines a pointer to this callback function. EnumWindowsProc is a placeholder for the application-defined function name.
FindWindow	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.
FindWindowEx	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.

Name	Description
GetAltTabInfo	Retrieves status information for the specified window if it is the application-switching (ALT+TAB) window.
GetAncestor	Retrieves the handle to the ancestor of the specified window.
GetClientRect	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).
GetDesktopWindow	Returns 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	Returns 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.
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.
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. If the specified window is a topmost window, the function retrieves a handle to the next (or previous) topmost window. If the specified window is a top-level window, the function retrieves a handle to the next (or previous) top-level window. If the specified window is a child window, the function searches for a handle to the next (or previous) child window.
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.
GetShellWindow	Returns a handle to the Shell's desktop window.
GetTitleBarInfo	Retrieves information about the specified title bar.

Name	Description
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.
GetWindowModuleFileName	Retrieves the full path and file name of the module associated with the specified window handle.
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.
GetWindowText	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.
GetWindowTextLength	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.
GetWindowThreadProcessId	Retrieves the identifier of the thread that created the specified window and, optionally, the identifier of the process that created the window.
IsChild	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.
IsGUIThread	Determines whether the calling thread is already a GUI thread. It can also optionally convert the thread to a GUI thread.

Name	Description
IsHungAppWindow	Determines whether Windows 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.
IsIconic	Determines whether the specified window is minimized (iconic).
IsProcessDPIAware	Gets a value that indicates if the current process is dots per inch (dpi) aware such that it adjusts the sizes of UI elements to compensate for the dpi setting.
IsWindow	Determines whether the specified window handle identifies an existing window.
IsWindowUnicode	Determines whether the specified window is a native Unicode window.
IsWindowVisible	Retrieves the visibility state of the specified window.
IsZoomed	Determines whether a window is maximized.
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.
MoveWindow	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.
OpenIcon	Restores a minimized (iconic) window to its previous size and position; it then activates the window.
PhysicalToLogicalPoint	Converts the physical coordinates of a point in a window to logical coordinates.
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.
RealGetWindowClass	Retrieves a string that specifies the window type.

Name	Description
RegisterShellHookWindow	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.
SetForegroundWindow	Puts 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.
SetLayeredWindowAttributes	Sets the opacity and transparency color key of a layered window.
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	Sets the current process as dpi aware.
SetWindowDisplayAffinity	Stores the display affinity setting in kernel mode on the hWnd associated with the window.
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.
SetWindowText	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.
ShowOwnedPopups	Shows or hides all pop-up windows owned by the specified window.
ShowWindow	Sets the specified window's show state.

Name	Description
ShowWindowAsync	Sets the show state of a window created by a different thread.
SoundSentry	Triggers a visual signal to indicate that a sound is playing.
SwitchToThisWindow	Switches focus to a specified window and bring it to the foreground.
TileWindows	Tiles the specified child windows of the specified parent window.
UpdateLayeredWindow	Updates the position, size, shape, content, and translucency of a layered window.
UpdateLayeredWindowIndirect	Updates the position, size, shape, content, and translucency of a layered 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.
WinMain	WinMain is the conventional name for the user-provided entry point for a Windows-based application.

Window Macros

Name	Description
GET_X_LPARAM	Retrieves the signed x-coordinate from the given LPARAM value.
GET_Y_LPARAM	Retrieves the signed y-coordinate from the given LPARAM value.
HIBYTE	Retrieves the high-order byte from the given 16-bit value.
HIWORD	Retrieves the high-order word from the given 32-bit value.
LOBYTE	Retrieves the low-order byte from the specified value.
LOWORD	Retrieves the low-order word from the specified value.
MAKELONG	Creates a LONG value by concatenating the specified values.
MAKELPARAM	Creates a value for use as an <i>lParam</i> parameter in a message. The macro concatenates the specified values.

Name	Description
MAKELRESULT	Creates a value for use as a return value from a window procedure. The macro concatenates the specified values.
MAKEWORD	Creates a WORD value by concatenating the specified values.
MAKEWPARAM	Creates a value for use as a <i>wParam</i> parameter in a message. The macro concatenates the specified values.

Window Messages

Name	Description
MN_GETHMENU	Gets the HMENU for the current window.
WM_GETFONT	Retrieves the font with which the control is currently drawing its text.
WM_GETTEXT	Copies the text that corresponds to a window into a buffer provided by the caller.
WM_GETTEXTLENGTH	Determine the length, in characters, of the text associated with a window.
WM_SETFONT	Specifies the font that a control is to use when drawing text.
WM_SETICON	Associates a new large or small icon with a window. The system displays the large icon in the ALT+TAB dialog box, and the small icon in the window caption.
WM_SETTEXT	Sets the text of a window.

Window Notifications

Name	Description
WM_ACTIVATEAPP	<p>Sent when a window belonging to a different application than the active window is about to be activated. The message is sent to the application whose window is being activated and to the application whose window is being deactivated.</p> <p>A window receives this message through its WindowProc function.</p>

Name	Description
WM_CANCELMODE	Sent to cancel certain modes, such as mouse capture. For example, the system sends this message to the active window when a dialog box or message box is displayed. Certain functions also send this message explicitly to the specified window regardless of whether it is the active window. For example, the EnableWindow function sends this message when disabling the specified window.
WM_CHILDACTIVATE	Sent to a child window when the user clicks the window's title bar or when the window is activated, moved, or sized.
WM_CLOSE	Sent as a signal that a window or an application should terminate.
WM_COMPACTING	Sent to all top-level windows when the system detects more than 12.5 percent of system time over a 30- to 60-second interval is being spent compacting memory. This indicates that system memory is low.
WM_CREATE	Sent when an application requests that a window be created by calling the CreateWindowEx or CreateWindow function. (The message is sent before the function returns.) The window procedure of the new window receives this message after the window is created, but before the window becomes visible.
WM_DESTROY	Sent when a window is being destroyed. It is sent to the window procedure of the window being destroyed after the window is removed from the screen. This message is sent first to the window being destroyed and then to the child windows (if any) as they are destroyed. During the processing of the message, it can be assumed that all child windows still exist.
WM_ENABLE	Sent when an application changes the enabled state of a window. It is sent to the window whose enabled state is changing. This message is sent before the EnableWindow function returns, but after the enabled state (WS_DISABLED style bit) of the window has changed.

Name	Description
WM_ENTERSIZEMOVE	Sent one time to a window after it enters the moving or sizing modal loop. The window enters the moving or sizing modal loop when the user clicks the window's title bar or sizing border, or when the window passes the WM_SYSCOMMAND message to the DefWindowProc function and the <i>wParam</i> parameter of the message specifies the SC_MOVE or SC_SIZE value. The operation is complete when DefWindowProc returns. The system sends the WM_ENTERSIZEMOVE message regardless of whether the dragging of full windows is enabled.
WM_ERASEBKGND	Sent when the window background must be erased (for example, when a window is resized). The message is sent to prepare an invalidated portion of a window for painting.
WM_EXITSIZEMOVE	Sent one time to a window, after it has exited the moving or sizing modal loop. The window enters the moving or sizing modal loop when the user clicks the window's title bar or sizing border, or when the window passes the WM_SYSCOMMAND message to the DefWindowProc function and the <i>wParam</i> parameter of the message specifies the SC_MOVE or SC_SIZE value. The operation is complete when DefWindowProc returns.
WM_GETICON	Sent to a window to retrieve a handle to the large or small icon associated with a window. The system displays the large icon in the ALT+TAB dialog, and the small icon in the window caption.
WM_GETMINMAXINFO	Sent to a window when the size or position of the window is about to change. An application can use this message to override the window's default maximized size and position, or its default minimum or maximum tracking size.
WM_INPUTLANGCHANGE	Sent to the topmost affected window after an application's input language has been changed. You should make any application-specific settings and pass the message to the DefWindowProc function, which passes the message to all first-level child windows. These child windows can pass the message to DefWindowProc to have it pass the message to their child windows, and so on.

Name	Description
WM_INPUTLANGCHANGEREQUEST	Posted to the window with the focus when the user chooses a new input language, either with the hotkey (specified in the Keyboard control panel application) or from the indicator on the system taskbar. An application can accept the change by passing the message to the DefWindowProc function or reject the change (and prevent it from taking place) by returning immediately.
WM_MOVE	Sent after a window has been moved.
WM_MOVING	Sent to a window that the user is moving. By processing this message, an application can monitor the position of the drag rectangle and, if needed, change its position.
WM_NCACTIVATE	Sent to a window when its nonclient area needs to be changed to indicate an active or inactive state.
WM_NCCALCSIZE	Sent when the size and position of a window's client area must be calculated. By processing this message, an application can control the content of the window's client area when the size or position of the window changes.
WM_NCCREATE	Sent prior to the WM_CREATE message when a window is first created.
WM_NCDESTROY	Informs a window that its nonclient area is being destroyed. The DestroyWindow function sends the WM_NCDESTROY message to the window following the WM_DESTROY message. WM_DESTROY is used to free the allocated memory object associated with the window. The WM_NCDESTROY message is sent after the child windows have been destroyed. In contrast, WM_DESTROY is sent before the child windows are destroyed.
WM_NULL	Performs no operation. An application sends the WM_NULL message if it wants to post a message that the recipient window will ignore.
WM_PARENTNOTIFY	Sent to the parent of a child window when the child window is created or destroyed, or when the user clicks a mouse button while the cursor is over the child window. When the child window is being created, the system sends WM_PARENTNOTIFY just before the CreateWindow or CreateWindowEx function that creates the window returns. When the child window is being destroyed, the system sends the message before any processing to destroy the window takes place.

Name	Description
WM_QUERYDRAGICON	Sent to a minimized (iconic) window. The window is about to be dragged by the user but does not have an icon defined for its class. An application can return a handle to an icon or cursor. The system displays this cursor or icon while the user drags the icon.
WM_QUERYOPEN	Sent to an icon when the user requests that the window be restored to its previous size and position.
WM_QUIT	Indicates a request to terminate an application, and is generated when the application calls the PostQuitMessage function. It causes the GetMessage function to return zero.
WM_SHOWWINDOW	Sent to a window when the window is about to be hidden or shown.
WM_SIZE	Sent to a window after its size has changed.
WM_SIZING	Sent to a window that the user is resizing. By processing this message, an application can monitor the size and position of the drag rectangle and, if needed, change its size or position.
WM_STYLECHANGED	Sent to a window after the SetWindowLong function has changed one or more of the window's styles.
WM_STYLECHANGING	Sent to a window when the SetWindowLong function is about to change one or more of the window's styles.
WM_THEMECHANGED	Broadcast to every window following a theme change event. Examples of theme change events are the activation of a theme, the deactivation of a theme, or a transition from one theme to another.
WM_USERCHANGED	Sent to all windows after the user has logged on or off. When the user logs on or off, the system updates the user-specific settings. The system sends this message immediately after updating the settings.
WM_WINDOWPOSCHANGED	Sent to a window whose size, position, or place in the Z order has changed as a result of a call to the SetWindowPos function or another window-management function.
WM_WINDOWPOSCHANGING	Sent to a window whose size, position, or place in the Z order is about to change as a result of a call to the SetWindowPos function or another window-management function.

Window Structures

Name	Description
ALTTABINFO	Contains status information for the application-switching (ALT+TAB) window.
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. An application passes a pointer to this structure as the <i>lpParam</i> parameter of the CreateWindow function when creating an MDI client window.
CREATESTRUCT	Defines the initialization parameters passed to the window procedure of an application. These members are identical to the parameters of the CreateWindowEx function.
GUITHREADINFO	Contains information about a GUI thread.
MINMAXINFO	Contains information about a window's maximized size and position and its minimum and maximum tracking size.
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.
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.

Window Constants

Name	Description
Extended Window Styles	Styles that can be specified wherever an extended window style is required.
Window Styles	Styles that can be specified wherever a window style is required. After the control has been created, these styles cannot be modified, except as noted.

Feedback

Was this page helpful?



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

Window Overviews

Article • 04/27/2021

- [About Windows](#)
 - [Window Features](#)
 - [Using Windows](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

About Windows

Article • 01/07/2021

This topic describes the programming elements that applications use to create and use windows; manage relationships between windows; and size, move, and display windows.

The overview includes the following topics.

- [Desktop Window](#)
- [Application Windows](#)
 - [Client Area](#)
 - [Nonclient Area](#)
- [Controls and Dialog Boxes](#)
- [Window Attributes](#)
 - [Class Name](#)
 - [Window Name](#)
 - [Window Style](#)
 - [Extended Window Style](#)
 - [Position](#)
 - [Size](#)
 - [Parent or Owner Window Handle](#)
 - [Menu Handle or Child-Window Identifier](#)
 - [Application Instance Handle](#)
 - [Creation Data](#)
 - [Window Handle](#)
- [Window Creation](#)
 - [Main Window Creation](#)
 - [Window-Creation Messages](#)
 - [Multithread Applications](#)

Desktop Window

When you start the system, it automatically creates the desktop window. The *desktop window* is a system-defined window that paints the background of the screen and serves as the base for all windows displayed by all applications.

The desktop window uses a bitmap to paint the background of the screen. The pattern created by the bitmap is called the *desktop wallpaper*. By default, the desktop window uses the bitmap from a .bmp file specified in the registry as the desktop wallpaper.

The [GetDesktopWindow](#) function returns a handle to the desktop window.

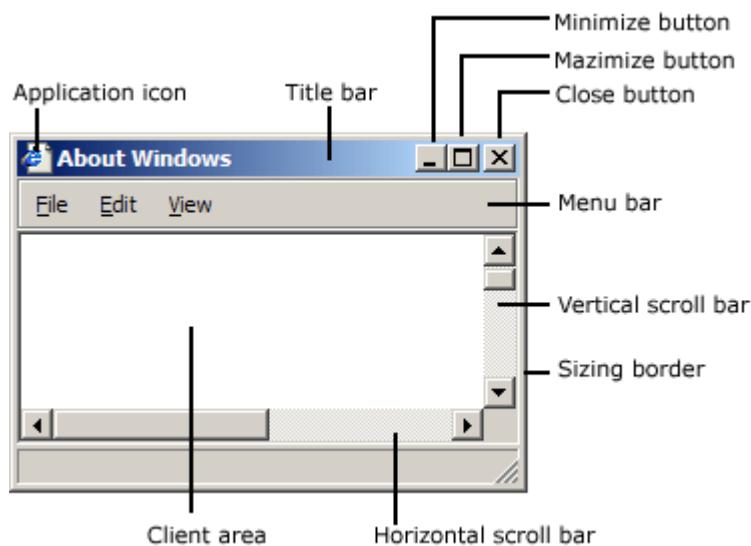
A system configuration application, such as a Control Panel item, changes the desktop wallpaper by using the [SystemParametersInfo](#) function with the *wAction* parameter set to **SPI_SETDESKWALLPAPER** and the *lParam* parameter specifying a bitmap file name. [SystemParametersInfo](#) then loads the bitmap from the specified file, uses the bitmap to paint the background of the screen, and enters the new file name in the registry.

Application Windows

Every graphical Windows-based application creates at least one window, called the *main window*, that serves as the primary interface between the user and the application. Most applications also create other windows, either directly or indirectly, to perform tasks related to the main window. Each window plays a part in displaying output and receiving input from the user.

When you start an application, the system also associates a taskbar button with the application. The *taskbar button* contains the program icon and title. When the application is active, its taskbar button is displayed in the pushed state.

An application window includes elements such as a title bar, a menu bar, the window menu (formerly known as the system menu), the minimize button, the maximize button, the restore button, the close button, a sizing border, a client area, a horizontal scroll bar, and a vertical scroll bar. An application's main window typically includes all of these components. The following illustration shows these components in a typical main window.



Client Area

The *client area* is the part of a window where the application displays output, such as text or graphics. For example, a desktop publishing application displays the current page

of a document in the client area. The application must provide a function, called a window procedure, to process input to the window and display output in the client area. For more information, see [Window Procedures](#).

Nonclient Area

The title bar, menu bar, window menu, minimize and maximize buttons, sizing border, and scroll bars are referred to collectively as the window's *nonclient area*. The system manages most aspects of the nonclient area; the application manages the appearance and behavior of its client area.

The *title bar* displays an application-defined icon and line of text; typically, the text specifies the name of the application or indicates the purpose of the window. An application specifies the icon and text when creating the window. The title bar also makes it possible for the user to move the window by using a mouse or other pointing device.

Most applications include a *menu bar* that lists the commands supported by the application. Items in the menu bar represent the main categories of commands. Clicking an item on the menu bar typically opens a pop-up menu whose items correspond to the tasks within a given category. By clicking a command, the user directs the application to carry out a task.

The *window menu* is created and managed by the system. It contains a standard set of menu items that, when chosen by the user, set a window's size or position, close the application, or perform tasks. For more information, see [Menus](#).

The buttons in the upper-right corner affect the size and position of the window. When you click the *maximize button*, the system enlarges the window to the size of the screen and positions the window, so it covers the entire desktop, minus the taskbar. At the same time, the system replaces the maximize button with the restore button. When you click the *restore button*, the system restores the window to its previous size and position. When you click the *minimize button*, the system reduces the window to the size of its taskbar button, positions the window over the taskbar button, and displays the taskbar button in its normal state. To restore the application to its previous size and position, click its taskbar button. When you click the *close button*, the application exits.

The *sizing border* is an area around the perimeter of the window that enables the user to size the window by using a mouse or other pointing device.

The *horizontal scroll bar* and *vertical scroll bar* convert mouse or keyboard input into values that an application uses to shift the contents of the client area either horizontally or vertically. For example, a word-processing application that displays a lengthy

document typically provides a vertical scroll bar to enable the user to page up and down through the document.

Controls and Dialog Boxes

An application can create several types of windows in addition to its main window, including controls and dialog boxes.

A *control* is a window that an application uses to obtain a specific piece of information from the user, such as the name of a file to open or the desired point size of a text selection. Applications also use controls to obtain information needed to control a particular feature of an application. For example, a word-processing application typically provides a control to let the user turn word wrapping on and off. For more information, see [Windows Controls](#).

Controls are always used in conjunction with another window—typically, a dialog box. A *dialog box* is a window that contains one or more controls. An application uses a dialog box to prompt the user for input needed to complete a command. For example, an application that includes a command to open a file would display a dialog box that includes controls in which the user specifies a path and file name. Dialog boxes do not typically use the same set of window components as does a main window. Most have a title bar, a window menu, a border (non-sizing), and a client area, but they typically do not have a menu bar, minimize and maximize buttons, or scroll bars. For more information, see [Dialog Boxes](#).

A *message box* is a special dialog box that displays a note, caution, or warning to the user. For example, a message box can inform the user of a problem the application has encountered while performing a task. For more information, see [Message Boxes](#).

Window Attributes

An application must provide the following information when creating a window. (With the exception of the [Window Handle](#), which the creation function returns to uniquely identify the new window.)

- [Class Name](#)
- [Window Name](#)
- [Window Style](#)
- [Extended Window Style](#)
- [Position](#)
- [Size](#)

- Parent or Owner Window Handle
- Menu Handle or Child-Window Identifier
- Application Instance Handle
- Creation Data
- Window Handle

These window attributes are described in the following sections.

Class Name

Every window belongs to a window class. An application must register a window class before creating any windows of that class. The *window class* defines most aspects of a window's appearance and behavior. The chief component of a window class is the *window procedure*, a function that receives and processes all input and requests sent to the window. The system provides the input and requests in the form of *messages*. For more information, see [Window Classes](#), [Window Procedures](#), and [Messages and Message Queues](#).

Window Name

A *window name* is a text string that identifies a window for the user. A main window, dialog box, or message box typically displays its window name in its title bar, if present. A control may display its window name, depending on the control's class. For example, buttons, edit controls, and static controls displays their window names within the rectangle occupied by the control. However, controls such as list boxes and combo boxes do not display their window names.

To change the window name after creating a window, use the [SetWindowText](#) function . This function uses the [GetWindowTextLength](#) and [GetWindowText](#) functions to retrieve the current window-name string from the window.

Window Style

Every window has one or more window styles. A window style is a named constant that defines an aspect of the window's appearance and behavior that is not specified by the window's class. An application usually sets window styles when creating windows. It can also set the styles after creating a window by using the [SetWindowLong](#) function.

The system and, to some extent, the window procedure for the class, interpret the window styles.

Some window styles apply to all windows, but most apply to windows of specific window classes. The general window styles are represented by constants that begin with the WS_ prefix; they can be combined with the OR operator to form different types of windows, including main windows, dialog boxes, and child windows. The class-specific window styles define the appearance and behavior of windows belonging to the predefined control classes. For example, the **SCROLLBAR** class specifies a scroll bar control, but the **SBS_HORZ** and **SBS_VERT** styles determine whether a horizontal or vertical scroll bar control is created.

For lists of styles that can be used by windows, see the following topics:

- [Window Styles](#)
- [Button Styles](#)
- [Combo Box Styles](#)
- [Edit Control Styles](#)
- [List Box Styles](#)
- [Rich Edit Control Styles](#)
- [Scroll Bar Control Styles](#)
- [Static Control Styles](#)

Extended Window Style

Every window can optionally have one or more extended window styles. An *extended window style* is a named constant that defines an aspect of the window's appearance and behavior that is not specified by the window class or the other window styles. An application usually sets extended window styles when creating windows. It can also set the styles after creating a window by using the [SetWindowLong](#) function.

For more information, see [CreateWindowEx](#).

Position

A window's position is defined as the coordinates of its upper left corner. These coordinates, sometimes called window coordinates, are always relative to the upper left corner of the screen or, for a child window, the upper left corner of the parent window's client area. For example, a top-level window having the coordinates (10,10) is placed 10 pixels to the right of the upper left corner of the screen and 10 pixels down from it. A child window having the coordinates (10,10) is placed 10 pixels to the right of the upper left corner of its parent window's client area and 10 pixels down from it.

The [WindowFromPoint](#) function retrieves a handle to the window occupying a particular point on the screen. Similarly, the [ChildWindowFromPoint](#) and

[ChildWindowFromPointEx](#) functions retrieve a handle to the child window occupying a particular point in the parent window's client area. Although [ChildWindowFromPointEx](#) can ignore invisible, disabled, and transparent child windows, [ChildWindowFromPoint](#) cannot.

Size

A window's size (width and height) is given in pixels. A window can have zero width or height. If an application sets a window's width and height to zero, the system sets the size to the default minimum window size. To discover the default minimum window size, an application uses the [GetSystemMetrics](#) function with the **SM_CXMIN** and **SM_CYMIN** flags.

An application may need to create a window with a client area of a particular size. The [AdjustWindowRect](#) and [AdjustWindowRectEx](#) functions calculate the required size of a window based on the desired size of the client area. The application can pass the resulting size values to the [CreateWindowEx](#) function.

An application can size a window so that it is extremely large; however, it should not size a window so that it is larger than the screen. Before setting a window's size, the application should check the width and height of the screen by using [GetSystemMetrics](#) with the **SM_CXSCREEN** and **SM_CYSCREEN** flags.

Parent or Owner Window Handle

A window can have a parent window. A window that has a parent is called a *child window*. The *parent window* provides the coordinate system used for positioning a child window. Having a parent window affects aspects of a window's appearance; for example, a child window is clipped so that no part of the child window can appear outside the borders of its parent window.

A window that has no parent, or whose parent is the desktop window, is called a *top-level window*. An application can use the [EnumWindows](#) function to obtain a handle to each top-level window on the screen. [EnumWindows](#) passes the handle to each top-level window, in turn, to an application-defined callback function, [EnumWindowsProc](#).

A top-level window can own, or be owned by, another window. An *owned window* always appears in front of its owner window, is hidden when its owner window is minimized, and is destroyed when its owner window is destroyed. For more information, see [Owned Windows](#).

Menu Handle or Child-Window Identifier

A child window can have a *child-window identifier*, a unique, application-defined value associated with the child window. Child-window identifiers are especially useful in applications that create multiple child windows. When creating a child window, an application specifies the identifier of the child window. After creating the window, the application can change the window's identifier by using the [SetWindowLong](#) function, or it can retrieve the identifier by using the [GetWindowLong](#) function.

Every window, except a child window, can have a menu. An application can include a menu by providing a menu handle either when registering the window's class or when creating the window.

Application Instance Handle

Every application has an instance handle associated with it. The system provides the instance handle to an application when the application starts. Because it can run multiple copies of the same application, the system uses instance handles internally to distinguish one instance of an application from another. The application must specify the instance handle in many different windows, including those that create windows.

Creation Data

Every window can have application-defined creation data associated with it. When the window is first created, the system passes a pointer to the data on to the window procedure of the window being created. The window procedure uses the data to initialize application-defined variables.

Window Handle

After creating a window, the creation function returns a *window handle* that uniquely identifies the window. A window handle has the **HWND** data type; an application must use this type when declaring a variable that holds a window handle. An application uses this handle in other functions to direct their actions to the window.

An application can use the [FindWindow](#) function to discover whether a window with the specified class name or window name exists in the system. If such a window exists, [FindWindow](#) returns a handle to the window. To limit the search to the child windows of a particular application, use the [FindWindowEx](#) function.

The [IsWindow](#) function determines whether a window handle identifies a valid, existing window. There are special constants that can replace a window handle in certain functions. For example, an application can use [HWND_BROADCAST](#) in the [SendMessage](#) and [SendMessageTimeout](#) functions, or [HWND_DESKTOP](#) in the [MapWindowPoints](#) function.

Window Creation

To create application windows, use the [CreateWindow](#) or [CreateWindowEx](#) function. You must provide the information required to define the window attributes. [CreateWindowEx](#) has a parameter, *dwExStyle*, that [CreateWindow](#) does not have; otherwise, the functions are identical. In fact, [CreateWindow](#) simply calls [CreateWindowEx](#) with the *dwExStyle* parameter set to zero. For this reason, the remainder of this overview refers only to [CreateWindowEx](#).

This section contains the following topics:

- [Main Window Creation](#)
- [Window-Creation Messages](#)
- [Multithread Applications](#)

ⓘ Note

There are additional functions for creating special-purpose windows such as dialog boxes and message boxes. For more information, see [DialogBox](#), [CreateDialog](#), and [MessageBox](#).

Main Window Creation

Every Windows-based application must have [WinMain](#) as its entry point function. [WinMain](#) performs a number of tasks, including registering the window class for the main window and creating the main window. [WinMain](#) registers the main window class by calling the [RegisterClass](#) function, and it creates the main window by calling the [CreateWindowEx](#) function.

Your [WinMain](#) function can also limit your application to a single instance. Create a named mutex using the [CreateMutex](#) function. If [GetLastError](#) returns [ERROR_ALREADY_EXISTS](#), another instance of your application exists (it created the mutex) and you should exit [WinMain](#).

The system does not automatically display the main window after creating it; instead, an application must use the [ShowWindow](#) function to display the main window. After creating the main window, the application's [WinMain](#) function calls [ShowWindow](#), passing it two parameters: a handle to the main window and a flag specifying whether the main window should be minimized or maximized when it is first displayed. Normally, the flag can be set to any of the constants beginning with the SW_ prefix. However, when [ShowWindow](#) is called to display the application's main window, the flag must be set to [SW_SHOWDEFAULT](#). This flag tells the system to display the window as directed by the program that started the application.

If a window class was registered with the Unicode version of [RegisterClass](#), the window receives only Unicode messages. To determine whether a window uses the Unicode character set or not, call [IsWindowUnicode](#).

Window-Creation Messages

When creating any window, the system sends messages to the window procedure for the window. The system sends the [WM_NCCREATE](#) message after creating the window's nonclient area and the [WM_CREATE](#) message after creating the client area. The window procedure receives both messages before the system displays the window. Both messages include a pointer to a [CREATESTRUCT](#) structure that contains all the information specified in the [CreateWindowEx](#) function. Typically, the window procedure performs initialization tasks upon receiving these messages.

When creating a child window, the system sends the [WM_PARENTNOTIFY](#) message to the parent window after sending the [WM_NCCREATE](#) and [WM_CREATE](#) messages. It also sends other messages while creating a window. The number and order of these messages depend on the window class and style and on the function used to create the window. These messages are described in other topics in this help file.

Multithread Applications

A Windows-based application can have multiple threads of execution, and each thread can create windows. The thread that creates a window must contain the code for its window procedure.

An application can use the [EnumThreadWindows](#) function to enumerate the windows created by a particular thread. This function passes the handle to each thread window, in turn, to an application-defined callback function, [EnumThreadWndProc](#).

The [GetWindowThreadProcessId](#) function returns the identifier of the thread that created a particular window.

To set the show state of a window created by another thread, use the [ShowWindowAsync](#) function.

Feedback

Was this page helpful?

 Yes

 No

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

Window Features

Article • 11/19/2022

This overview discusses features of windows such as window types, states, size, and position.

- [Window Types](#)
 - [Overlapped Windows](#)
 - [Pop-up Windows](#)
 - [Child Windows](#)
 - [Positioning](#)
 - [Clipping](#)
 - [Relationship to Parent Window](#)
 - [Messages](#)
 - [Layered Windows](#)
 - [Message-Only Windows](#)
- [Window Relationships](#)
 - [Foreground and Background Windows](#)
 - [Owned Windows](#)
 - [Z-Order](#)
- [Window Show State](#)
 - [Active Window](#)
 - [Disabled Windows](#)
 - [Window Visibility](#)
 - [Minimized, Maximized, and Restored Windows](#)
- [Window Size and Position](#)
 - [Default Size and Position](#)
 - [Tracking Size](#)
 - [System Commands](#)
 - [Size and Position Functions](#)
 - [Size and Position Messages](#)
- [Window Animation](#)
- [Window Layout and Mirroring](#)
 - [Mirroring Dialog Boxes and Message Boxes](#)
 - [Mirroring Device Contexts Not Associated with a Window](#)
- [Window Destruction](#)

Window Types

This section contains the following topics that describe window types.

- Overlapped Windows
- Pop-up Windows
- Child Windows
- Layered Windows
- Message-Only Windows

Overlapped Windows

An *overlapped window* is a top-level window (non-child window) that has a title bar, border, and client area; it is meant to serve as an application's main window. It can also have a window menu, minimize and maximize buttons, and scroll bars. An overlapped window used as a main window typically includes all of these components.

By specifying the [WS_OVERLAPPED](#) or [WS_OVERLAPPEDWINDOW](#) style in the [CreateWindowEx](#) function, an application creates an overlapped window. If you use the [WS_OVERLAPPED](#) style, the window has a title bar and border. If you use the [WS_OVERLAPPEDWINDOW](#) style, the window has a title bar, sizing border, window menu, and minimize and maximize buttons.

Pop-up Windows

A *pop-up window* is a special type of overlapped window used for dialog boxes, message boxes, and other temporary windows that appear outside an application's main window. Title bars are optional for pop-up windows; otherwise, pop-up windows are the same as overlapped windows of the [WS_OVERLAPPED](#) style.

You create a pop-up window by specifying the [WS_POPUP](#) style in [CreateWindowEx](#). To include a title bar, specify the [WS_CAPTION](#) style. Use the [WS_POPUPWINDOW](#) style to create a pop-up window that has a border and a window menu. The [WS_CAPTION](#) style must be combined with the [WS_POPUPWINDOW](#) style to make the window menu visible.

Child Windows

A *child window* has the [WS_CHILD](#) style and is confined to the client area of its parent window. An application typically uses child windows to divide the client area of a parent window into functional areas. You create a child window by specifying the [WS_CHILD](#) style in the [CreateWindowEx](#) function.

A child window must have a parent window. The parent window can be an overlapped window, a pop-up window, or even another child window. You specify the parent

window when you call [CreateWindowEx](#). If you specify the [WS_CHILD](#) style in [CreateWindowEx](#) but do not specify a parent window, the system does not create the window.

A child window has a client area but no other features, unless they are explicitly requested. An application can request a title bar, a window menu, minimize and maximize buttons, a border, and scroll bars for a child window, but a child window cannot have a menu. If the application specifies a menu handle, either when it registers the child's window class or creates the child window, the menu handle is ignored. If no border style is specified, the system creates a borderless window. An application can use borderless child windows to divide a parent window's client area while keeping the divisions invisible to the user.

This section discusses the following aspects of child windows:

- [Positioning](#)
- [Clipping](#)
- [Relationship to Parent Window](#)
- [Messages](#)

Positioning

The system always positions a child window relative to the upper left corner of its parent window's client area. No part of a child window ever appears outside the borders of its parent window. If an application creates a child window that is larger than the parent window or positions a child window so that some or all of the child window extends beyond the borders of the parent, the system clips the child window; that is, the portion outside the parent window's client area is not displayed. Actions that affect the parent window can also affect the child window, as follows.

Parent Window	Child Window
Destroyed	Destroyed before the parent window is destroyed.
Hidden	Hidden before the parent window is hidden. A child window is visible only when the parent window is visible.
Moved	Moved with the parent window's client area. The child window is responsible for painting its client area after the move.
Shown	Shown after the parent window is shown.

Clipping

The system does not automatically clip a child window from the parent window's client area. This means the parent window draws over the child window if it carries out any drawing in the same location as the child window. However, the system does clip the child window from the parent window's client area if the parent window has the [WS_CLIPCHILDREN](#) style. If the child window is clipped, the parent window cannot draw over it.

A child window can overlap other child windows in the same client area. A child window that shares the same parent window as one or more other child windows is called a *sibling window*. Sibling windows can draw in each other's client area, unless one of the child windows has the [WS_CLIPSIBLINGS](#) style. If a child window does have this style, any portion of its sibling window that lies within the child window is clipped.

If a window has either the [WS_CLIPCHILDREN](#) or [WS_CLIPSIBLINGS](#) style, a slight loss in performance occurs. Each window takes up system resources, so an application should not use child windows indiscriminately. For best performance, an application that needs to logically divide its main window should do so in the window procedure of the main window rather than by using child windows.

Relationship to Parent Window

An application can change the parent window of an existing child window by calling the [SetParent](#) function. In this case, the system removes the child window from the client area of the old parent window and moves it to the client area of the new parent window. If [SetParent](#) specifies a **NULL** handle, the desktop window becomes the new parent window. In this case, the child window is drawn on the desktop, outside the borders of any other window. The [GetParent](#) function retrieves a handle to a child window's parent window.

The parent window relinquishes a portion of its client area to a child window, and the child window receives all input from this area. The window class need not be the same for each of the child windows of the parent window. This means that an application can fill a parent window with child windows that look different and carry out different tasks. For example, a dialog box can contain many types of controls, each one a child window that accepts different types of data from the user.

A child window has only one parent window, but a parent can have any number of child windows. Each child window, in turn, can have child windows. In this chain of windows, each child window is called a descendant window of the original parent window. An

application uses the [IsChild](#) function to discover whether a given window is a child window or a descendant window of a given parent window.

The [EnumChildWindows](#) function enumerates the child windows of a parent window. Then, [EnumChildWindows](#) passes the handle to each child window to an application-defined callback function. Descendant windows of the given parent window are also enumerated.

Messages

The system passes a child window's input messages directly to the child window; the messages are not passed through the parent window. The only exception is if the child window has been disabled by the [EnableWindow](#) function. In this case, the system passes any input messages that would have gone to the child window to the parent window instead. This permits the parent window to examine the input messages and enable the child window, if necessary.

A child window can have a unique integer identifier. Child window identifiers are important when working with control windows. An application directs a control's activity by sending it messages. The application uses the control's child window identifier to direct the messages to the control. In addition, a control sends notification messages to its parent window. A notification message includes the control's child window identifier, which the parent uses to identify which control sent the message. An application specifies the child-window identifier for other types of child windows by setting the *hMenu* parameter of the [CreateWindowEx](#) function to a value rather than a menu handle.

Layered Windows

Using a layered window can significantly improve performance and visual effects for a window that has a complex shape, animates its shape, or wishes to use alpha blending effects. The system automatically composes and repaints layered windows and the windows of underlying applications. As a result, layered windows are rendered smoothly, without the flickering typical of complex window regions. In addition, layered windows can be partially translucent, that is, alpha-blended.

To create a layered window, specify the **WS_EX_LAYERED** extended window style when calling the [CreateWindowEx](#) function, or call the [SetWindowLong](#) function to set **WS_EX_LAYERED** after the window has been created. After the [CreateWindowEx](#) call, the layered window will not become visible until the [SetLayeredWindowAttributes](#) or [UpdateLayeredWindow](#) function has been called for this window.

Note

Beginning with Windows 8, **WS_EX_LAYERED** can be used with child windows and top-level windows. Previous Windows versions support **WS_EX_LAYERED** only for top-level windows.

To set the opacity level or the transparency color key for a given layered window, call **SetLayeredWindowAttributes**. After the call, the system may still ask the window to paint when the window is shown or resized. However, because the system stores the image of a layered window, the system will not ask the window to paint if parts of it are revealed as a result of relative window moves on the desktop. Legacy applications do not need to restructure their painting code if they want to add translucency or transparency effects for a window, because the system redirects the painting of windows that called **SetLayeredWindowAttributes** into off-screen memory and recomposes it to achieve the desired effect.

For faster and more efficient animation or if per-pixel alpha is needed, call **UpdateLayeredWindow**. **UpdateLayeredWindow** should be used primarily when the application must directly supply the shape and content of a layered window, without using the redirection mechanism the system provides through **SetLayeredWindowAttributes**. In addition, using **UpdateLayeredWindow** directly uses memory more efficiently, because the system does not need the additional memory required for storing the image of the redirected window. For maximum efficiency in animating windows, call **UpdateLayeredWindow** to change the position and the size of a layered window. Please note that after **SetLayeredWindowAttributes** has been called, subsequent **UpdateLayeredWindow** calls will fail until the layering style bit is cleared and set again.

Hit testing of a layered window is based on the shape and transparency of the window. This means that the areas of the window that are color-keyed or whose alpha value is zero will let the mouse messages through. However, if the layered window has the **WS_EX_TRANSPARENT** extended window style, the shape of the layered window will be ignored and the mouse events will be passed to other windows underneath the layered window.

Message-Only Windows

A *message-only window* enables you to send and receive messages. It is not visible, has no z-order, cannot be enumerated, and does not receive broadcast messages. The

window simply dispatches messages.

To create a message-only window, specify the [HWND_MESSAGE](#) constant or a handle to an existing message-only window in the *hWndParent* parameter of the [CreateWindowEx](#) function. You can also change an existing window to a message-only window by specifying [HWND_MESSAGE](#) in the *hWndNewParent* parameter of the [SetParent](#) function.

To find message-only windows, specify [HWND_MESSAGE](#) in the *hwndParent* parameter of the [FindWindowEx](#) function. In addition, [FindWindowEx](#) searches message-only windows as well as top-level windows if both the *hwndParent* and *hwndChildAfter* parameters are **NULL**.

Window Relationships

There are many ways that a window can relate to the user or another window. A window may be an owned window, foreground window, or background window. A window also has a z-order relative to other windows. For more information, see the following topics:

- [Foreground and Background Windows](#)
- [Owned Windows](#)
- [Z-Order](#)

Foreground and Background Windows

Each process can have multiple threads of execution, and each thread can create windows. The thread that created the window with which the user is currently working is called the foreground thread, and the window is called the *foreground window*. All other threads are background threads, and the windows created by background threads are called *background windows*.

Each thread has a priority level that determines the amount of CPU time the thread receives. Although an application can set the priority level of its threads, normally the foreground thread has a slightly higher priority level than the background threads. Because it has a higher priority, the foreground thread receives more CPU time than the background threads. The foreground thread has a normal base priority of 9; a background thread has a normal base priority of 7.

The user sets the foreground window by clicking a window, or by using the ALT+TAB or ALT+ESC key combination. To retrieve a handle to the foreground window, use the [GetForegroundWindow](#) function. To check if your application window is the foreground

window, compare the handle returned by [GetForegroundWindow](#) to that of your application window.

An application sets the foreground window by using the [SetForegroundWindow](#) function.

The system restricts which processes can set the foreground window. A process can set the foreground window only if:

- All of the following conditions are true:
 - The process calling [SetForegroundWindow](#) 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.

A process that can set the foreground window can enable another process to set the foreground window by calling the [AllowSetForegroundWindow](#) function, or by calling the [BroadcastSystemMessage](#) function with the **BSF_ALLOWSFW** flag. The foreground process can disable calls to [SetForegroundWindow](#) by calling the [LockSetForegroundWindow](#) function.

Owned Windows

An overlapped or pop-up window can be owned by another overlapped or pop-up window. Being owned places several constraints on a window.

- An owned window is always above its owner in the z-order.
- The system automatically destroys an owned window when its owner is destroyed.
- An owned window is hidden when its owner is minimized.

Only an overlapped or pop-up window can be an owner window; a child window cannot be an owner window. An application creates an owned window by specifying the owner's window handle as the *hwndParent* parameter of [CreateWindowEx](#) when it creates a window with the **WS_OVERLAPPED** or **WS_POPUP** style. The *hwndParent* parameter must identify an overlapped or pop-up window. If *hwndParent* identifies a child window, the system assigns ownership to the top-level parent window of the child window. After creating an owned window, an application cannot transfer ownership of the window to another window.

Dialog boxes and message boxes are owned windows by default. An application specifies the owner window when calling a function that creates a dialog box or message box.

An application can use the [GetWindow](#) function with the **GW_OWNER** flag to retrieve a handle to a window's owner.

Z-Order

The *z-order* of a window indicates the window's position in a stack of overlapping windows. This window stack is oriented along an imaginary axis, the z-axis, extending outward from the screen. The window at the top of the z-order overlaps all other windows. The window at the bottom of the z-order is overlapped by all other windows.

The system maintains the z-order in a single list. It adds windows to the z-order based on whether they are topmost windows, top-level windows, or child windows. A *topmost window* overlaps all other non-topmost windows, regardless of whether it is the active or foreground window. A topmost window has the **WS_EX_TOPMOST** style. All topmost windows appear in the z-order before any non-topmost windows. A child window is grouped with its parent in z-order.

When an application creates a window, the system puts it at the top of the z-order for windows of the same type. You can use the [BringWindowToTop](#) function to bring a window to the top of the z-order for windows of the same type. You can rearrange the z-order by using the [SetWindowPos](#) and [DeferWindowPos](#) functions.

The user changes the z-order by activating a different window. The system positions the active window at the top of the z-order for windows of the same type. When a window comes to the top of z-order, so do its child windows. You can use the [GetTopWindow](#) function to search all child windows of a parent window and return a handle to the child window that is highest in z-order. The [GetNextWindow](#) function retrieves a handle to the next or previous window in z-order.

Window Show State

At any one given time, a window may be active or inactive; hidden or visible; and minimized, maximized, or restored. These qualities are referred to collectively as the *window show state*. The following topics discuss the window show state:

- [Active Window](#)
- [Disabled Windows](#)
- [Window Visibility](#)
- [Minimized, Maximized, and Restored Windows](#)

Active Window

An *active window* is the top-level window of the application with which the user is currently working. To allow the user to easily identify the active window, the system places it at the top of the z-order and changes the color of its title bar and border to the system-defined active window colors. Only a top-level window can be an active window. When the user is working with a child window, the system activates the top-level parent window associated with the child window.

Only one top-level window in the system is active at a time. The user activates a top-level window by clicking it (or one of its child windows), or by using the ALT+ESC or ALT+TAB key combination. An application activates a top-level window by calling the [SetActiveWindow](#) function. Other functions can cause the system to activate a different top-level window, including [SetWindowPos](#), [DeferWindowPos](#), [SetWindowPlacement](#), and [DestroyWindow](#). Although an application can activate a different top-level window at any time, to avoid confusing the user, it should do so only in response to a user action. An application uses the [GetActiveWindow](#) function to retrieve a handle to the active window.

When the activation changes from a top-level window of one application to the top-level window of another, the system sends a [WM_ACTIVATEAPP](#) message to both applications, notifying them of the change. When the activation changes to a different top-level window in the same application, the system sends both windows a [WM_ACTIVATE](#) message.

Disabled Windows

A window can be disabled. A *disabled window* receives no keyboard or mouse input from the user, but it can receive messages from other windows, from other applications, and from the system. An application typically disables a window to prevent the user

from using the window. For example, an application may disable a push button in a dialog box to prevent the user from choosing it. An application can enable a disabled window at any time; enabling a window restores normal input.

By default, a window is enabled when created. An application can specify the [WS_DISABLED](#) style, however, to disable a new window. An application enables or disables an existing window by using the [EnableWindow](#) function. The system sends a [WM_ENABLE](#) message to a window when its enabled state is about to change. An application can determine whether a window is enabled by using the [IsWindowEnabled](#) function.

When a child window is disabled, the system passes the child's mouse input messages to the parent window. The parent uses the messages to determine whether to enable the child window. For more information, see [Mouse Input](#).

Only one window at a time can receive keyboard input; that window is said to have the keyboard focus. If an application uses the [EnableWindow](#) function to disable a keyboard-focus window, the window loses the keyboard focus in addition to being disabled. [EnableWindow](#) then sets the keyboard focus to **NULL**, meaning no window has the focus. If a child window, or other descendant window, has the keyboard focus, the descendant window loses the focus when the parent window is disabled. For more information, see [Keyboard Input](#).

Window Visibility

A window can be either visible or hidden. The system displays a *visible window* on the screen. It hides a *hidden window* by not drawing it. If a window is visible, the user can supply input to the window and view the window's output. If a window is hidden, it is effectively disabled. A hidden window can process messages from the system or from other windows, but it cannot process input from the user or display output. An application sets a window's visibility state when creating the window. Later, the application can change the visibility state.

A window is visible when the [WS_VISIBLE](#) style is set for the window. By default, the [CreateWindowEx](#) function creates a hidden window unless the application specifies the [WS_VISIBLE](#) style. Typically, an application sets the [WS_VISIBLE](#) style after it has created a window to keep details of the creation process hidden from the user. For example, an application may keep a new window hidden while it customizes the window's appearance. If the [WS_VISIBLE](#) style is specified in [CreateWindowEx](#), the system sends the [WM_SHOWWINDOW](#) message to the window after creating the window, but before displaying it.

An application can determine whether a window is visible by using the [IsWindowVisible](#) function. An application can show (make visible) or hide a window by using the [ShowWindow](#), [SetWindowPos](#), [DeferWindowPos](#), or [SetWindowPlacement](#) or [SetWindowLong](#) function. These functions show or hide a window by setting or removing the [WS_VISIBLE](#) style for the window. They also send the [WM_SHOWWINDOW](#) message to the window before showing or hiding it.

When an owner window is minimized, the system automatically hides the associated owned windows. Similarly, when an owner window is restored, the system automatically shows the associated owned windows. In both cases, the system sends the [WM_SHOWWINDOW](#) message to the owned windows before hiding or showing them. Occasionally, an application may need to hide the owned windows without having to minimize or hide the owner. In this case, the application uses the [ShowOwnedPopups](#) function. This function sets or removes the [WS_VISIBLE](#) style for all owned windows and sends the [WM_SHOWWINDOW](#) message to the owned windows before hiding or showing them. Hiding an owner window has no effect on the visibility state of the owned windows.

When a parent window is visible, its associated child windows are also visible. Similarly, when the parent window is hidden, its child windows are also hidden. Minimizing the parent window has no effect on the visibility state of the child windows; that is, the child windows are minimized along with the parent, but the [WS_VISIBLE](#) style is not changed.

Even if a window has the [WS_VISIBLE](#) style, the user may not be able to see the window on the screen; other windows may completely overlap it or it may have been moved beyond the edge of the screen. Also, a visible child window is subject to the clipping rules established by its parent-child relationship. If the window's parent window is not visible, it will also not be visible. If the parent window moves beyond the edge of the screen, the child window also moves because a child window is drawn relative to the parent's upper left corner. For example, a user may move the parent window containing the child window far enough off the edge of the screen that the user may not be able to see the child window, even though the child window and its parent window both have the [WS_VISIBLE](#) style.

Minimized, Maximized, and Restored Windows

A *maximized window* is a window that has the [WS_MAXIMIZE](#) style. By default, the system enlarges a maximized window so that it fills the screen or, in the case of a child window, the parent window's client area. Although a window's size can be set to the same size of a maximized window, a maximized window is slightly different. The system automatically moves the window's title bar to the top of the screen or to the top of the parent window's client area. Also, the system disables the window's sizing border and

the window-positioning capability of the title bar (so that the user cannot move the window by dragging the title bar).

A *minimized window* is a window that has the [WS_MINIMIZE](#) style. By default, the system reduces a minimized window to the size of its taskbar button and moves the minimized window to the taskbar. A *restored window* is a window that has been returned to its previous size and position, that is, the size it was before it was minimized or maximized.

If an application specifies the [WS_MAXIMIZE](#) or [WS_MINIMIZE](#) style in the [CreateWindowEx](#) function, the window is initially maximized or minimized. After creating a window, an application can use the [CloseWindow](#) function to minimize the window. The [ArrangeIconicWindows](#) function arranges the icons on the desktop, or it arranges a parent window's minimized child windows in the parent window. The [OpenIcon](#) function restores a minimized window to its previous size and position.

The [ShowWindow](#) function can minimize, maximize, or restore a window. It can also set the window's visibility and activation states. The [SetWindowPlacement](#) function includes the same functionality as [ShowWindow](#), but it can override the window's default minimized, maximized, and restored positions.

The [IsZoomed](#) and [IsIconic](#) functions determine whether a given window is maximized or minimized, respectively. The [GetWindowPlacement](#) function retrieves the minimized, maximized, and restored positions for the window, and also determines the window's show state.

When the system receives a command to maximize or restore a minimized window, it sends the window a [WM_QUERYOPEN](#) message. If the window procedure returns [FALSE](#), the system ignores the maximize or restore command.

The system automatically sets the size and position of a maximized window to the system-defined defaults for a maximized window. To override these defaults, an application can either call the [SetWindowPlacement](#) function or process the [WM_GETMINMAXINFO](#) message that is received by a window when the system is about to maximize the window. [WM_GETMINMAXINFO](#) includes a pointer to a [MINMAXINFO](#) structure containing values the system uses to set the maximized size and position. Replacing these values overrides the defaults.

Window Size and Position

A window's size and position are expressed as a bounding rectangle, given in coordinates relative to the screen or the parent window. The coordinates of a top-level

window are relative to the upper left corner of the screen; the coordinates of a child window are relative to the upper left corner of the parent window. An application specifies a window's initial size and position when it creates the window, but it can change the window's size and position at any time. For more information, see [Filled Shapes](#).

This section contains the following topics:

- [Default Size and Position](#)
- [Tracking Size](#)
- [System Commands](#)
- [Size and Position Functions](#)
- [Size and Position Messages](#)

Default Size and Position

An application can allow the system to calculate the initial size or position of a top-level window by specifying `CW_USEDEFAULT` in [CreateWindowEx](#). If the application sets the window's coordinates to `CW_USEDEFAULT` and has created no other top-level windows, the system sets the new window's position relative to the upper left corner of the screen; otherwise, it sets the position relative to the position of the top-level window that the application created most recently. If the width and height parameters are set to `CW_USEDEFAULT`, the system calculates the size of the new window. If the application has created other top-level windows, the system bases the size of the new window on the size of the application's most recently created top-level window. Specifying `CW_USEDEFAULT` when creating a child or pop-up window causes the system to set the window's size to the default minimum window size.

Tracking Size

The system maintains a minimum and maximum tracking size for a window of the [WS_THICKFRAME](#) style; a window with this style has a sizing border. The *minimum tracking size* is the smallest window size you can produce by dragging the window's sizing border. Similarly, the *maximum tracking size* is the largest window size you can produce by dragging the sizing border.

A window's minimum and maximum tracking sizes are set to system-defined default values when the system creates the window. An application can discover the defaults and override them by processing the [WM_GETMINMAXINFO](#) message. For more information, see [Size and Position Messages](#).

System Commands

An application that has a window menu can change the size and position of that window by sending system commands. System commands are generated when the user chooses commands from the window menu. An application can emulate the user action by sending a [WM_SYSCOMMAND](#) message to the window. The following system commands affect the size and position of a window.

Command	Description
SC_CLOSE	Closes the window. This command sends a WM_CLOSE message to the window. The window carries out any steps needed to clean up and destroy itself.
SC_MAXIMIZE	Maximizes the window.
SC_MINIMIZE	Minimizes the window.
SC_MOVE	Moves the window.
SC_RESTORE	Restores a minimized or maximized window to its previous size and position.
SC_SIZE	Starts a size command. To change the size of the window, use the mouse or keyboard.

Size and Position Functions

After creating a window, an application can set the window's size or position by calling one of several different functions, including [SetWindowPlacement](#), [MoveWindow](#), [SetWindowPos](#), and [DeferWindowPos](#). [SetWindowPlacement](#) sets a window's minimized position, maximized position, restored size and position, and show state. The [MoveWindow](#) and [SetWindowPos](#) functions are similar; both set the size or position of a single application window. The [SetWindowPos](#) function includes a set of flags that affect the window's show state; [MoveWindow](#) does not include these flags. Use the [BeginDeferWindowPos](#), [DeferWindowPos](#), and [EndDeferWindowPos](#) functions to simultaneously set the position of a number of windows, including the size, position, position in the z-order, and show state.

An application can retrieve the coordinates of a window's bounding rectangle by using the [GetWindowRect](#) function. [GetWindowRect](#) fills a [RECT](#) structure with the coordinates of the window's upper left and lower right corners. The coordinates are relative to the upper left corner of the screen, even for a child window. The [ScreenToClient](#) or [MapWindowPoints](#) function maps the screen coordinates of a child window's bounding rectangle to coordinates relative to the parent window's client area.

The [GetClientRect](#) function retrieves the coordinates of a window's client area. [GetClientRect](#) fills a [RECT](#) structure with the coordinates of the upper left and lower right corners of the client area, but the coordinates are relative to the client area itself. This means the coordinates of a client area's upper left corner are always (0,0), and the coordinates of the lower right corner are the width and height of the client area.

The [CascadeWindows](#) function cascades the windows on the desktop or cascades the child windows of the specified parent window. The [TileWindows](#) function tiles the windows on the desktop or tiles the child windows of the specified parent window.

Size and Position Messages

The system sends the [WM_GETMINMAXINFO](#) message to a window whose size or position is about to change. For example, the message is sent when the user clicks [Move](#) or [Size](#) from the window menu or clicks the sizing border or title bar; the message is also sent when an application calls [SetWindowPos](#) to move or size the window.

[WM_GETMINMAXINFO](#) includes a pointer to a [MINMAXINFO](#) structure containing the default maximized size and position for the window, as well as the default minimum and maximum tracking sizes. An application can override the defaults by processing [WM_GETMINMAXINFO](#) and setting the appropriate members of [MINMAXINFO](#). A window must have the [WS_THICKFRAME](#) or [WS_CAPTION](#) style to receive [WM_GETMINMAXINFO](#). A window with the [WS_THICKFRAME](#) style receives this message during the window-creation process, as well as when it is being moved or sized.

The system sends the [WM_WINDOWPOSCHANGING](#) message to a window whose size, position, position in the z-order, or show state is about to change. This message includes a pointer to a [WINDOWPOS](#) structure that specifies the window's new size, position, position in the z-order, and show state. By setting the members of [WINDOWPOS](#), an application can affect the window's new size, position, and appearance.

After changing a window's size, position, position in the z-order, or show state, the system sends the [WM_WINDOWPOSCHANGED](#) message to the window. This message includes a pointer to [WINDOWPOS](#) that informs the window of its new size, position, position in the z-order, and show state. Setting the members of the [WINDOWPOS](#) structure that is passed with [WM_WINDOWPOSCHANGED](#) has no effect on the window. A window that must process [WM_SIZE](#) and [WM_MOVE](#) messages must pass [WM_WINDOWPOSCHANGED](#) to the [DefWindowProc](#) function; otherwise, the system does not send [WM_SIZE](#) and [WM_MOVE](#) messages to the window.

The system sends the [WM_NCCALCSIZE](#) message to a window when the window is created or sized. The system uses the message to calculate the size of a window's client area and the position of the client area relative to the upper left corner of the window. A window typically passes this message to the default window procedure; however, this message can be useful in applications that customize a window's nonclient area or preserve portions of the client area when the window is sized. For more information, see [Painting and Drawing](#).

Window Animation

You can produce special effects when showing or hiding windows by using the [AnimateWindow](#) function. When the window is animated in this manner, the system will either roll, slide, or fade the window, depending on the flags you specify in a call to [AnimateWindow](#).

By default, the system uses *roll animation*. With this effect, the window appears to roll open (showing the window) or roll closed (hiding the window). You can use the *dwFlags* parameter to specify whether the window rolls horizontally, vertically, or diagonally.

When you specify the **AW_SLIDE** flag, the system uses *slide animation*. With this effect, the window appears to slide into view (showing the window) or slide out of view (hiding the window). You can use the *dwFlags* parameter to specify whether the window slides horizontally, vertically, or diagonally.

When you specify the **AW_BLEND** flag, the system uses an *alpha-blended fade*.

You can also use the **AW_CENTER** flag to make a window appear to collapse inward or expand outward.

Window Layout and Mirroring

The window layout defines how text and Windows Graphics Device Interface (GDI) objects are laid out in a window or device context (DC). Some languages, such as English, French, and German, require a left-to-right (LTR) layout. Other languages, such as Arabic and Hebrew, require right-to-left (RTL) layout. The window layout applies to text but also affects the other GDI elements of the window, including bitmaps, icons, the location of the origin, buttons, cascading tree controls, and whether the horizontal coordinate increases as you go left or right. For example, after an application has set RTL layout, the origin is positioned at the right edge of the window or device, and the number representing the horizontal coordinate increases as you move left. However, not all objects are affected by the layout of a window. For example, the layout for dialog

boxes, message boxes, and device contexts that are not associated with a window, such as metafile and printer DCs, must be handled separately. Specifics for these are mentioned later in this topic.

The window functions allow you to specify or change the window layout in Arabic and Hebrew versions of Windows. Note that changing to a RTL layout (also known as mirroring) is not supported for windows that have the style [CS_OWNDC](#) or for a DC with the [GM_ADVANCED](#) graphic mode.

By default, the window layout is left-to-right (LTR). To set the RTL window layout, call [CreateWindowEx](#) with the style [WS_EX_LAYOUTRTL](#). Also by default, a child window (that is, one created with the [WS_CHILD](#) style and with a valid parent *hWnd* parameter in the call to [CreateWindow](#) or [CreateWindowEx](#)) has the same layout as its parent. To disable inheritance of mirroring to all child windows, specify [WS_EX_NOINHERITLAYOUT](#) in the call to [CreateWindowEx](#). Note, mirroring is not inherited by owned windows (those created without the [WS_CHILD](#) style) or those created with the parent *hWnd* parameter in [CreateWindowEx](#) set to **NULL**. To disable inheritance of mirroring for an individual window, process the [WM_NCCREATE](#) message with [GetWindowLong](#) and [SetWindowLong](#) to turn off the [WS_EX_LAYOUTRTL](#) flag. This processing is in addition to whatever other processing is needed. The following code fragment shows how this is done.

```
SetWindowLong (hWnd,
               GWL_EXSTYLE,
               GetWindowLong(hWnd, GWL_EXSTYLE) & ~WS_EX_LAYOUTRTL)
```

You can set the default layout to RTL by calling [SetProcessDefaultLayout\(LAYOUTRTL\)](#). All windows created after the call will be mirrored, but existing windows are not affected. To turn off default mirroring, call [SetProcessDefaultLayout\(0\)](#).

Note, [SetProcessDefaultLayout](#) mirrors the DCs only of mirrored windows. To mirror any DC, call [SetLayout\(hdc, LAYOUTRTL\)](#). For more information, see the discussion on mirroring device contexts not associated with windows, which comes later in this topic.

Bitmaps and icons in a mirrored window are also mirrored by default. However, not all of these should be mirrored. For example, those with text, a business logo, or an analog clock should not be mirrored. To disable mirroring of bitmaps, call [SetLayout](#) with the [LAYOUT_BITMAPORIENTATIONPRESERVED](#) bit set in *dwLayout*. To disable mirroring in a DC, call [SetLayout\(hdc, 0\)](#).

To query the current default layout, call [GetProcessDefaultLayout](#). Upon a successful return, *pdwDefaultLayout* contains LAYOUT_RTL or 0. To query the layout settings of the device context, call [GetLayout](#). Upon a successful return, [GetLayout](#) returns a **DWORD** that indicates the layout settings by the settings of the LAYOUT_RTL and the LAYOUT_BITMAPORIENTATIONPRESERVED bits.

After a window has been created, you change the layout using the [SetWindowLong](#) function. For example, this is necessary when the user changes the user interface language of an existing window from Arabic or Hebrew to German. However, when changing the layout of an existing window, you must invalidate and update the window to ensure that the contents of the window are all drawn on the same layout. The following code example is from sample code that changes the window layout as needed:

```
// Using ANSI versions of GetWindowLong and SetWindowLong because Unicode
// is not needed for these calls

lExStyles = GetWindowLongA(hWnd, GWL_EXSTYLE);

// Check whether new layout is opposite the current layout
if (!!(pLState -> IsRTLLayout) != !!(lExStyles & WS_EX_LAYOUTRTL))
{
    // the following lines will update the window layout

    lExStyles ^= WS_EX_LAYOUTRTL;           // toggle layout
    SetWindowLongA(hWnd, GWL_EXSTYLE, lExStyles);
    InvalidateRect(hWnd, NULL, TRUE);      // to update layout in the client
area
}
```

In mirroring, you should think in terms of "near" and "far" instead of "left" and "right". Failure to do so can cause problems. One common coding practice that causes problems in a mirrored window occurs when mapping between screen coordinates and client coordinates. For example, applications often use code similar to the following to position a control in a window:

```
// DO NOT USE THIS IF APPLICATION MIRRORS THE WINDOW

// get coordinates of the window in screen coordinates
GetWindowRect(hControl, (LPRECT) &rControlRect);

// map screen coordinates to client coordinates in dialog
```

```
ScreenToClient(hDialog, (LPPOINT) &rControlRect.left);
ScreenToClient(hDialog, (LPPOINT) &rControlRect.right);
```

This causes problems in mirroring because the left edge of the rectangle becomes the right edge in a mirrored window, and vice versa. To avoid this problem, replace the [ScreenToClient](#) calls with a call to [MapWindowPoints](#) as follows:

```
// USE THIS FOR MIRRORING

GetWindowRect(hControl, (LPRECT) &rControlRect);
MapWindowPoints(NULL, hDialog, (LPPOINT) &rControlRect, 2)
```

This code works because, on platforms that support mirroring, [MapWindowPoints](#) is modified to swap the left and right point coordinates when the client window is mirrored. For more information, see the Remarks section of [MapWindowPoints](#).

Another common practice that can cause problems in mirrored windows is positioning objects in a client window using offsets in screen coordinates instead of client coordinates. For example, the following code uses the difference in screen coordinates as the x position in client coordinates to position a control in a dialog box.

```
// OK if LTR layout and mapping mode of client is MM_TEXT,
// but WRONG for a mirrored dialog

RECT rdDialog;
RECT rcControl;

HWND hControl = GetDlgItem(hDlg, IDD_CONTROL);
GetWindowRect(hDlg, &rcDialog);           // gets rect in screen
coordinates
GetWindowRect(hControl, &rcControl);
MoveWindow(hControl,
           rcControl.left - rcDialog.left, // uses x position in client
coords
           rcControl.top - rcDialog.top,
           nWidth,
           nHeight,
           FALSE);
```

This code is fine when the dialog window has left-to-right (LTR) layout and the mapping mode of the client is MM_TEXT, because the new x position in client coordinates corresponds to the difference in left edges of the control and the dialog in screen

coordinates. However, in a mirrored dialog, left and right are reversed, so instead you should use [MapWindowPoints](#) as follows:

```
RECT rcDialog;
RECT rcControl;

HWND hControl = GetDlgItem(hDlg, IDD_CONTROL);
GetWindowRect(hControl, &rcControl);

// MapWindowPoints works correctly in both mirrored and non-mirrored
// windows.
MapWindowPoints(NULL, hDlg, (LPOINT) &rcControl, 2);

// Now rcControl is in client coordinates.
MoveWindow(hControl, rcControl.left, rcControl.top, nWidth, nHeight, FALSE)
```

Mirroring Dialog Boxes and Message Boxes

Dialog boxes and message boxes do not inherit layout, so you must set the layout explicitly. To mirror a message box, call [MessageBox](#) or [MessageBoxEx](#) with the **MB_RTLREADING** option. To layout a dialog box right-to-left, use the extended style **WS_EX_LAYOUTRTL** in the dialog template structure [DLGTEMPLATEEX](#). Property sheets are a special case of dialog boxes. Each tab is treated as a separate dialog box, so you need to include the **WS_EX_LAYOUTRTL** style in every tab that you want mirrored.

Mirroring Device Contexts Not Associated with a Window

DCs that are not associated with a window, such as metafile or printer DCs, do not inherit layout, so you must set the layout explicitly. To change the device context layout, use the [SetLayout](#) function.

The [SetLayout](#) function is rarely used with windows. Typically, windows receive an associated DC only in processing a **WM_PAINT** message. Occasionally, a program creates a DC for a window by calling [GetDC](#). Either way, the initial layout for the DC is set by [BeginPaint](#) or [GetDC](#) according to the window's **WS_EX_LAYOUTRTL** flag.

The values returned by [GetWindowOrgEx](#), [GetWindowExtEx](#), [GetViewportOrgEx](#) and [GetViewportExtEx](#) are not affected by calling [SetLayout](#).

When the layout is RTL, [GetMapMode](#) will return **MM_ANISOTROPIC** instead of **MM_TEXT**. Calling [SetMapMode](#) with **MM_TEXT** will function correctly; only the return value from [GetMapMode](#) is affected. Similarly, calling [SetLayout\(hdc, LAYOUTRTL\)](#)

when the mapping mode is MM_TEXT causes the reported mapping mode to change to MM_ANISOTROPIC.

Window Destruction

In general, an application must destroy all the windows it creates. It does this by using the [DestroyWindow](#) function. When a window is destroyed, the system hides the window, if it is visible, and then removes any internal data associated with the window. This invalidates the window handle, which can no longer be used by the application.

An application destroys many of the windows it creates soon after creating them. For example, an application usually destroys a dialog box window as soon as the application has sufficient input from the user to continue its task. An application eventually destroys the main window of the application (before terminating).

Before destroying a window, an application should save or remove any data associated with the window, and it should release any system resources allocated for the window. If the application does not release the resources, the system will free any resources not freed by the application.

Destroying a window does not affect the window class from which the window is created. New windows can still be created using that class, and any existing windows of that class continue to operate. Destroying a window also destroys the window's descendant windows. The [DestroyWindow](#) function sends a [WM_DESTROY](#) message first to the window, then to its child windows and descendant windows. In this way, all descendant windows of the window being destroyed are also destroyed.

A window with a window menu receives a [WM_CLOSE](#) message when the user clicks Close. By processing this message, an application can prompt the user for confirmation before destroying the window. If the user confirms that the window should be destroyed, the application can call the [DestroyWindow](#) function to destroy the window.

If the window being destroyed is the active window, both the active and focus states are transferred to another window. The window that becomes the active window is the next window, as determined by the ALT+ESC key combination. The new active window then determines which window receives the keyboard focus.

Feedback

Was this page helpful?

 Yes

 No

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

Using Windows

Article • 02/06/2023

The examples in this section describe how to perform the following tasks:

- Creating a Main Window
 - Creating, Enumerating, and Sizing Child Windows
 - Destroying a Window
 - Using Layered Windows

Creating a Main Window

The first window an application creates is typically the main window. You create the main window by using the [CreateWindowEx](#) function, specifying the window class, window name, window styles, size, position, menu handle, instance handle, and creation data. A main window belongs to an application-defined window class, so you must register the window class and provide a window procedure for the class before creating the main window.

Most applications typically use the [WS_OVERLAPPEDWINDOW](#) style to create the main window. This style gives the window a title bar, a window menu, a sizing border, and minimize and maximize buttons. The [CreateWindowEx](#) function returns a handle that uniquely identifies the window.

The following example creates a main window belonging to an application-defined window class. The window name, **Main Window**, will appear in the window's title bar. By combining the **WS_VSCROLL** and **WS_HSCROLL** styles with the **WS_OVERLAPPEDWINDOW** style, the application creates a main window with horizontal and vertical scroll bars in addition to the components provided by the **WS_OVERLAPPEDWINDOW** style. The four occurrences of the **CW_USEDEFAULT** constant set the initial size and position of the window to the system-defined default values. By specifying **NULL** instead of a menu handle, the window will have the menu defined for the window class.

```

    "MainWClass",           // class name
    "Main Window",          // window name
    WS_OVERLAPPEDWINDOW |   // overlapped window
        WS_HSCROLL |       // horizontal scroll bar
        WS_VSCROLL,         // vertical scroll bar
    CW_USEDEFAULT,          // default horizontal position
    CW_USEDEFAULT,          // default vertical position
    CW_USEDEFAULT,          // default width
    CW_USEDEFAULT,          // default height
    (HWND) NULL,            // no parent or owner window
    (HMENU) NULL,            // class menu used
    hinst,                  // instance handle
    NULL);                 // no window creation data

if (!hwndMain)
    return FALSE;

// Show the window using the flag specified by the program
// that started the application, and send the application
// a WM_PAINT message.

ShowWindow(hwndMain, SW_SHOWDEFAULT);
UpdateWindow(hwndMain);

```

Notice that the preceding example calls the [ShowWindow](#) function after creating the main window. This is done because the system does not automatically display the main window after creating it. By passing the [SW_SHOWDEFAULT](#) flag to [ShowWindow](#), the application allows the program that started the application to set the initial show state of the main window. The [UpdateWindow](#) function sends the window its first [WM_PAINT](#) message.

Creating, Enumerating, and Sizing Child Windows

You can divide a window's client area into different functional areas by using child windows. Creating a child window is like creating a main window—you use the [CreateWindowEx](#) function. To create a window of an application-defined window class, you must register the window class and provide a window procedure before creating the child window. You must give the child window the [WS_CHILD](#) style and specify a parent window for the child window when you create it.

The following example divides the client area of an application's main window into three functional areas by creating three child windows of equal size. Each child window is the same height as the main window's client area, but each is one-third its width. The main window creates the child windows in response to the [WM_CREATE](#) message, which the main window receives during its own window-creation process. Because each child

window has the [WS_BORDER](#) style, each has a thin line border. Also, because the [WS_VISIBLE](#) style is not specified, each child window is initially hidden. Notice also that each child window is assigned a child-window identifier.

The main window sizes and positions the child windows in response to the [WM_SIZE](#) message, which the main window receives when its size changes. In response to [WM_SIZE](#), the main window retrieves the dimensions of its client area by using the [GetClientRect](#) function and then passes the dimensions to the [EnumChildWindows](#) function. [EnumChildWindows](#) passes the handle to each child window, in turn, to the application-defined [EnumChildProc](#) callback function. This function sizes and positions each child window by calling the [MoveWindow](#) function; the size and position are based on the dimensions of the main window's client area and the identifier of the child window. Afterward, [EnumChildProc](#) calls the [ShowWindow](#) function to make the window visible.

```
#define ID_FIRSTCHILD 100
#define ID_SECONDCHILD 101
#define ID_THIRDCHILD 102

LONG APIENTRY MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    RECT rcClient;
    int i;

    switch(uMsg)
    {
        case WM_CREATE: // creating main window

            // Create three invisible child windows.

            for (i = 0; i < 3; i++)
            {
                CreateWindowEx(0,
                               "ChildWClass",
                               (LPCTSTR) NULL,
                               WS_CHILD | WS_BORDER,
                               0,0,0,0,
                               hwnd,
                               (HMENU) (int) (ID_FIRSTCHILD + i),
                               hinst,
                               NULL);
            }

        return 0;

        case WM_SIZE:    // main window changed size
```

```

        // Get the dimensions of the main window's client
        // area, and enumerate the child windows. Pass the
        // dimensions to the child windows during enumeration.

        GetClientRect(hwnd, &rcClient);
        EnumChildWindows(hwnd, EnumChildProc, (LPARAM) &rcClient);
        return 0;

        // Process other messages.
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

BOOL CALLBACK EnumChildProc(HWND hwndChild, LPARAM lParam)
{
    LPRECT rcParent;
    int i, idChild;

    // Retrieve the child-window identifier. Use it to set the
    // position of the child window.

    idChild = GetWindowLong(hwndChild, GWL_ID);

    if (idChild == ID_FIRSTCHILD)
        i = 0;
    else if (idChild == ID_SECONDCHILD)
        i = 1;
    else
        i = 2;

    // Size and position the child window.

    rcParent = (LPRECT) lParam;
    MoveWindow(hwndChild,
               (rcParent->right / 3) * i,
               0,
               rcParent->right / 3,
               rcParent->bottom,
               TRUE);

    // Make sure the child window is visible.

    ShowWindow(hwndChild, SW_SHOW);

    return TRUE;
}

```

Destroying a Window

You can use the [DestroyWindow](#) function to destroy a window. Typically, an application sends the [WM_CLOSE](#) message before destroying a window, giving the window the

opportunity to prompt the user for confirmation before the window is destroyed. A window that includes a window menu automatically receives the **WM_CLOSE** message when the user clicks **Close** from the window menu. If the user confirms that the window should be destroyed, the application calls **DestroyWindow**. The system sends the **WM_DESTROY** message to the window after removing it from the screen. In response to **WM_DESTROY**, the window saves its data and frees any resources it allocated. A main window concludes its processing of **WM_DESTROY** by calling the **PostQuitMessage** function to quit the application.

The following example shows how to prompt for user confirmation before destroying a window. In response to **WM_CLOSE**, the example displays a dialog box that contains **Yes**, **No**, and **Cancel** buttons. If the user clicks **Yes**, **DestroyWindow** is called; otherwise, the window is not destroyed. Because the window being destroyed is a main window, the example calls **PostQuitMessage** in response to **WM_DESTROY**.

```
case WM_CLOSE:  
  
    // Create the message box. If the user clicks  
    // the Yes button, destroy the main window.  
  
    if (MessageBox(hwnd, szConfirm, szAppName, MB_YESNOCANCEL) == IDYES)  
        DestroyWindow(hwndMain);  
    else  
        return 0;  
  
case WM_DESTROY:  
  
    // Post the WM_QUIT message to  
    // quit the application terminate.  
  
    PostQuitMessage(0);  
    return 0;
```

Using Layered Windows

To have a dialog box come up as a translucent window, first create the dialog as usual. Then, on **WM_INITDIALOG**, set the layered bit of the window's extended style and call **SetLayeredWindowAttributes** with the desired alpha value. The code might look like this:

```
// Set WS_EX_LAYERED on this window  
SetWindowLong(hwnd,
```

```

        GWL_EXSTYLE,
        GetWindowLong(hwnd, GWL_EXSTYLE) | WS_EX_LAYERED);

// Make this window 70% alpha
SetLayeredWindowAttributes(hwnd, 0, (255 * 70) / 100, LWA_ALPHA);

```

Note that the third parameter of [SetLayeredWindowAttributes](#) is a value that ranges from 0 to 255, with 0 making the window completely transparent and 255 making it completely opaque. This parameter mimics the more versatile [BLENDFUNCTION](#) of the [AlphaBlend](#) function.

To make this window completely opaque again, remove the [WS_EX_LAYERED](#) bit by calling [SetWindowLong](#) and then ask the window to repaint. Removing the bit is desired to let the system know that it can free up some memory associated with layering and redirection. The code might look like this:

```

// Remove WS_EX_LAYERED from this window styles
SetWindowLong(hwnd,
              GWL_EXSTYLE,
              GetWindowLong(hwnd, GWL_EXSTYLE) & ~WS_EX_LAYERED);

// Ask the window and its children to repaint
RedrawWindow(hwnd,
              NULL,
              NULL,
              RDW_ERASE | RDW_INVALIDATE | RDW_FRAME | RDW_ALLCHILDREN);

```

In order to use layered child windows, the application has to declare itself Windows 8-aware in the manifest.

For windows 10/11, one can include this compatibility snippet in its `app.manifest` to make it Windows 10-aware :

```

...
<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
        <!-- Windows 10 GUID -->
        <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}" />
    </application>
</compatibility>
...

```

More about modifying app manifest can be read here : [Application manifests](#)

Feedback

Was this page helpful?

 Yes

 No

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

Window Reference

Article • 04/27/2021

- [Window Constants](#)
 - [Window Functions](#)
 - [Window Macros](#)
 - [Window Messages](#)
 - [Window Notifications](#)
 - [Window Structures](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

Window Constants

Article • 01/07/2021

- [Extended Window Styles](#)
- [Window Styles](#)

Feedback

Was this page helpful?

 Yes

 No

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

Extended Window Styles

Article • 04/07/2022

The following are the extended window styles.

Example

C++

```
virtual     BOOL     Create(HWND hWndParent, WCHAR* pwszClassName,
                           WCHAR* pwszWindowName, UINT uID, HICON hIcon,
                           DWORD dwStyle = WS_OVERLAPPEDWINDOW,
                           DWORD dwExStyle = WS_EX_APPWINDOW,
                           int x = CW_USEDEFAULT, int y = CW_USEDEFAULT,
                           int cx = CW_USEDEFAULT, int cy = CW_USEDEFAULT);
```

This code was taken from a sample in the [Windows classic samples](#) GitHub repo.

Constant/value	Description
WS_EX_ACCEPTFILES 0x00000010L	The window accepts drag-drop files.
WS_EX_APPWINDOW 0x00040000L	Forces a top-level window onto the taskbar when the window is visible.
WS_EX_CLIENTEDGE 0x00000200L	The window has a border with a sunken edge.
WS_EX_COMPOSITED 0x02000000L	Paints all descendants of a window in 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. This cannot be used if the window has a class style of either CS_OWNDC or CS_CLASSDC. Windows 2000: This style is not supported.

Constant/value	Description
WS_EX_CONTEXTHELP 0x00000400L	<p>The title bar of the window includes a question mark. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message. The child window should pass the message to the parent window procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the child window.</p> <p>WS_EX_CONTEXTHELP cannot be used with the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles.</p>
WS_EX_CONTROLPARENT 0x00010000L	<p>The window itself contains child windows that should take part in dialog box navigation. If this style is specified, the dialog manager recurses into children of this window when performing navigation operations such as handling the TAB key, an arrow key, or a keyboard mnemonic.</p>
WS_EX_DLGMODALFRAME 0x00000001L	<p>The window has a double border; the window can, optionally, be created with a title bar by specifying the WS_CAPTION style in the <i>dwStyle</i> parameter.</p>
WS_EX_LAYERED 0x00080000	<p>The window is a layered window. This style cannot be used if the window has a class style of either CS_OWNDC or CS_CLASSDC.</p> <p>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.</p>
WS_EX_LAYOUTRTL 0x00400000L	<p>If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the horizontal origin of the window is on the right edge. Increasing horizontal values advance to the left.</p>
WS_EX_LEFT 0x00000000L	<p>The window has generic left-aligned properties. This is the default.</p>
WS_EX_LEFTSCROLLBAR 0x00004000L	<p>If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the vertical scroll bar (if present) is to the left of the client area. For other languages, the style is ignored.</p>
WS_EX_LTRREADING 0x00000000L	<p>The window text is displayed using left-to-right reading-order properties. This is the default.</p>
WS_EX_MDICHILD 0x00000040L	<p>The window is a MDI child window.</p>

Constant/value	Description
WS_EX_NOACTIVATE 0x08000000L	<p>A top-level window created with this style does not become the foreground window when the user clicks it. The system does not bring this window to the foreground when the user minimizes or closes the foreground window.</p> <p>The window should not be activated through programmatic access or via keyboard navigation by accessible technology, such as Narrator.</p> <p>To activate the window, use the SetActiveWindow or SetForegroundWindow function.</p> <p>The window does not appear on the taskbar by default. To force the window to appear on the taskbar, use the WS_EX_APPWINDOW style.</p>
WS_EX_NOINHERITLAYOUT 0x00100000L	<p>The window does not pass its window layout to its child windows.</p>
WS_EX_NOPARENTNOTIFY 0x00000004L	<p>The child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.</p>
WS_EX_NOREDIRECTIONBITMAP 0x00200000L	<p>The window does not render to a redirection surface. This is for windows that do not have visible content or that use mechanisms other than surfaces to provide their visual.</p>
WS_EX_OVERLAPPEDWINDOW (WS_EX_WINDOWEDGE WS_EX_CLIENTEDGE)	<p>The window is an overlapped window.</p>
WS_EX_PALETTEWINDOW (WS_EX_WINDOWEDGE WS_EX_TOOLWINDOW WS_EX_TOPMOST)	<p>The window is palette window, which is a modeless dialog box that presents an array of commands.</p>
WS_EX_RIGHT 0x00001000L	<p>The window has generic "right-aligned" properties. This depends on the window class. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading-order alignment; otherwise, the style is ignored.</p> <p>Using the WS_EX_RIGHT style for static or edit controls has the same effect as using the SS_RIGHT or ES_RIGHT style, respectively. Using this style with button controls has the same effect as using BS_RIGHT and BS_RIGHTBUTTON styles.</p>
WS_EX_RIGHTSCROLLBAR 0x00000000L	<p>The vertical scroll bar (if present) is to the right of the client area. This is the default.</p>

Constant/value	Description
WS_EX_RTLREADING 0x00002000L	If the shell language is Hebrew, Arabic, or another language that supports reading-order alignment, the window text is displayed using right-to-left reading-order properties. For other languages, the style is ignored.
WS_EX_STATICEDGE 0x00020000L	The window has a three-dimensional border style intended to be used for items that do not accept user input.
WS_EX_TOOLWINDOW 0x00000080L	The window is intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the taskbar or in the dialog that appears when the user presses ALT+TAB. If a tool window has a system menu, its icon is not displayed on the title bar. However, you can display the system menu by right-clicking or by typing ALT+SPACE.
WS_EX_TOPMOST 0x00000008L	The window should be placed above all non-topmost windows and should stay above them, even when the window is deactivated. To add or remove this style, use the SetWindowPos function.
WS_EX_TRANSPARENT 0x00000020L	The window should not be painted until siblings beneath the window (that were created by the same thread) have been painted. The window appears transparent because the bits of underlying sibling windows have already been painted. To achieve transparency without these restrictions, use the SetWindowRgn function.
WS_EX_WINDOWEDGE 0x00000100L	The window has a border with a raised edge.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

Feedback



Was this page helpful?  Yes  No

Get help at Microsoft Q&A

Window Styles

Article • 08/19/2022

The following are the window styles. After the window has been created, these styles cannot be modified, except as noted.

Constant name	Constant value	Description
WS_BORDER	0x00800000L	The window has a thin-line border
WS_CAPTION	0x00C00000L	The window has a title bar (includes the WS_BORDER style).
WS_CHILD	0x40000000L	The window is a child window. A window with this style cannot have a menu bar. This style cannot be used with the WS_POPUP style.
WS_CHILDWINDOW	0x40000000L	Same as the WS_CHILD style.
WS_CLIPCHILDREN	0x02000000L	Excludes the area occupied by child windows when drawing occurs within the parent window. This style is used when creating the parent window.
WS_CLIPSIBLINGS	0x04000000L	Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, the WS_CLIPSIBLINGS style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.
WS_DISABLED	0x08000000L	The window is initially disabled. A disabled window cannot receive input from the user. To change this after a window has been created, use the EnableWindow function.
WS_DLGFREAME	0x00400000L	The window has a border of a style typically used with dialog boxes. A window with this style cannot have a title bar.

Constant name	Constant value	Description
WS_GROUP	0x00020000L	<p>The window is the first control of a group of controls. The group consists of this first control and all controls defined after it, up to the next control with the WS_GROUP style. The first control in each group usually has the WS_TABSTOP style so that the user can move from group to group. The user can subsequently change the keyboard focus from one control in the group to the next control in the group by using the direction keys.</p> <p>You can turn this style on and off to change dialog box navigation. To change this style after a window has been created, use the SetWindowLong function.</p>
WS_HSCROLL	0x00100000L	The window has a horizontal scroll bar.
WS_ICONIC	0x20000000L	The window is initially minimized. Same as the WS_MINIMIZE style.
WS_MAXIMIZE	0x01000000L	The window is initially maximized.
WS_MAXIMIZEBOX	0x00010000L	The window has a maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_MINIMIZE	0x20000000L	The window is initially minimized. Same as the WS_ICONIC style.
WS_MINIMIZEBOX	0x00020000L	The window has a minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_OVERLAPPED	0x00000000L	The window is an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style.
WS_OVERLAPPEDWINDOW	(WS_OVERLAPPED WS_CAPTION WS_SYSMENU WS_THICKFRAME WS_MINIMIZEBOX WS_MAXIMIZEBOX)	The window is an overlapped window. Same as the WS_TILEDWINDOW style.
WS_POPUP	0x80000000L	The window is a pop-up window. This style cannot be used with the WS_CHILD style.

Constant name	Constant value	Description
WS_POPUPWINDOW	(WS_POPUP WS_BORDER WS_SYSMENU)	The window is a pop-up window. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible.
WS_SIZEBOX	0x00040000L	The window has a sizing border. Same as the WS_THICKFRAME style.
WS_SYSMENU	0x00080000L	The window has a window menu on its title bar. The WS_CAPTION style must also be specified.
WS_TABSTOP	0x00010000L	<p>The window is a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control with the WS_TABSTOP style.</p> <p>You can turn this style on and off to change dialog box navigation. To change this style after a window has been created, use the SetWindowLong function. For user-created windows and modeless dialogs to work with tab stops, alter the message loop to call the IsDialogMessage function.</p>
WS_THICKFRAME	0x00040000L	The window has a sizing border. Same as the WS_SIZEBOX style.
WS_TILED	0x00000000L	The window is an overlapped window. An overlapped window has a title bar and a border. Same as the WS_OVERLAPPED style.
WS_TILEDWINDOW	(WS_OVERLAPPED WS_CAPTION WS_SYSMENU WS_THICKFRAME WS_MINIMIZEBOX WS_MAXIMIZEBOX)	The window is an overlapped window. Same as the WS_OVERLAPPEDWINDOW style.
WS_VISIBLE	0x10000000L	<p>The window is initially visible. This style can be turned on and off by using the ShowWindow or SetWindowPos function.</p>
WS_VSCROLL	0x00200000L	The window has a vertical scroll bar.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

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

Window functions

Article • 04/15/2021

- [AdjustWindowRect](#)
- [AdjustWindowRectEx](#)
- [AllowSetForegroundWindow](#)
- [AnimateWindow](#)
- [AnyPopup](#)
- [ArrangeIconicWindows](#)
- [BeginDeferWindowPos](#)
- [BringWindowToTop](#)
- [CalculatePopupWindowPosition](#)
- [CascadeWindows](#)
- [ChangeWindowMessageFilter](#)
- [ChangeWindowMessageFilterEx](#)
- [ChildWindowFromPoint](#)
- [ChildWindowFromPointEx](#)
- [CloseWindow](#)
- [CreateWindow](#)
- [CreateWindowEx](#)
- [DeferWindowPos](#)
- [DeregisterShellHookWindow](#)
- [DestroyWindow](#)
- [EndDeferWindowPos](#)
- [EndTask](#)
- [*EnumChildProc*](#)
- [EnumChildWindows](#)
- [EnumThreadWindows](#)
- [*EnumThreadWndProc*](#)
- [EnumWindows](#)
- [*EnumWindowsProc*](#)
- [FindWindow](#)
- [FindWindowEx](#)
- [GetAltTabInfo](#)
- [GetAncestor](#)
- [GetClientRect](#)
- [GetDesktopWindow](#)
- [GetForegroundWindow](#)
- [GetGUIThreadInfo](#)
- [GetLastActivePopup](#)

- [GetLayeredWindowAttributes](#)
- [GetNextWindow](#)
- [GetParent](#)
- [GetProcessDefaultLayout](#)
- [GetShellWindow](#)
- [GetSysColor](#)
- [GetTitleBarInfo](#)
- [GetTopWindow](#)
- [GetWindow](#)
- [GetWindowDisplayAffinity](#)
- [GetWindowInfo](#)
- [GetWindowModuleFileName](#)
- [GetWindowPlacement](#)
- [GetWindowRect](#)
- [GetWindowText](#)
- [GetWindowTextLength](#)
- [GetWindowThreadProcessId](#)
- [InternalGetWindowText](#)
- [IsChild](#)
- [IsGUIThread](#)
- [IsHungAppWindow](#)
- [IsIconic](#)
- [IsProcessDPIAware](#)
- [IsWindow](#)
- [IsWindowUnicode](#)
- [IsWindowVisible](#)
- [IsZoomed](#)
- [LockSetForegroundWindow](#)
- [LogicalToPhysicalPoint](#)
- [MoveWindow](#)
- [OpenIcon](#)
- [PhysicalToLogicalPoint](#)
- [RealChildWindowFromPoint](#)
- [RealGetWindowClass](#)
- [RegisterShellHookWindow](#)
- [SetForegroundWindow](#)
- [SetLayeredWindowAttributes](#)
- [SetParent](#)
- [SetProcessDefaultLayout](#)
- [SetProcessDPIAware](#)

- [SetSysColors](#)
 - [SetWindowDisplayAffinity](#)
 - [SetWindowFeedbackSettings](#)
 - [SetWindowPlacement](#)
 - [SetWindowPos](#)
 - [SetWindowText](#)
 - [ShowOwnedPopups](#)
 - [ShowWindow](#)
 - [ShowWindowAsync](#)
 - [SoundSentry](#)
 - [SwitchToThisWindow](#)
 - [TileWindows](#)
 - [UpdateLayeredWindow](#)
 - [UpdateLayeredWindowIndirect](#)
 - [WindowFromPhysicalPoint](#)
 - [WindowFromPoint](#)
 - [WinMain](#)
 - [WNDPROC](#)
-

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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?

[!\[\]\(b53ef8cd03d3c3f50b768b406d2d0cba_img.jpg\) Yes](#)

[!\[\]\(b21b6bf629ef3e7444a1b1325d2a3c8b_img.jpg\) 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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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

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](#)

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](#)

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)

Article10/13/2021

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](#)

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](#)

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](#)

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.

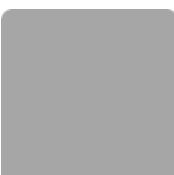
System colors for monochrome displays are usually interpreted as shades of gray.

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](#).

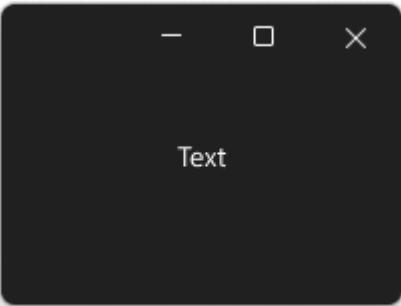
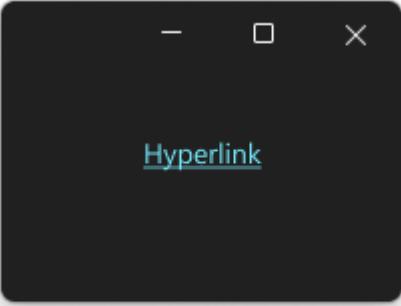
Windows 10/11 system colors

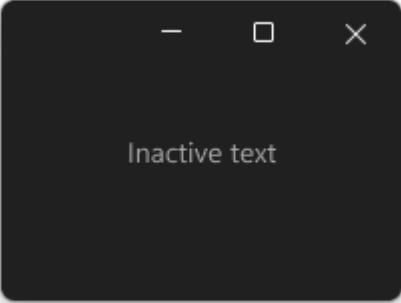
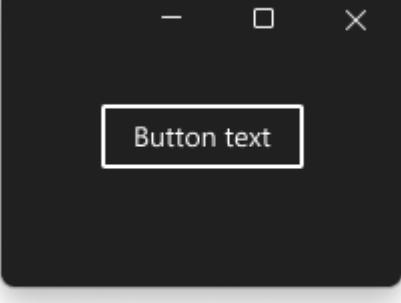
This table shows the values that are supported in Windows 10 and Windows 11 with color values from the Windows 11 *Aquatic* contrast theme.

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](#)

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](#)

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](#)

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](#)

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](#)

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)

Article10/13/2021

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)

Article10/13/2021

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](#)

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](#)

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](#)

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](#)

OpenIcon function (winuser.h)

Article 10/13/2021

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](#)

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](#)

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

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

Window Macros

Article • 06/30/2023

- [GET_X_LPARAM](#)
 - [GET_Y_LPARAM](#)
 - [GET_MOUSEORKEY_LPARAM](#)
 - [HIBYTE](#)
 - [HIWORD](#)
 - [LOBYTE](#)
 - [LOWORD](#)
 - [MAKELONG](#)
 - [MAKELPARAM](#)
 - [MAKELRESULT](#)
 - [MAKEWORD](#)
 - [MAKEWPARAM](#)
-

Feedback

Was this page helpful?

 Yes

 No

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](#)

GET_MOUSEORKEY_LPARAM macro

Article • 06/30/2023

Retrieves the input device type from the specified LPARAM value.

Syntax

```
C++  
  
WORD GET_MOUSEORKEY_LPARAM(  
    LPARAM lParam  
);
```

-param IParam

The value to be converted.

Return value

The return value is the bit of the high-order word representing the input device type. It can be one of the following values.

Return code/value	Description
FAPPCOMMAND_KEY 0	User pressed a key.
FAPPCOMMAND_MOUSE 0x8000	User clicked a mouse button.
FAPPCOMMAND_OEM 0x1000	An unidentified hardware source generated the event. It could be a mouse or a keyboard event.

Remarks

This macro is identical to the [GET_DEVICE_LPARAM macro](#) macro.

See also

[GET_DEVICE_LPARAM macro](#), [Mouse Input](#)

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](#)

Window Messages (Windows and Messages)

Article • 04/27/2021

In This Section

- [MN_GETHMENU](#)
- [WM_ERASEBKGND](#)
- [WM_GETFONT](#)
- [WM_GETTEXT](#)
- [WM_GETTEXTLENGTH](#)
- [WM_SETFONT](#)
- [WM_SETICON](#)
- [WM_SETTEXT](#)

Feedback

Was this page helpful?

 Yes

 No

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

MN_GETHMENU message

Article • 01/07/2021

Retrieves the menu handle for the current window.

C++

```
#define MN_GETHMENU 0x01E1
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: **HMENU**

If successful, the return value is the **HMENU** for the current window. If it fails, the return value is **NULL**.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

WM_ERASEBKGND message

Article • 01/07/2021

Sent when the window background must be erased (for example, when a window is resized). The message is sent to prepare an invalidated portion of a window for painting.

C++

```
#define WM_ERASEBKGND 0x0014
```

Parameters

wParam

A handle to the device context.

lParam

This parameter is not used.

Return value

Type: LRESULT

An application should return nonzero if it erases the background; otherwise, it should return zero.

Remarks

The [DefWindowProc](#) function erases the background by using the class background brush specified by the **hbrBackground** member of the [WNDCLASS](#) structure. If **hbrBackground** is **NULL**, the application should process the **WM_ERASEBKGND** message and erase the background.

An application should return nonzero in response to **WM_ERASEBKGND** if it processes the message and erases the background; this indicates that no further erasing is required. If the application returns zero, the window will remain marked for erasing. (Typically, this indicates that the **fErase** member of the [PAINTSTRUCT](#) structure will be **TRUE**.)

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[WNDCLASS](#)

Conceptual

[Icons](#)

Other Resources

[BeginPaint](#)

[PAINTSTRUCT](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_GETFONT message

Article • 01/07/2021

Retrieves the font with which the control is currently drawing its text.

C++

```
#define WM_GETFONT 0x0031
```

Parameters

wParam

This parameter is not used and must be zero.

lParam

This parameter is not used and must be zero.

Return value

Type: **HFONT**

The return value is a handle to the font used by the control, or **NULL** if the control is using the system font.

Requirements

Requirement	Value
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

Reference

[WM_SETFONT](#)

Conceptual

Windows

Feedback

Was this page helpful?

 Yes

 No

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

WM_GETTEXT message

Article • 01/07/2021

Copies the text that corresponds to a window into a buffer provided by the caller.

C++

```
#define WM_GETTEXT 0x000D
```

Parameters

wParam

The maximum number of characters to be copied, including the terminating null character.

ANSI applications may have the string in the buffer reduced in size (to a minimum of half that of the *wParam* value) due to conversion from ANSI to Unicode.

lParam

A pointer to the buffer that is to receive the text.

Return value

Type: LRESULT

The return value is the number of characters copied, not including the terminating null character.

Remarks

The [DefWindowProc](#) function copies the text associated with the window into the specified buffer and returns the number of characters copied. Note, for non-text static controls this gives you the text with which the control was originally created, that is, the ID number. However, it gives you the ID of the non-text static control as originally created. That is, if you subsequently used a [STM_SETIMAGE](#) to change it the original ID would still be returned.

For an edit control, the text to be copied is the content of the edit control. For a combo box, the text is the content of the edit control (or static-text) portion of the combo box.

For a button, the text is the button name. For other windows, the text is the window title. To copy the text of an item in a list box, an application can use the [LB_GETTEXT](#) message.

When the [WM_GETTEXT](#) message is sent to a static control with the [SS_ICON](#) style, a handle to the icon will be returned in the first four bytes of the buffer pointed to by *lParam*. This is true only if the [WM_SETTEXT](#) message has been used to set the icon.

Rich Edit: If the text to be copied exceeds 64K, use either the [EM_STREAMOUT](#) or [EM_GETSELTEXT](#) message.

Sending a [WM_GETTEXT](#) message to a non-text static control, such as a static bitmap or static icon control, does not return a string value. Instead, it returns zero. In addition, in early versions of Windows, applications could send a [WM_GETTEXT](#) message to a non-text static control to retrieve the control's ID. To retrieve a control's ID, applications can use [GetWindowLong](#) passing [GWL_ID](#) as the index value or [GetWindowLongPtr](#) using [GWLP_ID](#).

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[GetWindowLong](#)

[GetWindowLongPtr](#)

[GetWindowText](#)

[GetWindowTextLength](#)

[WM_GETTEXTLENGTH](#)

[WM_SETTEXT](#)

[Conceptual](#)

[Windows](#)

[Other Resources](#)

[EM_GETSELTEXT](#)

[EM_STREAMOUT](#)

[LB_GETTEXT](#)

Feedback

Was this page helpful?



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

WM_GETTEXTLENGTH message

Article • 01/07/2021

Determines the length, in characters, of the text associated with a window.

C++

```
#define WM_GETTEXTLENGTH 0x000E
```

Parameters

wParam

This parameter is not used and must be zero.

lParam

This parameter is not used and must be zero.

Return value

Type: LRESULT

The return value is the length of the text in characters, not including the terminating null character.

Remarks

For an edit control, the text to be copied is the content of the edit control. For a combo box, the text is the content of the edit control (or static-text) portion of the combo box. For a button, the text is the button name. For other windows, the text is the window title. To determine the length of an item in a list box, an application can use the [LB_GETTEXTLEN](#) message.

When the **WM_GETTEXTLENGTH** message is sent, the [DefWindowProc](#) function returns the length, in characters, of the text. Under certain conditions, the [DefWindowProc](#) function returns 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.

To obtain the exact length of the text, use the [WM_GETTEXT](#), [LB_GETTEXT](#), or [CB_GETLBTEXT](#) messages, or the [GetWindowText](#) function.

Sending a [WM_GETTEXTLENGTH](#) message to a non-text static control, such as a static bitmap or static icon control, does not return a string value. Instead, it returns zero.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[GetWindowText](#)

[GetWindowTextLength](#)

[WM_GETTEXT](#)

Conceptual

[Windows](#)

Other Resources

[CB_GETLBTEXT](#)

[LB_GETTEXT](#)

[LB_GETTEXTLEN](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SETFONT message

Article • 01/07/2021

Sets the font that a control is to use when drawing text.

C++

```
#define WM_SETFONT 0x0030
```

Parameters

wParam

A handle to the font (**HFONT**). If this parameter is **NULL**, the control uses the default system font to draw text.

lParam

The low-order word of *lParam* specifies whether the control should be redrawn immediately upon setting the font. If this parameter is **TRUE**, the control redraws itself.

Return value

Type: **LRESULT**

This message does not return a value.

Remarks

The **WM_SETFONT** message applies to all controls, not just those in dialog boxes.

The best time for the owner of a dialog box control to set the font of the control is when it receives the **WM_INITDIALOG** message. The application should call the **DeleteObject** function to delete the font when it is no longer needed; for example, after it destroys the control.

The size of the control does not change as a result of receiving this message. To avoid clipping text that does not fit within the boundaries of the control, the application should correct the size of the control window before it sets the font.

When a dialog box uses the [DS_SETFONT](#) style to set the text in its controls, the system sends the [WM_SETFONT](#) message to the dialog box procedure before it creates the controls. An application can create a dialog box that contains the DS_SETFONT style by calling any of the following functions:

- [CreateDialogIndirect](#)
- [CreateDialogIndirectParam](#)
- [DialogBoxIndirect](#)
- [DialogBoxIndirectParam](#)

Requirements

Requirement	Value
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

Reference

[CreateDialogIndirect](#)

[CreateDialogIndirectParam](#)

[DialogBoxIndirect](#)

[DialogBoxIndirectParam](#)

[DLGTEMPLATE](#)

[MAKELPARAM](#)

[WM_GETFONT](#)

[WM_INITDIALOG](#)

[Conceptual](#)

[Windows](#)

[Other Resources](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SETICON message

Article • 03/13/2023

Associates a new large or small icon with a window. The system displays the large icon in the ALT+TAB dialog box, and the small icon in the window caption.

C++

```
#define WM_SETICON 0x0080
```

Parameters

wParam

The type of icon to be set. This parameter can be one of the following values.

Value	Meaning
ICON_BIG 1	Set the large icon for the window.
ICON_SMALL 0	Set the small icon for the window.

lParam

A handle to the new large or small icon. If this parameter is **NULL**, the icon indicated by *wParam* is removed.

Return value

Type: **LRESULT**

The return value is a handle to the previous large or small icon, depending on the value of *wParam*. It is **NULL** if the window previously had no icon of the type indicated by *wParam*.

Remarks

The [DefWindowProc](#) function returns a handle to the previous large or small icon associated with the window, depending on the value of *wParam*.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[WM_GETICON](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SETTEXT message

Article • 01/07/2021

Sets the text of a window.

C++

```
#define WM_SETTEXT 0x000C
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a null-terminated string that is the window text.

Return value

Type: LRESULT

The return value is **TRUE** if the text is set. It is **FALSE** (for an edit control), **LB_ERRSPACE** (for a list box), or **CB_ERRSPACE** (for a combo box) if insufficient space is available to set the text in the edit control. It is **CB_ERR** if this message is sent to a combo box without an edit control.

Remarks

The [DefWindowProc](#) function sets and displays the window text. For an edit control, the text is the contents of the edit control. For a combo box, the text is the contents of the edit-control portion of the combo box. For a button, the text is the button name. For other windows, the text is the window title.

This message does not change the current selection in the list box of a combo box. An application should use the [CB_SELECTSTRING](#) message to select the item in a list box that matches the text in the edit control.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[WM_GETTEXT](#)

Conceptual

[Windows](#)

Other Resources

[CB_SELECTSTRING](#)

Feedback

Was this page helpful?

 Yes

 No

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

Window Notifications

Article • 04/27/2021

In This Section

- [WM_ACTIVATEAPP](#)
- [WM_CANCELMODE](#)
- [WM_CHILDACTIVATE](#)
- [WM_CLOSE](#)
- [WM_COMPACTING](#)
- [WM_CREATE](#)
- [WM_DESTROY](#)
- [WM_DPICHANGED](#)
- [WM_ENABLE](#)
- [WM_ENTERSIZEMOVE](#)
- [WM_EXITSIZEMOVE](#)
- [WM_GETICON](#)
- [WM_GETMINMAXINFO](#)
- [WM_INPUTLANGCHANGE](#)
- [WM_INPUTLANGCHANGEREQUEST](#)
- [WM_MOVE](#)
- [WM_MOVING](#)
- [WM_NCACTIVATE](#)
- [WM_NCCALCSIZE](#)
- [WM_NCCREATE](#)
- [WM_NCDESTROY](#)
- [WM_NULL](#)
- [WM_QUERYDRAGICON](#)
- [WM_QUERYOPEN](#)
- [WM_QUIT](#)
- [WM_SHOWWINDOW](#)
- [WM_SIZE](#)
- [WM_SIZING](#)
- [WM_STYLECHANGED](#)
- [WM_STYLECHANGING](#)
- [WM_THEMECHANGED](#)
- [WM_USERCHANGED](#)
- [WM_WINDOWPOSCHANGED](#)
- [WM_WINDOWPOSCHANGING](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_ACTIVATEAPP message

Article • 01/07/2021

Sent when a window belonging to a different application than the active window is about to be activated. The message is sent to the application whose window is being activated and to the application whose window is being deactivated.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_ACTIVATEAPP 0x001C
```

Parameters

wParam

Indicates whether the window is being activated or deactivated. This parameter is **TRUE** if the window is being activated; it is **FALSE** if the window is being deactivated.

lParam

The thread identifier. If the *wParam* parameter is **TRUE**, *lParam* is the identifier of the thread that owns the window being deactivated. If *wParam* is **FALSE**, *lParam* is the identifier of the thread that owns the window being activated.

Return value

Type: **LRESULT**

If an application processes this message, it should return zero.

Requirements

Requirement	Value
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

Reference

[WM_ACTIVATE](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_CANCELMODE message

Article • 01/07/2021

Sent to cancel certain modes, such as mouse capture. For example, the system sends this message to the active window when a dialog box or message box is displayed. Certain functions also send this message explicitly to the specified window regardless of whether it is the active window. For example, the [EnableWindow](#) function sends this message when disabling the specified window.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_CANCELMODE 0x001F
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Remarks

When the WM_CANCELMODE message is sent, the [DefWindowProc](#) function cancels internal processing of standard scroll bar input, cancels internal menu processing, and releases the mouse capture.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[DefWindowProc](#)

[EnableWindow](#)

[ReleaseCapture](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_CHILDACTIVATE message

Article • 01/07/2021

Sent to a child window when the user clicks the window's title bar or when the window is activated, moved, or sized.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_CHILDACTIVATE 0x0022
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Requirements

Requirement	Value
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

[Reference](#)

[MoveWindow](#)

[SetWindowPos](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_CLOSE message

Article • 04/07/2022

Sent as a signal that a window or an application should terminate.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_CLOSE          0x0010
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Example

C++

```
LRESULT CALLBACK WindowProc(
    __in HWND hWnd,
    __in UINT uMsg,
    __in WPARAM wParam,
    __in LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CLOSE:
            DestroyWindow(hWnd);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
    }
}
```

```
        break;
    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }

    return 0;
}
```

Example from [Windows Classic Samples](#) on GitHub.

Remarks

An application can prompt the user for confirmation, prior to destroying a window, by processing the **WM_CLOSE** message and calling the [DestroyWindow](#) function only if the user confirms the choice.

By default, the [DefWindowProc](#) function calls the [DestroyWindow](#) function to destroy the window.

Requirements

Requirement	Value
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

[Reference](#)

[DefWindowProc](#)

[DestroyWindow](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_COMPACTING message

Article • 01/07/2021

Sent to all top-level windows when the system detects more than 12.5 percent of system time over a 30- to 60-second interval is being spent compacting memory. This indicates that system memory is low.

A window receives this message through its [WindowProc](#) function.

ⓘ Note

This message is provided only for compatibility with 16-bit Windows-based applications.

C++

```
#define WM_COMPACTING      0x0041
```

Parameters

wParam

The ratio of central processing unit (CPU) time currently spent by the system compacting memory to CPU time currently spent by the system performing other operations. For example, 0x8000 represents 50 percent of CPU time spent compacting memory.

lParam

This parameter is not used.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Remarks

When an application receives this message, it should free as much memory as possible, taking into account the current level of activity of the application and the total number of applications running on the system.

Requirements

Requirement	Value
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

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_CREATE message

Article • 01/07/2021

Sent when an application requests that a window be created by calling the [CreateWindowEx](#) or [CreateWindow](#) function. (The message is sent before the function returns.) The window procedure of the new window receives this message after the window is created, but before the window becomes visible.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_CREATE 0x0001
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a [CREATESTRUCT](#) structure that contains information about the window being created.

Return value

Type: LRESULT

If an application processes this message, it should return zero to continue creation of the window. If the application returns –1, the window is destroyed and the [CreateWindowEx](#) or [CreateWindow](#) function returns a **NULL** handle.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Requirement	Value
Header	Winuser.h (include Windows.h)

See also

Reference

[CreateWindow](#)

[CreateWindowEx](#)

[CREATESTRUCT](#)

[WM_NCCREATE](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_DESTROY message

Article • 01/07/2021

Sent when a window is being destroyed. It is sent to the window procedure of the window being destroyed after the window is removed from the screen.

This message is sent first to the window being destroyed and then to the child windows (if any) as they are destroyed. During the processing of the message, it can be assumed that all child windows still exist.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_DESTROY      0x0002
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: [LRESULT](#)

If an application processes this message, it should return zero.

Remarks

If the window being destroyed is part of the clipboard viewer chain (set by calling the [SetClipboardViewer](#) function), the window must remove itself from the chain by processing the [ChangeClipboardChain](#) function before returning from the **WM_DESTROY** message.

Requirements

Requirement	Value
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

Reference

[ChangeClipboardChain](#)

[DestroyWindow](#)

[PostQuitMessage](#)

[SetClipboardViewer](#)

[WM_CLOSE](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_ENABLE message

Article • 01/07/2021

Sent when an application changes the enabled state of a window. It is sent to the window whose enabled state is changing. This message is sent before the [EnableWindow](#) function returns, but after the enabled state ([WS_DISABLED](#) style bit) of the window has changed.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_ENABLE 0x000A
```

Parameters

wParam

Indicates whether the window has been enabled or disabled. This parameter is **TRUE** if the window has been enabled or **FALSE** if the window has been disabled.

lParam

This parameter is not used.

Return value

Type: **LRESULT**

If an application processes this message, it should return zero.

Requirements

Requirement	Value
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

Reference

[EnableWindow](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_ENTERSIZEMOVE message

Article • 01/07/2021

Sent one time to a window after it enters the moving or sizing modal loop. The window enters the moving or sizing modal loop when the user clicks the window's title bar or sizing border, or when the window passes the [WM_SYSCOMMAND](#) message to the [DefWindowProc](#) function and the *wParam* parameter of the message specifies the SC_MOVE or SC_SIZE value. The operation is complete when [DefWindowProc](#) returns.

The system sends the WM_ENTERSIZEMOVE message regardless of whether the dragging of full windows is enabled.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_ENTERSIZEMOVE 0x0231
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: [LRESULT](#)

An application should return zero if it processes this message.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Requirement	Value
Header	Winuser.h (include Windows.h)

See also

Reference

[DefWindowProc](#)

[WM_EXITSIZEMOVE](#)

[WM_SYSCOMMAND](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_EXITSIZEMOVE message

Article • 01/07/2021

Sent one time to a window, after it has exited the moving or sizing modal loop. The window enters the moving or sizing modal loop when the user clicks the window's title bar or sizing border, or when the window passes the [WM_SYSCOMMAND](#) message to the [DefWindowProc](#) function and the *wParam* parameter of the message specifies the SC_MOVE or SC_SIZE value. The operation is complete when [DefWindowProc](#) returns.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_EXITSIZEMOVE 0x0232
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

An application should return zero if it processes this message.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[WM_ENTERSIZEMOVE](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_GETICON message

Article • 01/07/2021

Sent to a window to retrieve a handle to the large or small icon associated with a window. The system displays the large icon in the ALT+TAB dialog, and the small icon in the window caption.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_GETICON 0x007F
```

Parameters

wParam

The type of icon being retrieved. This parameter can be one of the following values.

Value	Meaning
ICON_BIG 1	Retrieve the large icon for the window.
ICON_SMALL 0	Retrieve the small icon for the window.
ICON_SMALL2 2	Retrieves the small icon provided by the application. If the application does not provide one, the system uses the system-generated icon for that window.

lParam

The DPI of the icon being retrieved. This can be used to provide different icons depending on the icon size.

Return value

Type: [HICON](#)

The return value is a handle to the large or small icon, depending on the value of *wParam*. When an application receives this message, it can return a handle to a large or

small icon, or pass the message to the [DefWindowProc](#) function.

Remarks

When an application receives this message, it can return a handle to a large or small icon, or pass the message to [DefWindowProc](#).

[DefWindowProc](#) returns a handle to the large or small icon associated with the window, depending on the value of *wParam*.

A window that has no icon explicitly set (with **WM_SETICON**) uses the icon for the registered window class, and in this case [DefWindowProc](#) will return 0 for a **WM_GETICON** message. If sending a **WM_GETICON** message to a window returns 0, next try calling the [GetClassLongPtr](#) function for the window. If that returns 0 then try the [LoadIcon](#) function.

Requirements

Requirement	Value
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

[Reference](#)

[DefWindowProc](#)

[WM_SETICON](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_GETMINMAXINFO message

Article • 01/07/2021

Sent to a window when the size or position of the window is about to change. An application can use this message to override the window's default maximized size and position, or its default minimum or maximum tracking size.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_GETMINMAXINFO 0x0024
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a [MINMAXINFO](#) structure that contains the default maximized position and dimensions, and the default minimum and maximum tracking sizes. An application can override the defaults by setting the members of this structure.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Remarks

The maximum tracking size is the largest window size that can be produced by using the borders to size the window. The minimum tracking size is the smallest window size that can be produced by using the borders to size the window.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[MoveWindow](#)

[SetWindowPos](#)

[MINMAXINFO](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_INPUTLANGCHANGE message

Article • 03/13/2023

Sent to the topmost affected window after an application's input language has been changed. You should make any application-specific settings and pass the message to the [DefWindowProc](#) function, which passes the message to all first-level child windows. These child windows can pass the message to [DefWindowProc](#) to have it pass the message to their child windows, and so on.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_INPUTLANGCHANGE 0x0051
```

Parameters

wParam

Type: [**WPARAM**](#)

The [BYTE](#) font character set for the input language.

See [iCharSet](#) parameter of the [CreateFont](#) function for a list of possible values.

lParam

Type: [**LPARAM**](#)

The [HKL](#) input locale identifier.

The low word contains a [Language Identifier](#) for the input language. The high word contains a device handle.

Return value

Type: [LRESULT](#)

An application should return nonzero if it processes this message.

Remarks

You can retrieve the [BCP 47 ↴ locale name](#) from the language identifier by calling the [LCIDToLocaleName](#) function. Once you have the locale name, you can then use [modern locale functions](#) to extract additional locale information.

C++

```
case WM_INPUTLANGCHANGE:  
{  
    HKL hkl = (HKL)lParam;  
    // LANGIDs are deprecated. Use BCP 47 locale names where possible.  
    LANGID langId = LOWORD(HandleToUlong(hkl));  
  
    WCHAR localeName[LOCALE_NAME_MAX_LENGTH];  
    LCIDToLocaleName(MAKELCID(langId, SORT_DEFAULT), localeName,  
    LOCALE_NAME_MAX_LENGTH, 0);  
  
    // Get the ISO abbreviated language name (for example, "eng").  
    WCHAR lang[9];  
    GetLocaleInfoEx(localeName, LOCALE_SISO639LANGNAME2, lang, 9);  
  
    // Get the keyboard layout identifier (for example, "00020409" for  
    United States-International keyboard layout)  
    WCHAR keyboardLayoutId[KL_NAMELENGTH];  
    GetKeyboardLayoutNameW(keyboardLayoutId);  
}
```

To get the the name of the currently active keyboard layout, call the [GetKeyboardLayoutName](#). For more information, see [Languages, Locales, and Keyboard Layouts](#).

For a list of the input layouts that are supplied with Windows, see [Keyboard Identifiers and Input Method Editors for Windows](#).

[Input Method Editor \(IME\)](#) profile changes may not be notified with [WM_INPUTLANGCHANGE](#). You can use [ITfActiveLanguageProfileNotifySink](#) from the [Text Services Framework](#) to handle active language or text service changes.

Starting with Windows 8

Some input layouts may not have assigned language identifiers and could be reported as transient language identifiers, such as `LOCALE_TRANSIENT_KEYBOARD1` (0x2000) or `LOCALE_TRANSIENT_KEYBOARD2` (0x2400), in the low word of `LPARAM`.

As these transient language identifiers can be re-assigned by the system at any time (such as when the user changes their Language Profile), and can identify a different locale based on the user and/or system, they cannot be considered durable identifiers. See [The deprecation of LCIDs](#) for more info.

Retrieve the language associated with the current keyboard layout or input method by calling [Windows.Globalization.Language.CurrentInputMethodLanguageTag](#). If your app passes language tags from [CurrentInputMethodLanguageTag](#) to any [National Language Support](#) functions, it must first convert the tags with [ResolveLocaleName](#). If you want to be notified about a language change in a UWP app, handle the [Windows.UI.Text.Core.CoreTextServicesManager.InputLanguageChanged](#) event.

Requirements

Requirement	Value
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

Reference

- [DefWindowProc](#)
- [WindowProc](#)
- [WM_INPUTLANGCHANGEREQUEST](#)
- [GetKeyboardLayout](#)
- [GetKeyboardLayoutList](#)
- [GetKeyboardLayoutName](#)
- [Windows.Globalization.Language.CurrentInputMethodLanguageTag](#)
- [Windows.UI.Text.Core.CoreTextServicesManager.InputLanguageChanged](#)

Conceptual

- [Windows](#)
- [Windows keyboard layouts](#)
- [Keyboard Identifiers and Input Method Editors for Windows](#)
- [Languages, Locales, and Keyboard Layouts](#)

Feedback

Was this page helpful?



Get help at Microsoft Q&A

WM_INPUTLANGCHANGEREQUEST message

Article • 01/07/2021

Posted to the window with the focus when the user chooses a new input language, either with the hotkey (specified in the Keyboard control panel application) or from the indicator on the system taskbar. An application can accept the change by passing the message to the [DefWindowProc](#) function or reject the change (and prevent it from taking place) by returning immediately.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_INPUTLANGCHANGEREQUEST 0x0050
```

Parameters

wParam

The new input locale. This parameter can be a combination of the following flags.

Value	Meaning
INPUTLANGCHANGE_BACKWARD 0x0004	A hot key was used to choose the previous input locale in the installed list of input locales. This flag cannot be used with the INPUTLANGCHANGE_FORWARD flag.
INPUTLANGCHANGE_FORWARD 0x0002	A hot key was used to choose the next input locale in the installed list of input locales. This flag cannot be used with the INPUTLANGCHANGE_BACKWARD flag.
INPUTLANGCHANGE_SYSCHARSET 0x0001	The new input locale's keyboard layout can be used with the system character set.

lParam

The input locale identifier. For more information, see [Languages, Locales, and Keyboard Layouts](#).

Return value

Type: LRESULT

This message is posted, not sent, to the application, so the return value is ignored. To accept the change, the application should pass the message to [DefWindowProc](#). To reject the change, the application should return zero without calling [DefWindowProc](#).

Remarks

When the [DefWindowProc](#) function receives the **WM_INPUTLANGCHANGEREQUEST** message, it activates the new input locale and notifies the application of the change by sending the [WM_INPUTLANGCHANGE](#) message.

The language indicator is present on the taskbar only if you have installed more than one keyboard layout and if you have enabled the indicator using the Keyboard control panel application.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[WM_INPUTLANGCHANGE](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MOVE message

Article • 11/19/2022

Sent after a window has been moved.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_MOVE          0x0003
```

Parameters

wParam

This parameter is not used.

lParam

The x and y coordinates of the upper-left corner of the client area of the window. The low-order word contains the x-coordinate while the high-order word contains the y coordinate.

Return value

Type: [LRESULT](#)

If an application processes this message, it should return zero.

Remarks

The parameters are given in screen coordinates for overlapped and pop-up windows and in parent-client coordinates for child windows.

The following example demonstrates how to obtain the position from the *lParam* parameter.

```
xPos = (int)(short) LOWORD(lParam); // horizontal position  
yPos = (int)(short) HIWORD(lParam); // vertical position
```

You can also use the [MAKEPOINTS](#) macro to convert the *lParam* parameter to a [POINTS](#) structure.

The [DefWindowProc](#) function sends the WM_SIZE and WM_MOVE messages when it processes the [WM_WINDOWPOSCHANGED](#) message. The WM_SIZE and WM_MOVE messages are not sent if an application handles the WM_WINDOWPOSCHANGED message without calling [DefWindowProc](#).

Requirements

Requirement	Value
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

Reference

[HIWORD](#)

[LOWORD](#)

[WM_WINDOWPOSCHANGED](#)

Conceptual

[Windows](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

WM_MOVING message

Article • 11/19/2022

Sent to a window that the user is moving. By processing this message, an application can monitor the position of the drag rectangle and, if needed, change its position.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_MOVING 0x0216
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a [RECT](#) structure with the current position of the window, in screen coordinates. To change the position of the drag rectangle, an application must change the members of this structure.

Return value

Type: [LRESULT](#)

An application should return [TRUE](#) if it processes this message.

Requirements

Requirement	Value
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

[Reference](#)

[WM_MOVE](#)

[WM_SIZING](#)

[Conceptual](#)

[Windows](#)

[Other Resources](#)

[RECT](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_NCACTIVATE message

Article • 01/07/2021

Sent to a window when its nonclient area needs to be changed to indicate an active or inactive state.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_NCACTIVATE 0x0086
```

Parameters

wParam

Indicates when a title bar or icon needs to be changed to indicate an active or inactive state. If an active title bar or icon is to be drawn, the *wParam* parameter is **TRUE**. If an inactive title bar or icon is to be drawn, *wParam* is **FALSE**.

lParam

When a [visual style](#) is active for this window, this parameter is not used.

When a visual style is not active for this window, this parameter is a handle to an optional update region for the nonclient area of the window. If this parameter is set to -1, [DefWindowProc](#) does not repaint the nonclient area to reflect the state change.

Return value

Type: **LRESULT**

When the *wParam* parameter is **FALSE**, an application should return **TRUE** to indicate that the system should proceed with the default processing, or it should return **FALSE** to prevent the change. When *wParam* is **TRUE**, the return value is ignored.

Remarks

Processing messages related to the nonclient area of a standard window is not recommended, because the application must be able to draw all the required parts of the nonclient area for the window. If an application does process this message, it must

return **TRUE** to direct the system to complete the change of active window. If the window is minimized when this message is received, the application should pass the message to the [DefWindowProc](#) function.

The [DefWindowProc](#) function draws the title bar or icon title in its active colors when the *wParam* parameter is **TRUE** and in its inactive colors when *wParam* is **FALSE**.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_NCCALCSIZE message

Article • 11/19/2022

Sent when the size and position of a window's client area must be calculated. By processing this message, an application can control the content of the window's client area when the size or position of the window changes.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_NCCALCSIZE 0x0083
```

Parameters

wParam

If *wParam* is **TRUE**, it specifies that the application should indicate which part of the client area contains valid information. The system copies the valid information to the specified area within the new client area.

If *wParam* is **FALSE**, the application does not need to indicate the valid part of the client area.

lParam

If *wParam* is **TRUE**, *lParam* points to an [NCCALCSIZE_PARAMS](#) structure that contains information an application can use to calculate the new size and position of the client rectangle.

If *wParam* is **FALSE**, *lParam* points to a [RECT](#) structure. On entry, the structure contains the proposed window rectangle for the window. On exit, the structure should contain the screen coordinates of the corresponding window client area.

Return value

Type: LRESULT

If the *wParam* parameter is **FALSE**, the application should return zero.

If *wParam* is **TRUE**, the application should return zero or a combination of the following values.

If *wParam* is TRUE and an application returns zero, the old client area is preserved and is aligned with the upper-left corner of the new client area.

Return code/value	Description
WVR_ALIGNTOP 0x0010	Specifies that the client area of the window is to be preserved and aligned with the top of the new position of the window. For example, to align the client area to the upper-left corner, return the WVR_ALIGNTOP and WVR_ALIGNLEFT values.
WVR_ALIGNRIGHT 0x0080	Specifies that the client area of the window is to be preserved and aligned with the right side of the new position of the window. For example, to align the client area to the lower-right corner, return the WVR_ALIGNRIGHT and WVR_ALIGNBOTTOM values.
WVR_ALIGNLEFT 0x0020	Specifies that the client area of the window is to be preserved and aligned with the left side of the new position of the window. For example, to align the client area to the lower-left corner, return the WVR_ALIGNLEFT and WVR_ALIGNBOTTOM values.
WVR_ALIGNBOTTOM 0x0040	Specifies that the client area of the window is to be preserved and aligned with the bottom of the new position of the window. For example, to align the client area to the top-left corner, return the WVR_ALIGNTOP and WVR_ALIGNLEFT values.
WVR_HREDRAW 0x0100	Used in combination with any other values, except WVR_VALIDRECTS , causes the window to be completely redrawn if the client rectangle changes size horizontally. This value is similar to CS_HREDRAW class style
WVR_VREDRAW 0x0200	Used in combination with any other values, except WVR_VALIDRECTS , causes the window to be completely redrawn if the client rectangle changes size vertically. This value is similar to CS_VREDRAW class style
WVR_REDRAW 0x0300	This value causes the entire window to be redrawn. It is a combination of WVR_HREDRAW and WVR_VREDRAW values.
WVR_VALIDRECTS 0x0400	<p>This value indicates that, upon return from WM_NCCALCSIZE, the rectangles specified by the <i>rgrc[1]</i> and <i>rgrc[2]</i> members of the NCCALCSIZE_PARAMS structure contain valid destination and source area rectangles, respectively. The system combines these rectangles to calculate the area of the window to be preserved. The system copies any part of the window image that is within the source rectangle and clips the image to the destination rectangle. Both rectangles are in parent-relative or screen-relative coordinates. This flag cannot be combined with any other flags.</p> <p>This return value allows an application to implement more elaborate client-area preservation strategies, such as centering or preserving a subset of the client area.</p>

Remarks

The window may be redrawn, depending on whether the [CS_HREDRAW](#) or [CS_VREDRAW](#) class style is specified. This is the default, backward-compatible processing of this message by the [DefWindowProc](#) function (in addition to the usual client rectangle calculation described in the preceding table).

When *wParam* is **TRUE**, simply returning 0 without processing the [NCCALCSIZE_PARAMS](#) rectangles will cause the client area to resize to the size of the window, including the window frame. This will remove the window frame and caption items from your window, leaving only the client area displayed.

Starting with Windows Vista, removing the standard frame by simply returning 0 when the *wParam* is **TRUE** does not affect frames that are extended into the client area using the [DwmExtendFrameIntoClientArea](#) function. Only the standard frame will be removed.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[MoveWindow](#)

[SetWindowPos](#)

[NCCALCSIZE_PARAMS](#)

Conceptual

[Windows](#)

Other Resources

Feedback

Was this page helpful?

 Yes

 No

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

WM_NCCREATE message

Article • 01/07/2021

Sent prior to the [WM_CREATE](#) message when a window is first created.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_NCCREATE 0x0081
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to the [CREATESTRUCT](#) structure that contains information about the window being created. The members of [CREATESTRUCT](#) are identical to the parameters of the [CreateWindowEx](#) function.

Return value

Type: [LRESULT](#)

If an application processes this message, it should return [TRUE](#) to continue creation of the window. If the application returns [FALSE](#), the [CreateWindow](#) or [CreateWindowEx](#) function will return a [NULL](#) handle.

Requirements

Requirement	Value
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

Reference

[CreateWindow](#)

[CreateWindowEx](#)

[DefWindowProc](#)

[CREATESTRUCT](#)

[WM_CREATE](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_NCDESTROY message

Article • 01/07/2021

Notifies a window that its nonclient area is being destroyed. The [DestroyWindow](#) function sends the WM_NCDESTROY message to the window following the WM_DESTROY message. WM_DESTROY is used to free the allocated memory object associated with the window.

The WM_NCDESTROY message is sent after the child windows have been destroyed. In contrast, WM_DESTROY is sent before the child windows are destroyed.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_NCDESTROY 0x0082
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Remarks

This message frees any memory internally allocated for the window.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[DestroyWindow](#)

[WM_DESTROY](#)

[WM_NCCREATE](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_NULL message

Article • 01/07/2021

Performs no operation. An application sends the WM_NULL message if it wants to post a message that the recipient window will ignore.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_NULL          0x0000
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

An application returns zero if it processes this message.

Remarks

For example, if an application has installed a [WH_GETMESSAGE](#) hook and wants to prevent a message from being processed, the [GetMsgProc](#) callback function can change the message number to WM_NULL so the recipient will ignore it.

As another example, an application can check if a window is responding to messages by sending the WM_NULL message with the [SendMessageTimeout](#) function.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_QUERYDRAGICON message

Article • 01/07/2021

Sent to a minimized (iconic) window. The window is about to be dragged by the user but does not have an icon defined for its class. An application can return a handle to an icon or cursor. The system displays this cursor or icon while the user drags the icon.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_QUERYDRAGICON 0x0037
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: [LRESULT](#)

An application should return a handle to a cursor or icon that the system is to display while the user drags the icon. The cursor or icon must be compatible with the display driver's resolution. If the application returns [NULL](#), the system displays the default cursor.

Remarks

When the user drags the icon of a window without a class icon, the system replaces the icon with a default cursor. If the application requires a different cursor to be displayed during dragging, it must return a handle to the cursor or icon compatible with the display driver's resolution. If an application returns a handle to a color cursor or icon, the system converts the cursor or icon to black and white. The application can call the

[LoadCursor](#) or [LoadIcon](#) function to load a cursor or icon from the resources in its executable (.exe) file and to retrieve this handle.

If a dialog box procedure handles this message, it should cast the desired return value to a **BOOL** and return the value directly. The **DWL_MSGRESULT** value set by the [SetWindowLong](#) function is ignored.

Requirements

Requirement	Value
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

Reference

[LoadCursor](#)

[LoadIcon](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_QUERYOPEN message

Article • 01/07/2021

Sent to an icon when the user requests that the window be restored to its previous size and position.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_QUERYOPEN 0x0013
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: LRESULT

If the icon can be opened, an application that processes this message should return **TRUE**; otherwise, it should return **FALSE** to prevent the icon from being opened.

Remarks

By default, the [DefWindowProc](#) function returns **TRUE**.

While processing this message, the application should not perform any action that would cause an activation or focus change (for example, creating a dialog box).

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[DefWindowProc](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_QUIT message

Article • 01/07/2021

Indicates a request to terminate an application, and is generated when the application calls the [PostQuitMessage](#) function. This message causes the [GetMessage](#) function to return zero.

C++

```
#define WM_QUIT      0x0012
```

Parameters

wParam

The exit code given in the [PostQuitMessage](#) function.

lParam

This parameter is not used.

Return value

Type: LRESULT

This message does not have a return value because it causes the message loop to terminate before the message is sent to the application's window procedure.

Remarks

The WM_QUIT message is not associated with a window and therefore will never be received through a window's window procedure. It is retrieved only by the [GetMessage](#) or [PeekMessage](#) functions.

Do not post the WM_QUIT message using the [PostMessage](#) function; use [PostQuitMessage](#).

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[GetMessage](#)

[PeekMessage](#)

[PostQuitMessage](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SHOWWINDOW message

Article • 01/07/2021

Sent to a window when the window is about to be hidden or shown.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_SHOWWINDOW 0x0018
```

Parameters

wParam

Indicates whether a window is being shown. If *wParam* is **TRUE**, the window is being shown. If *wParam* is **FALSE**, the window is being hidden.

lParam

The status of the window being shown. If *lParam* is zero, the message was sent because of a call to the [ShowWindow](#) function; otherwise, *lParam* is one of the following values.

Value	Meaning
SW_OTHERUNZOOM 4	The window is being uncovered because a maximize window was restored or minimized.
SW_OTHERZOOM 2	The window is being covered by another window that has been maximized.
SW_PARENTCLOSING 1	The window's owner window is being minimized.
SW_PARENTOPENING 3	The window's owner window is being restored.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Remarks

The [DefWindowProc](#) function hides or shows the window, as specified by the message. If a window has the [WS_VISIBLE](#) style when it is created, the window receives this message after it is created, but before it is displayed. A window also receives this message when its visibility state is changed by the [ShowWindow](#) or [ShowOwnedPopups](#) function.

The [WM_SHOWWINDOW](#) message is not sent under the following circumstances:

- When a top-level, overlapped window is created with the [WS_MAXIMIZE](#) or [WS_MINIMIZE](#) style.
- When the [SW_SHOWNORMAL](#) flag is specified in the call to the [ShowWindow](#) function.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[ShowOwnedPopups](#)

[ShowWindow](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SIZE message

Article • 04/07/2022

Sent to a window after its size has changed.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_SIZE          0x0005
```

Parameters

wParam

The type of resizing requested. This parameter can be one of the following values.

Value	Meaning
SIZE_MAXHIDE 4	Message is sent to all pop-up windows when some other window is maximized.
SIZE_MAXIMIZED 2	The window has been maximized.
SIZE_MAXSHOW 3	Message is sent to all pop-up windows when some other window has been restored to its former size.
SIZE_MINIMIZED 1	The window has been minimized.
SIZE_RESTORED 0	The window has been resized, but neither the SIZE_MINIMIZED nor SIZE_MAXIMIZED value applies.

lParam

The low-order word of *lParam* specifies the new width of the client area.

The high-order word of *lParam* specifies the new height of the client area.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Example

C++

```
/****************************************************************************
 * SimpleText::OnResize
 *
 * If the application receives a WM_SIZE message, this method
 * resize the render target appropriately.
 */
void SimpleText::OnResize(UINT width, UINT height)
{
    if (pRT_)
    {
        D2D1_SIZE_U size;
        size.width = width;
        size.height = height;
        pRT_->Resize(size);
    }
}

LRESULT CALLBACK SimpleText::WndProc(HWND hwnd, UINT message, WPARAM wParam,
LPARAM lParam)
{

    SimpleText *pSimpleText = reinterpret_cast<SimpleText *>(
        ::GetWindowLongPtr(hwnd, GWLP_USERDATA));

    if (pSimpleText)
    {
        switch(message)
        {
            case WM_SIZE:
            {
                UINT width = LOWORD(lParam);
                UINT height = HIWORD(lParam);
                pSimpleText->OnResize(width, height);
            }
            return 0;
        }
    }
    // ...
}
```

Example from [Windows classic samples](#) ↗ on GitHub.

Remarks

If the [SetScrollPos](#) or [MoveWindow](#) function is called for a child window as a result of the **WM_SIZE** message, the *bRedraw* or *bRepaint* parameter should be nonzero to cause the window to be repainted.

Although the width and height of a window are 32-bit values, the *lParam* parameter contains only the low-order 16 bits of each.

The [DefWindowProc](#) function sends the **WM_SIZE** and **WM_MOVE** messages when it processes the **WM_WINDOWPOSCHANGED** message. The **WM_SIZE** and **WM_MOVE** messages are not sent if an application handles the **WM_WINDOWPOSCHANGED** message without calling [DefWindowProc](#).

Requirements

Requirement	Value
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

[Reference](#)

[HIWORD](#)

[LOWORD](#)

[MoveWindow](#)

[WM_WINDOWPOSCHANGED](#)

[Conceptual](#)

[Windows](#)

[Other Resources](#)

[SetScrollPos](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SIZING message

Article • 11/19/2022

Sent to a window that the user is resizing. By processing this message, an application can monitor the size and position of the drag rectangle and, if needed, change its size or position.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_SIZING 0x0214
```

Parameters

wParam

The edge of the window that is being sized. This parameter can be one of the following values.

Value	Meaning
WMSZ_BOTTOM 6	Bottom edge
WMSZ_BOTTOMLEFT 7	Bottom-left corner
WMSZ_BOTTOMRIGHT 8	Bottom-right corner
WMSZ_LEFT 1	Left edge
WMSZ_RIGHT 2	Right edge
WMSZ_TOP 3	Top edge
WMSZ_TOPLEFT 4	Top-left corner
WMSZ_TOPRIGHT 5	Top-right corner

lParam

A pointer to a [RECT](#) structure with the screen coordinates of the drag rectangle. To change the size or position of the drag rectangle, an application must change the members of this structure.

Return value

Type: [LRESULT](#)

An application should return [TRUE](#) if it processes this message.

Requirements

Requirement	Value
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

Reference

[WM_MOVING](#)

[WM_SIZE](#)

Conceptual

[Windows](#)

Other Resources

[RECT](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_STYLECHANGED message

Article • 01/07/2021

Sent to a window after the [SetWindowLong](#) function has changed one or more of the window's styles.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_STYLECHANGED 0x007D
```

Parameters

wParam

Indicates whether the window's styles or extended window styles have changed. This parameter can be one or more of the following values.

Value	Meaning
GWL_EXSTYLE -20	The extended window styles have changed.
GWL_STYLE -16	The window styles have changed.

lParam

A pointer to a [STYLESTRUCT](#) structure that contains the new styles for the window. An application can examine the styles, but cannot change them.

Return value

Type: LRESULT

An application should return zero if it processes this message.

Requirements

Requirement	Value
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

Reference

[SetWindowLong](#)

[STYLESTRUCT](#)

[WM_STYLECHANGING](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_STYLECHANGING message

Article • 01/07/2021

Sent to a window when the [SetWindowLong](#) function is about to change one or more of the window's styles.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_STYLECHANGING 0x007C
```

Parameters

wParam

Indicates whether the window's styles or extended window styles are changing. This parameter can be one or more of the following values.

Value	Meaning
GWL_EXSTYLE -20	The extended window styles are changing.
GWL_STYLE -16	The window styles are changing.

lParam

A pointer to a [STYLESTRUCT](#) structure that contains the proposed new styles for the window. An application can examine the styles and, if necessary, change them.

Return value

Type: LRESULT

An application should return zero if it processes this message.

Requirements

Requirement	Value
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

[Reference](#)

[STYLESTRUCT](#)

[WM_STYLECHANGED](#)

[Conceptual](#)

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_THEMECHANGED message

Article • 01/07/2021

Broadcast to every window following a theme change event. Examples of theme change events are the activation of a theme, the deactivation of a theme, or a transition from one theme to another.

C++

```
#define WM_THEMECHANGED 0x031A
```

Parameters

wParam

This parameter is reserved.

lParam

This parameter is reserved.

Return value

Type: LRESULT

If an application processes this message, it should return zero.

Remarks

A window receives this message through its [WindowProc](#) function.

ⓘ Note

This message is posted by the operating system. Applications typically do not send this message.

Themes are specifications for the appearance of controls, so that the visual element of a control is treated separately from its functionality.

To release an existing theme handle, call [CloseThemeData](#). To acquire a new theme handle, use [OpenThemeData](#).

Following the **WM_THEMECHANGED** broadcast, any existing theme handles are invalid. A theme-aware window should release and reopen any of its pre-existing theme handles when it receives the **WM_THEMECHANGED** message. If the [OpenThemeData](#) function returns **NULL**, the window should paint unthemed.

Requirements

Requirement	Value
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

Other Resources

[CloseThemeData](#)

[IsThemeActive](#)

[OpenThemeData](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_USERCHANGED message

Article • 01/07/2021

Sent to all windows after the user has logged on or off. When the user logs on or off, the system updates the user-specific settings. The system sends this message immediately after updating the settings.

A window receives this message through its [WindowProc](#) function.

ⓘ Note

This message is not supported as of Windows Vista.

C++

```
#define WM_USERCHANGED 0x0054
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: [LRESULT](#)

An application should return zero if it processes this message.

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]

Requirement	Value
Minimum supported server	None supported
Header	Winuser.h (include Windows.h)

See also

[Windows Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_WINDOWPOSCHANGED message

Article • 01/07/2021

Sent to a window whose size, position, or place in the Z order has changed as a result of a call to the [SetWindowPos](#) function or another window-management function.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_WINDOWPOSCHANGED 0x0047
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a [WINDOWPOS](#) structure that contains information about the window's new size and position.

Return value

Type: [LRESULT](#)

If an application processes this message, it should return zero.

Remarks

By default, the [DefWindowProc](#) function sends the [WM_SIZE](#) and [WM_MOVE](#) messages to the window. The [WM_SIZE](#) and [WM_MOVE](#) messages are not sent if an application handles the [WM_WINDOWPOSCHANGED](#) message without calling [DefWindowProc](#). It is more efficient to perform any move or size change processing during the [WM_WINDOWPOSCHANGED](#) message without calling [DefWindowProc](#).

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[DefWindowProc](#)

[EndDeferWindowPos](#)

[SetWindowPos](#)

[WINDOWPOS](#)

[WM_MOVE](#)

[WM_SIZE](#)

[WM_WINDOWPOSCHANGING](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_WINDOWPOSCHANGING message

Article • 01/07/2021

Sent to a window whose size, position, or place in the Z order is about to change as a result of a call to the [SetWindowPos](#) function or another window-management function.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_WINDOWPOSCHANGING 0x0046
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a [WINDOWPOS](#) structure that contains information about the window's new size and position.

Return value

Type: [LRESULT](#)

If an application processes this message, it should return zero.

Remarks

For a window with the [WS_OVERLAPPED](#) or [WS_THICKFRAME](#) style, the [DefWindowProc](#) function sends the [WM_GETMINMAXINFO](#) message to the window. This is done to validate the new size and position of the window and to enforce the [CS_BYTEALIGNCLIENT](#) and [CS_BYTEALIGNWINDOW](#) client styles. By not passing the [WM_WINDOWPOSCHANGING](#) message to the [DefWindowProc](#) function, an application can override these defaults.

While this message is being processed, modifying any of the values in [WINDOWPOS](#) affects the window's new size, position, or place in the Z order. An application can

prevent changes to the window by setting or clearing the appropriate bits in the **flags** member of **WINDOWPOS**.

Requirements

Requirement	Value
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

Reference

[DefWindowProc](#)

[EndDeferWindowPos](#)

[SetWindowPos](#)

[WINDOWPOS](#)

[WM_GETMINMAXINFO](#)

[WM_MOVE](#)

[WM_SIZE](#)

[WM_WINDOWPOSCHANGED](#)

Conceptual

[Windows](#)

Feedback

Was this page helpful?

 Yes

 No

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

Window Structures

Article • 04/27/2021

- [ALTTABINFO](#)
- [CHANGEFILTERSTRUCT](#)
- [CLIENTCREATESTRUCT](#)
- [CREATESTRUCT](#)
- [GUITHREADINFO](#)
- [MINMAXINFO](#)
- [NCCALCSIZE_PARAMS](#)
- [STYLESTRUCT](#)
- [TITLEBARINFO](#)
- [TITLEBARINFOEX](#)
- [UPDATELAYEREDWINDOWINFO](#)
- [WINDOWINFO](#)
- [WINDOWPLACEMENT](#)
- [WINDOWPOS](#)

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](#)

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](#)

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](#)

CREATESTRUCTA structure (winuser.h)

Article 07/27/2022

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? [!\[\]\(a780acb6d486bc3d1a76c177d67e4f60_img.jpg\) Yes](#) [!\[\]\(494a9cc793f1d097304e8e22edb0bd80_img.jpg\) No](#)

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](#)

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

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](#)

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](#)

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](#)

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](#)

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
<code>WPF_ASYNCWINDOWPLACEMENT</code> 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](#)

Window Classes (Windows and Messages)

Article • 08/19/2021

This topic describes the types of window classes, how the system locates them, and the elements that define the default behavior of windows that belong to them.

A window class is a set of attributes that the system uses as a template to create a window. Every window is a member of a window class. All window classes are process specific.

In This Section

Name	Description
About Window Classes	Discusses window classes. Each window class has an associated window procedure shared by all windows of the same class. The window procedure processes messages for all windows of that class and therefore controls their behavior and appearance.
Using Window Classes	Demonstrates how to register a local window and use it to create a main window.
Window Class Reference	Contains the API reference.

Window Class Functions

Name	Description
GetClassInfoEx	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.
GetClassLong	Retrieves the specified 32-bit (long) value from the WNDCLASSEX structure associated with the specified window.
GetClassLongPtr	Retrieves the specified value from the WNDCLASSEX structure associated with the specified window.
GetClassName	Retrieves the name of the class to which the specified window belongs.

Name	Description
GetWindowLong	Retrieves information about the specified window. The function also retrieves the 32-bit (long) value at the specified offset into the extra window memory.
GetWindowLongPtr	Retrieves information about the specified window. The function also retrieves the value at a specified offset into the extra window memory.
RegisterClass	Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function.
RegisterClassEx	Registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function.
SetClassLongPtr	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.
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.
SetWindowLong	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.
SetWindowLongPtr	Changes an attribute of the specified window. The function also sets a value at the specified offset in the extra window memory.
UnregisterClass	Unregisters a window class, freeing the memory required for the class.

The following functions are obsolete.

Name	Description
GetClassInfo	<p>Retrieves information about a window class.</p> <p>[!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.</p>

Name	Description
GetClassWord	<p>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.</p> <p>[!Note] This function is deprecated for any use other than <i>nIndex</i> set to GCW_ATOM. The function is provided only for compatibility with 16-bit versions of Windows. Applications should use the GetClassLong function.</p>
SetClassLong	<p>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.</p> <p>[!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.</p>

Window Class Structures

Name	Description
WNDCLASS	<p>Contains the window class attributes that are registered by the RegisterClass function.</p> <p>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.</p>
WNDCLASSEX	<p>Contains window class information. It is used with the RegisterClassEx and GetClassInfoEx functions.</p> <p>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.</p>

Feedback

Was this page helpful?

 Yes

 No

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

Window Classes Overviews

Article • 04/27/2021

- [About Window Classes](#)
- [Using Window Classes](#)

Feedback

Was this page helpful?

 Yes

 No

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

About Window Classes

Article • 01/07/2021

Each window class has an associated window procedure shared by all windows of the same class. The window procedure processes messages for all windows of that class and therefore controls their behavior and appearance. For more information, see [Window Procedures](#).

A process must register a window class before it can create a window of that class. Registering a window class associates a window procedure, class styles, and other class attributes with a class name. When a process specifies a class name in the [CreateWindow](#) or [CreateWindowEx](#) function, the system creates a window with the window procedure, styles, and other attributes associated with that class name.

This section discusses the following topics.

- [Types of Window Classes](#)
 - [System Classes](#)
 - [Application Global Classes](#)
 - [Application Local Classes](#)
- [How the System Locates a Window Class](#)
- [Registering a Window Class](#)
- [Elements of a Window Class](#)
 - [Class Name](#)
 - [Window Procedure Address](#)
 - [Instance Handle](#)
 - [Class Cursor](#)
 - [Class Icons](#)
 - [Class Background Brush](#)
 - [Class Menu](#)
 - [Class Styles](#)
 - [Extra Class Memory](#)
 - [Extra Window Memory](#)

Types of Window Classes

There are three types of window classes:

- [System Classes](#)
- [Application Global Classes](#)
- [Application Local Classes](#)

These types differ in scope and in when and how they are registered and destroyed.

System Classes

A system class is a window class registered by the system. Many system classes are available for all processes to use, while others are used only internally by the system. Because the system registers these classes, a process cannot destroy them.

The system registers the system classes for a process the first time one of its threads calls a User or a Windows Graphics Device Interface (GDI) function.

Each application receives its own copy of the system classes. All 16-bit Windows-based applications in the same VDM share system classes, just as they do on 16-bit Windows.

The following table describes the system classes that are available for use by all processes.

Class	Description
Button	The class for a button.
ComboBox	The class for a combo box.
Edit	The class for an edit control.
ListBox	The class for a list box.
MDIClient	The class for an MDI client window.
ScrollBar	The class for a scroll bar.
Static	The class for a static control.

The following table describes the system classes that are available only for use by the system. They are listed here for completeness sake.

Class	Description
ComboLBox	The class for the list box contained in a combo box.
DDEMLEvent	The class for Dynamic Data Exchange Management Library (DDEML) events.
Message	The class for a message-only window.
#32768	The class for a menu.

Class	Description
#32769	The class for the desktop window.
#32770	The class for a dialog box.
#32771	The class for the task switch window.
#32772	The class for icon titles.

Application Global Classes

An [application global class](#) is a window class registered by an executable or DLL that is available to all other modules in the process. For example, your .dll can call the [RegisterClassEx](#) function to register a window class that defines a custom control as an application global class so that a process that loads the .dll can create instances of the custom control.

To create a class that can be used in every process, create the window class in a .dll and load the .dll in every process. To load the .dll in every process, add its name to the [AppInit_DLLs](#) value in following registry key:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows`

Whenever a process starts, the system loads the specified .dll in the context of the newly started process before calling its entry-point function. The .dll must register the class during its initialization procedure and must specify the [CS_GLOBALCLASS](#) style. For more information, see [Class Styles](#).

To remove an application global class and free the storage associated with it, use the [UnregisterClass](#) function.

Application Local Classes

An [application local class](#) is any window class that an executable or .dll registers for its exclusive use. Although you can register any number of local classes, it is typical to register only one. This window class supports the window procedure of the application's main window.

The system destroys a local class when the module that registered it closes. An application can also use the [UnregisterClass](#) function to remove a local class and free the storage associated with it.

How the System Locates a Window Class

The system maintains a list of structures for each of the three types of window classes. When an application calls the [CreateWindow](#) or [CreateWindowEx](#) function to create a window with a specified class, the system uses the following procedure to locate the class.

1. Search the list of application local classes for a class with the specified name whose instance handle matches the module's instance handle. (Several modules can use the same name to register local classes in the same process.)
2. If the name is not in the application local class list, search the list of application global classes.
3. If the name is not in the application global class list, search the list of system classes.

All windows created by the application use this procedure, including windows created by the system on the application's behalf, such as dialog boxes. It is possible to override system classes without affecting other applications. That is, an application can register an application local class having the same name as a system class. This replaces the system class in the context of the application but does not prevent other applications from using the system class.

Registering a Window Class

A window class defines the attributes of a window, such as its style, icon, cursor, menu, and window procedure. The first step in registering a window class is to fill in a [WNDCLASSEX](#) structure with the window class information. For more information, see [Elements of a Window Class](#). Next, pass the structure to the [RegisterClassEx](#) function. For more information, see [Using Window Classes](#).

To register an application global class, specify the CS_GLOBALCLASS style in the **style** member of the [WNDCLASSEX](#) structure. When registering an application local class, do not specify the CS_GLOBALCLASS style.

If you register the window class using the ANSI version of [RegisterClassEx](#), [RegisterClassExA](#), the application requests that the system pass text parameters of messages to the windows of the created class using the ANSI character set; if you register the class using the Unicode version of [RegisterClassEx](#), [RegisterClassExW](#), the application requests that the system pass text parameters of messages to the windows of the created class using the Unicode character set. 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](#).

The executable or DLL that registered the class is the owner of the class. The system determines class ownership from the **hInstance** member of the [WNDCLASSEX](#) structure passed to the [RegisterClassEx](#) function when the class is registered. For DLLs, the **hInstance** member must be the handle to the .dll instance.

The class is not destroyed when the .dll that owns it is unloaded. Therefore, if the system calls the window procedure for a window of that class, it will cause an access violation, because the .dll containing the window procedure is no longer in memory. The process must destroy all windows using the class before the .dll is unloaded and call the [UnregisterClass](#) function.

Elements of a Window Class

The elements of a window class define the default behavior of windows belonging to the class. The application that registers a window class assigns elements to the class by setting appropriate members in a [WNDCLASSEX](#) structure and passing the structure to the [RegisterClassEx](#) function. The [GetClassInfoEx](#) and [GetClassLong](#) functions retrieve information about a given window class. The [SetClassLong](#) function changes elements of a local or global class that the application has already registered.

Although a complete window class consists of many elements, the system requires only that an application supply a class name, the window-procedure address, and an instance handle. Use the other elements to define default attributes for windows of the class, such as the shape of the cursor and the content of the menu for the window. You must initialize any unused members of the [WNDCLASSEX](#) structure to zero or **NULL**. The window class elements are as shown in the following table.

Element	Purpose
Class Name	Distinguishes the class from other registered classes.
Window Procedure Address	Pointer to the function that processes all messages sent to windows in the class and defines the behavior of the window.
Instance Handle	Identifies the application or .dll that registered the class.
Class Cursor	Defines the mouse cursor that the system displays for a window of the class.
Class Icons	Defines the large icon and the small icon.

Element	Purpose
Class Background Brush	Defines the color and pattern that fill the client area when the window is opened or painted.
Class Menu	Specifies the default menu for windows that do not explicitly define a menu.
Class Styles	Defines how to update the window after moving or resizing it, how to process double-clicks of the mouse, how to allocate space for the device context, and other aspects of the window.
Extra Class Memory	Specifies the amount of extra memory, in bytes, that the system should reserve for the class. All windows in the class share the extra memory and can use it for any application-defined purpose. The system initializes this memory to zero.
Extra Window Memory	Specifies the amount of extra memory, in bytes, that the system should reserve for each window belonging to the class. The extra memory can be used for any application-defined purpose. The system initializes this memory to zero.

Class Name

Every window class needs a [Class Name](#) to distinguish one class from another. Assign a class name by setting the `lpszClassName` member of the [WNDCLASSEX](#) structure to the address of a null-terminated string that specifies the name. Because window classes are process specific, window class names need to be unique only within the same process. Also, because class names occupy space in the system's private atom table, you should keep class name strings as short as possible.

The [GetClassName](#) function retrieves the name of the class to which a given window belongs.

Window Procedure Address

Every class needs a window-procedure address to define the entry point of the window procedure used to process all messages for windows in the class. The system passes messages to the procedure when it requires the window to carry out tasks, such as painting its client area or responding to input from the user. A process assigns a window procedure to a class by copying its address to the `lpfnWndProc` member of the [WNDCLASSEX](#) structure. For more information, see [Window Procedures](#).

Instance Handle

Every window class requires an instance handle to identify the application or .dll that registered the class. The system requires instance handles to keep track of all of modules. The system assigns a handle to each copy of a running executable or .dll.

The system passes an instance handle to the entry-point function of each executable (see [WinMain](#)) and .dll (see [DlIMain](#)). The executable or .dll assigns this instance handle to the class by copying it to the **hInstance** member of the [WNDCLASSEX](#) structure.

Class Cursor

The *class cursor* defines the shape of the cursor when it is in the client area of a window in the class. The system automatically sets the cursor to the given shape when the cursor enters the window's client area and ensures it keeps that shape while it remains in the client area. To assign a cursor shape to a window class, load a predefined cursor shape by using the [LoadCursor](#) function and then assign the returned cursor handle to the **hCursor** member of the [WNDCLASSEX](#) structure. Alternatively, provide a custom cursor resource and use the [LoadCursor](#) function to load it from the application's resources.

The system does not require a class cursor. If an application sets the **hCursor** member of the [WNDCLASSEX](#) structure to **NULL**, no class cursor is defined. The system assumes the window sets the cursor shape each time the cursor moves into the window. A window can set the cursor shape by calling the [SetCursor](#) function whenever the window receives the [WM_MOUSEMOVE](#) message. For more information about cursors, see [Cursors](#).

Class Icons

A *class icon* is a picture that the system uses to represent a window of a particular class. An application can have two class icons—one large and one small. The system displays a window's *large class icon* in the task-switch window that appears when the user presses ALT+TAB, and in the large icon views of the task bar and explorer. The *small class icon* appears in a window's title bar and in the small icon views of the task bar and explorer.

To assign a large and small icon to a window class, specify the handles of the icons in the **hIcon** and **hIconSm** members of the [WNDCLASSEX](#) structure. The icon dimensions must conform to required dimensions for large and small class icons. For a large class icon, you can determine the required dimensions by specifying the **SM_CXICON** and **SM_CYICON** values in a call to the [GetSystemMetrics](#) function. For a small class icon, specify the **SM_CXSMICON** and **SM_CYSMICON** values. For information, see [Icons](#).

If an application sets the **hIcon** and **hIconSm** members of the [WNDCLASSEX](#) structure to **NULL**, the system uses the default application icon as the large and small class icons

for the window class. If you specify a large class icon but not a small one, the system creates a small class icon based on the large one. However, if you specify a small class icon but not a large one, the system uses the default application icon as the large class icon and the specified icon as the small class icon.

You can override the large or small class icon for a particular window by using the [WM_SETICON](#) message. You can retrieve the current large or small class icon by using the [WM_GETICON](#) message.

Class Background Brush

A *class background brush* prepares the client area of a window for subsequent drawing by the application. The system uses the brush to fill the client area with a solid color or pattern, thereby removing all previous images from that location whether they belong to the window or not. The system notifies a window that its background should be painted by sending the [WM_ERASEBKGND](#) message to the window. For more information, see [Brushes](#).

To assign a background brush to a class, create a brush by using the appropriate GDI functions and assign the returned brush handle to the **hbrBackground** member of the [WNDCLASSEX](#) structure.

Instead of creating a brush, an application can set the **hbrBackground** member to one of the standard system color values. For a list of the standard system color values, see [SetSysColors](#).

To use a standard system color, the application must increase the background-color value by one. For example, **COLOR_BACKGROUND + 1** is the system background color. Alternatively, you can use the [GetSysColorBrush](#) function to retrieve a handle to a brush that corresponds to a standard system color, and then specify the handle in the **hbrBackground** member of the [WNDCLASSEX](#) structure.

The system does not require that a window class have a class background brush. If this parameter is set to **NULL**, the window must paint its own background whenever it receives the [WM_ERASEBKGND](#) message.

Class Menu

A *class menu* defines the default menu to be used by the windows in the class if no explicit menu is given when the windows are created. A menu is a list of commands from which a user can choose actions for the application to carry out.

You can assign a menu to a class by setting the **IpszMenuName** member of the **WNDCLASSEX** structure to the address of a null-terminated string that specifies the resource name of the menu. The menu is assumed to be a resource in the given application. The system automatically loads the menu when it is needed. If the menu resource is identified by an integer and not by a name, the application can set the **IpszMenuName** member to that integer by applying the **MAKEINTRESOURCE** macro before assigning the value.

The system does not require a class menu. If an application sets the **IpszMenuName** member of the **WNDCLASSEX** structure to **NULL**, windows in the class have no menu bars. Even if no class menu is given, an application can still define a menu bar for a window when it creates the window.

If a menu is given for a class and a child window of that class is created, the menu is ignored. For more information, see [Menus](#).

Class Styles

The class styles define additional elements of the window class. Two or more styles can be combined by using the bitwise OR (|) operator. To assign a style to a window class, assign the style to the **style** member of the **WNDCLASSEX** structure. For a list of class styles, see [Window Class Styles](#).

Classes and Device Contexts

A *device context* is a special set of values that applications use for drawing in the client area of their windows. The system requires a device context for each window on the display but allows some flexibility in how the system stores and treats that device context.

If no device-context style is explicitly given, the system assumes each window uses a device context retrieved from a pool of contexts maintained by the system. In such cases, each window must retrieve and initialize the device context before painting and free it after painting.

To avoid retrieving a device context each time it needs to paint inside a window, an application can specify the **CS_OWNDC** style for the window class. This class style directs the system to create a private device context—that is, to allocate a unique device context for each window in the class. The application need only retrieve the context once and then use it for all subsequent painting.

Extra Class Memory

The system maintains a [WNDCLASSEX](#) structure internally for each window class in the system. When an application registers a window class, it can direct the system to allocate and append a number of additional bytes of memory to the end of the [WNDCLASSEX](#) structure. This memory is called *extra class memory* and is shared by all windows belonging to the class. Use the extra class memory to store any information pertaining to the class.

Because extra memory is allocated from the system's local heap, an application should use extra class memory sparingly. The [RegisterClassEx](#) function fails if the amount of extra class memory requested is greater than 40 bytes. If an application requires more than 40 bytes, it should allocate its own memory and store a pointer to the memory in the extra class memory.

The [SetClassWord](#) and [SetClassLong](#) functions copy a value to the extra class memory. To retrieve a value from the extra class memory, use the [GetClassWord](#) and [GetClassLong](#) functions. The **cbClsExtra** member of the [WNDCLASSEX](#) structure specifies the amount of extra class memory to allocate. An application that does not use extra class memory must initialize the **cbClsExtra** member to zero.

Extra Window Memory

The system maintains an internal data structure for each window. When registering a window class, an application can specify a number of additional bytes of memory, called *extra window memory*. When creating a window of the class, the system allocates and appends the specified amount of extra window memory to the end of the window's structure. An application can use this memory to store window-specific data.

Because extra memory is allocated from the system's local heap, an application should use extra window memory sparingly. The [RegisterClassEx](#) function fails if the amount of extra window memory requested is greater than 40 bytes. If an application requires more than 40 bytes, it should allocate its own memory and store a pointer to the memory in the extra window memory.

The [SetWindowLong](#) function copies a value to the extra memory. The [GetWindowLong](#) function retrieves a value from the extra memory. The **cbWndExtra** member of the [WNDCLASSEX](#) structure specifies the amount of extra window memory to allocate. An application that does not use the memory must initialize **cbWndExtra** to zero.

Feedback

Was this page helpful?

 Yes

 No

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

Using Window Classes

Article • 01/07/2021

This topic has a code example that shows how to register a local window and use it to create a main window.

Each process must register its own window classes. To register an application local class, use the [RegisterClassEx](#) function. You must define the window procedure, fill the members of the [WNDCLASSEX](#) structure, and then pass a pointer to the structure to the [RegisterClassEx](#) function.

The following example shows how to register a local window class and use it to create a main window.

```
#include <windows.h>

// Global variable

HINSTANCE hinst;

// Function prototypes.

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
InitApplication(HINSTANCE);
InitInstance(HINSTANCE, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

// Application entry point.

int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

    if (!InitApplication(hinstance))
        return FALSE;

    if (!InitInstance(hinstance, nCmdShow))
        return FALSE;

    BOOL fGotMessage;
    while ((fGotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 &&
fGotMessage != -1)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
```

```

        UNREFERENCED_PARAMETER(1pCmdLine);
    }

BOOL InitApplication(HINSTANCE hinstance)
{
    WNDCLASSEX wcx;

    // Fill in the window class structure with parameters
    // that describe the main window.

    wcx.cbSize = sizeof(wcx);           // size of structure
    wcx.style = CS_HREDRAW |           // redraw if size changes
                CS_VREDRAW;
    wcx.lpfnWndProc = MainWndProc;     // points to window procedure
    wcx.cbClsExtra = 0;                // no extra class memory
    wcx.cbWndExtra = 0;                // no extra window memory
    wcx.hInstance = hinstance;         // handle to instance
    wcx.hIcon = LoadIcon(NULL,
                         IDI_APPLICATION);          // predefined app. icon
    wcx.hCursor = LoadCursor(NULL,
                            IDC_ARROW);            // predefined arrow
    wcx.hbrBackground = GetStockObject(
        WHITE_BRUSH);                  // white background brush
    wcx.lpszMenuName = "MainMenu";    // name of menu resource
    wcx.lpszClassName = "MainWClass"; // name of window class
    wcx.hIconSm = LoadImage(hinstance, // small class icon
                           MAKEINTRESOURCE(5),
                           IMAGE_ICON,
                           GetSystemMetrics(SM_CXSMICON),
                           GetSystemMetrics(SM_CYSMICON),
                           LR_DEFAULTCOLOR);

    // Register the window class.

    return RegisterClassEx(&wcx);
}

BOOL InitInstance(HINSTANCE hinstance, int nCmdShow)
{
    HWND hwnd;

    // Save the application-instance handle.

    hinst = hinstance;

    // Create the main window.

    hwnd = CreateWindow(
        "MainWClass",           // name of window class
        "Sample",               // title-bar string
        WS_OVERLAPPEDWINDOW,   // top-level window
        CW_USEDEFAULT,          // default horizontal position
        CW_USEDEFAULT,          // default vertical position
        CW_USEDEFAULT,          // default width
        CW_USEDEFAULT);         // default height
}

```

```
(HWND) NULL,           // no owner window
(HMENU) NULL,          // use class menu
hinstance,             // handle to application instance
(LPVOID) NULL);        // no window-creation data

if (!hwnd)
    return FALSE;

// Show the window and send a WM_PAINT message to the window
// procedure.

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
return TRUE;

}
```

Registering an application global class is similar to registering an application local class, except that the **style** member of the [WNDCLASSEX](#) structure must specify the **CS_GLOBALCLASS** style.

Feedback

Was this page helpful?

 Yes

 No

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

Window Class Reference

Article • 04/27/2021

- [Window Class Functions](#)
- [Window Class Structures](#)

Feedback

Was this page helpful?

 Yes

 No

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

Window Class Functions

Article • 04/27/2021

- [GetClassInfo](#)
- [GetClassInfoEx](#)
- [GetClassLong](#)
- [GetClassLongPtr](#)
- [GetClassName](#)
- [GetClassWord](#)
- [GetWindowLong](#)
- [GetWindowLongPtr](#)
- [RegisterClass](#)
- [RegisterClassEx](#)
- [SetClassLong](#)
- [SetClassLongPtr](#)
- [SetClassWord](#)
- [SetWindowLong](#)
- [SetWindowLongPtr](#)
- [UnregisterClass](#)

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](#)

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](#)

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](#)

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](#)

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?

 Yes

 No

[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](#)

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](#)

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](#)

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

SetWindowLongA 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 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](#)

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

Window Class Structures

Article • 04/27/2021

- [WNDCLASS](#)
 - [WNDCLASSEX](#)
-

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](#)

Window Class Styles

Article • 04/07/2022

The class styles define additional elements of the window class. Two or more styles can be combined by using the bitwise OR (|) operator. To assign a style to a window class, assign the style to the **style** member of the [WNDCLASSEX](#) structure.

Example

C++

```
WNDCLASS wc = {};
wc.lpfWndProc = s_DropDownWndProc;
wc.cbWndExtra = sizeof(CTipACDialog *);
wc.hInstance = g_hInstance;
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wc.style = CS_SAVEBITS | CS_DROPSHADOW;
wc.lpszClassName = s_wzClassName;
RegisterClass(&wc);
```

Example from [Windows Classic Samples](#) on GitHub.

Constants

The following are the window class styles.

Constant/value	Description
CS_BYTEALIGNCLIENT 0x1000	Aligns the window's client area on a byte boundary (in the x direction). This style affects the width of the window and its horizontal placement on the display.
CS_BYTEALIGNWINDOW 0x2000	Aligns the window on a byte boundary (in the x direction). This style affects the width of the window and its horizontal placement on the display.
CS_CLASSDC 0x0040	Allocates one device context to be shared by all windows in the class. Because window classes are process specific, it is possible for multiple threads of an application to create a window of the same class. It is also possible for the threads to attempt to use the device context simultaneously. When this happens, the system allows only one thread to successfully finish its drawing operation.

Constant/value	Description
CS_DBCLKS 0x0008	Sends a double-click message to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class.
CS_DROPSHADOW 0x00020000	Enables the drop shadow effect on a window. The effect is turned on and off through SPI_SETDROPSHADOW . Typically, this is enabled for small, short-lived windows such as menus to emphasize their Z-order relationship to other windows. Windows created from a class with this style must be top-level windows; they may not be child windows.
CS_GLOBALCLASS 0x4000	Indicates that the window class is an application global class. For more information, see the "Application Global Classes" section of About Window Classes .
CS_HREDRAW 0x0002	Redraws the entire window if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE 0x0200	Disables Close on the window menu.
CS_OWNDC 0x0020	Allocates a unique device context for each window in the class.
CS_PARENTDC 0x0080	Sets the clipping rectangle of the child window to that of the parent window so that the child can draw on the parent. A window with the CS_PARENTDC style bit receives a regular device context from the system's cache of device contexts. It does not give the child the parent's device context or device context settings. Specifying CS_PARENTDC enhances an application's performance.
CS_SAVEBITS 0x0800	Saves, as a bitmap, the portion of the screen image obscured by a window of this class. When the window is removed, the system uses the saved bitmap to restore the screen image, including other windows that were obscured. Therefore, the system does not send WM_PAINT messages to windows that were obscured if the memory used by the bitmap has not been discarded and if other screen actions have not invalidated the stored image. This style is useful for small windows (for example, menus or dialog boxes) that are displayed briefly and then removed before other screen activity takes place. This style increases the time required to display the window, because the system must first allocate memory to store the bitmap.
CS_VREDRAW 0x0001	Redraws the entire window if a movement or size adjustment changes the height of the client area.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

Feedback

Was this page helpful?

 Yes

 No

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

Window Procedures

Article • 01/07/2021

Every window has an associated window procedure — a function that processes all messages sent or posted to all windows of the class. All aspects of a window's appearance and behavior depend on the window procedure's response to these messages.

In This Section

Name	Description
About Window Procedures	Discusses window procedures. Each window is a member of a particular window class. The window class determines the default window procedure that an individual window uses to process its messages.
Using Window Procedures	Covers how to perform the following tasks associated with window procedures.
Window Procedure Reference	Contains the API reference.

Functions

Name	Description
CallWindowProc	Passes message information to the specified window procedure.
DefWindowProc	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.
WindowProc	An application-defined function that processes messages sent to a window. The WNDPROC type defines a pointer to this callback function. WindowProc is a placeholder for the application-defined function name.

Feedback

Was this page helpful?

 Yes

 No

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

Window Procedure Overviews

Article • 04/27/2021

- [About Window Procedures](#)
- [Using Window Procedures](#)

Feedback

Was this page helpful?

 Yes

 No

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

About Window Procedures

Article • 01/07/2021

Each window is a member of a particular window class. The window class determines the default window procedure that an individual window uses to process its messages. All windows belonging to the same class use the same default window procedure. For example, the system defines a window procedure for the combo box class (**COMBOBOX**); all combo boxes then use that window procedure.

An application typically registers at least one new window class and its associated window procedure. After registering a class, the application can create many windows of that class, all of which use the same window procedure. Because this means several sources could simultaneously call the same piece of code, you must be careful when modifying shared resources from a window procedure. For more information, see [Window Classes](#).

Window procedures for dialog boxes (called dialog box procedures) have a similar structure and function as regular window procedures. All points referring to window procedures in this section also apply to dialog box procedures. For more information, see [Dialog Boxes](#).

This section discusses the following topics.

- [Structure of a Window Procedure](#)
- [Default Window Procedure](#)
- [Window Procedure Subclassing](#)
 - [Instance Subclassing](#)
 - [Global Subclassing](#)
- [Window Procedure Superclassing](#)

Structure of a Window Procedure

A window procedure is a function that has four parameters and returns a signed value. The parameters consist of a window handle, a **UINT** message identifier, and two message parameters declared with the **WPARAM** and **LPARAM** data types. For more information, see [WindowProc](#).

Message parameters often contain information in both their low-order and high-order words. There are several macros an application can use to extract information from the message parameters. The **LOWORD** macro, for example, extracts the low-order word

(bits 0 through 15) from a message parameter. Other macros include [HIWORD](#), [LOBYTE](#), and [HIBYTE](#).

The interpretation of the return value depends on the particular message. Consult the description of each message to determine the appropriate return value.

Because it is possible to call a window procedure recursively, it is important to minimize the number of local variables that it uses. When processing individual messages, an application should call functions outside the window procedure to avoid excessive use of local variables, possibly causing the stack to overflow during deep recursion.

Default Window Procedure

The default window procedure function, [DefWindowProc](#) defines certain fundamental behavior shared by all windows. The default window procedure provides the minimal functionality for a window. An application-defined window procedure should pass any messages that it does not process to the [DefWindowProc](#) function for default processing.

Window Procedure Subclassing

When an application creates a window, the system allocates a block of memory for storing information specific to the window, including the address of the window procedure that processes messages for the window. When the system needs to pass a message to the window, it searches the window-specific information for the address of the window procedure and passes the message to that procedure.

Subclassing is a technique that allows an application to intercept and process messages sent or posted to a particular window before the window has a chance to process them. By subclassing a window, an application can augment, modify, or monitor the behavior of the window. An application can subclass a window belonging to a system global class, such as an edit control or a list box. For example, an application could subclass an edit control to prevent the control from accepting certain characters. However, you cannot subclass a window or class that belongs to another application. All subclassing must be performed within the same process.

An application subclasses a window by replacing the address of the window's original window procedure with the address of a new window procedure, called the *subclass procedure*. Thereafter, the subclass procedure receives any messages sent or posted to the window.

The subclass procedure can take three actions upon receiving a message: it can pass the message to the original window procedure, modify the message and pass it to the original window procedure, or process the message and not pass it to the original window procedure. If the subclass procedure processes a message, it can do so before, after, or both before and after it passes the message to the original window procedure.

The system provides two types of subclassing: [instance](#) and [global](#). In *instance subclassing*, an application replaces the window procedure address of a single instance of a window. An application must use instance subclassing to subclass an existing window. In *global subclassing*, an application replaces the address of the window procedure in the [WNDCLASS](#) structure of a window class. All subsequent windows created with the class have the address of the subclass procedure, but existing windows of the class are not affected.

Instance Subclassing

An application subclasses an instance of a window by using the [SetWindowLong](#) function. The application passes the **GWL_WNDPROC** flag, the handle to the window to subclass, and the address of the subclass procedure to [SetWindowLong](#). The subclass procedure can reside in either the application's executable or a DLL.

[SetWindowLong](#) returns the address of the window's original window procedure. The application must save the address, using it in subsequent calls to the [CallWindowProc](#) function, to pass intercepted messages to the original window procedure. The application must also have the original window procedure address to remove the subclass from the window. To remove the subclass, the application calls [SetWindowLong](#) again, passing the address of the original window procedure with the **GWL_WNDPROC** flag and the handle to the window.

The system owns the system global classes, and aspects of the controls might change from one version of the system to the next. If the application must subclass a window that belongs to a system global class, the developer may need to update the application when a new version of the system is released.

Because instance subclassing occurs after a window is created, you cannot add any extra bytes to the window. Applications that subclass a window should use the window's property list to store any data needed for an instance of the subclassed window. For more information, see [Window Properties](#).

When an application subclasses a subclassed window, it must remove the subclasses in the reverse order they were performed. If the removal order is not reversed, an unrecoverable system error may occur.

Global Subclassing

To globally subclass a window class, the application must have a handle to a window of the class. The application also needs the handle to remove the subclass. To get the handle, an application typically creates a hidden window of the class to be subclassed. After obtaining the handle, the application calls the [SetClassLong](#) function, specifying the handle, the **GCL_WNDPROC** flag, and the address of the subclass procedure. [SetClassLong](#) returns the address of the original window procedure for the class.

The original window procedure address is used in global subclassing in the same way it is used in instance subclassing. The subclass procedure passes messages to the original window procedure by calling [CallWindowProc](#). The application removes the subclass from the window class by calling [SetClassLong](#) again, specifying the address of the original window procedure, the **GCL_WNDPROC** flag, and the handle to a window of the class being subclassed. An application that globally subclasses a control class must remove the subclass when the application terminates; otherwise, an unrecoverable system error may occur.

Global subclassing has the same limitations as instance subclassing, plus some additional restrictions. An application should not use the extra bytes for either the class or the window instance without knowing exactly how the original window procedure uses them. If the application must associate data with a window, it should use window properties.

Window Procedure Superclassing

Superclassing is a technique that allows an application to create a new window class with the basic functionality of the existing class, plus enhancements provided by the application. A superclass is based on an existing window class called the *base class*. Frequently, the base class is a system global window class such as an edit control, but it can be any window class.

A superclass has its own window procedure, called the superclass procedure. The *superclass procedure* can take three actions upon receiving a message: It can pass the message to the original window procedure, modify the message and pass it to the original window procedure, or process the message and not pass it to the original window procedure. If the superclass procedure processes a message, it can do so before, after, or both before and after it passes the message to the original window procedure.

Unlike a subclass procedure, a superclass procedure can process window creation messages ([WM_NCCREATE](#), [WM_CREATE](#), and so on), but it must also pass them to the

original base-class window procedure so that the base-class window procedure can perform its initialization procedure.

To superclass a window class, an application first calls the [GetClassInfo](#) function to retrieve information about the base class. [GetClassInfo](#) fills a [WNDCLASS](#) structure with the values from the [WNDCLASS](#) structure of the base class. Next, the application copies its own instance handle into the [hInstance](#) member of the [WNDCLASS](#) structure and copies the name of the superclass into the [lpszClassName](#) member. If the base class has a menu, the application must provide a new menu with the same menu identifiers and copy the menu name into the [lpszMenuName](#) member. If the superclass procedure processes the [WM_COMMAND](#) message and does not pass it to the window procedure of the base class, the menu need not have corresponding identifiers. [GetClassInfo](#) does not return the [lpszMenuName](#), [lpszClassName](#), or [hInstance](#) member of the [WNDCLASS](#) structure.

An application must also set the [lpfnWndProc](#) member of the [WNDCLASS](#) structure. The [GetClassInfo](#) function fills this member with the address of the original window procedure for the class. The application must save this address, to pass messages to the original window procedure, and then copy the address of the superclass procedure into the [lpfnWndProc](#) member. The application can, if necessary, modify any other members of the [WNDCLASS](#) structure. After it fills the [WNDCLASS](#) structure, the application registers the superclass by passing the address of the structure to the [RegisterClass](#) function. The superclass can then be used to create windows.

Because superclassing registers a new window class, an application can add to both the extra class bytes and the extra window bytes. The superclass must not use the original extra bytes for the base class or the window for the same reasons that an instance subclass or a global subclass should not use them. Also, if the application adds extra bytes for its use to either the class or the window instance, it must reference the extra bytes relative to the number of extra bytes used by the original base class. Because the number of bytes used by the base class may vary from one version of the base class to the next, the starting offset for the superclass's own extra bytes may also vary from one version of the base class to the next.

Feedback

Was this page helpful?



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

Using Window Procedures

Article • 01/07/2021

This section explains how to perform the following tasks associated with window procedures.

- [Designing a Window Procedure](#)
- [Associating a Window Procedure with a Window Class](#)
- [Subclassing a Window](#)

Designing a Window Procedure

The following example shows the structure of a typical window procedure. The window procedure uses the message argument in a **switch** statement with individual messages handled by separate **case** statements. Notice that each case returns a specific value for each message. For messages that it does not process, the window procedure calls the [DefWindowProc](#) function.

```
LRESULT CALLBACK MainWndProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // message identifier
    WPARAM wParam,      // first message parameter
    LPARAM lParam)      // second message parameter
{
    switch (uMsg)
    {
        case WM_CREATE:
            // Initialize the window.
            return 0;

        case WM_PAINT:
            // Paint the window's client area.
            return 0;

        case WM_SIZE:
            // Set the size and position of the window.
            return 0;

        case WM_DESTROY:
            // Clean up window-specific data objects.
            return 0;

        //
        // Process other messages.
    }
}
```

```
//  
  
default:  
    return DefWindowProc(hwnd, uMsg, wParam, lParam);  
}  
return 0;  
}
```

The [WM_NCCREATE](#) message is sent just after your window is created, but if an application responds to this message by returning FALSE, [CreateWindowEx](#) function fails. The [WM_CREATE](#) message is sent after your window is already created.

The [WM_DESTROY](#) message is sent when your window is about to be destroyed. The [DestroyWindow](#) function takes care of destroying any child windows of the window being destroyed. The [WM_NCDESTROY](#) message is sent just before a window is destroyed.

At the very least, a window procedure should process the [WM_PAINT](#) message to draw itself. Typically, it should handle mouse and keyboard messages as well. Consult the descriptions of individual messages to determine whether your window procedure should handle them.

Your application can call the [DefWindowProc](#) function as part of the processing of a message. In such a case, the application can modify the message parameters before passing the message to [DefWindowProc](#), or it can continue with the default processing after performing its own operations.

A dialog box procedure receives a [WM_INITDIALOG](#) message instead of a [WM_CREATE](#) message and does not pass unprocessed messages to the [DefDlgProc](#) function. Otherwise, a dialog box procedure is exactly the same as a window procedure.

Associating a Window Procedure with a Window Class

You associate a window procedure with a window class when registering the class. You must fill a [WNDCLASS](#) structure with information about the class, and the [lPfnWndProc](#) member must specify the address of the window procedure. To register the class, pass the address of [WNDCLASS](#) structure to the [RegisterClass](#) function. After the window class has been registered, the window procedure is automatically associated with each new window created with that class.

The following example shows how to associate the window procedure in the previous example with a window class.

```

int APIENTRY WinMain(
    HINSTANCE hinstance, // handle to current instance
    HINSTANCE hinstPrev, // handle to previous instance
    LPSTR lpCmdLine, // address of command-line string
    int nCmdShow) // show-window type
{
    WNDCLASS wc;

    // Register the main window class.
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC) MainWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hinstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = "MainMenu";
    wc.lpszClassName = "MainWindowClass";

    if (!RegisterClass(&wc))
        return FALSE;

    //
    // Process other messages.
    //

}


```

Subclassing a Window

To subclass an instance of a window, call the [SetWindowLong](#) function and specify the handle to the window to subclass the GWL_WNDPROC flag and a pointer to the subclass procedure. [SetWindowLong](#) returns a pointer to the original window procedure; use this pointer to pass messages to the original procedure. The subclass window procedure must use the [CallWindowProc](#) function to call the original window procedure.

Note

To write code that is compatible with both 32-bit and 64-bit versions of Windows, use the [SetWindowLongPtr](#) function.

The following example shows how to subclass an instance of an edit control in a dialog box. The subclass window procedure enables the edit control to receive all keyboard input, including the ENTER and TAB keys, whenever the control has the input focus.

```
WNDPROC wpOrigEditProc;

LRESULT APIENTRY EditBoxProc(
    HWND hwndDlg,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    HWND hwndEdit;

    switch(uMsg)
    {
        case WM_INITDIALOG:
            // Retrieve the handle to the edit control.
            hwndEdit = GetDlgItem(hwndDlg, ID_EDIT);

            // Subclass the edit control.
            wpOrigEditProc = (WNDPROC) SetWindowLong(hwndEdit,
                GWL_WNDPROC, (LONG) EditSubclassProc);
            //
            // Continue the initialization procedure.
            //
            return TRUE;

        case WM_DESTROY:
            // Remove the subclass from the edit control.
            SetWindowLong(hwndEdit, GWL_WNDPROC,
                (LONG) wpOrigEditProc);
            //
            // Continue the cleanup procedure.
            //
            break;
    }
    return FALSE;
    UNREFERENCED_PARAMETER(lParam);
}

// Subclass procedure
LRESULT APIENTRY EditSubclassProc(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    if (uMsg == WM_GETDLGCODE)
        return DLGC_WANTALLKEYS;
```

```
    return CallWindowProc(wpOrigEditProc, hwnd, uMsg,  
        wParam, lParam);  
}
```

Feedback

Was this page helpful?

 Yes

 No

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

Window Procedure Reference

Article • 04/27/2021

- [Window Procedure Functions](#)

Feedback

Was this page helpful?

 Yes

 No

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

Window Procedure Functions

Article • 04/27/2021

In This Section

- [CallWindowProc](#)
- [DefWindowProc](#)
- [*WindowProc*](#)

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](#)

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](#)

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](#)

Messages and Message Queues

Article • 03/13/2023

This section describes messages and message queues and how to use them in your applications.

In This Section

Name	Description
About Messages and Message Queues	This section discusses Windows messages and message queues.
Using Messages and Message Queues	The following code examples demonstrate how to perform the following tasks associated with Windows messages and message queues.
Message Reference	Contains the API reference.

System-Provided Messages

For lists of the system-provided messages, see [System-Defined Messages](#).

Message Functions

Name	Description
BroadcastSystemMessage	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.
BroadcastSystemMessageEx	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.
DispatchMessage	Dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

Name	Description
GetInputState	Determines whether there are mouse-button or keyboard messages in the calling thread's message queue.
GetMessage	<p>Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval.</p> <p>Unlike GetMessage, the PeekMessage function does not wait for a message to be posted before returning.</p>
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	<p>Retrieves the cursor position for the last message retrieved by the GetMessage function.</p> <p>To determine the current position of the cursor, use the GetCursorPos function.</p>
GetMessageTime	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).
GetQueueStatus	Indicates the type of messages found in the calling thread's message queue.
InSendMessage	<p>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.</p> <p>To obtain additional information about how the message was sent, use the InSendMessageEx function.</p>
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).
PeekMessage	Dispatches incoming sent messages, checks the thread message queue for a posted message, and retrieves the message (if any exist).
PostMessage	<p>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.</p> <p>To post a message in the message queue associated with a thread, use the PostThreadMessage function.</p>

Name	Description
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.
PostThreadMessage	Posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message.
RegisterWindowMessage	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.
ReplyMessage	Replies to a message sent through the SendMessage function without returning control to the function that called SendMessage .
SendAsyncProc	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.
SendMessage	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.
SendMessageCallback	Sends the specified message to a window or windows. It calls the window procedure for the specified window and returns immediately. 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.
SendMessageTimeout	Sends the specified message to one of more windows.
SendNotifyMessage	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.

Name	Description
SetMessageExtraInfo	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.
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.
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.

Message Constants

Name	Description
OCM_BASE	Used to define private messages for use by private window classes.
WM_APP	Used to define private messages.
WM_USER	Used to define private messages for use by private window classes.

Message Structures

Name	Description
BSMINFO	Contains information about a window that denied a request from BroadcastSystemMessageEx .
MSG	Contains message information from a thread's message queue.

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

Message and Message Queue Overviews

Article • 04/27/2021

- [About Messages and Message Queues](#)
- [Using Messages and Message Queues](#)

Feedback

Was this page helpful?

 Yes

 No

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

About Messages and Message Queues

Article • 08/19/2022

Unlike MS-DOS-based applications, Windows-based applications are event-driven. They do not make explicit function calls (such as C run-time library calls) to obtain input. Instead, they wait for the system to pass input to them.

The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure, that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. For more information about window procedures, see [Window Procedures](#).

If a top-level window stops responding to messages for more than several seconds, the system considers the window to be not responding. In this case, the system hides the window 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.

This section discusses the following topics:

- [Windows Messages](#)
- [Message Types](#)
 - [System-Defined Messages](#)
 - [Application-Defined Messages](#)
- [Message Routing](#)
 - [Queued Messages](#)
 - [Nonqueued Messages](#)
- [Message Handling](#)
 - [Message Loop](#)
 - [Window Procedure](#)
- [Message Filtering](#)
- [Posting and Sending Messages](#)
 - [Posting Messages](#)
 - [Sending Messages](#)
- [Message Deadlocks](#)
- [Broadcasting Messages](#)
- [Query Messages](#)

Windows Messages

The system passes input to a window procedure in the form of a *message*. Messages are generated by both the system and applications. The system generates a message at each input event—for example, when the user types, moves the mouse, or clicks a control such as a scroll bar. The system also generates messages in response to changes in the system brought about by an application, such as when an application changes the pool of system font resources or resizes one of its windows. An application can generate messages to direct its own windows to perform tasks or to communicate with windows in other applications.

The system sends a message to a window procedure with a set of four parameters: a window handle, a message identifier, and two values called *message parameters*. The *window handle* identifies the window for which the message is intended. The system uses it to determine which window procedure should receive the message.

A *message identifier* is a named constant that identifies the purpose of a message. When a window procedure receives a message, it uses a message identifier to determine how to process the message. For example, the message identifier **WM_PAINT** tells the window procedure that the window's client area has changed and must be repainted.

Message parameters specify data or the location of data used by a window procedure when processing a message. The meaning and value of the message parameters depend on the message. A message parameter can contain an integer, packed bit flags, a pointer to a structure containing additional data, and so on. When a message does not use message parameters, they are typically set to **NULL**. A window procedure must check the message identifier to determine how to interpret the message parameters.

Message Types

This section describes the two types of messages:

- [System-Defined Messages](#)
- [Application-Defined Messages](#)

System-Defined Messages

The system sends or posts a *system-defined message* when it communicates with an application. It uses these messages to control the operations of applications and to provide input and other information for applications to process. An application can also send or post system-defined messages. Applications generally use these messages to

control the operation of control windows created by using preregistered window classes.

Each system-defined message has a unique message identifier and a corresponding symbolic constant (defined in the software development kit (SDK) header files) that states the purpose of the message. For example, the [WM_PAINT](#) constant requests that a window paint its contents.

Symbolic constants specify the category to which system-defined messages belong. The prefix of the constant identifies the type of window that can interpret and process the message. Following are the prefixes and their related message categories.

Prefix	Message category	Documentation
ABM and ABN	Application desktop toolbar	Shell Messages and Notifications
ACM and ACN	Animation control	Animation Control Messages and Animation Control Notifications
BCM, BCN, BM, and BN	Button control	Button Control Messages and Button Control Notifications
CB and CBN	ComboBox control	ComboBox Control Messages and ComboBox Control Notifications
CBEM and CBEN	ComboBoxEx control	ComboBoxEx Messages and ComboBoxEx Notifications
CCM	General control	Control Messages
CDM	Common dialog box	Common Dialog Box Messages
DFM	Default context menu	Shell Messages and Notifications
DL	Drag list box	Drag List Box Notifications
DM	Default push button control	Dialog Box Messages
DTM and DTN	Date and time picker control	Date and Time Picker Messages and Date and Time Picker Notifications
EM and EN	Edit control	Edit Control Messages , Edit Control Notifications , Rich Edit Messages , and Rich Edit Notifications

Prefix	Message category	Documentation
HDM and HDN	Header control	Header Control Messages and Header Control Notifications
HKM	Hot key control	Hot Key Control Messages
IPM and IPN	IP address control	IP Address Messages and IP Address Notifications
LB and LBN	List box control	List Box Messages and List Box Notifications
LM	SysLink control	SysLink Control Messages
LVM and LVN	List view control	List View Messages and List View Notifications
MCM and MCN	Month calendar control	Month Calendar Messages and Month Calendar Notifications
PBM	Progress bar	Progress Bar Messages
PGM and PGN	Pager control	Pager Control Messages and Pager Control Notifications
PSM and PSN	Property sheet	Property Sheet Messages and Property Sheet Notifications
RB and RBN	Rebar control	Rebar Control Messages and Rebar Control Notifications
SB and SBN	Status bar window	Status Bar Messages and Status Bar Notifications
SBM	Scrollbar control	Scroll Bar Messages
SMC	Shell menu	Shell Messages and Notifications
STM and STN	Static control	Static Control Messages and Static Control Notifications
TB and TBN	Toolbar	Toolbar Control Messages and Toolbar Control Notifications
TBM and TRBN	Trackbar control	Trackbar Control Messages and Trackbar Control Notifications
TCM and TCN	Tab control	Tab Control Messages and Tab Control Notifications
TDM and TDN	Task dialog	Task Dialog Messages and Task Dialog Notifications
TTM and TTN	Tooltip control	Tooltip Control Messages and Tooltip Control Notifications
TVM and TVN	Tree-view control	Tree View Messages and Tree View Notifications

Prefix	Message category	Documentation
UDM and UDN	Up-down control	Up-Down Messages and Up-Down Notifications
WM	General	Clipboard Messages Clipboard Notifications Common Dialog Box Notifications Cursor Notifications Data Copy Message Desktop Window Manager Messages Device Management Messages Dialog Box Notifications Dynamic Data Exchange Messages Dynamic Data Exchange Notifications Hook Notifications Keyboard Accelerator Messages Keyboard Accelerator Notifications Keyboard Input Messages Keyboard Input Notifications Menu Notifications Mouse Input Notifications Multiple Document Interface Messages Raw Input Notifications Scroll Bar Notifications Timer Notifications Window Messages Window Notifications

General window messages cover a wide range of information and requests, including messages for mouse and keyboard input, menu and dialog box input, window creation and management, and Dynamic Data Exchange (DDE).

Application-Defined Messages

An application can create messages to be used by its own windows or to communicate with windows in other processes. If an application creates its own messages, the window procedure that receives them must interpret the messages and provide appropriate processing.

Message-identifier values are used as follows:

- The system reserves message-identifier values in the range 0x0000 through 0x03FF (the value of [WM_USER](#) – 1) for system-defined messages. Applications cannot use these values for private messages.

- Values in the range 0x0400 (the value of [WM_USER](#)) through 0x7FFF are available for message identifiers for private window classes.
- If your application is marked version 4.0, you can use message-identifier values in the range 0x8000 ([WM_APP](#)) through 0xBFFF for private messages.
- The system returns a message identifier in the range 0xC000 through 0xFFFF when an application calls the [RegisterWindowMessage](#) function to register a message. The message identifier returned by this function is guaranteed to be unique throughout the system. Use of this function prevents conflicts that can arise if other applications use the same message identifier for different purposes.

Message Routing

The system uses two methods to route messages to a window procedure: posting messages to a first-in, first-out queue called a *message queue*, a system-defined memory object that temporarily stores messages, and sending messages directly to a window procedure.

A messages that is posted to a message queue is called a *queued message*. These are primarily the result of user input entered through the mouse or keyboard, such as [WM_MOUSEMOVE](#), [WM_LBUTTONDOWN](#), [WM_KEYDOWN](#), and [WM_CHAR](#) messages. Other queued messages include the timer, paint, and quit messages: [WM_TIMER](#), [WM_PAINT](#), and [WM_QUIT](#). Most other messages, which are sent directly to a window procedure, are called *nonqueued messages*.

- [Queued Messages](#)
- [Nonqueued Messages](#)

Queued Messages

The system can display any number of windows at a time. To route mouse and keyboard input to the appropriate window, the system uses message queues.

The system maintains a single system message queue and one thread-specific message queue for each GUI thread. To avoid the overhead of creating a message queue for non-GUI threads, all threads are created initially without a message queue. The system creates a thread-specific message queue only when the thread makes its first call to one of the specific user functions; no GUI function calls result in the creation of a message queue.

Whenever the user moves the mouse, clicks the mouse buttons, or types on the keyboard, the device driver for the mouse or keyboard converts the input into messages and places them in the system message queue. The system removes the messages, one

at a time, from the system message queue, examines them to determine the destination window, and then posts them to the message queue of the thread that created the destination window. A thread's message queue receives all mouse and keyboard messages for the windows created by the thread. The thread removes messages from its queue and directs the system to send them to the appropriate window procedure for processing.

With the exception of the [WM_PAINT](#) message, the [WM_TIMER](#) message, and the [WM_QUIT](#) message, the system always posts messages at the end of a message queue. This ensures that a window receives its input messages in the proper first in, first out (FIFO) sequence. The [WM_PAINT](#) message, the [WM_TIMER](#) message, and the [WM_QUIT](#) message, however, are kept in the queue and are forwarded to the window procedure only when the queue contains no other messages. In addition, multiple [WM_PAINT](#) messages for the same window are combined into a single [WM_PAINT](#) message, consolidating all invalid parts of the client area into a single area. Combining [WM_PAINT](#) messages reduces the number of times a window must redraw the contents of its client area.

The system posts a message to a thread's message queue by filling an [MSG](#) structure and then copying it to the message queue. Information in [MSG](#) includes: the handle of the window for which the message is intended, the message identifier, the two message parameters, the time the message was posted, and the mouse cursor position. A thread can post a message to its own message queue or to the queue of another thread by using the [PostMessage](#) or [PostThreadMessage](#) function.

An application can remove a message from its queue by using the [GetMessage](#) function. To examine a message without removing it from its queue, an application can use the [PeekMessage](#) function. This function fills [MSG](#) with information about the message.

After removing a message from its queue, an application can use the [DispatchMessage](#) function to direct the system to send the message to a window procedure for processing. [DispatchMessage](#) takes a pointer to [MSG](#) that was filled by a previous call to the [GetMessage](#) or [PeekMessage](#) function. [DispatchMessage](#) passes the window handle, the message identifier, and the two message parameters to the window procedure, but it does not pass the time the message was posted or mouse cursor position. An application can retrieve this information by calling the [GetMessageTime](#) and [GetMessagePos](#) functions while processing a message.

A thread can use the [WaitMessage](#) function to yield control to other threads when it has no messages in its message queue. The function suspends the thread and does not return until a new message is placed in the thread's message queue.

You can call the [SetMessageExtraInfo](#) function to associate a value with the current thread's message queue. Then call the [GetMessageExtraInfo](#) function to get the value associated with the last message retrieved by the [GetMessage](#) or [PeekMessage](#) function.

Nonqueued Messages

Nonqueued messages are sent immediately to the destination window procedure, bypassing the system message queue and thread message queue. The system typically sends nonqueued messages to notify a window of events that affect it. For example, when the user activates a new application window, the system sends the window a series of messages, including [WM_ACTIVATE](#), [WM_SETFOCUS](#), and [WM_SETCURSOR](#). These messages notify the window that it has been activated, that keyboard input is being directed to the window, and that the mouse cursor has been moved within the borders of the window. Nonqueued messages can also result when an application calls certain system functions. For example, the system sends the [WM_WINDOWPOSCHANGED](#) message after an application uses the [SetWindowPos](#) function to move a window.

Some functions that send nonqueued messages are [BroadcastSystemMessage](#), [BroadcastSystemMessageEx](#), [SendMessage](#), [SendMessageTimeout](#), and [SendNotifyMessage](#).

Message Handling

An application must remove and process messages posted to the message queues of its threads. A single-threaded application usually uses a *message loop* in its [WinMain](#) function to remove and send messages to the appropriate window procedures for processing. Applications with multiple threads can include a message loop in each thread that creates a window. The following sections describe how a message loop works and explain the role of a window procedure:

- [Message Loop](#)
- [Window Procedure](#)

Message Loop

A simple message loop consists of one function call to each of these three functions: [GetMessage](#), [TranslateMessage](#), and [DispatchMessage](#). Note that if there is an error, [GetMessage](#) returns –1, thus the need for the special testing.

C++

```
MSG msg;
BOOL bRet;

while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

The [GetMessage](#) function retrieves a message from the queue and copies it to a structure of type [MSG](#). It returns a nonzero value, unless it encounters the [WM_QUIT](#) message, in which case it returns [FALSE](#) and ends the loop. In a single-threaded application, ending the message loop is often the first step in closing the application. An application can end its own loop by using the [PostQuitMessage](#) function, typically in response to the [WM_DESTROY](#) message in the window procedure of the application's main window.

If you specify a window handle as the second parameter of [GetMessage](#), only messages for the specified window are retrieved from the queue. [GetMessage](#) can also filter messages in the queue, retrieving only those messages that fall within a specified range. For more information about filtering messages, see [Message Filtering](#).

A thread's message loop must include [TranslateMessage](#) if the thread is to receive character input from the keyboard. The system generates virtual-key messages ([WM_KEYDOWN](#) and [WM_KEYUP](#)) each time the user presses a key. A virtual-key message contains a virtual-key code that identifies which key was pressed, but not its character value. To retrieve this value, the message loop must contain [TranslateMessage](#), which translates the virtual-key message into a character message ([WM_CHAR](#)) and places it back into the application message queue. The character message can then be removed upon a subsequent iteration of the message loop and dispatched to a window procedure.

The [DispatchMessage](#) function sends a message to the window procedure associated with the window handle specified in the [MSG](#) structure. If the window handle is [HWND_TOPMOST](#), [DispatchMessage](#) sends the message to the window procedures of

all top-level windows in the system. If the window handle is **NULL**, **DispatchMessage** does nothing with the message.

An application's main thread starts its message loop after initializing the application and creating at least one window. After it is started, the message loop continues to retrieve messages from the thread's message queue and to dispatch them to the appropriate windows. The message loop ends when the [GetMessage](#) function removes the [WM_QUIT](#) message from the message queue.

Only one message loop is needed for a message queue, even if an application contains many windows. **DispatchMessage** always dispatches the message to the proper window; this is because each message in the queue is an [MSG](#) structure that contains the handle of the window to which the message belongs.

You can modify a message loop in a variety of ways. For example, you can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages not specifying a window. You can also direct [GetMessage](#) to search for specific messages, leaving other messages in the queue. This is useful if you must temporarily bypass the usual FIFO order of the message queue.

An application that uses accelerator keys must be able to translate keyboard messages into command messages. To do this, the application's message loop must include a call to the [TranslateAccelerator](#) function. For more information about accelerator keys, see [Keyboard Accelerators](#).

If a thread uses a modeless dialog box, the message loop must include the [IsDialogMessage](#) function so that the dialog box can receive keyboard input.

Window Procedure

A window procedure is a function that receives and processes all messages sent to the window. Every window class has a window procedure, and every window created with that class uses that same window procedure to respond to messages.

The system sends a message to a window procedure by passing the message data as arguments to the procedure. The window procedure then performs an appropriate action for the message; it checks the message identifier and, while processing the message, uses the information specified by the message parameters.

A window procedure does not usually ignore a message. If it does not process a message, it must send the message back to the system for default processing. The window procedure does this by calling the [DefWindowProc](#) function, which performs a default action and returns a message result. The window procedure must then return

this value as its own message result. Most window procedures process just a few messages and pass the others on to the system by calling [DefWindowProc](#).

Because a window procedure is shared by all windows belonging to the same class, it can process messages for several different windows. To identify the specific window affected by the message, a window procedure can examine the window handle passed with a message. For more information about window procedures, see [Window Procedures](#).

Message Filtering

An application can choose specific messages to retrieve from the message queue (while ignoring other messages) by using the [GetMessage](#) or [PeekMessage](#) function to specify a message filter. The filter is a range of message identifiers (specified by a first and last identifier), a window handle, or both. [GetMessage](#) and [PeekMessage](#) use a message filter to select which messages to retrieve from the queue. Message filtering is useful if an application must search the message queue for messages that have arrived later in the queue. It is also useful if an application must process input (hardware) messages before processing posted messages.

The **WM_KEYFIRST** and **WM_KEYLAST** constants can be used as filter values to retrieve all keyboard messages; the **WM_MOUSEFIRST** and **WM_MOUSELAST** constants can be used to retrieve all mouse messages.

Any application that filters messages must ensure that a message satisfying the message filter can be posted. For example, if an application filters for a **WM_CHAR** message in a window that does not receive keyboard input, the [GetMessage](#) function does not return. This effectively "hangs" the application.

Posting and Sending Messages

Any application can post and send messages. Like the system, an application posts a message by copying it to a message queue and sends a message by passing the message data as arguments to a window procedure. To post messages, an application uses the [PostMessage](#) function. An application can send a message by calling the [SendMessage](#), [BroadcastSystemMessage](#), [SendMessageCallback](#), [SendMessageTimeout](#), [SendNotifyMessage](#), or [SendDlgItemMessage](#) function.

Posting Messages

An application typically posts a message to notify a specific window to perform a task. [PostMessage](#) creates an **MSG** structure for the message and copies the message to the message queue. The application's message loop eventually retrieves the message and dispatches it to the appropriate window procedure.

An application can post a message without specifying a window. If the application supplies a **NULL** window handle when calling [PostMessage](#), the message is posted to the queue associated with the current thread. Because no window handle is specified, the application must process the message in the message loop. This is one way to create a message that applies to the entire application, instead of to a specific window.

Occasionally, you may want to post a message to all top-level windows in the system. An application can post a message to all top-level windows by calling [PostMessage](#) and specifying **HWND_TOPMOST** in the *hwnd* parameter.

A common programming error is to assume that the [PostMessage](#) function always posts a message. This is not true when the message queue is full. An application should check the return value of the [PostMessage](#) function to determine whether the message has been posted and, if it has not been, repost it.

Sending Messages

An application typically sends a message to notify a window procedure to perform a task immediately. The [SendMessage](#) function sends the message to the window procedure corresponding to the given window. The function waits until the window procedure completes processing and then returns the message result. Parent and child windows often communicate by sending messages to each other. For example, a parent window that has an edit control as its child window can set the text of the control by sending a message to it. The control can notify the parent window of changes to the text that are carried out by the user by sending messages back to the parent.

The [SendMessageCallback](#) function also sends a message to the window procedure corresponding to the given window. However, this function returns immediately. After the window procedure processes the message, the system calls the specified callback function. For more information about the callback function, see the [SendAsyncProc](#) function.

Occasionally, you may want to send a message to all top-level windows in the system. For example, if the application changes the system time, it must notify all top-level windows about the change by sending a [WM_TIMECHANGE](#) message. An application can send a message to all top-level windows by calling [SendMessage](#) and specifying **HWND_TOPMOST** in the *hwnd* parameter. You can also broadcast a message to all

applications by calling the [BroadcastSystemMessage](#) function and specifying **BSM_APPLICATIONS** in the *lpdwRecipients* parameter.

By using the [InSendMessage](#) or [InSendMessageEx](#) function, a window procedure can determine whether it is processing a message sent by another thread. This capability is useful when message processing depends on the origin of the message.

Message Deadlocks

A thread that calls the [SendMessage](#) function to send a message to another thread cannot continue executing until the window procedure that receives the message returns. If the receiving thread yields control while processing the message, the sending thread cannot continue executing, because it is waiting for [SendMessage](#) to return. If the receiving thread is attached to the same queue as the sender, it can cause an application deadlock to occur. (Note that journal hooks attach threads to the same queue.)

Note that the receiving thread need not yield control explicitly; calling any of the following functions can cause a thread to yield control implicitly.

- [DialogBox](#)
- [DialogBoxIndirect](#)
- [DialogBoxIndirectParam](#)
- [DialogBoxParam](#)
- [GetMessage](#)
- [MessageBox](#)
- [PeekMessage](#)
- [SendMessage](#)

To avoid potential deadlocks in your application, consider using the [SendNotifyMessage](#) or [SendMessageTimeout](#) functions. Otherwise, a window procedure can determine whether a message it has received was sent by another thread by calling the [InSendMessage](#) or [InSendMessageEx](#) function. Before calling any of the functions in the preceding list while processing a message, the window procedure should first call [InSendMessage](#) or [InSendMessageEx](#). If this function returns **TRUE**, the window procedure must call the [ReplyMessage](#) function before any function that causes the thread to yield control.

Broadcasting Messages

Each message consists of a message identifier and two parameters, *wParam* and *lParam*. The message identifier is a unique value that specifies the message purpose. The parameters provide additional information that is message-specific, but the *wParam* parameter is generally a type value that provides more information about the message.

A *message broadcast* is simply the sending of a message to multiple recipients in the system. To broadcast a message from an application, use the [BroadcastSystemMessage](#) function, specifying the recipients of the message. Rather than specify individual recipients, you must specify one or more types of recipients. These types are applications, installable drivers, network drivers, and system-level device drivers. The system sends broadcast messages to all members of each specified type.

The system typically broadcasts messages in response to changes that take place within system-level device drivers or related components. The driver or related component broadcasts the message to applications and other components to notify them of the change. For example, the component responsible for disk drives broadcasts a message whenever the device driver for the floppy disk drive detects a change of media such as when the user inserts a disk in the drive.

The system broadcasts messages to recipients in this order: system-level device drivers, network drivers, installable drivers, and applications. This means that system-level device drivers, if chosen as recipients, always get the first opportunity to respond to a message. Within a given recipient type, no driver is guaranteed to receive a given message before any other driver. This means that a message intended for a specific driver must have a globally-unique message identifier so that no other driver unintentionally processes it.

You can also broadcast messages to all top-level windows by specifying `HWND_BROADCAST` in the [SendMessage](#), [SendMessageCallback](#), [SendMessageTimeout](#), or [SendNotifyMessage](#) function.

Applications receive messages through the window procedure of their top-level windows. Messages are not sent to child windows. Services can receive messages through a window procedure or their service control handlers.

 **Note**

System-level device drivers use a related, system-level function to broadcast system messages.

Query Messages

You can create your own custom messages and use them to coordinate activities between your applications and other components in the system. This is especially useful if you have created your own installable drivers or system-level device drivers. Your custom messages can carry information to and from your driver and the applications that use the driver.

To poll recipients for permission to carry out a given action, use a *query message*. You can generate your own query messages by setting the **BSF_QUERY** value in the *dwFlags* parameter when calling [BroadcastSystemMessage](#). Each recipient of the query message must return **TRUE** for the function to send the message to the next recipient. If any recipient returns **BROADCAST_QUERY_DENY**, the broadcast ends immediately and the function returns a zero.

Feedback

Was this page helpful?



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

Using Messages and Message Queues

Article • 02/06/2023

The following code examples demonstrate how to perform the following tasks associated with Windows messages and message queues.

- [Creating a Message Loop](#)
- [Examining a Message Queue](#)
- [Posting a Message](#)
- [Sending a Message](#)

Creating a Message Loop

The system does not automatically create a message queue for each thread. Instead, the system creates a message queue only for threads that perform operations which require a message queue. If the thread creates one or more windows, a message loop must be provided; this message loop retrieves messages from the thread's message queue and dispatches them to the appropriate window procedures.

Because the system directs messages to individual windows in an application, a thread must create at least one window before starting its message loop. Most applications contain a single thread that creates windows. A typical application registers the window class for its main window, creates and shows the main window, and then starts its message loop — all in the [WinMain](#) function.

You create a message loop by using the [GetMessage](#) and [DispatchMessage](#) functions. If your application must obtain character input from the user, include the [TranslateMessage](#) function in the loop. [TranslateMessage](#) translates virtual-key messages into character messages. The following example shows the message loop in the [WinMain](#) function of a simple Windows-based application.

C++

```
HINSTANCE hinst;
HWND hwndMain;

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    MSG msg;
    BOOL bRet;
    WNDCLASS wc;
    UNREFERENCED_PARAMETER(lpszCmdLine);
```

```
// Register the window class for the main window.

if (!hPrevInstance)
{
    wc.style = 0;
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon((HINSTANCE) NULL,
        IDI_APPLICATION);
    wc.hCursor = LoadCursor((HINSTANCE) NULL,
        IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = "MainMenu";
    wc.lpszClassName = "MainWndClass";

    if (!RegisterClass(&wc))
        return FALSE;
}

hinst = hInstance; // save instance handle

// Create the main window.

hwndMain = CreateWindow("MainWndClass", "Sample",
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL,
    (HMENU) NULL, hinst, (LPVOID) NULL);

// If the main window cannot be created, terminate
// the application.

if (!hwndMain)
    return FALSE;

// Show the window and paint its contents.

ShowWindow(hwndMain, nCmdShow);
UpdateWindow(hwndMain);

// Start the message loop.

while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0 )
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

```
// Return the exit code to the system.  
  
    return msg.wParam;  
}
```

The following example shows a message loop for a thread that uses accelerators and displays a modeless dialog box. When [TranslateAccelerator](#) or [IsDialogMessage](#) returns TRUE (indicating that the message has been processed), [TranslateMessage](#) and [DispatchMessage](#) are not called. The reason for this is that [TranslateAccelerator](#) and [IsDialogMessage](#) perform all necessary translating and dispatching of messages.

C++

```
HWND hwndMain;  
HWND hwndDlgModeless = NULL;  
MSG msg;  
BOOL bRet;  
HACCEL haccel;  
//  
// Perform initialization and create a main window.  
//  
  
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)  
{  
    if (bRet == -1)  
    {  
        // handle the error and possibly exit  
    }  
    else  
    {  
        if (hwndDlgModeless == (HWND) NULL ||  
            !IsDialogMessage(hwndDlgModeless, &msg) &&  
            !TranslateAccelerator(hwndMain, haccel,  
                                  &msg))  
        {  
            TranslateMessage(&msg);  
            DispatchMessage(&msg);  
        }  
    }  
}
```

Examining a Message Queue

Occasionally, an application needs to examine the contents of a thread's message queue from outside the thread's message loop. For example, if an application's window procedure performs a lengthy drawing operation, you may want the user to be able to interrupt the operation. Unless your application periodically examines the message

queue during the operation for mouse and keyboard messages, it will not respond to user input until after the operation has completed. The reason for this is that the [DispatchMessage](#) function in the thread's message loop does not return until the window procedure finishes processing a message.

You can use the [PeekMessage](#) function to examine a message queue during a lengthy operation. [PeekMessage](#) is similar to the [GetMessage](#) function; both check a message queue for a message that matches the filter criteria and then copy the message to an [MSG](#) structure. The main difference between the two functions is that [GetMessage](#) does not return until a message matching the filter criteria is placed in the queue, whereas [PeekMessage](#) returns immediately regardless of whether a message is in the queue.

The following example shows how to use [PeekMessage](#) to examine a message queue for mouse clicks and keyboard input during a lengthy operation.

C++

```
HWND hwnd;
BOOL fDone;
MSG msg;

// Begin the operation and continue until it is complete
// or until the user clicks the mouse or presses a key.

fDone = FALSE;
while (!fDone)
{
    fDone = DoLengthyOperation(); // application-defined function

    // Remove any messages that may be in the queue. If the
    // queue contains any mouse or keyboard
    // messages, end the operation.

    while (PeekMessage(&msg, hwnd, 0, 0, PM_REMOVE))
    {
        switch(msg.message)
        {
            case WM_LBUTTONDOWN:
            case WM_RBUTTONDOWN:
            case WM_KEYDOWN:
                //
                // Perform any required cleanup.
                //
                fDone = TRUE;
        }
    }
}
```

Other functions, including [GetQueueStatus](#) and [GetInputState](#), also allow you to examine the contents of a thread's message queue. [GetQueueStatus](#) returns an array of flags that indicates the types of messages in the queue; using it is the fastest way to discover whether the queue contains any messages. [GetInputState](#) returns TRUE if the queue contains mouse or keyboard messages. Both of these functions can be used to determine whether the queue contains messages that need to be processed.

Posting a Message

You can post a message to a message queue by using the [PostMessage](#) function. [PostMessage](#) places a message at the end of a thread's message queue and returns immediately, without waiting for the thread to process the message. The function's parameters include a window handle, a message identifier, and two message parameters. The system copies these parameters to an [MSG](#) structure, fills the **time** and **pt** members of the structure, and places the structure in the message queue.

The system uses the window handle passed with the [PostMessage](#) function to determine which thread message queue should receive the message. If the handle is **HWND_TOPMOST**, the system posts the message to the thread message queues of all top-level windows.

You can use the [PostThreadMessage](#) function to post a message to a specific thread message queue. [PostThreadMessage](#) is similar to [PostMessage](#), except the first parameter is a thread identifier rather than a window handle. You can retrieve the thread identifier by calling the [GetCurrentThreadId](#) function.

Use the [PostQuitMessage](#) function to exit a message loop. [PostQuitMessage](#) posts the **WM_QUIT** message to the currently executing thread. The thread's message loop terminates and returns control to the system when it encounters the **WM_QUIT** message. An application usually calls [PostQuitMessage](#) in response to the **WM_DESTROY** message, as shown in the following example.

C++

```
case WM_DESTROY:  
  
    // Perform cleanup tasks.  
  
    PostQuitMessage(0);  
    break;
```

Sending a Message

The [SendMessage](#) function is used to send a message directly to a window procedure. [SendMessage](#) calls a window procedure and waits for that procedure to process the message and return a result.

A message can be sent to any window in the system; all that is required is a window handle. The system uses the handle to determine which window procedure should receive the message.

Before processing a message that may have been sent from another thread, a window procedure should first call the [InSendMessage](#) function. If this function returns **TRUE**, the window procedure should call [ReplyMessage](#) before any function that causes the thread to yield control, as shown in the following example.

C++

```
case WM_USER + 5:  
    if (InSendMessage())  
        ReplyMessage(TRUE);  
  
    DialogBox(hInst, "MyDialogBox", hwndMain, (DLGPROC) MyDlgProc);  
    break;
```

A number of messages can be sent to controls in a dialog box. These control messages set the appearance, behavior, and content of controls or retrieve information about controls. For example, the [CB_ADDSTRING](#) message can add a string to a combo box, and the [BM_SETCHECK](#) message can set the check state of a check box or radio button.

Use the [SendDlgItemMessage](#) function to send a message to a control, specifying the identifier of the control and the handle of the dialog box window that contains the control. The following example, taken from a dialog box procedure, copies a string from a combo box's edit control into its list box. The example uses [SendDlgItemMessage](#) to send a [CB_ADDSTRING](#) message to the combo box.

C++

```
HWND hwndCombo;  
int cTxtLen;  
PSTR pszMem;  
  
switch (uMsg)  
{  
    case WM_COMMAND:  
        switch (LOWORD(wParam))  
        {  
            case IDD_ADDCBITEM:  
                // Get the handle of the combo box and the  
                // length of the string in the edit control
```

```

// of the combo box.

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);

// Add the string to the list box of the
// combo box and remove the string from the
// edit control of the combo box.

if (pszMem != NULL)
{
    SendDlgItemMessage(hwndDlg, IDD_COMBO,
        CB_ADDSTRING, 0,
        (DWORD) ((LPSTR) pszMem));
    SetWindowText(hwndCombo, (LPSTR) NULL);
}

// Free the memory and return.

VirtualFree(pszMem, 0, MEM_RELEASE);
return TRUE;
//
// Process other dialog box commands.
//
}

//
// Process other dialog box messages.
//
}

}

```

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

Message Reference

Article • 04/27/2021

In This Section

- [Message Functions](#)
- [Message Structures](#)
- [Message Constants](#)

Feedback

Was this page helpful?



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

Message Functions (Windows and Messages)

Article • 04/27/2021

- [BroadcastSystemMessage](#)
- [BroadcastSystemMessageEx](#)
- [DispatchMessage](#)
- [GetInputState](#)
- [GetMessage](#)
- [GetMessageExtraInfo](#)
- [GetMessagePos](#)
- [GetMessageTime](#)
- [GetQueueStatus](#)
- [InSendMessage](#)
- [InSendMessageEx](#)
- [PeekMessage](#)
- [PostMessage](#)
- [PostQuitMessage](#)
- [PostThreadMessage](#)
- [RegisterWindowMessage](#)
- [ReplyMessage](#)
- [SendAsyncProc](#)
- [SendMessage](#)
- [SendMessageCallback](#)
- [SendMessageTimeout](#)
- [SendNotifyMessage](#)
- [SetMessageExtraInfo](#)
- [TranslateMessage](#)
- [WaitMessage](#)

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](#)

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](#)

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 *lpmsg* 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

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](#)

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](#)

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](#)

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](#)

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

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](#)

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](#)

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](#)

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](#)

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)

Article 04/02/2021

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](#)

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](#)

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](#)

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](#)

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](#)

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](#)

TranslateMessageEx function

Article • 04/13/2023

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++

```
WINUSERAPI  
BOOL  
WINAPI  
TranslateMessageEx(  
    _In_ CONST MSG *lpMsg,  
    _In_ UINT flags);
```

Parameters

lpMsg [in]

A pointer to a [MSG](#) structure that contains message information retrieved from the calling thread's message queue by using the [GetMessage](#) or [PeekMessage](#) function.

flags [in]

The behavior of the function.

If bit 0 is set, a menu is active. In this mode Alt+Numeric keypad key combinations are not handled.

If bit 1 is set, **TranslateMessageEx** will return FALSE when it does not post WM_CHAR or WM_SYSCHAR to the message loop.

If bit 2 is set, keyboard state is not changed (Windows 10, version 1607 and newer)

All other bits (through 31) are reserved.

Return value

Returns a boolean value. If the message is translated (that is, a character message is posted to the thread's message queue), the return value is nonzero.

Unlike [TranslateMessage](#), [TranslateMessageEx](#) will return FALSE for the case where it doesn't post a WM_CHAR.

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](#).

This function is not defined in an SDK header and must be declared by the caller. This function is exported from user32.dll.

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](#)

Message Structures

Article • 04/27/2021

- [BSMINFO](#)
 - [MSG](#)
-

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](#)

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

Message Constants

Article • 04/27/2021

In This Section

- [OCM_BASE](#)
- [WM_APP](#)
- [WM_USER](#)

Feedback

Was this page helpful?



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

OCM_BASE

Article • 06/24/2021

Used to define private messages for use by private window classes, usually of the form OCM_BASE+x, where x is an integer value.

Syntax

```
#define WM_USER          0x0400  
#define OCM_BASE          (WM_USER+0x1c00)
```

Remarks

The following are the ranges of message numbers.

Range	Meaning
0 through WM_USER-1	Messages reserved for use by the system.
WM_USER through 0xFFFF	Integer messages for use by private window classes.
WM_APP through 0xBFFF	Messages available for use by applications.
0xC000 through 0xFFFF	String messages for use by applications.
Greater than 0xFFFF	Reserved by the system.

Message numbers in the first range (0 through WM_USER-1) are defined by the system. Values in this range that are not explicitly defined are reserved by the system.

Message numbers in the second range (WM_USER through 0xFFFF) can be defined and used by an application to send messages within a private window class. These values cannot be used to define messages that are meaningful throughout an application because some predefined window classes already define values in this range. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** may use these values. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are available for applications to use as private messages. Messages in this range do not conflict with system messages.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the [RegisterWindowMessage](#) function to retrieve a message number for a string. All applications that register the same string can use the associated message number for exchanging messages. The actual message number, however, is not a constant and cannot be assumed to be the same between different sessions.

Message numbers in the fifth range (greater than 0xFFFF) are reserved by the system.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Olectl.h

See also

Reference

[RegisterWindowMessage](#)

[WM_APP](#)

Conceptual

[Messages and Message Queues](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_APP

Article • 01/07/2021

Used to define private messages, usually of the form WM_APP+x, where x is an integer value.

syntax

```
#define WM_APP          0x8000
```

Remarks

The WM_APP constant is used to distinguish between message values that are reserved for use by the system and values that can be used by an application to send messages within a private window class. The following are the ranges of message numbers available.

Range	Meaning
0 through WM_USER –1	Messages reserved for use by the system.
WM_USER through 0x7FFF	Integer messages for use by private window classes.
WM_APP through 0xBFFF	Messages available for use by applications.
0xC000 through 0xFFFF	String messages for use by applications.
Greater than 0xFFFF	Reserved by the system.

Message numbers in the first range (0 through WM_USER –1) are defined by the system. Values in this range that are not explicitly defined are reserved by the system.

Message numbers in the second range (WM_USER through 0x7FFF) can be defined and used by an application to send messages within a private window class. These values cannot be used to define messages that are meaningful throughout an application because some predefined window classes already define values in this range. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** may use these values. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are available for applications to use as private messages. Messages in this range do not conflict with system messages.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the [RegisterWindowMessage](#) function to retrieve a message number for a string. All applications that register the same string can use the associated message number for exchanging messages. The actual message number, however, is not a constant and cannot be assumed to be the same between different sessions.

Message numbers in the fifth range (greater than 0xFFFF) are reserved by the system.

Requirements

Requirement	Value
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

Reference

[RegisterWindowMessage](#)

[WM_USER](#)

Conceptual

[Messages and Message Queues](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

WM_USER

Article • 01/07/2021

Used to define private messages for use by private window classes, usually of the form **WM_USER+x**, where *x* is an integer value.

Syntax

```
#define WM_USER          0x0400
```

Remarks

The following are the ranges of message numbers.

Range	Meaning
0 through WM_USER –1	Messages reserved for use by the system.
WM_USER through 0x7FFF	Integer messages for use by private window classes.
WM_APP (0x8000) through 0xBFFF	Messages available for use by applications.
0xC000 through 0xFFFF	String messages for use by applications.
Greater than 0xFFFF	Reserved by the system.

Message numbers in the first range (0 through **WM_USER** –1) are defined by the system. Values in this range that are not explicitly defined are reserved by the system.

Message numbers in the second range (**WM_USER** through 0x7FFF) can be defined and used by an application to send messages within a private window class. These values cannot be used to define messages that are meaningful throughout an application because some predefined window classes already define values in this range. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** may use these values. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are available for applications to use as private messages. Messages in this range do not conflict with system messages.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the [RegisterWindowMessage](#) function to retrieve a message number for a string. All applications that register the same string can use the associated message number for exchanging messages. The actual message number, however, is not a constant and cannot be assumed to be the same between different sessions.

Message numbers in the fifth range (greater than 0xFFFF) are reserved by the system.

Requirements

Requirement	Value
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

Reference

[RegisterWindowMessage](#)

[WM_APP](#)

Conceptual

[Messages and Message Queues](#)

Feedback

Was this page helpful?

 Yes

 No

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

Timers

Article • 01/07/2021

A timer is an internal routine that repeatedly measures a specified interval, in milliseconds.

In This Section

Name	Description
About Timers	Describes how to create, identify, set, and delete timers.
Using Timers	Discusses how to create and destroy timers, and how to use a timer to trap mouse input at specified intervals.
Timer Reference	Contains the API reference.

Timer Functions

Name	Description
KillTimer	Destroys the specified timer.
SetTimer	Creates a timer with the specified time-out value.
TimerProc	An application-defined callback function that processes WM_TIMER messages. The TIMERPROC type defines a pointer to this callback function. <i>TimerProc</i> is a placeholder for the application-defined function name.

Timer Notifications

Name	Description
WM_TIMER	Posted to the installing thread's message queue when a timer expires. The message is posted by the GetMessage or PeekMessage function.

Feedback

Was this page helpful?

 Yes

 No

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

Timer Overviews

Article • 04/27/2021

- [About Timers](#)
- [Using Timers](#)

Feedback

Was this page helpful?

 Yes

 No

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

About Timers

Article • 01/07/2021

This topic describes how to create, identify, set, and delete timers. An application uses a timer to schedule an event for a window after a specified time has elapsed. Each time the specified interval (or time-out value) for a timer elapses, the system notifies the window associated with the timer. Because a timer's accuracy depends on the system clock rate and how often the application retrieves messages from the message queue, the time-out value is only approximate.

This topic includes the following sections.

- [Timer Operations](#)
- [High-Resolution Timer](#)
- [Waitable Timer Objects](#)
- [Related topics](#)

Timer Operations

Applications create timers by using the [SetTimer](#) function. A new timer starts timing the interval as soon as it is created. An application can change a timer's time-out value by using [SetTimer](#) and can destroy a timer by using the [KillTimer](#) function. To use system resources efficiently, applications should destroy timers that are no longer necessary.

Each timer has a unique identifier. When creating a timer, an application can either specify an identifier or have the system create a unique value. The first parameter of a [WM_TIMER](#) message contains the identifier of the timer that posted the message.

If you specify a window handle in the call to [SetTimer](#), the application associates the timer with that window. Whenever the time-out value for the timer elapses, the system posts a [WM_TIMER](#) message to the window associated with the timer. If no window handle is specified in the call to [SetTimer](#), the application that created the timer must monitor its message queue for [WM_TIMER](#) messages and dispatch them to the appropriate window. If 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](#).

If you need to be notified when a timer elapses, use a waitable timer. For more information, see [Waitable Timer Objects](#).

High-Resolution Timer

A counter is a general term used in programming to refer to an incrementing variable. Some systems include a high-resolution performance counter that provides high-resolution elapsed times.

If a high-resolution performance counter exists on the system, you can use the [QueryPerformanceFrequency](#) function to express the frequency, in counts per second. The value of the count is processor dependent. On some processors, for example, the count might be the cycle rate of the processor clock.

The [QueryPerformanceCounter](#) function retrieves the current value of the high-resolution performance counter. By calling this function at the beginning and end of a section of code, an application essentially uses the counter as a high-resolution timer. For example, suppose that [QueryPerformanceFrequency](#) indicates that the frequency of the high-resolution performance counter is 50,000 counts per second. If the application calls [QueryPerformanceCounter](#) immediately before and immediately after the section of code to be timed, the counter values might be 1500 counts and 3500 counts, respectively. These values would indicate that .04 seconds (2000 counts) elapsed while the code executed.

Waitable Timer Objects

A waitable timer object is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer.

A thread uses the [CreateWaitableTimer](#) or [CreateWaitableTimerEx](#) function to create a timer object. The creating thread specifies whether the timer is a manual-reset timer or a synchronization timer. The creating thread can specify a name for the timer object. Threads in other processes can open a handle to an existing timer by specifying its name in a call to the [OpenWaitableTimer](#) function. Any thread with a handle to a timer object can use one of the wait functions to wait for the timer state to be set to signaled.

For more information about using waitable timer objects for thread synchronization, see [Waitable Timer Objects](#).

Related topics

[Using Timers](#)

Feedback

Was this page helpful?

 Yes

 No

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

Using Timers

Article • 01/07/2021

This topic shows how to create and destroy timers, and how to use a timer to trap mouse input at specified intervals.

This topic contains the following sections.

- [Creating a Timer](#)
- [Destroying a Timer](#)
- [Using Timer Functions to Trap Mouse Input](#)
- [Related topics](#)

Creating a Timer

The following example uses the [SetTimer](#) function to create two timers. The first timer is set for every 10 seconds, the second for every five minutes.

```
// Set two timers.

SetTimer(hwnd,          // handle to main window
         IDT_TIMER1,    // timer identifier
         10000,         // 10-second interval
         (TIMERPROC) NULL); // no timer callback

SetTimer(hwnd,          // handle to main window
         IDT_TIMER2,    // timer identifier
         300000,        // five-minute interval
         (TIMERPROC) NULL); // no timer callback
```

To process the [WM_TIMER](#) messages generated by these timers, add a **WM_TIMER** case statement to the window procedure for the *hwnd* parameter.

```
case WM_TIMER:

    switch (wParam)
    {
        case IDT_TIMER1:
            // process the 10-second timer

            return 0;
```

```
case IDT_TIMER2:  
    // process the five-minute timer  
  
    return 0;  
}
```

An application can also create a timer whose **WM_TIMER** messages are processed not by the main window procedure but by an application-defined callback function, as in the following code sample, which creates a timer and uses the callback function **MyTimerProc** to process the timer's **WM_TIMER** messages.

```
// Set the timer.  
  
SetTimer(hwnd,                  // handle to main window  
        IDT_TIMER3,            // timer identifier  
        5000,                 // 5-second interval  
        (TIMERPROC) MyTimerProc); // timer callback
```

The calling convention for **MyTimerProc** must be based on the **TimerProc** callback function.

If your application creates a timer without specifying a window handle, your application must monitor the message queue for **WM_TIMER** messages and dispatch them to the appropriate window.

```
HWND hwndTimer;    // handle to window for timer messages  
MSG msg;          // message structure  
  
while (GetMessage(&msg, // message structure  
                  NULL,           // handle to window to receive the message  
                  0,                // lowest message to examine  
                  0))              // highest message to examine  
{  
  
    // Post WM_TIMER messages to the hwndTimer procedure.  
  
    if (msg.message == WM_TIMER)  
    {  
        msg.hwnd = hwndTimer;  
    }  
  
    TranslateMessage(&msg); // translates virtual-key codes  
    DispatchMessage(&msg); // dispatches message to window  
}
```

Destroying a Timer

Applications should use the [KillTimer](#) function to destroy timers that are no longer necessary. The following example destroys the timers identified by the constants IDT_TIMER1, IDT_TIMER2, and IDT_TIMER3.

```
// Destroy the timers.  
  
KillTimer(hwnd, IDT_TIMER1);  
KillTimer(hwnd, IDT_TIMER2);  
KillTimer(hwnd, IDT_TIMER3);
```

Using Timer Functions to Trap Mouse Input

Sometimes it is necessary to prevent more input while you have a mouse pointer on the screen. One way to accomplish this is to create a special routine that traps mouse input until a specific event occurs. Many developers refer to this routine as "building a mousetrap."

The following example uses the [SetTimer](#) and [KillTimer](#) functions to trap mouse input. [SetTimer](#) creates a timer that sends a [WM_TIMER](#) message every 10 seconds. Each time the application receives a [WM_TIMER](#) message, it records the mouse pointer location. If the current location is the same as the previous location and the application's main window is minimized, the application moves the mouse pointer to the icon. When the application closes, [KillTimer](#) stops the timer.

```
HICON hIcon1;           // icon handle  
POINT ptOld;           // previous cursor location  
UINT uResult;          // SetTimer's return value  
HINSTANCE hinstance;    // handle to current instance  
  
//  
// Perform application initialization here.  
//  
  
wc.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(400));  
wc.hCursor = LoadCursor(hinstance, MAKEINTRESOURCE(200));  
  
// Record the initial cursor position.  
  
GetCursorPos(&ptOld);
```

```
// Set the timer for the mousetrap.

uResult = SetTimer(hwnd,                  // handle to main window
                  IDT_MOUSETRAP,    // timer identifier
                  10000,            // 10-second interval
                  (TIMERPROC) NULL); // no timer callback

if (uResult == 0)
{
    ErrorHandler("No timer is available.");
}

LONG APIENTRY MainWndProc(
    HWND hwnd,          // handle to main window
    UINT message,       // type of message
    WPARAM wParam,     // additional information
    LPARAM lParam)     // additional information
{
    HDC hdc;           // handle to device context
    POINT pt;          // current cursor location
    RECT rc;           // location of minimized window

    switch (message)
    {
        //
        // Process other messages.
        //

        case WM_TIMER:
            // If the window is minimized, compare the current
            // cursor position with the one from 10 seconds
            // earlier. If the cursor position has not changed,
            // move the cursor to the icon.

            if (IsIconic(hwnd))
            {
                GetCursorPos(&pt);

                if ((pt.x == ptOld.x) && (pt.y == ptOld.y))
                {
                    GetWindowRect(hwnd, &rc);
                    SetCursorPos(rc.left, rc.top);
                }
                else
                {
                    ptOld.x = pt.x;
                    ptOld.y = pt.y;
                }
            }

            return 0;

        case WM_DESTROY:
```

```

    // Destroy the timer.

    KillTimer(hwnd, IDT_MOUSETRAP);
    PostQuitMessage(0);
    break;

    //
    // Process other messages.
    //

}

```

Although the following example also shows you how to trap mouse input, it processes the [WM_TIMER](#) message through the application-defined callback function [MyTimerProc](#), rather than through the application's message queue.

```

UINT uResult;           // SetTimer's return value
HICON hIcon1;          // icon handle
POINT ptOld;           // previous cursor location
HINSTANCE hinstance;   // handle to current instance

//
// Perform application initialization here.
//

wc.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(400));
wc.hCursor = LoadCursor(hinstance, MAKEINTRESOURCE(200));

// Record the current cursor position.

GetCursorPos(&ptOld);

// Set the timer for the mousetrap.

uResult = SetTimer(hwnd,      // handle to main window
                  IDT_MOUSETRAP, // timer identifier
                  10000,        // 10-second interval
                  (TIMERPROC) MyTimerProc); // timer callback

if (uResult == 0)
{
    ErrorHandler("No timer is available.");
}

LONG APIENTRY MainWndProc(
    HWND hwnd,           // handle to main window
    UINT message,        // type of message
    WPARAM wParam,       // additional information
    LPARAM lParam)      // additional information
{

```

```
HDC hdc;           // handle to device context

switch (message)
{
// 
// Process other messages.
//

    case WM_DESTROY:
        // Destroy the timer.

        KillTimer(hwnd, IDT_MOUSETRAP);
        PostQuitMessage(0);
        break;

//
// Process other messages.
//


}

// MyTimerProc is an application-defined callback function that
// processes WM_TIMER messages.

VOID CALLBACK MyTimerProc(
    HWND hwnd,          // handle to window for timer messages
    UINT message,       // WM_TIMER message
    UINT idTimer,       // timer identifier
    DWORD dwTime)      // current system time
{

    RECT rc;
    POINT pt;

    // If the window is minimized, compare the current
    // cursor position with the one from 10 seconds earlier.
    // If the cursor position has not changed, move the
    // cursor to the icon.

    if (IsIconic(hwnd))
    {
        GetCursorPos(&pt);

        if ((pt.x == ptOld.x) && (pt.y == ptOld.y))
        {
            GetWindowRect(hwnd, &rc);
            SetCursorPos(rc.left, rc.top);
        }
        else
        {
            ptOld.x = pt.x;
            ptOld.y = pt.y;
        }
    }
}
```

```
    }  
}
```

Related topics

[About Timers](#)

Feedback

Was this page helpful?

 Yes

 No

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

Timer Reference

Article • 04/27/2021

In This Section

- [Timer Functions](#)
- [Timer Notifications](#)

Feedback

Was this page helpful?

 Yes

 No

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

Timer Functions

Article • 04/27/2021

In This Section

- [KillTimer](#)
- [SetCoalescableTimer](#)
- [SetTimer](#)
- [*TimerProc*](#)

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

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 (0xFFFFFFFF), 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](#)

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](#)

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](#)

Timer Notifications

Article • 04/27/2021

- [WM_TIMER](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_TIMER message

Article • 01/07/2021

Posted to the installing thread's message queue when a timer expires. The message is posted by the [GetMessage](#) or [PeekMessage](#) function.

C++

```
#define WM_TIMER          0x0113
```

Parameters

wParam [in]

The timer identifier.

lParam [in]

A pointer to an application-defined callback function that was passed to the [SetTimer](#) function when the timer was installed.

Return value

Type: [LRESULT](#)

An application should return zero if it processes this message.

Remarks

You can process the message by providing a WM_TIMER case in the window procedure. Otherwise, [DispatchMessage](#) will call the *TimerProc* callback function specified in the call to the [SetTimer](#) function used to install the timer.

The WM_TIMER message is a low-priority message. The [GetMessage](#) and [PeekMessage](#) functions post this message only when no other higher-priority messages are in the thread's message queue.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[GetMessage](#)

[PeekMessage](#)

[SetTimer](#)

[TimerProc](#)

Conceptual

[Timers](#)

Feedback

Was this page helpful?

 Yes

 No

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

Window Properties

Article • 01/07/2021

A window property is any data assigned to a window. A window property is usually a handle of the window-specific data, but it may be any value. Each window property is identified by a string name.

In This Section

Name	Description
About Window Properties	Discusses window properties.
Using Window Properties	Explains how to perform the following tasks associated with window properties.
Window Property Reference	Contains the API reference.

Window Property Functions

Name	Description
EnumProps	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.
EnumPropsEx	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.
GetProp	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.
PropEnumProc	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.

Name	Description
PropEnumProcEx	An 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. <i>PropEnumProcEx</i> is a placeholder for the application-defined function name.
RemoveProp	Removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed.
SetProp	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.

Feedback

Was this page helpful?



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

Window Property Overviews

Article • 04/27/2021

- [About Window Properties](#)
- [Using Window Properties](#)

Feedback

Was this page helpful?

 Yes

 No

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

About Window Properties

Article • 01/07/2021

A *window property* is any data assigned to a window. A window property is usually a handle of the window-specific data, but it may be any value. Each window property is identified by a string name. There are several functions that enable applications to use window properties. This overview discusses the following topics:

- [Advantages of Using Window Properties](#)
- [Assigning Window Properties](#)
- [Enumerating Window Properties](#)

Advantages of Using Window Properties

Window properties are typically used to associate data with a subclassed window or a window in a multiple-document interface (MDI) application. In either case, it is not convenient to use the extra bytes specified in the [CreateWindow](#) function or class structure for the following two reasons:

- An application might not know how many extra bytes are available or how the space is being used. By using window properties, the application can associate data with a window without accessing the extra bytes.
- An application must access the extra bytes by using offsets. However, window properties are accessed by their string identifiers, not by offsets.

For more information about subclassing, see [Window Procedure Subclassing](#). For more information about MDI windows, see [Multiple Document Interface](#).

Assigning Window Properties

The [SetProp](#) function assigns a window property and its string identifier to a window. The [GetProp](#) function retrieves the window property identified by the specified string. The [RemoveProp](#) function destroys the association between a window and a window property but does not destroy the data itself. To destroy the data itself, use the appropriate function to free the handle that is returned by [RemoveProp](#).

Enumerating Window Properties

The [EnumProps](#) and [EnumPropsEx](#) functions enumerate all of a window's properties by using an application-defined callback function. For more information about the callback

function, see [PropEnumProc](#).

EnumPropsEx includes an extra parameter for application-defined data used by the callback function. For more information about the callback function, see [PropEnumProcEx](#).

Feedback

Was this page helpful?

 Yes

 No

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

Using Window Properties

Article • 01/07/2021

This section explains how to perform the following tasks associated with window properties.

- [Adding a Window Property](#)
- [Retrieving a Window Property](#)
- [Listing Window Properties for a Given Window](#)
- [Deleting a Window Property](#)

Adding a Window Property

The following example loads an icon and then a cursor and allocates memory for a buffer. The example then uses the [SetProp](#) function to assign the resulting icon, cursor, and memory handles as window properties for the window identified by the application-defined hwndSubclass variable. The properties are identified by the strings PROP_ICON, PROP_CURSOR, and PROP_BUFFER.

```
#define BUFFER 4096

HINSTANCE hinst;          // handle of current instance
HWND hwndSubclass;        // handle of a subclassed window
HANDLE hIcon, hCursor;
HGLOBAL hMem;
char *lpMem;
TCHAR tchPath[] = "c:\\winnt\\samples\\winprop.c";
HRESULT hResult;

// Load resources.

hIcon = LoadIcon(hinst, MAKEINTRESOURCE(400));
hCursor = LoadCursor(hinst, MAKEINTRESOURCE(220));

// Allocate and fill a memory buffer.

hMem = GlobalAlloc(GPTR, BUFFER);
lpMem = GlobalLock(hMem);
if (lpMem == NULL)
{
    // TODO: write error handler
}
hResult = StringCchCopy(lpMem, STRSAFE_MAX_CCH, tchPath);
if (FAILED(hResult))
{
```

```

// TO DO: write error handler if function fails.
}

GlobalUnlock(hMem);

// Set the window properties for hwndSubclass.

SetProp(hwndSubclass, "PROP_ICON", hIcon);
SetProp(hwndSubclass, "PROP_CURSOR", hCursor);
SetProp(hwndSubclass, "PROP_BUFFER", hMem);

```

Retrieving a Window Property

A window can create handles to its window property data and use the data for any purpose. The following example uses [GetProp](#) to obtain handles to the window properties identified by PROP_ICON, PROP_CURSOR, and PROP_BUFFER. The example then displays the contents of the newly obtained memory buffer, cursor, and icon in the window's client area.

```

#define PATHLENGTH 256

HWND hwndSubclass;      // handle of a subclassed window
HANDLE hIconProp, hCursProp;
HGLOBAL hMemProp;
char *lpFilename;
TCHAR tchBuffer[PATHLENGTH];
size_t * nSize;
HDC hdc;
HRESULT hResult;

// Get the window properties, then use the data.

hIconProp = (HICON) GetProp(hwndSubclass, "PROP_ICON");
TextOut(hdc, 10, 40, "PROP_ICON", 9);
DrawIcon(hdc, 90, 40, hIconProp);

hCursProp = (HCURSOR) GetProp(hwndSubclass, "PROP_CURSOR");
TextOut(hdc, 10, 85, "PROP_CURSOR", 9);
DrawIcon(hdc, 110, 85, hCursProp);

hMemProp = (HGLOBAL) GetProp(hwndSubclass, "PROP_BUFFER");
lpFilename = GlobalLock(hMemProp);
hResult = StringCchPrintf(tchBuffer, PATHLENGTH,
    "Path to file: %s", lpFilename);
if (FAILED(hResult))
{
// TODO: write error handler if function fails.
}
hResult = StringCchLength(tchBuffer, PATHLENGTH, nSize)

```

```
if (FAILED(hResult))
{
// TODO: write error handler if function fails.
}
TextOut(hdc, 10, 10, tchBuffer, *nSize);
```

Listing Window Properties for a Given Window

In the following example, the [EnumPropsEx](#) function lists the string identifiers of the window properties for the window identified by the application-defined hwndSubclass variable. This function relies on the application-defined callback function WinPropProc to display the strings in the window's client area.

```
EnumPropsEx(hwndSubclass, WinPropProc, NULL);

// WinPropProc is an application-defined callback function
// that lists a window property.

BOOL CALLBACK WinPropProc(
    HWND hwndSubclass, // handle of window with property
    LPCSTR lpszString, // property string or atom
    HANDLE hData)      // data handle
{
    static int nProp = 1; // property counter
    TCHAR tchBuffer[BUFFER]; // expanded-string buffer
    size_t * nSize;          // size of string in buffer
    HDC hdc;                // device-context handle
    HRESULT hResult;

    hdc = GetDC(hwndSubclass);

    // Display window property string in client area.
    hResult = StringCchPrintf(tchBuffer, BUFFER, "WinProp %d: %s", nProp++, lpszString);
    if (FAILED(hResult))
    {
        // TO DO: write error handler if function fails.
    }
    hResult = StringCchLength(tchBuffer, BUFFER, nSize);
    if (FAILED(hResult))
    {
        // TO DO: write error handler if function fails.
    }
    TextOut(hdc, 10, nProp * 20, tchBuffer, *nSize);

    ReleaseDC(hwndSubclass, hdc);
```

```
    return TRUE;  
}
```

Deleting a Window Property

When a window is destroyed, it must destroy any window properties it set. The following example uses the [EnumPropsEx](#) function and the application-defined callback function `DelPropProc` to destroy the properties associated with the window identified by the application-defined `hwndSubclass` variable. The callback function, which uses the [RemoveProp](#) function, is also shown.

```
case WM_DESTROY:  
  
    EnumPropsEx(hwndSubclass, DelPropProc, NULL);  
  
    PostQuitMessage(0);  
    break;  
  
// DelPropProc is an application-defined callback function  
// that deletes a window property.  
  
BOOL CALLBACK DelPropProc(  
    HWND hwndSubclass, // handle of window with property  
    LPCSTR lpszString, // property string or atom  
    HANDLE hData)      // data handle  
{  
    hData = RemoveProp(hwndSubclass, lpszString);  
    //  
    // if appropriate, free the handle hData  
    //  
  
    return TRUE;  
}
```

Feedback

Was this page helpful?



Yes



No

Get help at Microsoft Q&A

Window Property Reference

Article • 04/27/2021

- [Window Property Functions](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

Window Property Functions

Article • 04/27/2021

- [EnumProps](#)
 - [EnumPropsEx](#)
 - [GetProp](#)
 - [*PropEnumProc*](#)
 - [*PropEnumProcEx*](#)
 - [RemoveProp](#)
 - [SetProp](#)
-

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](#)

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? [!\[\]\(2706f10a6ff2f3ff67206c0342037de7_img.jpg\) Yes](#) [!\[\]\(5fca115314313abf5199bec28345f955_img.jpg\) 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](#)

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](#)

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](#)

Configuration (Windows and Messages)

Article • 06/18/2021

Display elements are the parts of a window and the display that appear on the system display screen. *System metrics* are the dimensions of various display elements. Typical system metrics include the window border width, icon height, and so on. System metrics also describe other aspects of the system, such as whether a mouse is installed, double-byte characters are supported, or a debugging version of the operating system is installed. The [GetSystemMetrics](#) function retrieves the specified system metric.

Applications can also retrieve and set the color of window elements such as menus, scroll bars, and buttons by using the [GetSysColor](#) and [SetSysColors](#) functions, respectively.

The [SystemParametersInfo](#) function retrieves or sets various system attributes, such as double-click time, screen saver time-out, window border width, and desktop pattern. When an application uses [SystemParametersInfo](#) to set a parameter, the change takes place immediately. This function also enables applications to update the user profile, so changes to the system will be preserved when the system is restarted.

Feedback

Was this page helpful?

 Yes

 No

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

Configuration Reference

Article • 01/07/2021

The following functions can be used to control aspects of the configuration of display elements:

- [GetSystemMetrics](#)
- [SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

Configuration Constants

Article • 01/07/2021

This section provides the reference specifications for [SystemParametersInfo](#) constants related to [Configuration](#) system attributes.

In this section

Topic	Description
Contact Visualization	The following constants are used by applications or UI frameworks to identify how UI feedback is processed when an input contact is detected.
Gesture Visualization	The following constants are used by applications or UI frameworks to identify how UI feedback is processed when one of the listed gestures is detected.
Pen Visualization	The following constants are used by applications or UI frameworks to identify how UI feedback is processed when one of the listed pen gestures is detected.

Related topics

[Configuration Reference](#)

[Input Feedback Configuration](#)

Feedback

Was this page helpful?

 Yes

 No

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

Contact Visualization

Article • 01/07/2021

The following constants are used by applications or UI frameworks to identify how UI feedback is processed when an input contact is detected.

These constants are used with the **SPI_GETCONTACTVISUALIZATION** and **SPI_SETCOMTACTVISUALIZATION** parameters and the [SystemParametersInfo](#) function.

CONTACTVISUALIZATION_OFF

0x0000

Specifies UI feedback for all contacts is off.

CONTACTVISUALIZATION_ON

0x0001

Specifies UI feedback for all contacts is on.

CONTACTVISUALIZATION_PRESENTATIONMODE

0x0002

Specifies UI feedback for all contacts is on with presentation mode visuals.

Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	Winuser.h

See also

[Configuration Constants](#)

[Gesture Visualization](#)

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

Gesture Visualization

Article • 01/07/2021

The following constants are used by applications or UI frameworks to identify how UI feedback is processed when one of the listed gestures is detected.

These constants are used with the **SPI_GETGESTUREVISUALIZATION** and **SPI_SETGESTUREVISUALIZATION** parameters and the [SystemParametersInfo](#) function.

Note

For retrieving or setting pen visualization info, we recommend that you use the **SPI_GETPENVISUALIZATION** and **SPI_SETPENVISUALIZATION** parameters and the constants listed in [Pen Visualization](#).

GESTUREVISUALIZATION_OFF

0x0000

Specifies that UI feedback for all gestures is off.

GESTUREVISUALIZATION_ON

0x001F

Specifies that UI feedback for all gestures is on.

GESTUREVISUALIZATION_TAP

0x0001

Specifies UI feedback for a tap.

GESTUREVISUALIZATION_DOUBLETAP

0x0002

Specifies UI feedback for a double tap.

GESTUREVISUALIZATION_PRESSANDTAP

0x0004

Specifies UI feedback for a press and tap.

GESTUREVISUALIZATION_PRESSANDHOLD

0x0008

Specifies UI feedback for a press and hold.

GESTUREVISUALIZATION_RIGHTTAP

0x0010

Specifies UI feedback for a right tap.

Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	Winuser.h

See also

[Configuration Constants](#)

[Contact Visualization](#)

[SystemParametersInfo](#)

[Input Feedback Configuration](#)

Feedback

Was this page helpful?

 Yes

 No

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

Pen Visualization

Article • 01/07/2021

The following constants are used by applications or UI frameworks to identify how UI feedback is processed when one of the listed pen gestures is detected.

These constants are used with the **SPI_GETPENVISUALIZATION** and **SPI_SETPENVISUALIZATION** parameters and the [SystemParametersInfo](#) function.

PENVISUALIZATION_OFF

0x0000

Specifies that UI feedback for all pen gestures is off.

PENVISUALIZATION_ON

0x0023

Specifies that UI feedback for all pen gestures is on.

PENVISUALIZATION_TAP

0x0001

Specifies UI feedback for a pen tap.

PENVISUALIZATION_DOUBLETAP

0x0002

Specifies UI feedback for a pen double tap.

PENVISUALIZATION_CURSOR

0x0020

Specifies UI feedback for the pen cursor.

Requirements

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]

Requirement	Value
Header	Winuser.h

See also

[Configuration Constants](#)

[Contact Visualization](#)

[SystemParametersInfo](#)

[Input Feedback Configuration](#)

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](#)

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_GETCLEARTEXTURE 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.
	<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_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>

	Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This parameter is not supported.
SPI_SETPENDRAGOUTTHRESHOLD 0x0087	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. 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_SETPENSIDEMOVETHRESHOLD 0x008B	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. 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_SETSHOWIMEUI 0x006F	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.
SPI_SETSNAPSIZING 0x008F	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. 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_SETWINARRANGING 0x0083	Sets whether window arrangement is enabled. Set <i>pvParam</i> to TRUE for on or FALSE for off. 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.

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](#)

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](#)

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](#)

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](#)

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](#)

WM_SETTINGCHANGE message

Article • 01/07/2021

A message that is sent to all top-level windows when the [SystemParametersInfo](#) function changes a system-wide setting or when policy settings have changed.

Applications should send WM_SETTINGCHANGE to all top-level windows when they make changes to system parameters. (This message cannot be sent directly to a window.) To send the WM_SETTINGCHANGE message to all top-level windows, use the [SendMessageTimeout](#) function with the *hwnd* parameter set to **HWND_BROADCAST**.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_WININICHANGE 0x001A  
#define WM_SETTINGCHANGE WM_WININICHANGE
```

Parameters

wParam

When the system sends this message as a result of a [SystemParametersInfo](#) call, the *wParam* parameter is the value of the *uiAction* parameter passed to the [SystemParametersInfo](#) function. For a list of values, see [SystemParametersInfo](#).

When the system sends this message as a result of a change in policy settings, this parameter indicates the type of policy that was applied. This value is 1 if computer policy was applied or zero if user policy was applied.

When the system sends this message as a result of a change in locale settings, this parameter is zero.

When an application sends this message, this parameter must be **NULL**.

lParam

When the system sends this message as a result of a [SystemParametersInfo](#) call, *lParam* is a pointer to a string that indicates the area containing the system parameter that was changed. This parameter does not usually indicate which specific system parameter changed. (Note that some applications send this message with *lParam* set to **NULL**.) In

general, when you receive this message, you should check and reload any system parameter settings that are used by your application.

This string can be the name of a registry key or the name of a section in the Win.ini file. When the string is a registry name, it typically indicates only the leaf node in the registry, not the full path.

When the system sends this message as a result of a change in policy settings, this parameter points to the string "Policy".

When the system sends this message as a result of a change in locale settings, this parameter points to the string "intl".

To effect a change in the environment variables for the system or the user, broadcast this message with *lParam* set to the string "Environment".

Return value

Type: LRESULT

If you process this message, return zero.

Remarks

The *lParam* parameter indicates which system metric has changed, for example, "ConvertibleSlateMode" if the CONVERTIBLESLATEMODE indicator was toggled or "SystemDockMode" if the DOCKED indicator was toggled.

Requirements

Requirement	Value
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

[Policy Events](#)

[SendMessageTimeout](#)

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_WININICHANGE message

Article • 01/07/2021

An application sends the **WM_WININICHANGE** message to all top-level windows after making a change to the WIN.INI file. The [SystemParametersInfo](#) function sends this message after an application uses the function to change a setting in WIN.INI.

ⓘ Note

The **WM_WININICHANGE** message is provided only for compatibility with earlier versions of the system. Applications should use the **WM_SETTINGCHANGE** message.

A window receives this message through its [WindowProc](#) function.

C++

```
#define WM_WININICHANGE 0x001A
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to a string containing the name of the system parameter that was changed. For example, this string can be the name of a registry key or the name of a section in the Win.ini file. This parameter is not particularly useful in determining which system parameter changed. For example, when the string is a registry name, it typically indicates only the leaf node in the registry, not the whole path. In addition, some applications send this message with *lParam* set to **NULL**. In general, when you receive this message, you should check and reload any system parameter settings that are used by your application.

Return value

Type: LRESULT

If you process this message, return zero.

Remarks

To send the **WM_WININICHANGE** message to all top-level windows, use the [SendMessage](#) function with the *hWnd* parameter set to **HWND_BROADCAST**.

Calls to functions that change WIN.INI may be mapped to the registry instead. This mapping occurs when WIN.INI and the section being changed are specified in the registry under the following key:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\IniFileMapping`

The change in the storage location has no effect on the behavior of this message.

Requirements

Requirement	Value
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](#)

Hooks

Article • 01/07/2021

A hook is a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure.

In This Section

Name	Description
Hook Overview	Discusses how hooks should be used.
Using Hooks	Demonstrates how to perform tasks associated with hooks.
Hook Reference	Contains the API reference.

Hook Functions

Name	Description
CallMsgFilter	Passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hook procedures.
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.
CallWndProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function before calling the window procedure to process a message sent to the thread.
CallWndRetProc	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.

Name	Description
CBTProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the keyboard focus; or before synchronizing with the system message queue. A computer-based training (CBT) application uses this hook procedure to receive useful notifications from the system.
DebugProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function before calling the hook procedures associated with any type of hook. The system passes information about the hook to be called to the DebugProc hook procedure, which examines the information and determines whether to allow the hook to be called.
ForegroundIdleProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function whenever the foreground thread is about to become idle.
GetMsgProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function whenever the GetMessage or PeekMessage function has retrieved a message from an application message queue. Before returning the retrieved message to the caller, the system passes the message to the hook procedure.
JournalPlaybackProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. Typically, an application uses this function to play back a series of mouse and keyboard messages recorded previously by the JournalRecordProc hook procedure. As long as a JournalPlaybackProc hook procedure is installed, regular mouse and keyboard input is disabled.
JournalRecordProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The function records messages the system removes from the system message queue. Later, an application can use a JournalPlaybackProc hook procedure to play back the messages.
KeyboardProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function whenever an application calls the GetMessage or PeekMessage function and there is a keyboard message (WM_KEYUP or WM_KEYDOWN) to be processed.

Name	Description
LowLevelKeyboardProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function every time a new keyboard input event is about to be posted into a thread input queue.
LowLevelMouseProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function every time a new mouse input event is about to be posted into a thread input queue.
MessageProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function after an input event occurs in a dialog box, message box, menu, or scroll bar, but before the message generated by the input event is processed. The hook procedure can monitor messages for a dialog box, message box, menu, or scroll bar created by a particular application or all applications.
MouseProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function whenever an application calls the GetMessage or PeekMessage function and there is a mouse message to be processed.
SetWindowsHookEx	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.
ShellProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The function receives notifications of Shell events from the system.
SysMsgProc	An application-defined or library-defined callback function used with the SetWindowsHookEx function. The system calls this function after an input event occurs in a dialog box, message box, menu, or scroll bar, but before the message generated by the input event is processed. The function can monitor messages for any dialog box, message box, menu, or scroll bar in the system.
UnhookWindowsHookEx	Removes a hook procedure installed in a hook chain by the SetWindowsHookEx function.

Hook Notifications

Name	Description
------	-------------

Name	Description
WM_CANCELJOURNAL	Posted to an application when a user cancels the application's journaling activities. The message is posted with a N ULL window handle.
WM_QUEUESYNC	Sent by a CBT application to separate user-input messages from other messages sent through the WH_JOURNALPLAYBACK procedure.

Hook Structures

Name	Description
CBT_CREATEWND	Contains information passed to a WH_CBT hook procedure, CBTProc , before a window is created.
CBTACTIVATESTRUCT	Contains information passed to a WH_CBT hook procedure, CBTProc , before a window is activated.
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.
KBDLLHOOKSTRUCT	Contains information about a low-level keyboard input event.
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 .
MSLLHOOKSTRUCT	Contains information about a low-level mouse input event.

Related topics

[SetWinEventHook](#)

Feedback

Was this page helpful?

 Yes

 No

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

Hooks Overview

Article • 09/02/2022

A *hook* is a mechanism by which an application can intercept events, such as messages, mouse actions, and keystrokes. A function that intercepts a particular type of event is known as a *hook procedure*. A hook procedure can act on each event it receives, and then modify or discard the event.

The following some example uses for hooks:

- Monitor messages for debugging purposes
- Provide support for recording and playback of macros
- Provide support for a help key (F1)
- Simulate mouse and keyboard input
- Implement a computer-based training (CBT) application

ⓘ Note

Hooks tend to slow down the system because they increase the amount of processing the system must perform for each message. You should install a hook only when necessary, and remove it as soon as possible.

This section discusses the following:

- [Hook Chains](#)
- [Hook Procedures](#)
- [Hook Types](#)
 - [WH_CALLWNDPROC and WH_CALLWNDPROCRET](#)
 - [WH_CBT](#)
 - [WH_DEBUG](#)
 - [WH_FOREGROUNDIDLE](#)
 - [WH_GETMESSAGE](#)
 - [WH_JOURNALPLAYBACK](#)
 - [WH_JOURNALRECORD](#)
 - [WH_KEYBOARD_LL](#)
 - [WH_KEYBOARD](#)
 - [WH_MOUSE_LL](#)
 - [WH_MOUSE](#)
 - [WH_MSGFILTER and WH_SYSMSGFILTER](#)

- o WH_SHELL

Hook Chains

The system supports many different types of hooks; each type provides access to a different aspect of its message-handling mechanism. For example, an application can use the [WH_MOUSE](#) hook to monitor the message traffic for mouse messages.

The system maintains a separate hook chain for each type of hook. A *hook chain* is a list of pointers to special, application-defined callback functions called *hook procedures*. When a message occurs that is associated with a particular type of hook, the system passes the message to each hook procedure referenced in the hook chain, one after the other. The action a hook procedure can take depends on the type of hook involved. The hook procedures for some types of hooks can only monitor messages; others can modify messages or stop their progress through the chain, preventing them from reaching the next hook procedure or the destination window.

Hook Procedures

To take advantage of a particular type of hook, the developer provides a hook procedure and uses the [SetWindowsHookEx](#) function to install it into the chain associated with the hook. A hook procedure must have the following syntax:

syntax

```
LRESULT CALLBACK HookProc(
    int nCode,
    WPARAM wParam,
    LPARAM lParam
)
{
    // process event
    ...

    return CallNextHookEx(NULL, nCode, wParam, lParam);
}
```

HookProc is a placeholder for an application-defined name.

The *nCode* parameter is a hook code that the hook procedure uses to determine the action to perform. The value of the hook code depends on the type of the hook; each type has its own characteristic set of hook codes. The values of the *wParam* and *lParam* parameters depend on the hook code, but they typically contain information about a message that was sent or posted.

The [SetWindowsHookEx](#) function always installs a hook procedure at the beginning of a hook chain. When an event occurs that is monitored by a particular type of hook, the system calls the procedure at the beginning of the hook chain associated with the hook. Each hook procedure in the chain determines whether to pass the event to the next procedure. A hook procedure passes an event to the next procedure by calling the [CallNextHookEx](#) function.

Note that the hook procedures for some types of hooks can only monitor messages. the system passes messages to each hook procedure, regardless of whether a particular procedure calls [CallNextHookEx](#).

A *global hook* monitors messages for all threads in the same desktop as the calling thread. A *thread-specific hook* monitors messages for only an individual thread. A global hook procedure can be called in the context of any application in the same desktop as the calling thread, so the procedure must be in a separate DLL module. A thread-specific hook procedure is called only in the context of the associated thread. If an application installs a hook procedure for one of its own threads, the hook procedure can be in either the same module as the rest of the application's code or in a DLL. If the application installs a hook procedure for a thread of a different application, the procedure must be in a DLL. For information, see [Dynamic-Link Libraries](#).

ⓘ Note

You should use global hooks only for debugging purposes; otherwise, you should avoid them. Global hooks hurt system performance and cause conflicts with other applications that implement the same type of global hook.

Hook Types

Each type of hook enables an application to monitor a different aspect of the system's message-handling mechanism. The following sections describe the available hooks.

- [WH_CALLWNDPROC](#) and [WH_CALLWNDPROCRET](#)
- [WH_CBT](#)
- [WH_DEBUG](#)
- [WH_FOREGROUNDIDLE](#)
- [WH_GETMESSAGE](#)
- [WH_JOURNALPLAYBACK](#)
- [WH_JOURNALRECORD](#)

- [WH_KEYBOARD_LL](#)
- [WH_KEYBOARD](#)
- [WH_MOUSE_LL](#)
- [WH_MOUSE](#)
- [WH_MSGFILTER](#) and [WH_SYSMSGFILTER](#)
- [WH_SHELL](#)

WH_CALLWNDPROC and WH_CALLWNDPROCRET

The **WH_CALLWNDPROC** and **WH_CALLWNDPROCRET** hooks enable you to monitor messages sent to window procedures. The system calls a **WH_CALLWNDPROC** hook procedure before passing the message to the receiving window procedure, and calls the **WH_CALLWNDPROCRET** hook procedure after the window procedure has processed the message.

The **WH_CALLWNDPROCRET** hook passes a pointer to a [CWPRETSTRUCT](#) structure to the hook procedure. The structure contains the return value from the window procedure that processed the message, as well as the message parameters associated with the message. Subclassing the window does not work for messages set between processes.

For more information, see the [CallWndProc](#) and [CallWndRetProc](#) callback functions.

WH_CBT

The system calls a **WH_CBT** hook procedure before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the input focus; or before synchronizing with the system message queue. The value the hook procedure returns determines whether the system allows or prevents one of these operations. The **WH_CBT** hook is intended primarily for computer-based training (CBT) applications.

For more information, see the [CBTPROC](#) callback function.

For information, see [WinEvents](#).

WH_DEBUG

The system calls a **WH_DEBUG** hook procedure before calling hook procedures associated with any other hook in the system. You can use this hook to determine whether to allow the system to call hook procedures associated with other types of hooks.

For more information, see the [DebugProc](#) callback function.

WH_FOREGROUNDIDLE

The **WH_FOREGROUNDIDLE** hook enables you to perform low priority tasks during times when its foreground thread is idle. The system calls a **WH_FOREGROUNDIDLE** hook procedure when the application's foreground thread is about to become idle.

For more information, see the [ForegroundIdleProc](#) callback function.

WH_GETMESSAGE

The **WH_GETMESSAGE** hook enables an application to monitor messages about to be returned by the [GetMessage](#) or [PeekMessage](#) function. You can use the **WH_GETMESSAGE** hook to monitor mouse and keyboard input and other messages posted to the message queue.

For more information, see the [GetMsgProc](#) callback function.

WH_JOURNALPLAYBACK

Warning

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.

The **WH_JOURNALPLAYBACK** hook enables an application to insert messages into the system message queue. You can use this hook to play back a series of mouse and keyboard events recorded earlier by using [WH_JOURNALRECORD](#). Regular mouse and keyboard input is disabled as long as a **WH_JOURNALPLAYBACK** hook is installed. A **WH_JOURNALPLAYBACK** hook is a global hook—it cannot be used as a thread-specific hook.

The **WH_JOURNALPLAYBACK** hook returns a time-out value. This value tells the system how many milliseconds to wait before processing the current message from the playback hook. This enables the hook to control the timing of the events it plays back.

For more information, see the [JournalPlaybackProc](#) callback function.

WH_JOURNALRECORD

Warning

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.

The **WH_JOURNALRECORD** hook enables you to monitor and record input events. Typically, you use this hook to record a sequence of mouse and keyboard events to play back later by using [WH_JOURNALPLAYBACK](#). The **WH_JOURNALRECORD** hook is a global hook—it cannot be used as a thread-specific hook.

For more information, see the [*JournalRecordProc*](#) callback function.

WH_KEYBOARD_LL

The **WH_KEYBOARD_LL** hook enables you to monitor keyboard input events about to be posted in a thread input queue.

For more information, see the [*LowLevelKeyboardProc*](#) callback function.

WH_KEYBOARD

The **WH_KEYBOARD** hook enables an application to monitor message traffic for [WM_KEYDOWN](#) and [WM_KEYUP](#) messages about to be returned by the [GetMessage](#) or [PeekMessage](#) function. You can use the **WH_KEYBOARD** hook to monitor keyboard input posted to a message queue.

For more information, see the [*KeyboardProc*](#) callback function.

WH_MOUSE_LL

The **WH_MOUSE_LL** hook enables you to monitor mouse input events about to be posted in a thread input queue.

For more information, see the [*LowLevelMouseProc*](#) callback function.

WH_MOUSE

The **WH_MOUSE** hook enables you to monitor mouse messages about to be returned by the [GetMessage](#) or [PeekMessage](#) function. You can use the **WH_MOUSE** hook to monitor mouse input posted to a message queue.

For more information, see the [MouseProc](#) callback function.

WH_MSGFILTER and WH_SYSMSGFILTER

The **WH_MSGFILTER** and **WH_SYSMSGFILTER** hooks enable you to monitor messages about to be processed by a menu, scroll bar, message box, or dialog box, and to detect when a different window is about to be activated as a result of the user's pressing the ALT+TAB or ALT+ESC key combination. The **WH_MSGFILTER** hook can only monitor messages passed to a menu, scroll bar, message box, or dialog box created by the application that installed the hook procedure. The **WH_SYSMSGFILTER** hook monitors such messages for all applications.

The **WH_MSGFILTER** and **WH_SYSMSGFILTER** hooks enable you to perform message filtering during modal loops that is equivalent to the filtering done in the main message loop. For example, an application often examines a new message in the main loop between the time it retrieves the message from the queue and the time it dispatches the message, performing special processing as appropriate. However, during a modal loop, the system retrieves and dispatches messages without allowing an application the chance to filter the messages in its main message loop. If an application installs a **WH_MSGFILTER** or **WH_SYSMSGFILTER** hook procedure, the system calls the procedure during the modal loop.

An application can call the **WH_MSGFILTER** hook directly by calling the [CallMsgFilter](#) function. By using this function, the application can use the same code to filter messages during modal loops as it uses in the main message loop. To do so, encapsulate the filtering operations in a **WH_MSGFILTER** hook procedure and call [CallMsgFilter](#) between the calls to the [GetMessage](#) and [DispatchMessage](#) functions.

syntax

```
while (GetMessage(&msg, (HWND) NULL, 0, 0))
{
    if (!CallMsgFilter(&qmsg, 0))
        DispatchMessage(&qmsg);
}
```

The last argument of [CallMsgFilter](#) is simply passed to the hook procedure; you can enter any value. The hook procedure, by defining a constant such as **MSGF_MAINLOOP**, can use this value to determine where the procedure was called from.

For more information, see the [MessageProc](#) and [SysMsgProc](#) callback functions.

WH_SHELL

A shell application can use the **WH_SHELL** hook to receive important notifications. The system calls a **WH_SHELL** hook procedure when the shell application is about to be activated and when a top-level window is created or destroyed.

Note that custom shell applications do not receive **WH_SHELL** messages. Therefore, any application that registers itself as the default shell must call the [SystemParametersInfo](#) function before it (or any other application) can receive **WH_SHELL** messages. This function must be called with **SPI_SETMINIMIZEDMETRICS** and a [MINIMIZEDMETRICS](#) structure. Set the **iArrange** member of this structure to **ARW_HIDE**.

For more information, see the [*ShellProc*](#) callback function.

Feedback

Was this page helpful?



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

Using Hooks

Article • 01/07/2021

The following code examples demonstrate how to perform the following tasks associated with hooks:

- [Installing and Releasing Hook Procedures](#)
- [Monitoring System Events](#)

Installing and Releasing Hook Procedures

You can install a hook procedure by calling the [SetWindowsHookEx](#) function and specifying the type of hook calling the procedure, whether the procedure should be associated with all threads in the same desktop as the calling thread or with a particular thread, and a pointer to the procedure entry point.

You must place a global hook procedure in a DLL separate from the application installing the hook procedure. The installing application must have the handle to the DLL module before it can install the hook procedure. To retrieve a handle to the DLL module, call the [LoadLibrary](#) function with the name of the DLL. After you have obtained the handle, you can call the [GetProcAddress](#) function to retrieve a pointer to the hook procedure. Finally, use [SetWindowsHookEx](#) to install the hook procedure address in the appropriate hook chain. [SetWindowsHookEx](#) passes the module handle, a pointer to the hook-procedure entry point, and 0 for the thread identifier, indicating that the hook procedure should be associated with all threads in the same desktop as the calling thread. This sequence is shown in the following example.

syntax

```
HOOKPROC hkprcSysMsg;
static HINSTANCE hinstDLL;
static HHOOK hhookSysMsg;

hinstDLL = LoadLibrary(TEXT("c:\\myapp\\sysmsg.dll"));
hkprcSysMsg = (HOOKPROC)GetProcAddress(hinstDLL, "SysMessageProc");

hhookSysMsg = SetWindowsHookEx(
    WH_SYSMSGFILTER,
    hkprcSysMsg,
    hinstDLL,
    0);
```

You can release a thread-specific hook procedure (remove its address from the hook chain) by calling the [UnhookWindowsHookEx](#) function, specifying the handle to the hook procedure to release. Release a hook procedure as soon as your application no longer needs it.

You can release a global hook procedure by using [UnhookWindowsHookEx](#), but this function does not free the DLL containing the hook procedure. This is because global hook procedures are called in the process context of every application in the desktop, causing an implicit call to the [LoadLibrary](#) function for all of those processes. Because a call to the [FreeLibrary](#) function cannot be made for another process, there is then no way to free the DLL. The system eventually frees the DLL after all processes explicitly linked to the DLL have either terminated or called [FreeLibrary](#) and all processes that called the hook procedure have resumed processing outside the DLL.

An alternative method for installing a global hook procedure is to provide an installation function in the DLL, along with the hook procedure. With this method, the installing application does not need the handle to the DLL module. By linking with the DLL, the application gains access to the installation function. The installation function can supply the DLL module handle and other details in the call to [SetWindowsHookEx](#). The DLL can also contain a function that releases the global hook procedure; the application can call this hook-releasing function when terminating.

Monitoring System Events

The following example uses a variety of thread-specific hook procedures to monitor the system for events affecting a thread. It demonstrates how to process events for the following types of hook procedures:

- **WH_CALLWNDPROC**
- **WH_CBT**
- **WH_DEBUG**
- **WH_GETMESSAGE**
- **WH_KEYBOARD**
- **WH_MOUSE**
- **WH_MSGFILTER**

The user can install and remove a hook procedure by using the menu. When a hook procedure is installed and an event that is monitored by the procedure occurs, the procedure writes information about the event to the client area of the application's main window.

```

#include <windows.h>
#include <strsafe.h>
#include "app.h"

#pragma comment( lib, "user32.lib")
#pragma comment( lib, "gdi32.lib")

#define NUMHOOKS 7

// Global variables

typedef struct _MYHOOKDATA
{
    int nType;
    HOOKPROC hkprc;
    HHOOK hhook;
} MYHOOKDATA;

MYHOOKDATA myhookdata[NUMHOOKS];

HWND gh_hwndMain;

// Hook procedures

LRESULT WINAPI CallWndProc(int, WPARAM, LPARAM);
LRESULT WINAPI CBTProc(int, WPARAM, LPARAM);
LRESULT WINAPI DebugProc(int, WPARAM, LPARAM);
LRESULT WINAPI GetMsgProc(int, WPARAM, LPARAM);
LRESULT WINAPI KeyboardProc(int, WPARAM, LPARAM);
LRESULT WINAPI MouseProc(int, WPARAM, LPARAM);
LRESULT WINAPI MessageProc(int, WPARAM, LPARAM);

void LookUpTheMessage(PMSG, LPTSTR);

LRESULT WINAPI MainWndProc(HWND hwndMain, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static BOOL afHooks[NUMHOOKS];
    int index;
    static HMENU hmenu;

    gh_hwndMain = hwndMain;

    switch (uMsg)
    {
        case WM_CREATE:

            // Save the menu handle

            hmenu = GetMenu(hwndMain);

            // Initialize structures with hook data. The menu-item
            identifiers are
            // defined as 0 through 6 in the header file app.h. They can be

```

```

used to
    // identify array elements both here and during the WM_COMMAND
message.

myhookdata[IDM_CALLWNDPROC].nType = WH_CALLWNDPROC;
myhookdata[IDM_CALLWNDPROC].hkprc = CallWndProc;
myhookdata[IDM_CBT].nType = WH_CBT;
myhookdata[IDM_CBT].hkprc = CBTProc;
myhookdata[IDM_DEBUG].nType = WH_DEBUG;
myhookdata[IDM_DEBUG].hkprc = DebugProc;
myhookdata[IDM_GETMESSAGE].nType = WH_GETMESSAGE;
myhookdata[IDM_GETMESSAGE].hkprc = GetMsgProc;
myhookdata[IDM_KEYBOARD].nType = WH_KEYBOARD;
myhookdata[IDM_KEYBOARD].hkprc = KeyboardProc;
myhookdata[IDM_MOUSE].nType = WH_MOUSE;
myhookdata[IDM_MOUSE].hkprc = MouseProc;
myhookdata[IDM_MSGFILTER].nType = WH_MSGFILTER;
myhookdata[IDM_MSGFILTER].hkprc = MessageProc;

// Initialize all flags in the array to FALSE.

memset(afHooks, FALSE, sizeof(afHooks));

return 0;

case WM_COMMAND:
    switch (LOWORD(wParam))
    {
        // The user selected a hook command from the menu.

        case IDM_CALLWNDPROC:
        case IDM_CBT:
        case IDM_DEBUG:
        case IDM_GETMESSAGE:
        case IDM_KEYBOARD:
        case IDM_MOUSE:
        case IDM_MSGFILTER:

            // Use the menu-item identifier as an index
            // into the array of structures with hook data.

            index = LOWORD(wParam);

            // If the selected type of hook procedure isn't
            // installed yet, install it and check the
            // associated menu item.

            if (!afHooks[index])
            {
                myhookdata[index].hhook = SetWindowsHookEx(
                    myhookdata[index].nType,
                    myhookdata[index].hkprc,
                    (HINSTANCE) NULL, GetCurrentThreadId());
                CheckMenuItem(hmenu, index,
                    MF_BYCOMMAND | MF_CHECKED);
            }
        }
    }
}

```

```

        afHooks[index] = TRUE;
    }

    // If the selected type of hook procedure is
    // already installed, remove it and remove the
    // check mark from the associated menu item.

    else
    {
        UnhookWindowsHookEx(myhookdata[index].hhook);
        CheckMenuItem(hmenu, index,
                      MF_BYCOMMAND | MF_UNCHECKED);
        afHooks[index] = FALSE;
    }

    default:
        return (DefWindowProc(hwndMain, uMsg, wParam,
                             lParam));
    }
    break;

    //
    // Process other messages.
    //

    default:
        return DefWindowProc(hwndMain, uMsg, wParam, lParam);
    }
    return NULL;
}

//*****************************************************************************
WH_CALLWNDPROC hook procedure
*****
```

LRESULT WINAPI CallWndProc(int nCode, WPARAM wParam, LPARAM lParam)

{

CHAR szCWPBuf[256];
CHAR szMsg[16];
HDC hdc;
static int c = 0;
size_t cch;
HRESULT hResult;

if (nCode < 0) // do not process message
 return CallNextHookEx(myhookdata[IDM_CALLWNDPROC].hhook, nCode,
wParam, lParam);

// Call an application-defined function that converts a message
// constant to a string and copies it to a buffer.

LookUpTheMessage((PMSG) lParam, szMsg);

hdc = GetDC(gh_hwndMain);

```

switch (nCode)
{
    case HC_ACTION:
        hResult = StringCchPrintf(szCWPBuf, 256/sizeof(TCHAR),
            "CALLWNDPROC - tsk: %ld, msg: %s, %d times    ",
            wParam, szMsg, c++);
        if (FAILED(hResult))
        {
            // TODO: writer error handler
        }
        hResult = StringCchLength(szCWPBuf, 256/sizeof(TCHAR), &cch);
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        TextOut(hdc, 2, 15, szCWPBuf, cch);
        break;

    default:
        break;
}

ReleaseDC(gh_hwndMain, hdc);

return CallNextHookEx(myhookdata[IDM_CALLWNDPROC].hhook, nCode, wParam,
lParam);
}

/*********************************************
WH_GETMESSAGE hook procedure
********************************************/


LRESULT CALLBACK GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    CHAR szMSGBuf[256];
    CHAR szRem[16];
    CHAR szMsg[16];
    HDC hdc;
    static int c = 0;
    size_t cch;
    HRESULT hResult;

    if (nCode < 0) // do not process message
        return CallNextHookEx(myhookdata[IDM_GETMESSAGE].hhook, nCode,
            wParam, lParam);

    switch (nCode)
    {
        case HC_ACTION:
            switch (wParam)
            {
                case PM_REMOVE:
                    hResult = StringCchCopy(szRem, 16/sizeof(TCHAR),
"PM_REMOVE");
                    if (FAILED(hResult))

```

```

    {
        // TODO: write error handler
    }
    break;

    case PM_NOREMOVE:
        hResult = StringCchCopy(szRem, 16/sizeof(TCHAR),
"PM_NOREMOVE");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    default:
        hResult = StringCchCopy(szRem, 16/sizeof(TCHAR),
"Unknown");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;
    }

    // Call an application-defined function that converts a
    // message constant to a string and copies it to a
    // buffer.

    LookUpTheMessage((PMSG) lParam, szMsg);

    hdc = GetDC(gh_hwndMain);
    hResult = StringCchPrintf(szMSGBuf, 256/sizeof(TCHAR),
        "GETMESSAGE - wParam: %s, msg: %s, %d times    ",
        szRem, szMsg, c++);
    if (FAILED(hResult))
    {
        // TODO: write error handler
    }
    hResult = StringCchLength(szMSGBuf, 256/sizeof(TCHAR), &cch);
    if (FAILED(hResult))
    {
        // TODO: write error handler
    }
    TextOut(hdc, 2, 35, szMSGBuf, cch);
    break;

    default:
        break;
}

ReleaseDC(gh_hwndMain, hdc);

return CallNextHookEx(myhookdata[IDM_GETMESSAGE].hhook, nCode, wParam,
lParam);
}

```

```

/*****WH_DEBUG hook procedure*****
*****WH_DEBUG hook procedure*****/

LRESULT CALLBACK DebugProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    CHAR szBuf[128];
    HDC hdc;
    static int c = 0;
    size_t cch;
    HRESULT hResult;

    if (nCode < 0) // do not process message
        return CallNextHookEx(myhookdata[IDM_DEBUG].hhook, nCode,
            wParam, lParam);

    hdc = GetDC(gh_hwndMain);

    switch (nCode)
    {
        case HC_ACTION:
            hResult = StringCchPrintf(szBuf, 128/sizeof(TCHAR),
                "DEBUG - nCode: %d, tsk: %ld, %d times    ",
                nCode,wParam, c++);
            if (FAILED(hResult))
            {
                // TODO: write error handler
            }
            hResult = StringCchLength(szBuf, 128/sizeof(TCHAR), &cch);
            if (FAILED(hResult))
            {
                // TODO: write error handler
            }
            TextOut(hdc, 2, 55, szBuf, cch);
            break;

        default:
            break;
    }

    ReleaseDC(gh_hwndMain, hdc);

    return CallNextHookEx(myhookdata[IDM_DEBUG].hhook, nCode, wParam,
        lParam);
}

/*****WH_CBT hook procedure*****
*****WH_CBT hook procedure*****/

LRESULT CALLBACK CBTProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    CHAR szBuf[128];
    CHAR szCode[128];

```

```
HDC hdc;
static int c = 0;
size_t cch;
HRESULT hResult;

if (nCode < 0) // do not process message
    return CallNextHookEx(myhookdata[IDM_CBT].hhook, nCode, wParam,
        lParam);

hdc = GetDC(gh_hwndMain);

switch (nCode)
{
    case HCBT_ACTIVATE:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_ACTIVATE");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_CLICKSKIPPED:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_CLICKSKIPPED");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_CREATEWND:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_CREATEWND");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_DESTROYWND:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_DESTROYWND");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_KEYSKIPPED:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_KEYSKIPPED");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;
}
```

```

        }

        break;

    case HCBT_MINMAX:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_MINMAX");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_MOVESIZE:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_MOVESIZE");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_QS:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR), "HCBT_QS");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_SETFOCUS:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_SETFOCUS");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    case HCBT_SYSCOMMAND:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR),
"HCBT_SYSCOMMAND");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;

    default:
        hResult = StringCchCopy(szCode, 128/sizeof(TCHAR), "Unknown");
        if (FAILED(hResult))
        {
            // TODO: write error handler
        }
        break;
}

```

```

    hResult = StringCchPrintf(szBuf, 128/sizeof(TCHAR), "CBT - nCode: %s,
tsk: %ld, %d times    ",
                           szCode, wParam, c++);
if (FAILED(hResult))
{
// TODO: write error handler
}
hResult = StringCchLength(szBuf, 128/sizeof(TCHAR), &cch);
if (FAILED(hResult))
{
// TODO: write error handler
}
TextOut(hdc, 2, 75, szBuf, cch);
ReleaseDC(gh_hwndMain, hdc);

return CallNextHookEx(myhookdata[IDM_CBT].hhook, nCode, wParam, lParam);
}

/*****
 * WH_MOUSE hook procedure
 *****/
LRESULT CALLBACK MouseProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    CHAR szBuf[128];
    CHAR szMsg[16];
    HDC hdc;
    static int c = 0;
    size_t cch;
    HRESULT hResult;

    if (nCode < 0) // do not process the message
        return CallNextHookEx(myhookdata[IDM_MOUSE].hhook, nCode,
                           wParam, lParam);

    // Call an application-defined function that converts a message
    // constant to a string and copies it to a buffer.

    LookUpTheMessage((PMSG) lParam, szMsg);

    hdc = GetDC(gh_hwndMain);
    hResult = StringCchPrintf(szBuf, 128/sizeof(TCHAR),
                           "MOUSE - nCode: %d, msg: %s, x: %d, y: %d, %d times    ",
                           nCode, szMsg, LOWORD(lParam), HIWORD(lParam), c++);
if (FAILED(hResult))
{
// TODO: write error handler
}
hResult = StringCchLength(szBuf, 128/sizeof(TCHAR), &cch);
if (FAILED(hResult))
{
// TODO: write error handler
}
TextOut(hdc, 2, 95, szBuf, cch);
ReleaseDC(gh_hwndMain, hdc);
}

```

```

        return CallNextHookEx(myhookdata[IDM_MOUSE].hhook, nCode, wParam,
lParam);
}

/*********************************************
 WH_KEYBOARD hook procedure
***** */

```

LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam)

- {
- CHAR szBuf[128];
- HDC hdc;
- static int c = 0;
- size_t cch;
- HRESULT hResult;
-
- if (nCode < 0) // do not process message
- return CallNextHookEx(myhookdata[IDM_KEYBOARD].hhook, nCode,
- wParam, lParam);
-
- hdc = GetDC(gh_hwndMain);
- hResult = StringCchPrintf(szBuf, 128/sizeof(TCHAR), "KEYBOARD - nCode:
- %d, vk: %d, %d times ", nCode, wParam, c++);
- if (FAILED(hResult))
- {
- // TODO: write error handler
- }
- hResult = StringCchLength(szBuf, 128/sizeof(TCHAR), &cch);
- if (FAILED(hResult))
- {
- // TODO: write error handler
- }
- TextOut(hdc, 2, 115, szBuf, cch);
- ReleaseDC(gh_hwndMain, hdc);
-
- return CallNextHookEx(myhookdata[IDM_KEYBOARD].hhook, nCode, wParam,
- lParam);
- }
-
- *****
- WH_MSGFILTER hook procedure
- ***** */

LRESULT CALLBACK MessageProc(int nCode, WPARAM wParam, LPARAM lParam)

- {
- CHAR szBuf[128];
- CHAR szMsg[16];
- CHAR szCode[32];
- HDC hdc;
- static int c = 0;
- size_t cch;
- HRESULT hResult;
-
- if (nCode < 0) // do not process message

```

        return CallNextHookEx(myhookdata[IDM_MSGFILTER].hhook, nCode,
                           wParam, lParam);

    switch (nCode)
    {
        case MSGF_DIALOGBOX:
            hResult = StringCchCopy(szCode, 32/sizeof(TCHAR),
"MSGF_DIALOGBOX");
                if (FAILED(hResult))
                {
                    // TODO: write error handler
                }
            break;

        case MSGF_MENU:
            hResult = StringCchCopy(szCode, 32/sizeof(TCHAR), "MSGF_MENU");
                if (FAILED(hResult))
                {
                    // TODO: write error handler
                }
            break;

        case MSGF_SCROLLBAR:
            hResult = StringCchCopy(szCode, 32/sizeof(TCHAR),
"MSGF_SCROLLBAR");
                if (FAILED(hResult))
                {
                    // TODO: write error handler
                }
            break;

        default:
            hResult = StringCchPrintf(szCode, 128/sizeof(TCHAR), "Unknown:
%d", nCode);
            if (FAILED(hResult))
            {
                // TODO: write error handler
            }
            break;
    }

    // Call an application-defined function that converts a message
    // constant to a string and copies it to a buffer.

    LookUpTheMessage((PMSG) lParam, szMsg);

    hdc = GetDC(gh_hwndMain);
    hResult = StringCchPrintf(szBuf, 128/sizeof(TCHAR),
        "MSGFILTER nCode: %s, msg: %s, %d times    ",
        szCode, szMsg, c++);
    if (FAILED(hResult))
    {
        // TODO: write error handler
    }
    hResult = StringCchLength(szBuf, 128/sizeof(TCHAR), &cch);
}

```

```
if (FAILED(hResult))
{
// TODO: write error handler
}
TextOut(hdc, 2, 135, szBuf, cch);
ReleaseDC(gh_hwndMain, hdc);

return CallNextHookEx(myhookdata[IDM_MSGFILTER].hhook, nCode, wParam,
lParam);
}
```

Feedback

Was this page helpful?

 Yes

 No

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

Hook Reference

Article • 04/27/2021

- [Hook Functions](#)
 - [Hook Notifications](#)
 - [Hook Structures](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

Hook Functions

Article • 04/27/2021

- [CallIMsgFilter](#)
- [CallNextHookEx](#)
- [*CallWndProc*](#)
- [*CallWndRetProc*](#)
- [*CBTProc*](#)
- [DebugProc](#)
- [*ForegroundIdleProc*](#)
- [GetMsgProc](#)
- [*JournalPlaybackProc*](#)
- [*JournalRecordProc*](#)
- [*KeyboardProc*](#)
- [*LowLevelKeyboardProc*](#)
- [*LowLevelMouseProc*](#)
- [*MessageProc*](#)
- [*MouseProc*](#)
- [SetWindowsHookEx](#)
- [*ShellProc*](#)
- [SysMsgProc](#)
- [UnhookWindowsHookEx](#)

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](#)

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](#)

CallWndProc callback function

Article • 06/30/2023

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function before calling the window procedure to process a message sent to the thread.

The **HOOKPROC** type defines a pointer to this callback function. *CallWndProc* is a placeholder for the application-defined or library-defined function name.

Syntax

```
C++  
  
LRESULT CALLBACK CallWndProc(  
    _In_ int nCode,  
    _In_ WPARAM wParam,  
    _In_ LPARAM lParam  
);
```

Parameters

- *nCode* [in]

Type: **int**

Specifies whether the hook procedure must process the message. If *nCode* is **HC_ACTION**, the hook procedure must process the message. If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and must return the value returned by [CallNextHookEx](#).

- *wParam* [in]

Type: **WPARAM**

Specifies whether the message was sent by the current thread. If the message was sent by the current thread, it is nonzero; otherwise, it is zero.

- *lParam* [in]

Type: **LPARAM**

A pointer to a [CWPSTRUCT](#) structure that contains details about the message.

Return value

Type: ****

Type: LRESULT

If *nCode* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *nCode* is greater than or equal to zero, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_CALLWNDPROC](#) 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

The *CallWndProc* hook procedure can examine the message, but it cannot modify it. After the hook procedure returns control to the system, the message is passed to the window procedure.

An application installs the hook procedure by specifying the [WH_CALLWNDPROC](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) 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

[CallNextHookEx](#)

[CWPSTRUCT](#)

[SendMessage](#)

[SetWindowsHookExA](#)/[SetWindowsHookExW](#)

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

CBTProc callback function

Article • 06/30/2023

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the keyboard focus; or before synchronizing with the system message queue. A computer-based training (CBT) application uses this hook procedure to receive useful notifications from the system.

Syntax

C++

```
LRESULT CALLBACK CBTProc(  
    _In_ int      nCode,  
    _In_ WPARAM  wParam,  
    _In_ LPARAM  lParam  
) ;
```

Parameters

- *nCode* [in]

Type: [int](#)

The code that the hook procedure uses to determine how to process the message.

If *nCode* is less than zero, the hook procedure must pass the message to the

[CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#). This parameter can be one of the following values.

Value	Meaning
HCBT_ACTIVATE 5	The system is about to activate a window.
HCBT_CLICKSKIPPED 6	The system has removed a mouse message from the system message queue. Upon receiving this hook code, a CBT application must install a WH_JOURNALPLAYBACK hook procedure in response to the mouse message.

Value	Meaning
HCBT_CREATEWND 3	<p>A window is about to be created. The system calls the hook procedure before sending the WM_CREATE or WM_NCCREATE message to the window. If the hook procedure returns a nonzero value, the system destroys the window; the CreateWindow function returns <code>NULL</code>, but the WM_DESTROY message is not sent to the window. If the hook procedure returns zero, the window is created normally.</p> <p>At the time of the HCBT_CREATEWND notification, the window has been created, but its final size and position may not have been determined and its parent window may not have been established. It is possible to send messages to the newly created window, although it has not yet received WM_NCCREATE or WM_CREATE messages. It is also possible to change the position in the z-order of the newly created window by modifying the <code>hwndInsertAfter</code> member of the CBT_CREATEWND structure.</p>
HCBT_DESTROYWND 4	A window is about to be destroyed.
HCBT_KEYSKIPPED 7	The system has removed a keyboard message from the system message queue. Upon receiving this hook code, a CBT application must install a WH_JOURNALPLAYBACK hook procedure in response to the keyboard message.
HCBT_MINMAX 1	A window is about to be minimized or maximized.
HCBT_MOVESIZE 0	A window is about to be moved or sized.
HCBT_QS 2	The system has retrieved a WM_QUEUESYNC message from the system message queue.
HCBT_SETFOCUS 9	A window is about to receive the keyboard focus.
HCBT_SYSCOMMAND 8	A system command is about to be carried out. This allows a CBT application to prevent task switching by means of hot keys.

- *wParam* [in]
Type: **WPARAM**

Depends on the *nCode* parameter. For details, see the following Remarks section.

- *lParam* [in]
Type: **LPARAM**

Depends on the *nCode* parameter. For details, see the following Remarks section.

Return value

Type: ****

Type: LRESULT

The value returned by the hook procedure determines whether the system allows or prevents one of these operations. For operations corresponding to the following CBT hook codes, the return value must be 0 to allow the operation, or 1 to prevent it.

- **HCBT_ACTIVATE**
- **HCBT_CREATEWND**
- **HCBT_DESTROYWND**
- **HCBT_MINMAX**
- **HCBT_MOVESIZE**
- **HCBT_SETFOCUS**
- **HCBT_SYSCOMMAND**

For operations corresponding to the following CBT hook codes, the return value is ignored.

- **HCBT_CLICKSKIPPED**
- **HCBT_KEYSKIPPED**
- **HCBT_QS**

Remarks

The **HOOKPROC** type defines a pointer to this callback function. *CBTProc* is a placeholder for the application-defined or library-defined function name.

The hook procedure should not install a **WH_JOURNALPLAYBACK** hook procedure except in the situations described in the preceding list of hook codes.

An application installs the hook procedure by specifying the **WH_CBT** hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA/SetWindowsHookExW](#) function.

The following table describes the *wParam* and *lParam* parameters for each **HCBT_** hook code.

Value	wParam	lParam
HCBT_ACTIVATE	Specifies the handle to the window about to be activated.	Specifies a long pointer to a CBTACTIVATESTRUCT structure containing the handle to the active window and specifies whether the activation is changing because of a mouse click.
HCBT_CLICKSKIPPED	Specifies the mouse message removed from the system message queue.	Specifies a long pointer to a MOUSEHOOKSTRUCT structure containing the hit-test code and the handle to the window for which the mouse message is intended.
HCBT_CREATEWND	Specifies the handle to the new window.	Specifies a long pointer to a CBT_CREATEWND structure containing initialization parameters for the window. The parameters include the coordinates and dimensions of the window. By changing these parameters, a <i>CBTProc</i> hook procedure can set the initial size and position of the window.
HCBT_DESTROYWND	Specifies the handle to the window about to be destroyed.	Is undefined and must be set to zero.
HCBT_KEYSKIPPED	Specifies the virtual-key code.	Specifies the repeat count, scan code, key-transition code, previous key state, and context code. The HCBT_KEYSKIPPED value is sent to a <i>CBTProc</i> hook procedure only if a WH_KEYBOARD hook is installed. For more information, see WM_KEYUP or WM_KEYDOWN .
HCBT_MINMAX	Specifies the handle to the window being minimized or maximized.	Specifies, in the low-order word, a show-window value (SW_) specifying the operation. For a list of show-window values, see the ShowWindow . The high-order word is undefined.

Value	wParam	lParam
HCBT_MOVESIZE	Specifies the handle to the window to be moved or sized.	Specifies a long pointer to a RECT structure containing the coordinates of the window. By changing the values in the structure, a <i>CBTProc</i> hook procedure can set the final coordinates of the window.
HCBT_QS	Is undefined and must be zero.	Is undefined and must be zero.
HCBT_SETFOCUS	Specifies the handle to the window gaining the keyboard focus.	Specifies the handle to the window losing the keyboard focus.
HCBT_SYSCOMMAND	Specifies a system-command value (<i>SC_</i>) specifying the system command. For more information about system-command values, see WM_SYSCOMMAND .	Contains the same data as the <i>lParam</i> value of a WM_SYSCOMMAND message: If a system menu command is chosen with the mouse, the low-order word contains the x-coordinate of the cursor, in screen coordinates, and the high-order word contains the y-coordinate; otherwise, the parameter is not used.

For more details, see [Windows Events](#).

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

[CallNextHookEx](#)

[CreateWindowA/CreateWindowW](#)

[SetWindowsHookExA/SetWindowsHookExW](#)

[WM_SYSCOMMAND message](#)

Feedback

Was this page helpful?

 Yes

 No

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

DebugProc function

Article • 06/30/2023

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function before calling the hook procedures associated with any type of hook. The system passes information about the hook to be called to the *DebugProc* hook procedure, which examines the information and determines whether to allow the hook to be called.

The **HOOKPROC** type defines a pointer to this callback function. *DebugProc* is a placeholder for the application-defined or library-defined function name.

Syntax

```
c++  
  
LRESULT CALLBACK DebugProc(  
    _In_ int      nCode,  
    _In_ WPARAM   wParam,  
    _In_ LPARAM   lParam  
);
```

Parameters

- *nCode* [in]

Type: **int**

Specifies whether the hook procedure must process the message. If *nCode* is HC_ACTION, the hook procedure must process the message. If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

- *wParam* [in]

Type: **WPARAM**

The type of hook about to be called. This parameter can be one of the following values.

Value	Meaning
-------	---------

Value	Meaning
WH_CALLWNDPROC 4	Installs a hook procedure that monitors messages sent to a window procedure. For more information, see the description of the [*CallWndProc*](callwndproc.md) hook procedure.
WH_CALLWNDPROCRET 12	Installs a hook procedure that monitors messages that have just been processed by a window procedure. For more information, see the description of the CallWndRetProc hook procedure.
WH_CBT 5	Installs a hook procedure that receives notifications useful to a CBT application. For more information, see the description of the [**CBTProc**](cbtproc.md) hook procedure.
WH_DEBUG 9	Installs a hook procedure useful for debugging other hook procedures. For more information, see the description of the DebugProc hook procedure.
WH_GETMESSAGE 3	Installs a hook procedure that monitors messages posted to a message queue. For more information, see the description of the GetMsgProc hook procedure.
WH_JOURNALPLAYBACK	Installs a hook procedure that posts messages previously recorded by a WH_JOURNALRECORD hook procedure. For more information, see the description of the JournalPlaybackProc hook procedure.
WH_JOURNALRECORD 0	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 description of the JournalRecordProc hook procedure.
WH_KEYBOARD 2	Installs a hook procedure that monitors keystroke messages. For more information, see the description of the KeyboardProc hook procedure.
WH_MOUSE 7	Installs a hook procedure that monitors mouse messages. For more information, see the description of the [*MouseProc*](mouseproc.md) hook procedure.

Value	Meaning
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. The hook procedure monitors these messages only for the application that installed the hook procedure. For more information, see the [*MessageProc*](messageproc.md) hook procedure.
WH_SHELL 10	Installs a hook procedure that receives notifications useful to a Shell application. For more information, see the description of the [*ShellProc*](shellproc.md) hook procedure and the WH_SHELL hook section.
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 system. For more information, see the description of the [*SysMsgProc*](sysmsgproc.md) hook procedure.

- *lParam* [in]

Type: **LPARAM**

A pointer to a [DEBUGHOOKINFO](#) structure that contains the parameters to be passed to the destination hook procedure.

Return value

Type: ****

Type: **LRESULT**

To prevent the system from calling the hook, the hook procedure must return a nonzero value. Otherwise, the hook procedure must call [CallNextHookEx](#).

Remarks

An application installs this hook procedure by specifying the [WH_DEBUG](#) hook type and the pointer to the hook procedure in a call to the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) 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

[CallNextHookEx](#)

CallWndProc

CallWndRetProc

CBTProc

[DEBUGHOOKINFO](#)

GetMsgProc

JournalPlaybackProc

JournalRecordProc

KeyboardProc

MessageProc

MouseProc

[SetWindowsHookExA/SetWindowsHookExW](#)

ShellProc

SysMsgProc

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

ForegroundIdleProc callback function

Article • 06/30/2023

An application-defined or library-defined callback function used with the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) function. The system calls this function whenever the foreground thread is about to become idle.

Syntax

```
C++  
  
DWORD CALLBACK ForegroundIdleProc(  
    _In_ int code,  
    DWORD wParam,  
    LONG lParam  
);
```

Parameters

- *code* [in]

Type: **int**

If *code* is **HC_ACTION**, the hook procedure must process the message. If *code* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

- *wParam*

Type: **DWORD**

This parameter is not used.

- *lParam*

Type: **LONG**

This parameter is not used.

Return value

Type: ****

Type: DWORD

If *code* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *code* is greater than or equal to zero, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_FOREGROUNDIDLE](#) 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

The **HOOKPROC** type defines a pointer to this callback function. *ForegroundIdleProc* is a placeholder for the application-defined or library-defined function name.

An application installs this hook procedure by specifying the [WH_FOREGROUNDIDLE](#) hook type and the pointer to the hook procedure in a call to the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) function.

While processing this callback function, avoid calling any functions that retrieve window messages from the calling thread's message queue. This includes [GetMessage](#), [PeekMessageA](#)/[PeekMessageW](#), modal dialog box, and COM functions. Calling such functions may result in the thread not returning from [GetMessage](#) or [WaitMessage](#) when there are messages in the calling thread's message queue.

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

Reference

[CallNextHookEx](#)

[SetWindowsHookExA](#)/[SetWindowsHookExW](#)

Conceptual

Feedback

Was this page helpful?

 Yes

 No

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

GetMsgProc function

Article • 06/30/2023

-description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function whenever the [GetMessage](#) or [PeekMessageA/PeekMessageW](#) function has retrieved a message from an application message queue. Before returning the retrieved message to the caller, the system passes the message to the hook procedure. Before returning the retrieved message to the caller, the system passes the message to the hook procedure.

The **HOOKPROC** type defines a pointer to this callback function. *GetMsgProc* is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK GetMsgProc(
    _In_ int      code,
    _In_ WPARAM  wParam,
    _In_ LPARAM  lParam
);
```

-parameters

code [in]

Type: **int**

Specifies whether the hook procedure must process the message. If *code* is **HC_ACTION**, the hook procedure must process the message. If *code* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

wParam [in]

Type: **WPARAM**

Specifies whether the message has been removed from the queue. This parameter can be one of the following values.

Value	Meaning
PM_NOREMOVE 0x0000	The message has not been removed from the queue. (An application called the PeekMessage function, specifying the PM_NOREMOVE flag.)
PM_REMOVE 0x0001	The message has been removed from the queue. (An application called GetMessage , or it called the PeekMessage function, specifying the PM_REMOVE flag.)

IParam [in]

Type: **LPARAM**

A pointer to an [MSG](#) structure that contains details about the message.

-returns

If *code* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *code* is greater than or equal to zero, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_GETMESSAGE](#) 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

The [GetMsgProc](#) hook procedure can examine or modify the message.

After the hook procedure returns control to the system, the [GetMessage](#) or [PeekMessageA/PeekMessageW](#) function returns the message, along with any modifications, to the application that originally called it.

An application installs this hook procedure by specifying the [WH_GETMESSAGE](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA/SetWindowsHookExW](#) function.

See also

[CallNextHookEx](#)

[GetMessage](#)

[MSG](#)

[PeekMessage](#)

[SetWindowsHookEx](#)

[Hooks](#)

Feedback

Was this page helpful?

[!\[\]\(1211315d3f8f154a702f4c244ae46241_img.jpg\) Yes](#)

[!\[\]\(f22ca5cc76c9a530e26f3ecc38b85344_img.jpg\) No](#)

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

JournalPlaybackProc function

Article • 06/30/2023

Description

⚠ Warning

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.

An application-defined or library-defined callback function used with the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) function. The function records messages the system removes from the system message queue. Later, an application can use a [JournalPlaybackProc](#) hook procedure to play back the messages.

ⓘ Note

Typically, an application uses this function to play back a series of mouse and keyboard messages recorded previously by the [JournalRecordProc](#) hook procedure. As long as a [JournalPlaybackProc](#) hook procedure is installed, regular mouse and keyboard input is disabled.

The **HOOKPROC** type defines a pointer to this callback function. *JournalRecordProc* is a placeholder for the application-defined or library-defined function name.

```
c++  
  
LRESULT CALLBACK JournalRecordProc(  
    _In_ int     code,  
    WPARAM wParam,  
    _In_ LPARAM lParam  
);
```

Parameters

code [in]

Type: int

A code the hook procedure uses to determine how to process the message.

If *code* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
HC_GETNEXT 1	The hook procedure must copy the current mouse or keyboard message to the EVENTMSG structure pointed to by the <i>lParam</i> parameter.
HC_NOREMOVE 3	An application has called the PeekMessage function with <i>wRemoveMsg</i> set to PM_NOREMOVE , indicating that the message is not removed from the message queue after PeekMessage processing.
HC_NOREMOVE 2	The hook procedure must prepare to copy the next mouse or keyboard message to the EVENTMSG structure pointed to by <i>lParam</i> . Upon receiving the HC_GETNEXT code, the hook procedure must copy the message to the structure.
HC_SYSMODALOFF 5	A system-modal dialog box has been destroyed. The hook procedure must resume playing back the messages.
HC_SYSMODALON 4	A system-modal dialog box is being displayed. Until the dialog box is destroyed, the hook procedure must stop playing back messages.

wParam

Type: WPARAM

This parameter is not used.

lParam

Type: LPARAM

A pointer to an [EVENTMSG](#) structure that represents a message being processed by the hook procedure. This parameter is valid only when the *code* parameter is **HC_GETNEXT**.

Returns

Type: LRESULT

To have the system wait before processing the message, the return value must be the amount of time, in clock ticks, that the system should wait.

(This value can be computed by calculating the difference between the time members in the current and previous input messages.)

To process the message immediately, the return value should be zero. The return value is used only if the hook code is **HC_GETNEXT**; otherwise, it is ignored.

Remarks

A **JournalPlaybackProc** hook procedure should copy an input message to The *lParam* parameter. The message must have been previously recorded by using a **JournalRecordProc** hook procedure, which should not modify the message.

To retrieve the same message over and over, the hook procedure can be called several times with the *code* parameter set to **HC_GETNEXT** without an intervening call with *code* set to **HC_SKIP**.

If *code* is **HC_GETNEXT** and the return value is greater than zero, the system sleeps for the number of milliseconds specified by the return value. When the system continues, it calls the hook procedure again with *code* set to **HC_GETNEXT** to retrieve the same message. The return value from this new call to **JournalPlaybackProc** should be zero; otherwise, the system will go back to sleep for the number of milliseconds specified by the return value, call **JournalPlaybackProc** again, and so on. The system will appear to be not responding.

Unlike most other global hook procedures, the **JournalRecordProc** and **JournalPlaybackProc** hook procedures are always called in the context of the thread that set the hook.

After the hook procedure returns control to the system, the message continues to be processed. If *code* is **HC_SKIP**, the hook procedure must prepare to return the next recorded event message on its next call.

Install the **JournalPlaybackProc** hook procedure by specifying the **WH_JOURNALPLAYBACK** type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

If the user presses CTRL+ESC OR CTRL+ALT+DEL during journal playback, the system stops the playback, unhooks the journal playback procedure, and posts a **WM_CANCELJOURNAL** message to the journaling application.

If the hook procedure returns a message in the range **WM_KEYFIRST** to **WM_KEYLAST**, the following conditions apply:

- The **paramL** member of the **EVENTMSG** structure specifies the virtual key code of the key that was pressed.
- The **paramH** member of the **EVENTMSG** structure specifies the scan code.
- There's no way to specify a repeat count. The event is always taken to represent one key event.

See also

[CallNextHookEx](#)

[EVENTMSG](#)

[JournalRecordProc](#)

[PeekMessage](#)

[SetWindowsHookEx](#)

[WM_CANCELJOURNAL](#)

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

JournalRecordProc function

Article • 07/10/2020

Description

An application-defined or library-defined callback function used with the [SetWindowsHookEx](#) function. The function records messages the system removes from the system message queue. Later, an application can use a [JournalPlaybackProc](#) hook procedure to play back the messages.

The **HOOKPROC** type defines a pointer to this callback function. **JournalRecordProc** is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK JournalRecordProc(
    _In_ int     code,
    WPARAM wParam,
    _In_ LPARAM lParam
);
```

Parameters

code [in]

Type: **int**

Specifies how to process the message. If *code* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#). This parameter can be one of the following values.

Value	Meaning
HC_ACTION 0	The <i>lParam</i> parameter is a pointer to an EVENTMSG structure containing information about a message removed from the system queue. The hook procedure must record the contents of the structure by copying them to a buffer or file.
HC_SYSMODALOFF 5	A system-modal dialog box has been destroyed. The hook procedure must resume recording.

Value	Meaning
HC_SYSMODALON 4	A system-modal dialog box is being displayed. Until the dialog box is destroyed, the hook procedure must stop recording.

wParam

Type: **WPARAM**

This parameter is not used.

lParam [in]

Type: **LPARAM**

A pointer to an **EVENTMSG** structure that contains the message to be recorded.

Returns

Type: **LRESULT**

The return value is ignored.

Remarks

A **JournalRecordProc** hook procedure must copy but not modify the messages. After the hook procedure returns control to the system, the message continues to be processed.

Install the **JournalRecordProc** hook procedure by specifying the [WH_JOURNALRECORD](#) type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

A **JournalRecordProc** hook procedure does not need to live in a dynamic-link library. A **JournalRecordProc** hook procedure can live in the application itself.

Unlike most other global hook procedures, the **JournalRecordProc** and **JournalPlaybackProc** hook procedures are always called in the context of the thread that set the hook.

An application that has installed a **JournalRecordProc** hook procedure should watch for the [VK_CANCEL](#) virtual key code (which is implemented as the CTRL+BREAK key combination on most keyboards). This virtual key code should be interpreted by the

application as a signal that the user wishes to stop journal recording. The application should respond by ending the recording sequence and removing the **JournalRecordProc** hook procedure. Removal is important. It prevents a journaling application from locking up the system by hanging inside a hook procedure.

This role as a signal to stop journal recording means that a CTRL+BREAK key combination cannot itself be recorded. Since the CTRL+C key combination has no such role as a journaling signal, it can be recorded. There are two other key combinations that cannot be recorded: CTRL+ESC and CTRL+ALT+DEL. Those two key combinations cause the system to stop all journaling activities (record or playback), remove all journaling hooks, and post a [WM_CANCELJOURNAL](#) message to the journaling application.

See also

[CallNextHookEx](#)

[EVENTMSG](#)

[JournalPlaybackProc](#)

[SetWindowsHookEx](#)

[WM_CANCELJOURNAL](#)

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

KeyboardProc callback function

Article • 06/30/2023

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function.

The system calls this function whenever an application calls the [GetMessage](#) or [PeekMessageA/PeekMessageW](#) function and there is a keyboard message ([WM_KEYUP](#) or [WM_KEYDOWN](#)) to be processed.

The **HOOKPROC** type defines a pointer to this callback function. *KeyboardProc* is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK KeyboardProc(
    _In_ int     code,
    _In_ WPARAM wParam,
    _In_ LPARAM lParam
);
```

Parameters

code [in]

Type: **int**

A code the hook procedure uses to determine how to process the message.

If *code* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
HC_ACTION 0	The <i>wParam</i> and <i>lParam</i> parameters contain information about a keystroke message.

Value	Meaning
HC_NOREMOVE 3	The <i>wParam</i> and <i>lParam</i> parameters contain information about a keystroke message, and the keystroke message has not been removed from the message queue. (An application called the PeekMessage function, specifying the PM_NOREMOVE flag.)

wParam [in]

Type: **WPARAM**

The [virtual-key code](#) of the key that generated the keystroke message.

lParam [in]

Type: **LPARAM**

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag. For more information about The *lParam* parameter, see [Keystroke Message Flags](#). The following table describes the bits of this value.

Bits	Description
0- 15	The repeat count. The value is the number of times the keystroke is repeated as a result of the user's holding down the key.
16- 23	The scan code. The value depends on the OEM.
24	Indicates 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.
25- 28	Reserved.
29	The context code. The value is 1 if the ALT key is down; otherwise, it is 0.
30	The previous key state. The value is 1 if the key is down before the message is sent; it is 0 if the key is up.
31	The transition state. The value is 0 if the key is being pressed and 1 if it is being released.

Returns

Type: **LRESULT**

If *code* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *code* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_KEYBOARD](#) hooks will not receive hook notifications and may behave incorrectly as a result.

If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the [WH_KEYBOARD](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA/SetWindowsHookExW](#) function.

This hook may be called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

See also

- [CallNextHookEx](#)
- [GetMessage](#)
- [PeekMessage](#)
- [SetWindowsHookEx](#)
- [WM_KEYUP](#)
- [WM_KEYDOWN](#)
- [Hooks](#)
- [Keyboard Input \(Keyboard and Mouse Input\)](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful?



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

LowLevelKeyboardProc function

Article • 06/30/2023

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function every time a new keyboard input event is about to be posted into a thread input queue.

Note

When this callback function is called in response to a change in the state of a key, the callback function is called before the asynchronous state of the key is updated. Consequently, the asynchronous state of the key cannot be determined by calling [GetAsyncKeyState](#) from within the callback function.

The **HOOKPROC** type defines a pointer to this callback function. **LowLevelKeyboardProc** is a placeholder for the application-defined or library-defined function name.

c++

```
LRESULT CALLBACK LowLevelKeyboardProc(
    _In_ int nCode,
    _In_ WPARAM wParam,
    _In_ LPARAM lParam
);
```

Parameters

code [in]

Type: **int**

A code the hook procedure uses to determine how to process the message.

If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
HC_ACTION 0	The <i>wParam</i> and <i>lParam</i> parameters contain information about a keyboard message.

wParam [in]

Type: **WPARAM**

The identifier of the keyboard message.

This parameter can be one of the following messages: [WM_KEYDOWN](#), [WM_KEYUP](#), [WM_SYSKEYDOWN](#), or [WM_SYSKEYUP](#).

lParam [in]

Type: **LPARAM**

A pointer to a [KBDLLHOOKSTRUCT](#) structure.

Returns

Type: **LRESULT**

If *nCode* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_KEYBOARD_LL](#) hooks will not receive hook notifications and may behave incorrectly as a result.

If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the [WH_KEYBOARD_LL](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) function.

This hook is called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

The keyboard input can come from the local keyboard driver or from calls to the [keybd_event](#) function. If the input comes from a call to [keybd_event](#), the input was "injected". However, the [WH_KEYBOARD_LL](#) hook is not injected into another process. Instead, the context switches back to the process that installed the hook and it is called in its original context. Then the context switches back to the application that generated the event.

The hook procedure should process a message in less time than the data entry specified in the [LowLevelHooksTimeout](#) value in the following registry key:

```
HKEY_CURRENT_USER**\Control Panel**\Desktop
```

The value is in milliseconds. If the hook procedure times out, the system passes the message to the next hook. However, on Windows 7 and later, the hook is silently removed without being called. There is no way for the application to know whether the hook is removed.

Windows 10 version 1709 and later The maximum timeout value the system allows is 1000 milliseconds (1 second). The system will default to using a 1000 millisecond timeout if the [LowLevelHooksTimeout](#) value is set to a value larger than 1000.

Note

Debug hooks cannot track this type of low level keyboard hooks. If the application must use low level hooks, it should run the hooks on a dedicated thread that passes the work off to a worker thread and then immediately returns. In most cases where the application needs to use low level hooks, it should monitor raw input instead. This is because raw input can asynchronously monitor mouse and keyboard messages that are targeted for other threads more effectively than low level hooks can. For more information on raw input, see [Raw Input](#).

See also

[CallNextHookEx](#)

[KBDLLHOOKSTRUCT](#)

[keybd_event](#)

[SetWindowsHookEx](#)

[WM_KEYDOWN](#)

[WM_KEYUP](#)

[WM_SYSKEYDOWN](#)

[WM_SYSKEYUP](#)

[Hooks](#)

[About Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

LowLevelMouseProc function

Article • 06/30/2023

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function every time a new mouse input event is about to be posted into a thread input queue.

The **HOOKPROC** type defines a pointer to this callback function. *LowLevelMouseProc* is a placeholder for the application-defined or library-defined function name.

LowLevelMouseProc is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK LowLevelMouseProc(
    _In_ int nCode,
    _In_ WPARAM wParam,
    _In_ LPARAM lParam
);
```

Parameters

nCode [in]

Type: **int**

A code the hook procedure uses to determine how to process the message.

If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
HC_ACTION 0	The <i>wParam</i> and <i>lParam</i> parameters contain information about a mouse message.

wParam [in]

Type: WPARAM

The identifier of the mouse message.

This parameter can be one of the following messages: [WM_LBUTTONDOWN](#), [WM_LBUTTONUP](#), [WM_MOUSEMOVE](#), [WM_MOUSEWHEEL](#), [WM_RBUTTONDOWN](#) or [WM_RBUTTONUP](#).

lParam [in]

Type: LPARAM

A pointer to an [MSLLHOOKSTRUCT](#) structure.

Returns

Type: LRESULT

If *nCode* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_MOUSE_LL](#) hooks will not receive hook notifications and may behave incorrectly as a result.

If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the [WH_MOUSE_LL](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA/SetWindowsHookExW](#) function.

This hook is called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

The mouse input can come from the local mouse driver or from calls to the [mouse_event](#) function. If the input comes from a call to [mouse_event](#), the input was "injected". However, the [WH_MOUSE_LL](#) hook is not injected into another process. Instead, the context switches back to the process that installed the hook and it is called in its original context. Then the context switches back to the application that generated the event.

The hook procedure should process a message in less time than the data entry specified in the [LowLevelHooksTimeout](#) value in the following registry key:

```
HKEY_CURRENT_USER\Control Panel\Desktop
```

The value is in milliseconds. If the hook procedure times out, the system passes the message to the next hook. However, on Windows 7 and later, the hook is silently removed without being called. There is no way for the application to know whether the hook is removed.

Windows 10 version 1709 and later The maximum timeout value the system allows is 1000 milliseconds (1 second). The system will default to using a 1000 millisecond timeout if the [LowLevelHooksTimeout](#) value is set to a value larger than 1000.

Note

Debug hooks cannot track this type of low level mouse hooks. If the application must use low level hooks, it should run the hooks on a dedicated thread that passes the work off to a worker thread and then immediately returns. In most cases where the application needs to use low level hooks, it should monitor raw input instead. This is because raw input can asynchronously monitor mouse and keyboard messages that are targeted for other threads more effectively than low level hooks can. For more information on raw input, see [Raw Input](#).

See also

[CallNextHookEx](#)

[mouse_event](#)

[KBDLLHOOKSTRUCT](#)

[MSLLHOOKSTRUCT](#)

[SetWindowsHookEx](#)

[WM_LBUTTONDOWN](#)

[WM_LBUTTONUP](#)

[WM_MOUSEMOVE](#)

[WM_MOUSEWHEEL](#)

[WM_RBUTTONDOWN](#)

[WM_RBUTTONUP](#)

[Hooks](#)

[About Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

MessageProc function

Article • 06/30/2023

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function after an input event occurs in a dialog box, message box, menu, or scroll bar, but before the message generated by the input event is processed. The hook procedure can monitor messages for a dialog box, message box, menu, or scroll bar created by a particular application or all applications.

The **HOOKPROC** type defines a pointer to this callback function. *MessageProc* is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK MessageProc(
    _In_ int      code,
    _In_ WPARAM  wParam,
    _In_ LPARAM  lParam
);
```

Parameters

code [in]

Type: **int**

The type of input event that generated the message.

If *code* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
-------	---------

Value	Meaning
MSGF_DDEMGR 0x8001	The input event occurred while the Dynamic Data Exchange Management Library (DDEML) was waiting for a synchronous transaction to finish. For more information about DDEML, see Dynamic Data Exchange Management Library .
MSGF_DIALOGBOX 0	The input event occurred in a message box or dialog box.
MSGF_MENU 2	The input event occurred in a menu.
MSGF_SCROLLBAR 5	The input event occurred in a scroll bar.

wParam

Type: **WPARAM**

This parameter is not used.

lParam [in]

Type: **LPARAM**

A pointer to an [MSG](#) structure.

Returns

Type: **LRESULT**

If *code* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *code* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_MSGFILTER](#) hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the [WH_MSGFILTER](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA/](#)[SetWindowsHookExW](#) function.

If an application that uses the DDEML and performs synchronous transactions must process messages before they are dispatched, it must use the [WH_MSGFILTER](#) hook.

See also

[CallNextHookEx](#)

[SetWindowsHookEx](#)

[MSG](#)

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

MouseProc function

Article • 06/30/2023

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function whenever an application calls the [GetMessage](#) or [PeekMessageA/PeekMessageW](#) function and there is a mouse message to be processed.

The **HOOKPROC** type defines a pointer to this callback function. *MouseProc* is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK MouseProc(
    _In_ int      nCode,
    _In_ WPARAM  wParam,
    _In_ LPARAM  lParam
);
```

Parameters

nCode [in]

Type: **int**

A code that the hook procedure uses to determine how to process the message.

If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
HC_ACTION 0	The <i>wParam</i> and <i>lParam</i> parameters contain information about a mouse message.

Value	Meaning
HC_NOREMOVE 3	The <i>wParam</i> and <i>lParam</i> parameters contain information about a mouse message, and the mouse message has not been removed from the message queue. (An application called the PeekMessage function, specifying the PM_NOREMOVE flag.)

wParam [in]

Type: **WPARAM**

The identifier of the mouse message.

lParam [in]

Type: **LPARAM**

A pointer to a [MOUSEHOOKSTRUCT](#) structure.

Returns

Type: **LRESULT**

If *nCode* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_MOUSE](#) hooks will not receive hook notifications and may behave incorrectly as a result.

If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the target window procedure.

Remarks

An application installs the hook procedure by specifying the [WH_MOUSE](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) function.

The hook procedure must not install a [WH_JOURNALPLAYBACK](#) callback function.

This hook may be called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

See also

[CallNextHookEx](#)

[GetMessage](#)

[MOUSEHOOKSTRUCT](#)

[PeekMessage](#)

[SetWindowsHookEx](#)

[Hooks](#)

[About Hooks](#)

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](#)

ShellProc function

Article • 11/19/2022

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The function receives notifications of Shell events from the system.

The **HOOKPROC** type defines a pointer to this callback function. *ShellProc* is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK ShellProc(
    _In_ int      nCode,
    _In_ WPARAM  wParam,
    _In_ LPARAM  lParam
);
```

Parameters

nCode [in]

Type: **int**

The hook code.

If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
HSHELL_ACCESSIBILITYSTATE 11	The accessibility state has changed.
HSHELL_ACTIVATESHELLWINDOW	The shell should activate its main window.

Value	Meaning
HSHELL_APPCOMMAND 12	The user completed an input event (for example, pressed an application command button on the mouse or an application command key on the keyboard), and the application did not handle the WM_APPCOMMAND message generated by that input. If the Shell procedure handles the WM_COMMAND message, it should not call CallNextHookEx . See the Return Value section for more information.
HSHELL_GETMINRECT 5	A window is being minimized or maximized. The system needs the coordinates of the minimized rectangle for the window.
HSHELL_LANGUAGE 8	Keyboard language was changed or a new keyboard layout was loaded.
HSHELL_REDRAW 6	The title of a window in the task bar has been redrawn.
HSHELL_TASKMAN 7	The user has selected the task list. A shell application that provides a task list should return TRUE to prevent Windows from starting its task list.
HSHELL_WINDOWACTIVATED 4	The activation has changed to a different top-level, unowned window.
HSHELL_WINDOWCREATED 1	A top-level, unowned window has been created. The window exists when the system calls this hook.
HSHELL_WINDOWDESTROYED 2	A top-level, unowned window is about to be destroyed. The window still exists when the system calls this hook.
HSHELL_WINDOWREPLACED 13	A top-level window is being replaced. The window exists when the system calls this hook.

wParam [in]

Type: **WPARAM**

This parameter depends on the value of the *nCode* parameter, as shown in the following table.

nCode	wParam
HSHELL_ACCESSIBILITYSTATE	Indicates which accessibility feature has changed state. This value is one of the following: ACCESS_FILTERKEYS , ACCESS_MOUSEKEYS , or ACCESS_STICKYKEYS .

nCode	wParam
HSHELL_APPCOMMAND	Indicates where the WM_APPCOMMAND message was originally sent; for example, the handle to a window. For more information, see <i>cmd</i> parameter in WM_APPCOMMAND .
HSHELL_GETMINRECT	A handle to the minimized or maximized window.
HSHELL_LANGUAGE	A handle to the window.
HSHELL_REDRAW	A handle to the redrawn window.
HSHELL_WINDOWACTIVATED	A handle to the activated window.
HSHELL_WINDOWCREATED	A handle to the created window.
HSHELL_WINDOWDESTROYED	A handle to the destroyed window.
HSHELL_WINDOWREPLACED	A handle to the window being replaced. Windows 2000: Not supported.

IParam [in]

Type: **LPARAM**

This parameter depends on the value of the *nCode* parameter, as shown in the following table.

nCode	IParam
HSHELL_APPCOMMAND	<code>GET_APPCOMMAND LPARAM(1Param)</code> is the application command corresponding to the input event. <code>GET_DEVICE LPARAM(1Param)</code> indicates what generated the input event; for example, the mouse or keyboard. For more information, see the <i>uDevice</i> parameter description under WM_APPCOMMAND . <code>GET_FLAGS LPARAM(1Param)</code> depends on the value of <i>cmd</i> in WM_APPCOMMAND . For example, it might indicate which virtual keys were held down when the WM_APPCOMMAND message was originally sent. For more information, see the <i>dwCmdFlags</i> description parameter under WM_APPCOMMAND .
HSHELL_GETMINRECT	A pointer to a RECT structure.
HSHELL_LANGUAGE	A handle to a keyboard layout.
HSHELL_MONITORCHANGED	A handle to the window that moved to a different monitor.
HSHELL_REDRAW	The value is TRUE if the window is flashing, or FALSE otherwise.

nCode	IParam
HSHELL_WINDOWACTIVATED	The value is TRUE if the window is in full-screen mode, or FALSE otherwise.
HSHELL_WINDOWREPLACED	A handle to the new window. Windows 2000: Not supported.

Returns

Type: LRESULT

The return value should be zero unless the value of *nCode* is HSHELL_APPCOMMAND and the shell procedure handles the WM_COMMAND message. In this case, the return should be nonzero.

Remarks

Install this hook procedure by specifying the WH_SHELL hook type and a pointer to the hook procedure in a call to the SetWindowsHookEx function.

See also

[CallNextHookEx](#)

[SendMessage](#)

[SetWindowsHookEx](#)

[WM_APPCOMMAND](#)

[WM_COMMAND](#)

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

SysMsgProc function

Article • 06/30/2023

Description

An application-defined or library-defined callback function used with the [SetWindowsHookExA/SetWindowsHookExW](#) function. The system calls this function after an input event occurs in a dialog box, message box, menu, or scroll bar, but before the message generated by the input event is processed. The function can monitor messages for any dialog box, message box, menu, or scroll bar in the system.

The **HOOKPROC** type defines a pointer to this callback function. *SysMsgProc* is a placeholder for the application-defined or library-defined function name.

C++

```
LRESULT CALLBACK SysMsgProc(
    _In_ int      nCode,
    WPARAM wParam,
    _In_ LPARAM lParam
);
```

Parameters

nCode [in]

Type: **int**

The type of input event that generated the message.

If *nCode* is less than zero, the hook procedure must pass the message to the [CallNextHookEx](#) function without further processing and should return the value returned by [CallNextHookEx](#).

This parameter can be one of the following values.

Value	Meaning
MSGF_DIALOGBOX 0	The input event occurred in a message box or dialog box.
MSGF_MENU 2	The input event occurred in a menu.

Value	Meaning
MSGF_SCROLLBAR 5	The input event occurred in a scroll bar.

wParam

Type: **WPARAM**

This parameter is not used.

lParam [in]

Type: **LPARAM**

A pointer to an [MSG](#) message structure.

Returns

Type: **LRESULT**

If *nCode* is less than zero, the hook procedure must return the value returned by [CallNextHookEx](#).

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call [CallNextHookEx](#) and return the value it returns; otherwise, other applications that have installed [WH_SYSMSGFILTER](#) hooks will not receive hook notifications and may behave incorrectly as a result.

If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the target window procedure.

Remarks

An application installs the hook procedure by specifying the [WH_SYSMSGFILTER](#) hook type and a pointer to the hook procedure in a call to the [SetWindowsHookExA](#)/[SetWindowsHookExW](#) function.

See also

[CallNextHookEx](#)

[MSG](#)

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](#)

Hook Notifications

Article • 04/27/2021

- [WM_CANCELJOURNAL](#)
- [WM_QUEUESYNC](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_CANCELJOURNAL message

Article • 05/24/2022

⚠ Warning

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.

Posted to an application when a user cancels the application's journaling activities. The message is posted with a **NULL** window handle.

C++

```
#define WM_CANCELJOURNAL 0x004B
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: **void**

This message does not return a value. It is meant to be processed from within an application's main loop or a [GetMessage](#) hook procedure, not from a window procedure.

Remarks

Journal record and playback modes are modes imposed on the system that let an application sequentially record or play back user input. The system enters these modes when an application installs a [JournalRecordProc](#) or [JournalPlaybackProc](#) hook.

procedure. When the system is in either of these journaling modes, applications must take turns reading input from the input queue. If any one application stops reading input while the system is in a journaling mode, other applications are forced to wait.

To ensure a robust system, one that cannot be made unresponsive by any one application, the system automatically cancels any journaling activities when a user presses CTRL+ESC or CTRL+ALT+DEL. The system then unhooks any journaling hook procedures, and posts a **WM_CANCELJOURNAL** message, with a **NULL** window handle, to the application that set the journaling hook.

The **WM_CANCELJOURNAL** message has a **NULL** window handle, therefore it cannot be dispatched to a window procedure. There are two ways for an application to see a **WM_CANCELJOURNAL** message: If the application is running in its own main loop, it must catch the message between its call to [GetMessage](#) or [PeekMessage](#) and its call to [DispatchMessage](#). If the application is not running in its own main loop, it must set a [GetMsgProc](#) hook procedure (through a call to [SetWindowsHookEx](#) specifying the **WH_GETMESSAGE** hook type) that watches for the message.

When an application sees a **WM_CANCELJOURNAL** message, it can assume two things: the user has intentionally canceled the journal record or playback mode, and the system has already unhooked any journal record or playback hook procedures.

Note that the key combinations mentioned above (CTRL+ESC or CTRL+ALT+DEL) cause the system to cancel journaling. If any one application is made unresponsive, they give the user a means of recovery. The **VK_CANCEL** virtual key code (usually implemented as the CTRL+BREAK key combination) is what an application that is in journal record mode should watch for as a signal that the user wishes to cancel the journaling activity. The difference is that watching for **VK_CANCEL** is a suggested behavior for journaling applications, whereas CTRL+ESC or CTRL+ALT+DEL cause the system to cancel journaling regardless of a journaling application's behavior.

Requirements

Requirement	Value
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

Reference

[JournalPlaybackProc](#)

[JournalRecordProc](#)

[GetMsgProc](#)

[SetWindowsHookEx](#)

[Conceptual](#)

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_QUEUESYNC message

Article • 01/07/2021

Sent by a computer-based training (CBT) application to separate user-input messages from other messages sent through the [WH_JOURNALPLAYBACK](#) procedure.

C++

```
#define WM_QUEUESYNC 0x0023
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

Type: **void**

A CBT application should return zero if it processes this message.

Remarks

Whenever a CBT application uses the [WH_JOURNALPLAYBACK](#) procedure, the first and last messages are **WM_QUEUESYNC**. This allows the CBT application to intercept and examine user-initiated messages without doing so for events that it sends.

If an application specifies a **NULL** window handle, the message is posted to the message queue of the active window.

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]

Requirement	Value
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Reference

[JournalPlaybackProc](#)

[SetWindowsHookEx](#)

Conceptual

[Hooks](#)

Feedback

Was this page helpful?

 Yes

 No

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

Hook Structures

Article • 04/27/2021

- [CBT_CREATEWND](#)
- [CBTACTIVATESTRUCT](#)
- [CWPRETSTRUCT](#)
- [CWPSTRUCT](#)
- [DEBUGHOOKINFO](#)
- [EVENTMSG](#)
- [KBDLLHOOKSTRUCT](#)
- [MOUSEHOOKSTRUCT](#)
- [MOUSEHOOKSTRUCTEX](#)
- [MSLLHOOKSTRUCT](#)

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](#)

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](#)

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](#)

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](#)

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](#)

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

MSLLHOOKSTRUCT structure (winuser.h)

Article 11/19/2022

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](#)

Multiple Document Interface

Article • 01/07/2021

[Many new and intermediate users find it difficult to learn to use MDI applications. Therefore, you should consider other models for your user interface. However, you can use MDI for applications which do not easily fit into an existing model.]

The multiple-document interface (MDI) is a specification that defines a user interface for applications that enable the user to work with more than one document at the same time.

In This Section

Topic	Description
About the Multiple Document Interface	Describes the Multiple Document Interface.
Using the Multiple Document Interface	Explains how to perform tasks associated with the Multiple Document Interface.
MDI Reference	Contains the API reference.

MDI Functions

Name	Description
CreateMDIWindow	Creates a MDI child window.
DefFrameProc	Provides default processing for any window messages that the window procedure of a 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.
DefMDIChildProc	Provides default processing for any window message that the window procedure of a 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.
TranslateMDISysAccel	Processes accelerator keystrokes for window menu commands of the 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.

MDI Messages

Name	Description
WM_MDIACTIVATE	Sent to a MDI client window to instruct the client window to activate a different MDI child window.
WM_MDICASCADE	Sent to a MDI client window to arrange all its child windows in a cascade format.
WM_MDICREATE	Sent to a MDI client window to create an MDI child window.
WM_MDIDESTROY	Sent to a MDI client window to close an MDI child window.
WM_MDIGETACTIVE	Sent to a MDI client window to retrieve the handle to the active MDI child window.
WM_MDIICONARRANGE	Sent to a MDI client window to arrange all minimized MDI child windows. It does not affect child windows that are not minimized.
WM_MDIMAXIMIZE	Sent to a MDI client window to maximize an MDI child window. The system resizes the child window to make its client area fill the client window. The system places the child window's window menu icon in the rightmost position of the frame window's menu bar, and places the child window's restore icon in the leftmost position. The system also appends the title bar text of the child window to that of the frame window.
WM_MDINEXT	Sent to a MDI client window to activate the next or previous child window.
WM_MDIRRESHMENU	Sent to a MDI client window to refresh the window menu of the MDI frame window.
WM_MDIRESTORE	Sent to a MDI client window to restore an MDI child window from maximized or minimized size.
WM_MDISETMENU	Sent to a MDI client window to replace the entire menu of an MDI frame window, to replace the window menu of the frame window, or both.
WM_MDTILE	Sent to a MDI client window to arrange all of its MDI child windows in a tile format.

MDI Structures

Name	Description
------	-------------

Name	Description
MDICREATESTRUCT	Contains information about the class, title, owner, location, and size of a MDI child window.

Feedback

Was this page helpful?



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

MDI Overviews

Article • 04/27/2021

- [About the Multiple Document Interface](#)
- [Using the Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

About the Multiple Document Interface

Article • 01/07/2021

Each document in an multiple-document interface (MDI) application is displayed in a separate child window within the client area of the application's main window. Typical MDI applications include word-processing applications that allow the user to work with multiple text documents, and spreadsheet applications that allow the user to work with multiple charts and spreadsheets. For more information, see the following topics.

- [Frame, Client, and Child Windows](#)
- [Child Window Creation](#)
- [Child Window Activation](#)
- [Multiple Document Menus](#)
- [Multiple Document Accelerators](#)
- [Child Window Size and Arrangement](#)
- [Icon Title Windows](#)
- [Child Window Data](#)
 - [Window Structure](#)
 - [Window Properties](#)

Frame, Client, and Child Windows

An MDI application has three kinds of windows: a frame window, an MDI client window, as well as a number of child windows. The *frame window* is like the main window of the application: it has a sizing border, a title bar, a window menu, a minimize button, and a maximize button. The application must register a window class for the frame window and provide a window procedure to support it.

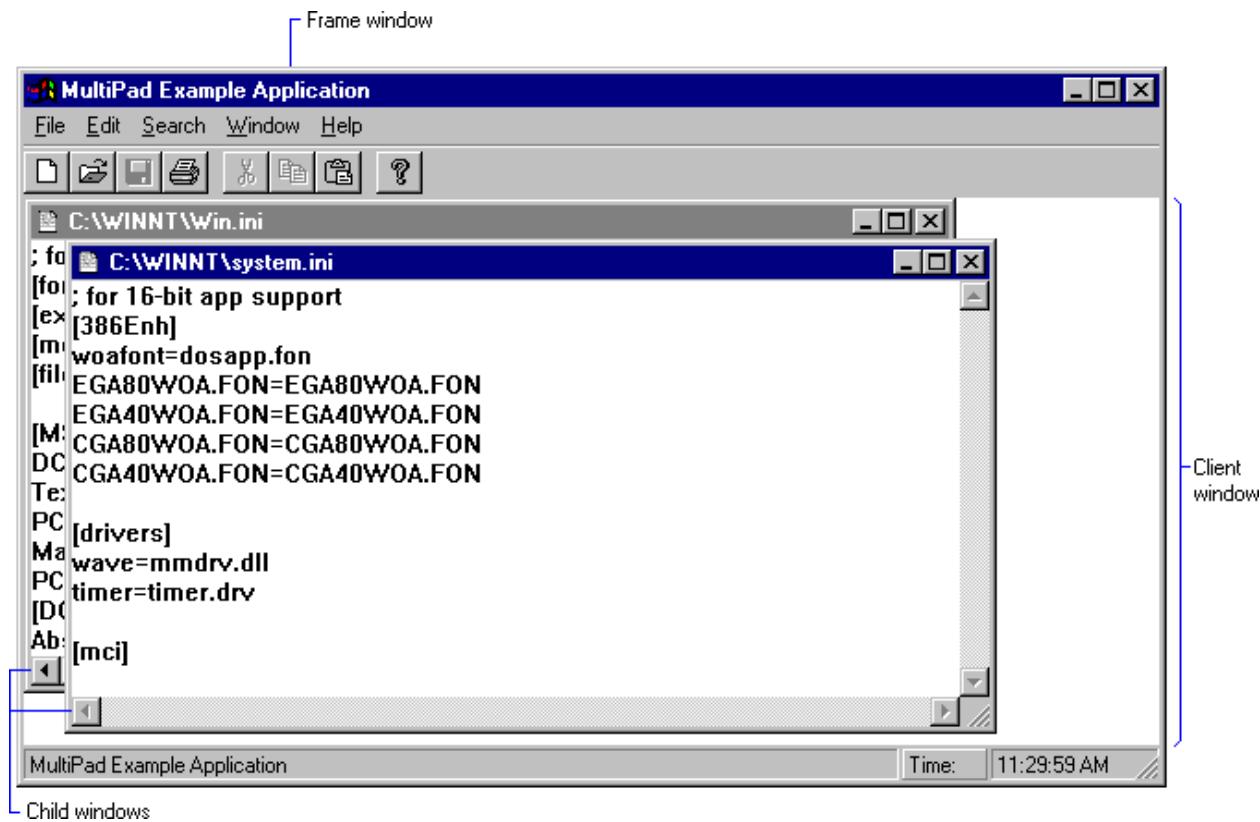
An MDI application does not display output in the client area of the frame window. Instead, it displays the MDI client window. An *MDI client window* is a special type of child window belonging to the preregistered window class **MDICLIENT**. The client window is a child of the frame window; it serves as the background for child windows. It also provides support for creating and manipulating child windows. For example, an MDI application can create, activate, or maximize child windows by sending messages to the MDI client window.

When the user opens or creates a document, the client window creates a child window for the document. The client window is the parent window of all MDI child windows in the application. Each child window has a sizing border, a title bar, a window menu, a

minimize button, and a maximize button. Because a child window is clipped, it is confined to the client window and cannot appear outside it.

An MDI application can support more than one kind of document. For example, a typical spreadsheet application enables the user to work with both charts and spreadsheets. For each type of document that it supports, an MDI application must register a child window class and provide a window procedure to support the windows belonging to that class. For more information about window classes, see [Window Classes](#). For more information about window procedures, see [Window Procedures](#).

Following is a typical MDI application. It is named Multipad.



Child Window Creation

To create a child window, an MDI application either calls the [CreateMDIWindow](#) function or sends the [WM_MDICREATE](#) message to the MDI client window. A more efficient way to create an MDI child window is to call the [CreateWindowEx](#) function, specifying the [WS_EX_MDICHILD](#) extended style.

To destroy a child window, an MDI application sends a [WM_MDIDESTROY](#) message to the MDI client window.

Child Window Activation

Any number of child windows can appear in the client window at any one time, but only one can be active. The active child window is positioned in front of all other child windows, and its border is highlighted.

The user can activate an inactive child window by clicking it. An MDI application activates a child window by sending a [WM_MDIACTIVATE](#) message to the MDI client window. As the client window processes this message, it sends a [WM_MDIACTIVATE](#) message to the window procedure of the child window to be activated and to the window procedure of the child window being deactivated.

To prevent a child window from activating, handle the [WM_NCACTIVATE](#) message to the child window by returning **FALSE**.

The system keeps track of each child window's position in the stack of overlapping windows. This stacking is known as the [Z-Order](#). The user can activate the next child window in the Z order by clicking **Next** from the window menu in the active window. An application activates the next (or previous) child window in the Z order by sending a [WM_MDINEXT](#) message to the client window.

To retrieve the handle to the active child window, the MDI application sends a [WM_MDIGETACTIVE](#) message to the client window.

Multiple Document Menus

The frame window of an MDI application should include a menu bar with a window menu. The window menu should include items that arrange the child windows within the client window or that close all child windows. The window menu of a typical MDI application might include the items in the following table.

Menu item	Purpose
Tile	Arranges child windows in a tile format so that each appears in its entirety in the client window.
Cascade	Arranges child windows in a cascade format. The child windows overlap one another, but the title bar of each is visible.
Arrange Icons	Arranges the icons of minimized child windows along the bottom of the client window.
Close All	Closes all child windows.

Whenever a child window is created, the system automatically appends a new menu item to the window menu. The text of the menu item is the same as the text on the menu bar of the new child window. By clicking the menu item, the user can activate the corresponding child window. When a child window is destroyed, the system automatically removes the corresponding menu item from the window menu.

The system can add up to ten menu items to the window menu. When the tenth child window is created, the system adds the **More Windows** item to the window menu. Clicking this item displays the **Select Window** dialog box. The dialog box contains a list box with the titles of all MDI child windows currently available. The user can activate a child window by clicking its title in the list box.

If your MDI application supports several types of child windows, tailor the menu bar to reflect the operations associated with the active window. To do this, provide separate menu resources for each type of child window the application supports. When a new type of child window is activated, the application should send a **WM_MDISETMENU** message to the client window, passing to it the handle to the corresponding menu.

When no child window exists, the menu bar should contain only items used to create or open a document.

When the user is navigating through an MDI application's menus by using cursor keys, the keys behave differently than when the user is navigating through a typical application's menus. In an MDI application, control passes from the application's window menu to the window menu of the active child window, and then to the first item on the menu bar.

Multiple Document Accelerators

To receive and process accelerator keys for its child windows, an MDI application must include the **TranslateMDISysAccel** function in its message loop. The loop must call **TranslateMDISysAccel** before calling the **TranslateAccelerator** or **DispatchMessage** function.

Accelerator keys on the window menu for an MDI child window are different from those for a non-MDI child window. In an MDI child window, the ALT+ – (minus) key combination opens the window menu, the CTRL+F4 key combination closes the active child window, and the CTRL+F6 key combination activates the next child window.

Child Window Size and Arrangement

An MDI application controls the size and position of its child windows by sending messages to the MDI client window. To maximize the active child window, the application sends the [WM_MDIMAXIMIZE](#) message to the client window. When a child window is maximized, its client area completely fills the MDI client window. In addition, the system automatically hides the child window's title bar, and adds the child window's window menu icon and Restore button to the MDI application's menu bar. The application can restore the client window to its original (premaximized) size and position by sending the client window a [WM_MDIRESTORE](#) message.

An MDI application can arrange its child windows in either a cascade or tile format. When the child windows are cascaded, the windows appear in a stack. The window on the bottom of the stack occupies the upper left corner of the screen, and the remaining windows are offset vertically and horizontally so that the left border and title bar of each child window is visible. To arrange child windows in the cascade format, an MDI application sends the [WM_MDICASCADE](#) message. Typically, the application sends this message when the user clicks **Cascade** on the window menu.

When the child windows are tiled, the system displays each child window in its entirety — overlapping none of the windows. All of the windows are sized, as necessary, to fit within the client window. To arrange child windows in the tile format, an MDI application sends a [WM_MDTILE](#) message to the client window. Typically, the application sends this message when the user clicks **Tile** on the window menu.

An MDI application should provide a different icon for each type of child window it supports. The application specifies an icon when registering the child window class. The system automatically displays a child window's icon in the lower portion of the client window when the child window is minimized. An MDI application directs the system to arrange child window icons by sending a [WM_MDIICONARRANGE](#) message to the client window. Typically, the application sends this message when the user clicks **Arrange Icons** on the window menu.

Icon Title Windows

Because MDI child windows may be minimized, an MDI application must avoid manipulating icon title windows as if they were normal MDI child windows. Icon title windows appear when the application enumerates child windows of the MDI client window. Icon title windows differ from other child windows, however, in that they are owned by an MDI child window.

To determine whether a child window is an icon title window, use the [GetWindow](#) function with the **GW_OWNER** index. Non-title windows return **NULL**. Note that this test

is insufficient for top-level windows, because menus and dialog boxes are owned windows.

Child Window Data

Because the number of child windows varies depending on how many documents the user opens, an MDI application must be able to associate data (for example, the name of the current file) with each child window. There are two ways to do this:

- Store child window data in the window structure.
- Use window properties.

Window Structure

When an MDI application registers a window class, it may reserve extra space in the window structure for application data specific to this particular class of windows. To store and retrieve data in this extra space, the application uses the [GetWindowLong](#) and [SetWindowLong](#) functions.

To maintain a large amount of data for a child window, an application can allocate memory for a data structure and then store the handle to the memory containing the structure in the extra space associated with the child window.

Window Properties

An MDI application can also store per-document data by using window properties. *Per-document data* is data specific to the type of document contained in a particular child window. Properties are different from extra space in the window structure in that you need not allocate extra space when registering the window class. A window can have any number of properties. Also, where offsets are used to access the extra space in window structures, properties are referred to by string names. For more information about window properties, see [Window Properties](#).

Feedback

Was this page helpful?



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

Using the Multiple Document Interface

Article • 01/07/2021

This section explains how to perform the following tasks:

- [Registering Child and Frame Window Classes](#)
- [Creating Frame and Child Windows](#)
- [Writing the Main Message Loop](#)
- [Writing the Frame Window Procedure](#)
- [Writing the Child Window Procedure](#)
- [Creating a Child Window](#)

To illustrate these tasks, this section includes examples from Multipad, a typical multiple-document interface (MDI) application.

Registering Child and Frame Window Classes

A typical MDI application must register two window classes: one for its frame window and one for its child windows. If an application supports more than one type of document (for example, a spreadsheet and a chart), it must register a window class for each type.

The class structure for the frame window is similar to the class structure for the main window in non-MDI applications. The class structure for the MDI child windows differs slightly from the structure for child windows in non-MDI applications as follows:

- The class structure should have an icon, because the user can minimize an MDI child window as if it were a normal application window.
- The menu name should be **NULL**, because an MDI child window cannot have its own menu.
- The class structure should reserve extra space in the window structure. With this space, the application can associate data, such as a filename, with a particular child window.

The following example shows how Multipad registers its frame and child window classes.

```
BOOL WINAPI InitializeApplication()
{
    WNDCLASS wc;
```

```

// Register the frame window class.

wc.style      = 0;
wc.lpfnWndProc = (WNDPROC) MPFrameWndProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance   = hInst;
wc.hIcon       = LoadIcon(hInst, IDMULTIPAD);
wc.hCursor     = LoadCursor((HANDLE) NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH) (COLOR_APPWORKSPACE + 1);
wc.lpszMenuName = IDMULTIPAD;
wc.lpszClassName = szFrame;

if (!RegisterClass (&wc) )
    return FALSE;

// Register the MDI child window class.

wc.lpfnWndProc = (WNDPROC) MPMDIChildWndProc;
wc.hIcon       = LoadIcon(hInst, IDNOTE);
wc.lpszMenuName = (LPCTSTR) NULL;
wc.cbWndExtra = CBWNDEXTRA;
wc.lpszClassName = szChild;

if (!RegisterClass(&wc))
    return FALSE;

return TRUE;
}

```

Creating Frame and Child Windows

After registering its window classes, an MDI application can create its windows. First, it creates its frame window by using the [CreateWindow](#) or [CreateWindowEx](#) function. After creating its frame window, the application creates its client window, again by using [CreateWindow](#) or [CreateWindowEx](#). The application should specify **MDICLIENT** as the client window's class name; **MDICLIENT** is a preregistered window class defined by the system. The *lpvParam* parameter of [CreateWindow](#) or [CreateWindowEx](#) should point to a [CLIENTCREATESTRUCT](#) structure. This structure contains the members described in the following table:

Member	Description
hWindowMenu	Handle to the window menu used for controlling MDI child windows. As child windows are created, the application adds their titles to the window menu as menu items. The user can then activate a child window by clicking its title on the window menu.

Member	Description
idFirstChild	Specifies the identifier of the first MDI child window. The first MDI child window created is assigned this identifier. Additional windows are created with incremented window identifiers. When a child window is destroyed, the system immediately reassigns the window identifiers to keep their range contiguous.

When a child window's title is added to the window menu, the system assigns an identifier to the child window. When the user clicks a child window's title, the frame window receives a **WM_COMMAND** message with the identifier in the *wParam* parameter. You should specify a value for the **idFirstChild** member that does not conflict with menu-item identifiers in the frame window's menu.

Multipad's frame window procedure creates the MDI client window while processing the **WM_CREATE** message. The following example shows how the client window is created.

```

case WM_CREATE:
{
    CLIENTCREATESTRUCT ccs;

    // Retrieve the handle to the window menu and assign the
    // first child window identifier.

    ccs.hWindowMenu = GetSubMenu(GetMenu(hwnd), WINDOWMENU);
    ccs.idFirstChild = IDM_WINDOWCHILD;

    // Create the MDI client window.

    hwndMDIClient = CreateWindow( "MDICLIENT", (LPCTSTR) NULL,
        WS_CHILD | WS_CLIPCHILDREN | WS_VSCROLL | WS_HSCROLL,
        0, 0, 0, 0, hwnd, (HMENU) 0xCAC, hInst, (LPSTR) &ccs);

    ShowWindow(hwndMDIClient, SW_SHOW);
}
break;

```

Titles of child windows are added to the bottom of the window menu. If the application adds strings to the window menu by using the **AppendMenu** function, these strings can be overwritten by the titles of the child windows when the window menu is repainted (whenever a child window is created or destroyed). An MDI application that adds strings to its window menu should use the **InsertMenu** function and verify that the titles of child windows have not overwritten these new strings.

Use the **WS_CLIPCHILDREN** style to create the MDI client window to prevent the window from painting over its child windows.

Writing the Main Message Loop

The main message loop of an MDI application is similar to that of a non-MDI application handling accelerator keys. The difference is that the MDI message loop calls the [TranslateMDISysAccel](#) function before checking for application-defined accelerator keys or before dispatching the message.

The following example shows the message loop of a typical MDI application. Note that [GetMessage](#) can return -1 if there is an error.

```
MSG msg;
BOOL bRet;

while ((bRet = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        if (!TranslateMDISysAccel(hwndMDIClient, &msg) &&
            !TranslateAccelerator(hwndFrame, hAccel, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}
```

The [TranslateMDISysAccel](#) function translates [WM_KEYDOWN](#) messages into [WM_SYSCOMMAND](#) messages and sends them to the active MDI child window. If the message is not an MDI accelerator message, the function returns **FALSE**, in which case the application uses the [TranslateAccelerator](#) function to determine whether any of the application-defined accelerator keys were pressed. If not, the loop dispatches the message to the appropriate window procedure.

Writing the Frame Window Procedure

The window procedure for an MDI frame window is similar to that of a non-MDI application's main window. The difference is that a frame window procedure passes all messages it does not handle to the [DefFrameProc](#) function rather than to the [DefWindowProc](#) function. In addition, the frame window procedure must also pass some messages that it does handle, including those listed in the following table.

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.

The frame window procedure in Multipad is called MPFrameWndProc. The handling of other messages by MPFrameWndProc is similar to that of non-MDI applications.

[WM_COMMAND](#) messages in Multipad are handled by the locally defined CommandHandler function. For command messages Multipad does not handle, CommandHandler calls the [DefFrameProc](#) function. If Multipad does not use [DefFrameProc](#) by default, the user can't activate a child window from the window menu, because the [WM_COMMAND](#) message sent by clicking the window's menu item would be lost.

Writing the Child Window Procedure

Like the frame window procedure, an MDI child window procedure uses a special function for processing messages by default. All messages that the child window procedure does not handle must be passed to the [DefMDIChildProc](#) function rather than to the [DefWindowProc](#) function. In addition, some window-management messages must be passed to [DefMDIChildProc](#), even if the application handles the message, in order for MDI to function correctly. Following are the messages the application must pass to [DefMDIChildProc](#).

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 (formerly known as system) menu commands: SC_NEXTWINDOW, SC_PREVWINDOW, SC_MOVE, SC_SIZE, and SC_MAXIMIZE.

Creating a Child Window

To create an MDI child window, an application can either call the [CreateMDIWindow](#) function or send an [WM_MDICREATE](#) message to the MDI client window. (The application can use the [CreateWindowEx](#) function with the **WS_EX_MDICHILD** style to create MDI child windows.) A single-threaded MDI application can use either method to create a child window. A thread in a multithreaded MDI application must use the [CreateMDIWindow](#) or [CreateWindowEx](#) function to create a child window in a different thread.

The *lParam* parameter of a [WM_MDICREATE](#) message is a far pointer to an [MDICREATESTRUCT](#) structure. The structure includes four dimension members: **x** and **y**, which indicate the horizontal and vertical positions of the window, and **cx** and **cy**, which indicate the horizontal and vertical extents of the window. Any of these members may be assigned explicitly by the application, or they may be set to **CW_USEDEFAULT**, in which case the system selects a position, size, or both, according to a cascading algorithm. In any case, all four members must be initialized. Multipad uses **CW_USEDEFAULT** for all dimensions.

The last member of the [MDICREATESTRUCT](#) structure is the **style** member, which may contain style bits for the window. To create an MDI child window that can have any

combination of window styles, specify the **MDIS_ALLCHILDSTYLES** window style. When this style is not specified, an MDI child window has the **WS_MINIMIZE**, **WS_MAXIMIZE**, **WS_HSCROLL**, and **WS_VSCROLL** styles as default settings.

Multipad creates its MDI child windows by using its locally defined AddFile function (located in the source file MPFILE.C). The AddFile function sets the title of the child window by assigning the **szTitle** member of the window's **MDICREATESTRUCT** structure to either the name of the file being edited or to "Untitled." The **szClass** member is set to the name of the MDI child window class registered in Multipad's InitializeApplication function. The **hOwner** member is set to the application's instance handle.

The following example shows the AddFile function in Multipad.

```
HWND APIENTRY AddFile(pName)
TCHAR * pName;
{
    HWND hwnd;
    TCHAR sz[160];
    MDICREATESTRUCT mcs;

    if (!pName)
    {

        // If the pName parameter is NULL, load the "Untitled"
        // string from the STRINGTABLE resource and set the szTitle
        // member of MDICREATESTRUCT.

        LoadString(hInst, IDS_UNTITLED, sz, sizeof(sz)/sizeof(TCHAR));
        mcs.szTitle = (LPCTSTR) sz;
    }
    else

        // Title the window with the full path and filename,
        // obtained by calling the OpenFile function with the
        // OF_PARSE flag, which is called before AddFile().

        mcs.szTitle = of.szPathName;

    mcs.szClass = szChild;
    mcs.hOwner = hInst;

    // Use the default size for the child window.

    mcs.x = mcs.cx = CW_USEDEFAULT;
    mcs.y = mcs.cy = CW_USEDEFAULT;

    // Give the child window the default style. The styleDefault
    // variable is defined in MULTIPAD.C.
```

```
mcs.style = styleDefault;

// Tell the MDI client window to create the child window.

hwnd = (HWND) SendMessage (hwndMDIClient, WM_MDICREATE, 0,
                           (LONG) (LPMDICREATESTRUCT) &mcs);

// If the file is found, read its contents into the child
// window's client area.

if (pName)
{
    if (!LoadFile(hwnd, pName))
    {

        // Cannot load the file; close the window.

        SendMessage(hwndMDIClient, WM_MDIDESTROY,
                    (DWORD) hwnd, 0L);
    }
}
return hwnd;
}
```

The pointer passed in the *IParam* parameter of the **WM_MDICREATE** message is passed to the **CreateWindow** function and appears as the first member in the **CREATESTRUCT** structure, passed in the **WM_CREATE** message. In Multipad, the child window initializes itself during **WM_CREATE** message processing by initializing document variables in its extra data and by creating the edit control's child window.

Feedback

Was this page helpful?

 Yes

 No

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

MDI Reference

Article • 04/27/2021

- [MDI Functions](#)
 - [MDI Messages](#)
 - [MDI Structures](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

MDI Functions

Article • 04/27/2021

- [CreateMDIWindow](#)
 - [DefFrameProc](#)
 - [DefMDIChildProc](#)
 - [TranslateMDISysAccel](#)
-

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](#)

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

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

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](#)

MDI Messages

Article • 04/27/2021

- [WM_MDIACTIVATE](#)
- [WM_MDICASCADE](#)
- [WM_MDICREATE](#)
- [WM_MDIDESTROY](#)
- [WM_MDIGETACTIVE](#)
- [WM_MDIICONARRANGE](#)
- [WM_MDIMAXIMIZE](#)
- [WM_MDINEXT](#)
- [WM_MDIREFRESHMENU](#)
- [WM_MDIRESTORE](#)
- [WM_MDISETMENU](#)
- [WM_MDTILE](#)

Feedback

Was this page helpful?



Get help at Microsoft Q&A

WM_MDIACTIVATE message

Article • 01/07/2021

An application sends the **WM_MDIACTIVATE** message to a multiple-document interface (MDI) client window to instruct the client window to activate a different MDI child window.

C++

```
#define WM_MDIACTIVATE 0x0222
```

Parameters

wParam

A handle to the MDI child window to be activated.

lParam

This parameter is not used.

Return value

Type: LRESULT

If an application sends this message to an MDI client window, the return value is zero.

An MDI child window should return zero if it processes this message.

Remarks

As the client window processes this message, it sends **WM_MDIACTIVATE** to the child window being deactivated and to the child window being activated. The message parameters received by an MDI child window are as follows:

wParam

A handle to the MDI child window being deactivated.

lParam

A handle to the MDI child window being activated.

An MDI child window is activated independently of the MDI frame window. When the frame window becomes active, the child window last activated by using the **WM_MDIACTIVATE** message receives the **WM_NCACTIVATE** message to draw an active window frame and title bar; the child window does not receive another **WM_MDIACTIVATE** message.

Requirements

Requirement	Value
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

Reference

[WM_MDIGETACTIVE](#)

[WM_MDINEXT](#)

[WM_NCACTIVATE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDICASCADE message

Article • 01/07/2021

An application sends the **WM_MDICASCADE** message to a multiple-document interface (MDI) client window to arrange all its child windows in a cascade format.

C++

```
#define WM_MDICASCADE 0x0227
```

Parameters

wParam

The cascade behavior. 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.

lParam

This parameter is not used.

Return value

Type: **BOOL**

If the message succeeds, the return value is **TRUE**.

If the message fails, the return value is **FALSE**.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[WM_MDIICONARRANGE](#)

[WM_MDITILE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDICREATE message

Article • 01/07/2021

An application sends the WM_MDICREATE message to a multiple-document interface (MDI) client window to create an MDI child window.

C++

```
#define WM_MDICREATE 0x0220
```

Parameters

wParam

This parameter is not used.

lParam

A pointer to an [MDICREATESTRUCT](#) structure containing information that the system uses to create the MDI child window.

Return value

Type: [HWND](#)

If the message succeeds, the return value is the handle to the new child window.

If the message fails, the return value is [NULL](#).

Remarks

The MDI child window is created with the [window style](#) bits [WS_CHILD](#), [WS_CLIPSIBLINGS](#), [WS_CLIPCHILDREN](#), [WS_SYSMENU](#), [WS_CAPTION](#), [WS_THICKFRAME](#), [WS_MINIMIZEBOX](#), and [WS_MAXIMIZEBOX](#), plus additional style bits specified in the [MDICREATESTRUCT](#) structure. The system adds the title of the new child window to the window menu of the frame window. An application should use this message to create all child windows of the client window.

If an MDI client window receives any message that changes the activation of its child windows while the active child window is maximized, the system restores the active child window and maximizes the newly activated child window.

When an MDI child window is created, the system sends the [WM_CREATE](#) message to the window. The *lParam* parameter 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.

An application should not send a second [WM_MDICREATE](#) message while a [WM_MDICREATE](#) message is still being processed. For example, it should not send a [WM_MDICREATE](#) message while an MDI child window is processing its [WM_MDICREATE](#) message.

Requirements

Requirement	Value
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

Reference

[CreateMDIWindow](#)

[CREATESTRUCT](#)

[MDICREATESTRUCT](#)

[WM_CREATE](#)

[WM_MDIDESTROY](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDIDESTROY message

Article • 01/07/2021

An application sends the WM_MDIDESTROY message to a multiple-document interface (MDI) client window to close an MDI child window.

C++

```
#define WM_MDIDESTROY 0x0221
```

Parameters

wParam

A handle to the MDI child window to be closed.

lParam

This parameter is not used.

Return value

Type: zero

This message always returns zero.

Remarks

This message removes the title of the MDI child window from the MDI frame window and deactivates the child window. An application should use this message to close all MDI child windows.

If an MDI client window receives a message that changes the activation of its child windows and the active MDI child window is maximized, the system restores the active child window and maximizes the newly activated child window.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[WM_MDICREATE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDIGETACTIVE message

Article • 01/07/2021

An application sends the **WM_MDIGETACTIVE** message to a multiple-document interface (MDI) client window to retrieve the handle to the active MDI child window.

C++

```
#define WM_MDIGETACTIVE 0x0229
```

Parameters

wParam

This parameter is not used.

lParam

The maximized state. If this parameter is not **NULL**, it is a pointer to a value that indicates the maximized state of the MDI child window. If the value is **TRUE**, the window is maximized; a value of **FALSE** indicates that it is not. If this parameter is **NULL**, the parameter is ignored.

Return value

Type: **HWND**

The return value is the handle to the active MDI child window.

Requirements

Requirement	Value
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

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDIICONARRANGE message

Article • 01/07/2021

An application sends the WM_MDIICONARRANGE message to a multiple-document interface (MDI) client window to arrange all minimized MDI child windows. It does not affect child windows that are not minimized.

C++

```
#define WM_MDIICONARRANGE 0x0228
```

Parameters

wParam

This parameter is not used; it must be zero.

lParam

This parameter is not used; it must be zero.

Requirements

Requirement	Value
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

Reference

[WM_MDICASCADE](#)

[WM_MDTILE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDIMAXIMIZE message

Article • 01/07/2021

An application sends the WM_MDIMAXIMIZE message to a multiple-document interface (MDI) client window to maximize an MDI child window. The system resizes the child window to make its client area fill the client window. The system places the child window's window menu icon in the rightmost position of the frame window's menu bar, and places the child window's restore icon in the leftmost position. The system also appends the title bar text of the child window to that of the frame window.

C++

```
#define WM_MDIMAXIMIZE 0x0225
```

Parameters

wParam

A handle to the MDI child window to be maximized.

lParam

This parameter is not used.

Return value

Type: zero

The return value is always zero.

Remarks

If an MDI client window receives any message that changes the activation of its child windows while the currently active MDI child window is maximized, the system restores the active child window and maximizes the newly activated child window.

Requirements

Requirement	Value
-------------	-------

Requirement	Value
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

Reference

[WM_MDIRESTORE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDINEXT message

Article • 01/07/2021

An application sends the WM_MDINEXT message to a multiple-document interface (MDI) client window to activate the next or previous child window.

C++

```
#define WM_MDINEXT 0x0224
```

Parameters

wParam

A handle to the MDI child window. The system activates the child window that is immediately before or after the specified child window, depending on the value of the *lParam* parameter. If the *wParam* parameter is **NULL**, the system activates the child window that is immediately before or after the currently active child window.

lParam

If this parameter is zero, the system activates the next MDI child window and places the child window identified by the *wParam* parameter behind all other child windows. If this parameter is nonzero, the system activates the previous child window, placing it in front of the child window identified by *wParam*.

Return value

Type: **zero**

The return value is always zero.

Remarks

If an MDI client window receives any message that changes the activation of its child windows while the active MDI child window is maximized, the system restores the active child window and maximizes the newly activated child window.

Requirements

Requirement	Value
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

Reference

[WM_MDIACTIVATE](#)

[WM_MDIGETACTIVE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDIREFRESHMENU message

Article • 01/07/2021

An application sends the WM_MDIREFRESHMENU message to a multiple-document interface (MDI) client window to refresh the window menu of the MDI frame window.

C++

```
#define WM_MDIREFRESHMENU 0x0234
```

Parameters

wParam

This parameter is not used and must be zero.

lParam

This parameter is not used and must be zero.

Return value

Type: HMENU

If the message succeeds, the return value is the handle to the frame window menu.

If the message fails, the return value is **NULL**.

Remarks

After sending this message, an application must call the [DrawMenuBar](#) function to update the menu bar.

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Requirement	Value
Header	Winuser.h (include Windows.h)

See also

Reference

[DrawMenuBar](#)

[WM_MDISETMENU](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDIRESTORE message

Article • 01/07/2021

An application sends the **WM_MDIRESTORE** message to a multiple-document interface (MDI) client window to restore an MDI child window from maximized or minimized size.

C++

```
#define WM_MDIRESTORE 0x0223
```

Parameters

wParam

A handle to the MDI child window to be restored.

lParam

This parameter is not used.

Return value

Type: **zero**

The return value is always zero.

Requirements

Requirement	Value
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

Reference

[WM_MDIMAXIMIZE](#)

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

WM_MDISETMENU message

Article • 01/07/2021

An application sends the **WM_MDISETMENU** message to a multiple-document interface (MDI) client window to replace the entire menu of an MDI frame window, to replace the window menu of the frame window, or both.

C++

```
#define WM_MDISETMENU 0x0230
```

Parameters

wParam

A handle to the new frame window menu. If this parameter is **NULL**, the frame window menu is not changed.

lParam

A handle to the new window menu. If this parameter is **NULL**, the window menu is not changed.

Return value

Type: **HMENU**

If the message succeeds, the return value is the handle to the old frame window menu.

If the message fails, the return value is zero.

Remarks

After sending this message, an application must call the [DrawMenuBar](#) function to update the menu bar.

If this message replaces the window menu, the MDI child window menu items are removed from the previous window menu and added to the new window menu.

If an MDI child window is maximized and this message replaces the MDI frame window menu, the window menu icon and restore icon are removed from the previous frame

window menu and added to the new frame window menu.

Requirements

Requirement	Value
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

Reference

[DrawMenuBar](#)

[WM_MDIREFRESHMENU](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_MDITILE message

Article • 01/07/2021

An application sends the **WM_MDITILE** message to a multiple-document interface (MDI) client window to arrange all of its MDI child windows in a tile format.

C++

```
#define WM_MDITILE      0x0226
```

Parameters

wParam

The tiling option. 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.

lParam

This parameter is not used.

Return value

Type: **BOOL**

If the message succeeds, the return value is **TRUE**.

If the message fails, the return value is **FALSE**.

Requirements

Requirement	Value
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

Reference

[WM_MDICASCADE](#)

[WM_MDIICONARRANGE](#)

Conceptual

[Multiple Document Interface](#)

Feedback

Was this page helpful?

 Yes

 No

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

MDI Structures

Article • 04/27/2021

- [MDICREATESTRUCT](#)

Feedback

Was this page helpful?



[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](#)