Contents

- Windows and Messages
  - Windows
    - Window Overviews
      - About Windows
      - Window Features
      - Using Windows
    - Window Reference
      - Window Constants
        - Extended Window Styles
        - Window Styles
      - Window Functions
        - 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

Window Macros
- GET_X_LPARAM
- GET_Y_LPARAM
- HIBYTE
- HIWORD
- LOBYTE
- LOWORD
- MAKELONG
- MAKELPARAM
- MAKELRESULT
- MAKEWORD
- MAKEWPARAM

Window Messages
- MN_GETHMENU
- WM_ERASEBKGND
- WM_GETFONT
- WM_GETTEXT
- WM_GETTEXTLENGTH
- WM_SETFONT
- WM_SETICON

WM_SETTEXT
Window Notifications
WM_ACTIVATEAPP
WM_CANCELMODE
WM_CHILDACTIVATE
WM_CLOSE
WM_COMPACTING
WM_CREATE
WM_DESTROY
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
Window Structures
ALTTABINFO
CHANGEFILTERSTRUCT
CLIENTCREATESTRUCT
CREATESTRUCT
GUITHREADINFO
MINMAXINFO
NCCALCSIZE_PARAMS
STYLESTRUCT
TITLEBARINFO
TITLEBARINFOEX
UPDATELAYEREDWINDOWINFO
WINDOWINFO
WINDOWPLACEMENT
WINDOWPOS
Window Classes
Window Class Overviews
About Window Classes
Using Window Classes
Window Class Reference
Window Class Functions
GetClassInfo
GetClassInfoEx
GetClassLong
GetClassLongPtr
GetClassName
GetClassWord
GetWindowLong
GetWindowLongPtr

            - RegisterClass
            - RegisterClassEx
            - SetClassLong
            - SetClassLongPtr
            - SetClassWord
            - SetWindowLong
            - SetWindowLongPtr
            - UnregisterClass
        - Window Class Structures
            - WNDCLASS
            - WNDCLASSEX
        - Window Class Styles
- Window Procedures
    - Window Procedure Overviews
        - About Window Procedures
        - Using Window Procedures
    - Window Procedure Reference
        - Window Procedure Functions
            - CallWindowProc
            - DefWindowProc
            - WindowProc
- Messages and Message Queues
    - Message and Message Queue Overviews
        - About Messages and Message Queues
        - Using Messages and Message Queues
    - Message Reference
        - Message Functions
            - 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

Message Structures
- BSMINFO
- MSG

Message Constants
- OCM__BASE
- WM_APP
- WM_USER

Timers

Timer Overviews
- About Timers
- Using Timers

Timer Reference

- Timer Functions
  - KillTimer
  - SetCoalescableTimer
  - SetTimer
  - TimerProc
- Timer Notifications
  - WM_TIMER

Window Properties

- Window Property Overviews
  - About Window Properties
  - Using Window Properties
- Window Property Reference
  - Window Property Functions
    - EnumProps
    - EnumPropsEx
    - GetProp
    - PropEnumProc
    - PropEnumProcEx
    - RemoveProp
    - SetProp

Configuration

- Configuration Reference
  - Configuration Constants
    - Contact Visualization
    - Gesture Visualization
    - Pen Visualization
  - GetSystemMetrics
  - SystemParametersInfo
    - ANIMATIONINFO
    - AUDIODESCRIPTION
    - MINIMIZEDMETRICS
    - NONCLIENTMETRICS

      - WM_SETTINGCHANGE
      - WM_WININICHANGE
- Hooks
  - Hook Overview
  - Using Hooks
  - Hook Reference
    - Hook Functions
      - CallMsgFilter
      - CallNextHookEx
      - CallWndProc
      - CallWndRetProc
      - CBTProc
      - DebugProc
      - ForegroundIdleProc
      - GetMsgProc
      - JournalPlaybackProc
      - JournalRecordProc
      - KeyboardProc
      - LowLevelKeyboardProc
      - LowLevelMouseProc
      - MessageProc
      - MouseProc
      - SetWindowsHookEx
      - ShellProc
      - SysMsgProc
      - UnhookWindowsHookEx
    - Hook Notifications
      - WM_CANCELJOURNAL
      - WM_QUEUESYNC
    - Hook Structures
      - CBT_CREATEWND
      - CBTACTIVATESTRUCT

- CWPRETSTRUCT
- CWPSTRUCT
- DEBUGHOOKINFO
- EVENTMSG
- KBDLLHOOKSTRUCT
- MOUSEHOOKSTRUCT
- MOUSEHOOKSTRUCTEX
- MSLLHOOKSTRUCT

Multiple Document Interface (MDI)
- MDI Overviews
  - About the Multiple Document Interface
  - Using the Multiple Document Interface
- MDI Reference
  - MDI Functions
    - CreateMDIWindow
    - DefFrameProc
    - DefMDIChildProc
    - TranslateMDISysAccel
  - MDI Messages
    - WM_MDIACTIVATE
    - WM_MDICASCADE
    - WM_MDICREATE
    - WM_MDIDESTROY
    - WM_MDIGETACTIVE
    - WM_MDIICONARRANGE
    - WM_MDIMAXIMIZE
    - WM_MDINEXT
    - WM_MDIREFRESHMENU
    - WM_MDIRESTORE
    - WM_MDISETMENU
    - WM_MDITILE
  - MDI Structures

MDICREATESTRUCT

Windows and Messages

2/22/2020 • 2 minutes to read • Edit Online

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.

Windows

7/30/2020 • 21 minutes to read • Edit Online

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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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 *lParam* parameter in a message. The macro concatenates the specified values.
MAKELRESULT	Creates a value for use as a return value from a window procedure. The macro concatenates the specified values.
MAKEWORD	Creates a **WORD** value by concatenating the specified values.
MAKEWPARAM	Creates a value for use as a *wParam* 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	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.
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 *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.
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 *wParam* 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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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	Ssent 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 *lpParam* 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.

NAME	DESCRIPTION
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.

Window Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About Windows
- Window Features
- Using Windows

About Windows

2/22/2020 • 15 minutes to read • Edit Online

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.

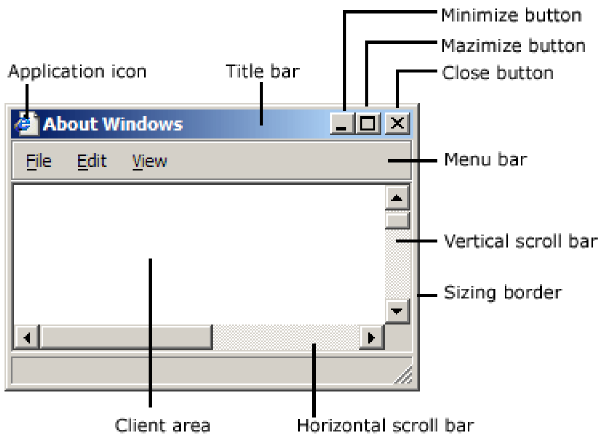
A system configuration application, such as a Control Panel item, changes the desktop wallpaper by using the **SystemParametersInfo** function with the *wAction* parameter set to **SPI_SETDESKWALLPAPER** and the *lpvParam* parameter specifying a bitmap file name. **SystemParametersInfo** then loads the 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 uses the **EnumWindows** function to obtain a handle to each of its top-level windows. **EnumWindows** passes the handle to each top-level window, in turn, to an application-defined callback function, **EnumWindowsProc**.

A 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.

Window Features

7/30/2020 • 37 minutes to read • Edit Online

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

- Window Types
  - Overlapped Windows
  - Pop-up Windows
  - Child Windows
  - 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 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:

- 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.

PARENT WINDOW	CHILD WINDOW
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 one of the following conditions is true:

- The process is the foreground process.
- The process was started by the foreground process.
- The process received the last input event.
- There is no foreground process.
- The foreground process is being debugged.
- The foreground is not locked (see **LockSetForegroundWindow**).
- The foreground lock time-out has expired (see **SPI_GETFOREGROUNDLOCKTIMEOUT** in **SystemParametersInfo**).
- No menus are active.

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.

COMMAND	DESCRIPTION
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(LAYOUT_RTL). 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, LAYOUT_RTL). 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, (LPPOINT) &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, LAYOUT_RTL) 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.

Using Windows

2/22/2020 • 6 minutes to read • Edit Online

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.

```
HINSTANCE hinst;
HWND hwndMain;

// Create the main window.

hwndMain = CreateWindowEx(
    0,                      // no extended styles
    "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
    hinstance,              // 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.

Window Reference

2/22/2020 • 2 minutes to read • Edit Online

- Window Constants
- Window Functions
- Window Macros
- Window Messages
- Window Notifications
- Window Structures

Window Constants

2/22/2020 • 2 minutes to read • Edit Online

- Extended Window Styles
- Window Styles

Extended Window Styles

7/30/2020 • 5 minutes to read • Edit Online

The following are the extended window styles.

Example

```
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 descendents 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.
WS_EX_CONTEXTHELP 0x00000400L	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. **WS_EX_CONTEXTHELP** cannot be used with the **WS_MAXIMIZEBOX** or **WS_MINIMIZEBOX** styles.

CONSTANT/VALUE	DESCRIPTION
WS_EX_CONTROLPARENT 0x00010000L	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.
WS_EX_DLGMODALFRAME 0x00000001L	The window has a double border; the window can, optionally, be created with a title bar by specifying the **WS_CAPTION** style in the *dwStyle* parameter.
WS_EX_LAYERED 0x00080000	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**. **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.
WS_EX_LAYOUTRTL 0x00400000L	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.
WS_EX_LEFT 0x00000000L	The window has generic left-aligned properties. This is the default.
WS_EX_LEFTSCROLLBAR 0x00004000L	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.
WS_EX_LTRREADING 0x00000000L	The window text is displayed using left-to-right reading-order properties. This is the default.
WS_EX_MDICHILD 0x00000040L	The window is a MDI child window.
WS_EX_NOACTIVATE 0x08000000L	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. The window should not be activated through programmatic access or via keyboard navigation by accessible technology, such as Narrator. To activate the window, use the **SetActiveWindow** or **SetForegroundWindow** function. 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.

CONSTANT/VALUE	DESCRIPTION
WS_EX_NOINHERITLAYOUT 0x00100000L	The window does not pass its window layout to its child windows.
WS_EX_NOPARENTNOTIFY 0x00000004L	The child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.
WS_EX_NOREDIRECTIONBITMAP 0x00200000L	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.
WS_EX_OVERLAPPEDWINDOW (WS_EX_WINDOWEDGE | WS_EX_CLIENTEDGE)	The window is an overlapped window.
WS_EX_PALETTEWINDOW (WS_EX_WINDOWEDGE | WS_EX_TOOLWINDOW | WS_EX_TOPMOST)	The window is palette window, which is a modeless dialog box that presents an array of commands.
WS_EX_RIGHT 0x00001000L	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. 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.
WS_EX_RIGHTSCROLLBAR 0x00000000L	The vertical scroll bar (if present) is to the right of the client area. This is the default.
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.

CONSTANT/VALUE	DESCRIPTION
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

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

Window Styles

7/30/2020 • 3 minutes to read • Edit Online

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

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_DLGFRAME 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/VALUE	DESCRIPTION
WS_GROUP 0x00020000L	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. 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.
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.
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.

CONSTANT/VALUE	DESCRIPTION
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	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. 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.
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	The window is initially visible. This style can be turned on and off by using the **ShowWindow** or **SetWindowPos** function.
WS_VSCROLL 0x00200000L	The window has a vertical scroll bar.

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)

Window Functions

2/22/2020 • 2 minutes to read • Edit Online

- 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

Window Macros

2/22/2020 • 2 minutes to read • Edit Online

- GET_X_LPARAM
- GET_Y_LPARAM
- HIBYTE
- HIWORD
- LOBYTE
- LOWORD
- MAKELONG
- MAKELPARAM
- MAKELRESULT
- MAKEWORD
- MAKEWPARAM

Window Messages

2/22/2020 • 2 minutes to read • Edit Online

In This Section

- MN_GETHMENU
- WM_ERASEBKGND
- WM_GETFONT
- WM_GETTEXT
- WM_GETTEXTLENGTH
- WM_SETFONT
- WM_SETICON
- WM_SETTEXT

MN_GETHMENU message

2/22/2020 • 2 minutes to read • Edit Online

Retrieves the menu handle for the current window.

```
#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

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

WM_ERASEBKGND message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_GETFONT message

2/22/2020 • 2 minutes to read • Edit Online

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

```
#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

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

WM_GETTEXT message

7/30/2020 • 2 minutes to read • Edit Online

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

```
#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

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

WM_GETTEXTLENGTH message

7/30/2020 • 2 minutes to read • Edit Online

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

```
#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 controlc, does not return a string value. Instead, it returns zero.

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

DefWindowProc

GetWindowText

GetWindowTextLength

WM_GETTEXT

Conceptual

Windows

Other Resources

CB_GETLBTEXT

LB_GETTEXT

LB_GETTEXTLEN

WM_SETFONT message

2/22/2020 • 2 minutes to read • Edit Online

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

```
#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

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

DeleteObject

WM_SETICON message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_SETTEXT message

7/30/2020 • 2 minutes to read • Edit Online

Sets the text of a window.

```
#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

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

Window Notifications

2/22/2020 • 2 minutes to read • Edit Online

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

WM_ACTIVATEAPP message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_CANCELMODE message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_CHILDACTIVATE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_CLOSE message

7/30/2020 • 2 minutes to read • Edit Online

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

A window receives this message through its **WindowProc** function.

```
#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

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

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

WM_COMPACTING message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_CREATE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

CREATESTRUCT

WM_NCCREATE

Conceptual

Windows

WM_DESTROY message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_ENABLE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_ENTERSIZEMOVE message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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_EXITSIZEMOVE

WM_SYSCOMMAND

Conceptual

Windows

WM_EXITSIZEMOVE message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_GETICON message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_GETMINMAXINFO message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_INPUTLANGCHANGE message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#define WM_INPUTLANGCHANGE              0x0051
```

Parameters

wParam

The character set of the new locale.

lParam

The input locale identifier. For more information, see Languages, Locales, and Keyboard Layouts.

Return value

Type: **LRESULT**

An application should return nonzero if it processes this 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

Reference

DefWindowProc

WM_INPUTLANGCHANGEREQUEST

Conceptual

Windows

WM_INPUTLANGCHANGEREQUEST message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_MOVE message

7/30/2020 • 2 minutes to read • Edit Online

Sent after a window has been moved.

A window receives this message through its **WindowProc** function.

```
#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

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

WM_MOVING message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_NCACTIVATE message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_NCCALCSIZE message

7/30/2020 • 3 minutes to read • Edit Online

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.

```
#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.

RETURN CODE/VALUE	DESCRIPTION
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	This value indicates that, upon return from **WM_NCCALCSIZE**, the rectangles specified by the **rgrc**[1] and **rgrc**[2] 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. 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.

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

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

RECT

WM_NCCREATE message

7/30/2020 • 2 minutes to read • Edit Online

Sent prior to the WM_CREATE message when a window is first created.

A window receives this message through its WindowProc function.

```
#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

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

WM_NCDESTROY message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_NULL message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_QUERYDRAGICON message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_QUERYOPEN message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_QUIT message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_SHOWWINDOW message

7/30/2020 • 2 minutes to read • Edit Online

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

A window receives this message through its **WindowProc** function.

```
#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

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

WM_SIZE message

7/30/2020 • 2 minutes to read • Edit Online

Sent to a window after its size has changed.

A window receives this message through its **WindowProc** function.

```
#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

```
/******************************************************************
*                                                                 *
*  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

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

WM_SIZING message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_STYLECHANGED message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_STYLECHANGING message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_THEMECHANGED message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_USERCHANGED message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	None supported
Header	Winuser.h (include Windows.h)

See also

Windows Overview

WM_WINDOWPOSCHANGED message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_WINDOWPOSCHANGING message

7/30/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

Window Structures

2/22/2020 • 2 minutes to read • Edit Online

- ALTTABINFO
- CHANGEFILTERSTRUCT
- CLIENTCREATESTRUCT
- CREATESTRUCT
- GUITHREADINFO
- MINMAXINFO
- NCCALCSIZE_PARAMS
- STYLESTRUCT
- TITLEBARINFO
- TITLEBARINFOEX
- UPDATELAYEREDWINDOWINFO
- WINDOWINFO
- WINDOWPLACEMENT
- WINDOWPOS

Window Classes

2/22/2020 • 3 minutes to read • Edit Online

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.
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.

NAME	DESCRIPTION
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	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.
GetClassWord	Retrieves the 16-bit (**WORD**) value at the specified offset into the extra class memory for the window class to which the specified window belongs. > [!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 function.

NAME	DESCRIPTION
SetClassLong	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**.

Window Class Structures

NAME	DESCRIPTION
WNDCLASS	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.
WNDCLASSEX	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.

Window Classes Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About Window Classes
- Using Window Classes

About Window Classes

2/22/2020 • 16 minutes to read • Edit Online

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.
#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.
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 a 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 **DllMain**). 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 **lpszMenuName** 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 **lpszMenuName** 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 **lpszMenuName** 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.

Using Window Classes

2/22/2020 • 2 minutes to read • Edit Online

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(lpCmdLine);
}

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 |
        CS_VREDRAW;                    // redraw if size changes
    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.

Window Class Reference

2/22/2020 • 2 minutes to read • Edit Online

- Window Class Functions
- Window Class Structures

Window Class Functions

2/22/2020 • 2 minutes to read • Edit Online

- GetClassInfo
- GetClassInfoEx
- GetClassLong
- GetClassLongPtr
- GetClassName
- GetClassWord
- GetWindowLong
- GetWindowLongPtr
- RegisterClass
- RegisterClassEx
- SetClassLong
- SetClassLongPtr
- SetClassWord
- SetWindowLong
- SetWindowLongPtr
- UnregisterClass

Window Class Structures

2/22/2020 • 2 minutes to read • Edit Online

- WNDCLASS
- WNDCLASSEX

Window Class Styles

7/30/2020 • 2 minutes to read • Edit Online

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

```
WNDCLASS wc = {};
wc.lpfnWndProc = 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.
CS_DBLCLKS 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.

CONSTANT/VALUE	DESCRIPTION
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

Minimum supported client	Windows 2000 Professional [desktop apps only]

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

Window Procedures

7/30/2020 • 2 minutes to read • Edit Online

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.

Window Procedure Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About Window Procedures
- Using Window Procedures

About Window Procedures

7/30/2020 • 9 minutes to read • Edit Online

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.

Using Window Procedures

7/30/2020 • 3 minutes to read • Edit Online

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);
}
```

Window Procedure Reference

2/22/2020 • 2 minutes to read • Edit Online

- Window Procedure Functions

Window Procedure Functions

7/30/2020 • 2 minutes to read • Edit Online

In This Section

- CallWindowProc
- DefWindowProc
- *WindowProc*

Messages and Message Queues

2/22/2020 • 4 minutes to read • Edit Online

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.
GetInputState	Determines whether there are mouse-button or keyboard messages in the calling thread's message queue.
GetMessage	Retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval. Unlike GetMessage, the PeekMessage function does not wait for a message to be posted before returning.

NAME	DESCRIPTION
GetMessageExtraInfo	Retrieves the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue.
GetMessagePos	Retrieves the cursor position for the last message retrieved by the GetMessage function. To determine the current position of the cursor, use the GetCursorPos function.
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	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.
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	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.
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**.

NAME	DESCRIPTION
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.
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 todefine 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.

Message and Message Queue Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About Messages and Message Queues
- Using Messages and Message Queues

About Messages and Message Queues

7/30/2020 • 22 minutes to read • Edit Online

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

PREFIX	MESSAGE CATEGORY	DOCUMENTATION
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
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	Scroll bar 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

PREFIX	MESSAGE CATEGORY	DOCUMENTATION
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
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.

```
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.

Using Messages and Message Queues

2/22/2020 • 7 minutes to read • Edit Online

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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.
    //

}
```

Message Reference

2/22/2020 • 2 minutes to read • Edit Online

In This Section

- Message Functions
- Message Structures
- Message Constants

Message Functions

2/22/2020 • 2 minutes to read • Edit Online

- 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

Message Structures

2/22/2020 • 2 minutes to read • Edit Online

- BSMINFO
- MSG

Message Constants

2/22/2020 • 2 minutes to read • Edit Online

In This Section

- OCM__BASE
- WM_APP
- WM_USER

OCM__BASE

2/22/2020 • 2 minutes to read • Edit Online

Used to define private messages for use by private window classes, usually of the form **OCM__BASE**+*x*, where *x* is an integer value.

```
#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 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

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

WM_APP

2/22/2020 • 2 minutes to read • Edit Online

Used to define private messages, usually of the form **WM_APP**+*x*, where *x* is an integer value.

```
#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

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

WM_USER

2/22/2020 • 2 minutes to read • Edit Online

Used to define private messages for use by private window classes, usually of the form **WM_USER**+*x*, where *x* is an integer value.

```
#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

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

Timers

2/22/2020 • 2 minutes to read • Edit Online

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. *TimerProc* 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.

Timer Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About Timers
- Using Timers

About Timers

2/26/2020 • 3 minutes to read • Edit Online

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

Using Timers

2/22/2020 • 5 minutes to read • Edit Online

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
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.
        // 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

Timer Reference

2/22/2020 • 2 minutes to read • Edit Online

In This Section

- Timer Functions
- Timer Notifications

Timer Functions

2/22/2020 • 2 minutes to read • Edit Online

In This Section

- KillTimer
- SetCoalescableTimer
- SetTimer
- *TimerProc*

Timer Notifications

2/22/2020 • 2 minutes to read • Edit Online

- WM_TIMER

WM_TIMER message

2/22/2020 • 2 minutes to read • Edit Online

Posted to the installing thread's message queue when a timer expires. The message is posted by the **GetMessage** or **PeekMessage** function.

```
#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

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

Window Properties

2/22/2020 • 2 minutes to read • Edit Online

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 **PROPENUMPROC** type defines a pointer to this callback function. *PropEnumProc* is a placeholder for the application-defined function name.
PropEnumProcEx	An application-defined callback function used with the **EnumPropsEx** function. The function receives property entries from a window's property list. The **PROPENUMPROCEX** type defines a pointer to this callback function. *PropEnumProcEx* 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.

NAME	DESCRIPTION
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.

Window Property Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About Window Properties
- Using Window Properties

About Window Properties

2/22/2020 • 2 minutes to read • Edit Online

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*.

Using Window Properties

2/22/2020 • 3 minutes to read • Edit Online

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;
}
```

Window Property Reference

2/22/2020 • 2 minutes to read • Edit Online

- Window Property Functions

Window Property Functions

2/22/2020 • 2 minutes to read • Edit Online

- EnumProps
- EnumPropsEx
- GetProp
- *PropEnumProc*
- *PropEnumProcEx*
- RemoveProp
- SetProp

Configuration

2/22/2020 • 2 minutes to read • Edit Online

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.

Configuration Reference

2/22/2020 • 2 minutes to read • Edit Online

The following functions can be used to control aspects of the configuration of display elements:

- GetSystemMetrics
- SystemParametersInfo

Configuration Constants

2/22/2020 • 2 minutes to read • Edit Online

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

Contact Visualization

2/22/2020 • 2 minutes to read • Edit Online

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_SETCONTACTVISUALIZATION** 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

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

Input Feedback Configuration

Gesture Visualization

2/22/2020 • 2 minutes to read • Edit Online

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

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

Pen Visualization

2/22/2020 • 2 minutes to read • Edit Online

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

Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	Winuser.h

See also

Configuration Constants

Contact Visualization

SystemParametersInfo

Input Feedback Configuration

WM_SETTINGCHANGE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_WININICHANGE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

Hooks

2/22/2020 • 5 minutes to read • Edit Online

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.
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.

NAME	DESCRIPTION
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.
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.

NAME	DESCRIPTION
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
WM_CANCELJOURNAL	Posted to an application when a user cancels the application's journaling activities. The message is posted with a **NULL** 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.

NAME	DESCRIPTION
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

Hooks Overview

4/13/2020 • 9 minutes to read • Edit Online

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
  - 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:

```
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

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

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.

```
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.

Using Hooks

2/22/2020 • 10 minutes to read • Edit Online

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.

```
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
 ****************************************************************/

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
 ****************************************************************/

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;
    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;

        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),
    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);
}
```

Hook Reference

2/22/2020 • 2 minutes to read • Edit Online

- Hook Functions
- Hook Notifications
- Hook Structures

Hook Functions

2/22/2020 • 2 minutes to read • Edit Online

- **CallMsgFilter**
- **CallNextHookEx**
- *CallWndProc*
- *CallWndRetProc*
- *CBTProc*
- *DebugProc*
- *ForegroundIdleProc*
- *GetMsgProc*
- *JournalPlaybackProc*
- *JournalRecordProc*
- *KeyboardProc*
- *LowLevelKeyboardProc*
- *LowLevelMouseProc*
- *MessageProc*
- *MouseProc*
- **SetWindowsHookEx**
- *ShellProc*
- *SysMsgProc*
- **UnhookWindowsHookEx**

Hook Notifications

2/22/2020 • 2 minutes to read • Edit Online

- WM_CANCELJOURNAL
- WM_QUEUESYNC

WM_CANCELJOURNAL message

2/22/2020 • 2 minutes to read • Edit Online

Posted to an application when a user cancels the application's journaling activities. The message is posted with a **NULL** window handle.

```
#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

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

WM_QUEUESYNC message

2/22/2020 • 2 minutes to read • Edit Online

Sent by a computer-based training (CBT) application to separate user-input messages from other messages sent through the **WH_JOURNALPLAYBACK** procedure.

```
#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

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

See also

Reference

JournalPlaybackProc

SetWindowsHookEx

Conceptual

Hooks

Hook Structures

2/22/2020 • 2 minutes to read • Edit Online

- CBT_CREATEWND
- CBTACTIVATESTRUCT
- CWPRETSTRUCT
- CWPSTRUCT
- DEBUGHOOKINFO
- EVENTMSG
- KBDLLHOOKSTRUCT
- MOUSEHOOKSTRUCT
- MOUSEHOOKSTRUCTEX
- MSLLHOOKSTRUCT

Multiple Document Interface

7/30/2020 • 2 minutes to read • Edit Online

[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.

NAME	DESCRIPTION
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_MDIREFRESHMENU	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_MDITILE	Sent to a MDI client window to arrange all of its MDI child windows in a tile format.

MDI Structures

NAME	DESCRIPTION
MDICREATESTRUCT	Contains information about the class, title, owner, location, and size of a MDI child window.

MDI Overviews

2/22/2020 • 2 minutes to read • Edit Online

- About the Multiple Document Interface
- Using the Multiple Document Interface

About the Multiple Document Interface

2/22/2020 • 9 minutes to read • Edit Online

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

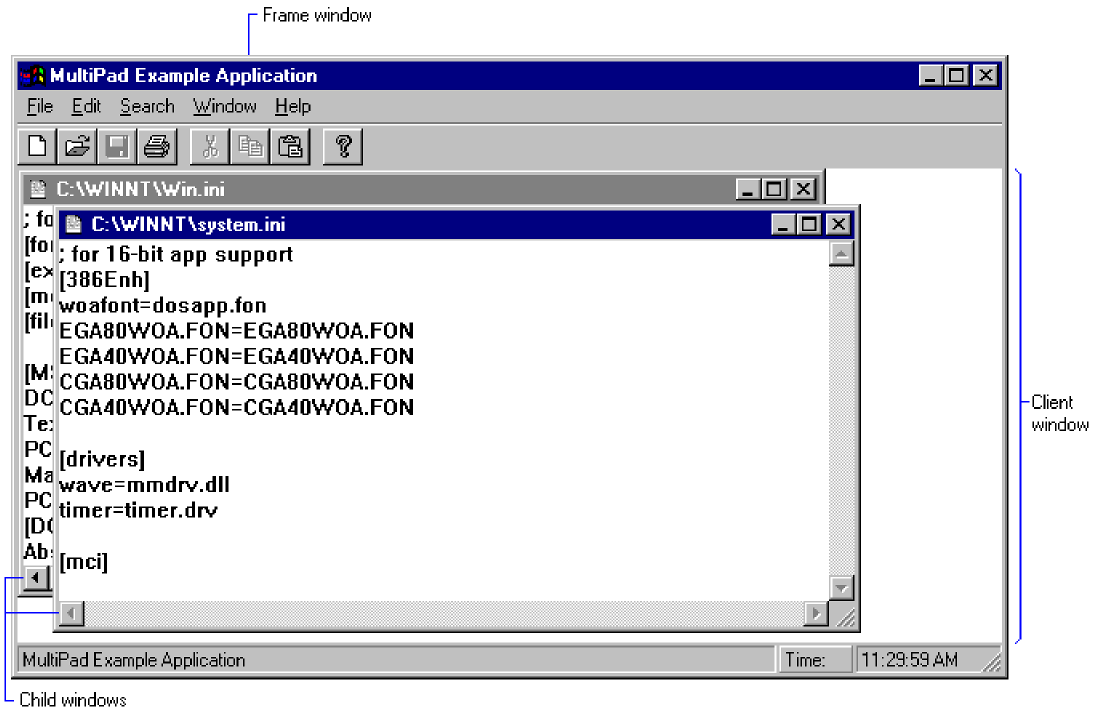
An MDI application has three kinds of windows: a frame window, an MDI client window, as well as a number of child windows. The *frame window* is like the main window of the application: it has a sizing border, a title bar, a window menu, a minimize button, and a maximize button. The application must 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.



```
; for 16-bit app support
[386Enh]
woafont=dosapp.fon
EGA80WOA.FON=EGA80WOA.FON
EGA40WOA.FON=EGA40WOA.FON
CGA80WOA.FON=CGA80WOA.FON
CGA40WOA.FON=CGA40WOA.FON

[drivers]
wave=mmdrv.dll
timer=timer.drv

[mci]
```

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_MDITILE** 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.

Using the Multiple Document Interface

7/30/2020 • 9 minutes to read • Edit Online

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.
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 *lParam* 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.

MDI Reference

2/22/2020 • 2 minutes to read • Edit Online

- MDI Functions
- MDI Messages
- MDI Structures

MDI Functions

2/22/2020 • 2 minutes to read • Edit Online

- CreateMDIWindow
- DefFrameProc
- DefMDIChildProc
- TranslateMDISysAccel

MDI Messages

2/22/2020 • 2 minutes to read • Edit Online

- WM_MDIACTIVATE
- WM_MDICASCADE
- WM_MDICREATE
- WM_MDIDESTROY
- WM_MDIGETACTIVE
- WM_MDIICONARRANGE
- WM_MDIMAXIMIZE
- WM_MDINEXT
- WM_MDIREFRESHMENU
- WM_MDIRESTORE
- WM_MDISETMENU
- WM_MDITILE

WM_MDIACTIVATE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_MDICASCADE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_MDICREATE message

2/22/2020 • 2 minutes to read • Edit Online

An application sends the **WM_MDICREATE** message to a multiple-document interface (MDI) client window to create an MDI child window.

```
#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

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

WM_MDIDESTROY message

2/22/2020 • 2 minutes to read • Edit Online

An application sends the **WM_MDIDESTROY** message to a multiple-document interface (MDI) client window to close an MDI child window.

```
#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

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

WM_MDIGETACTIVE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

Multiple Document Interface Overview

WM_MDIICONARRANGE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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_MDITILE

Conceptual

Multiple Document Interface

WM_MDIMAXIMIZE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_MDINEXT message

2/22/2020 • 2 minutes to read • Edit Online

An application sends the **WM_MDINEXT** message to a multiple-document interface (MDI) client window to activate the next or previous child window.

```
#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

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

WM_MDIREFRESHMENU message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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_MDISETMENU

Conceptual

Multiple Document Interface

WM_MDIRESTORE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

Conceptual

Multiple Document Interface

WM_MDISETMENU message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

WM_MDITILE message

2/22/2020 • 2 minutes to read • Edit Online

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.

```
#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

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

MDI Structures

2/22/2020 • 2 minutes to read • Edit Online

- MDICREATESTRUCT