Contents

WM_LBUTTONDBLCLK

WM_LBUTTONDOWN

WM_LBUTTONUP

WM_MBUTTONDBLCLK

WM_MBUTTONDOWN

WM_MBUTTONUP

WM_MOUSEACTIVATE

WM_MOUSEHOVER

WM_MOUSEHWHEEL

WM_MOUSELEAVE

WM_MOUSEMOVE

WM_MOUSEWHEEL

WM_NCHITTEST

WM_NCLBUTTONDBLCLK

WM_NCLBUTTONDOWN

WM_NCLBUTTONUP

WM_NCMBUTTONDBLCLK

WM_NCMBUTTONDOWN

WM_NCMBUTTONUP

WM_NCMOUSEHOVER

WM_NCMOUSELEAVE

WM_NCMOUSEMOVE

WM_NCRBUTTONDBLCLK

WM_NCRBUTTONDOWN

WM_NCRBUTTONUP

WM_NCXBUTTONDBLCLK

WM_NCXBUTTONDOWN

WM_NCXBUTTONUP

WM_RBUTTONDBLCLK

WM_RBUTTONDOWN

WM_RBUTTONUP

WM_XBUTTONDBLCLK

RID_DEVICE_INFO_HID

RID_DEVICE_INFO_KEYBOARD

RID_DEVICE_INFO_MOUSE

Keyboard and Mouse Input

4/13/2022 • 2 minutes to read • Edit Online

The following sections describe methods of capturing user input.

In This Section

NAME	DESCRIPTION
Keyboard Input	Discusses how the system generates keyboard input and how an application receives and processes that input.
Mouse Input	Discusses how the system provides mouse input to your application and how the application receives and processes that input.
Raw Input	Discusses how the system provides raw input to your application and how an application receives and processes that input.

Keyboard Input (Keyboard and Mouse Input)

4/13/2022 • 7 minutes to read • Edit Online

This section describes how the system generates keyboard input and how an application receives and processes that input.

In This Section

NAME	DESCRIPTION
About Keyboard Input	Discusses keyboard input.
Using Keyboard Input	Covers tasks that are associated with keyboard input.
Keyboard Input Reference	Contains the API reference.

Functions

NAME	DESCRIPTION
ActivateKeyboardLayout	Sets the input locale identifier (formerly called the keyboard layout handle) for the calling thread or the current process. The input locale identifier specifies a locale as well as the physical layout of the keyboard.
BlockInput	Blocks keyboard and mouse input events from reaching applications.
EnableWindow	Enables or disables mouse and keyboard input to the specified window or control. When input is disabled, the window does not receive input such as mouse clicks and key presses. When input is enabled, the window receives all input.
GetActiveWindow	Retrieves the window handle to the active window attached to the calling thread's message queue.
GetAsyncKeyState	Determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to GetAsyncKeyState.
GetFocus	Retrieves the handle to the window that has the keyboard focus, if the window is attached to the calling thread's message queue.
GetKeyboardLayout	Retrieves the active input locale identifier (formerly called the keyboard layout) for the specified thread. If the *idThread* parameter is zero, the input locale identifier for the active thread is returned.

NAME	DESCRIPTION
GetKeyboardLayoutList	Retrieves the input locale identifiers (formerly called keyboard layout handles) corresponding to the current set of input locales in the system. The function copies the identifiers to the specified buffer.
GetKeyboardLayoutName	Retrieves the name of the active input locale identifier (formerly called the keyboard layout).
GetKeyboardState	Copies the status of the 256 virtual keys to the specified buffer.
GetKeyNameText	Retrieves a string that represents the name of a key.
GetKeyState	Retrieves the status of the specified virtual key. The status specifies whether the key is up, down, or toggled (on, off alternating each time the key is pressed).
GetLastInputInfo	Retrieves the time of the last input event.
IsWindowEnabled	Determines whether the specified window is enabled for mouse and keyboard input.
LoadKeyboardLayout	Loads a new input locale identifier (formerly called the keyboard layout) into the system. Several input locale identifiers can be loaded at a time, but only one per process is active at a time. Loading multiple input locale identifiers makes it possible to rapidly switch between them.
MapVirtualKey	Translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. To specify a handle to the keyboard layout to use for translating the specified code, use the MapVirtualKeyEx function.
MapVirtualKeyEx	Maps a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. The function translates the codes using the input language and an input locale identifier.
OemKeyScan	Maps OEMASCII codes 0 through 0x0FF into the OEM scan codes and shift states. The function provides information that allows a program to send OEM text to another program by simulating keyboard input.
RegisterHotKey	Defines a system-wide hot key.
SendInput	Synthesizes keystrokes, mouse motions, and button clicks.
SetActiveWindow	Activates a window. The window must be attached to the calling thread's message queue.
SetFocus	Sets the keyboard focus to the specified window. The window must be attached to the calling thread's message queue.

NAME	DESCRIPTION
SetKeyboardState	Copies a 256-byte array of keyboard key states into the calling thread's keyboard input-state table. This is the same table accessed by the GetKeyboardState and GetKeyState functions. Changes made to this table do not affect keyboard input to any other thread.
ToAscii	Translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the keyboard layout handle. To specify a handle to the keyboard layout to use to translate the specified code, use the ToAsciiEx function.
ToAsciiEx	Translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the input locale identifier.
ToUnicode	Translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters. To specify a handle to the keyboard layout to use to translate the specified code, use the ToUnicodeEx function.
ToUnicodeEx	Translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.
UnloadKeyboardLayout	Unloads an input locale identifier (formerly called a keyboard layout).
UnregisterHotKey	Frees a hot key previously registered by the calling thread.
VkKeyScanEx	Translates a character to the corresponding virtual-key code and shift state. The function translates the character using the input language and physical keyboard layout identified by the input locale identifier.

The following functions are obsolete.

FUNCTION	DESCRIPTION
GetKBCodePage	Retrieves the current code page.
keybd_event	Synthesizes a keystroke. The system can use such a synthesized keystroke to generate a WM_KEYUP or WM_KEYDOWN message. The keyboard driver's interrupt handler calls the keybd_event function.
VkKeyScan	Translates a character to the corresponding virtual-key code and shift state for the current keyboard.

Messages

NAME	DESCRIPTION
WM_GETHOTKEY	Determines the hot key associated with a window.

NAME	DESCRIPTION
WM_SETHOTKEY	Associates a hot key with the window. When the user presses the hot key, the system activates the window.

Notifications

NAME	DESCRIPTION
WM_ACTIVATE	Sent to both the window being activated and the window being deactivated. If the windows use the same input queue, the message is sent synchronously, first to the window procedure of the top-level window being deactivated, then to the window procedure of the top-level window being activated. If the windows use different input queues, the message is sent asynchronously, so the window is activated immediately.
WM_APPCOMMAND	Notifies a window that the user generated an application command event, for example, by clicking an application command button using the mouse or typing an application command key on the keyboard.
WM_CHAR	Posted to the window with the keyboard focus when a WM_KEYDOWN message is translated by the TranslateMessage function. The WM_CHAR message contains the character code of the key that was pressed.
WM_DEADCHAR	Posted to the window with the keyboard focus when a WM_KEYUP message is translated by the TranslateMessage function. WM_DEADCHAR specifies a character code generated by a dead key. A dead key is a key that generates a character, such as the umlaut (double-dot), that is combined with another character to form a composite character. For example, the umlaut-O character ( ) is generated by typing the dead key for the umlaut character, and then typing the O key.
WM_HOTKEY	Posted when the user presses a hot key registered by the RegisterHotKey function. The message is placed at the top of the message queue associated with the thread that registered the hot key.
WM_KEYDOWN	Posted to the window with the keyboard focus when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is not pressed.
WM_KEYUP	Posted to the window with the keyboard focus when a nonsystem key is released. A nonsystem key is a key that is pressed when the ALT key is not pressed, or a keyboard key that is pressed when a window has the keyboard focus.
WM_KILLFOCUS	Sent to a window immediately before it loses the keyboard focus.
WM_SETFOCUS	Sent to a window after it has gained the keyboard focus.

NAME	DESCRIPTION
WM_SYSDEADCHAR	Sent to the window with the keyboard focus when a WM_SYSKEYDOWN message is translated by the TranslateMessage function. WM_SYSDEADCHAR specifies the character code of a system dead key that is, a dead key that is pressed while holding down the ALT key.
WM_SYSKEYDOWN	Posted to the window with the keyboard focus when the user presses the F10 key (which activates the menu bar) or holds down the ALT key and then presses another key. It also occurs when no window currently has the keyboard focus; in this case, the WM_SYSKEYDOWN message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.
WM_SYSKEYUP	Posted to the window with the keyboard focus when the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the keyboard focus; in this case, the WM_SYSKEYUP message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.
WM_UNICHAR	Posted to the window with the keyboard focus when a WM_KEYDOWN message is translated by the TranslateMessage function. The WM_UNICHAR message contains the character code of the key that was pressed.

Structures

NAME	DESCRIPTION
HARDWAREINPUT	Contains information about a simulated message generated by an input device other than a keyboard or mouse.
INPUT	Contains information used for synthesizing input events such as keystrokes, mouse movement, and mouse clicks.
KEYBDINPUT	Contains information about a simulated keyboard event.
LASTINPUTINFO	Contains the time of the last input.
MOUSEINPUT	Contains information about a simulated mouse event.

Constants

NAME	DESCRIPTION
Virtual-Key Codes	The symbolic constant names, hexadecimal values, and mouse or keyboard equivalents for the virtual-key codes used by the system. The codes are listed in numeric order.

About Keyboard Input

4/13/2022 • 20 minutes to read • Edit Online

Applications should accept user input from the keyboard as well as from the mouse. An application receives keyboard input in the form of messages posted to its windows.

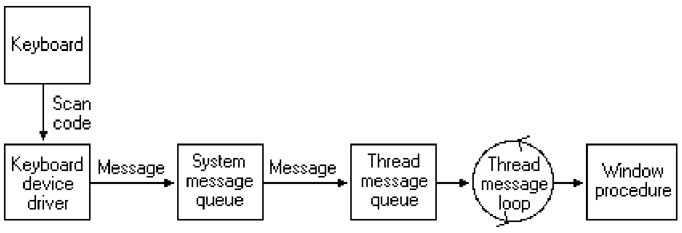
This section covers the following topics:

Keyboard Input Model

The system provides device-independent keyboard support for applications by installing a keyboard device driver appropriate for the current keyboard. The system provides language-independent keyboard support by using the language-specific keyboard layout currently selected by the user or the application. The keyboard device driver receives scan codes from the keyboard, which are sent to the keyboard layout where they are translated into messages and posted to the appropriate windows in your application.

Assigned to each key on a keyboard is a unique value called a *scan code*, a device-dependent identifier for the key on the keyboard. A keyboard generates two scan codes when the user types a key—one when the user presses the key and another when the user releases the key.

The keyboard device driver interprets a scan code and translates (maps) it to a *virtual-key code*, a device-independent value defined by the system that identifies the purpose of a key. After translating a scan code, the keyboard layout creates a message that includes the scan code, the virtual-key code, and other information about the keystroke, and then places the message in the system message queue. The system removes the message from the system message queue and posts it to the message queue of the appropriate thread. Eventually, the thread's message loop removes the message and passes it to the appropriate window procedure for processing. The following figure illustrates the keyboard input model.



Keyboard Focus and Activation

The system posts keyboard messages to the message queue of the foreground thread that created the window with the keyboard focus. The *keyboard focus* is a temporary property of a window. The system shares the keyboard among all windows on the display by shifting the keyboard focus, at the user's direction, from one window to another. The window that has the keyboard focus receives (from the message queue of the thread that created it) all keyboard messages until the focus changes to a different window.

A thread can call the **GetFocus** function to determine which of its windows (if any) currently has the keyboard focus. A thread can give the keyboard focus to one of its windows by calling the **SetFocus** function. When the keyboard focus changes from one window to another, the system sends a **WM_KILLFOCUS** message to the window that has lost the focus, and then sends a **WM_SETFOCUS** message to the window that has gained the focus.

The concept of keyboard focus is related to that of the active window. The *active window* is the top-level window the user is currently working with. The window with the keyboard focus is either the active window, or a child window of the active window. To help the user identify the active window, the system places it at the top of the Z order and highlights its title bar (if it has one) and border.

The user can activate a top-level window by clicking it, selecting it using the ALT+TAB or ALT+ESC key combination, or selecting it from the Task List. A thread can activate a top-level window by using the **SetActiveWindow** function. It can determine whether a top-level window it created is active by using the **GetActiveWindow** function.

When one window is deactivated and another activated, the system sends the **WM_ACTIVATE** message. The low-order word of the *wParam* parameter is zero if the window is being deactivated and nonzero if it is being activated. When the default window procedure receives the **WM_ACTIVATE** message, it sets the keyboard focus to the active window.

To block keyboard and mouse input events from reaching applications, use **BlockInput**. Note, the **BlockInput** function will not interfere with the asynchronous keyboard input-state table. This means that calling the **SendInput** function while input is blocked will change the asynchronous keyboard input-state table.

Keystroke Messages

Pressing a key causes a **WM_KEYDOWN** or **WM_SYSKEYDOWN** message to be placed in the thread message queue attached to the window that has the keyboard focus. Releasing a key causes a **WM_KEYUP** or **WM_SYSKEYUP** message to be placed in the queue.

Key-up and key-down messages typically occur in pairs, but if the user holds down a key long enough to start the keyboard's automatic repeat feature, the system generates a number of **WM_KEYDOWN** or **WM_SYSKEYDOWN** messages in a row. It then generates a single **WM_KEYUP** or **WM_SYSKEYUP** message when the user releases the key.

This section covers the following topics:

- System and Nonsystem Keystrokes
- Virtual-Key Codes Described
- Keystroke Message Flags

System and Nonsystem Keystrokes

The system makes a distinction between system keystrokes and nonsystem keystrokes. System keystrokes produce system keystroke messages, WM_SYSKEYDOWN and WM_SYSKEYUP. Nonsystem keystrokes produce nonsystem keystroke messages, WM_KEYDOWN and WM_KEYUP.

If your window procedure must process a system keystroke message, make sure that after processing the message the procedure passes it to the DefWindowProc function. Otherwise, all system operations involving the ALT key will be disabled whenever the window has the keyboard focus. That is, the user won't be able to access the window's menus or System menu, or use the ALT+ESC or ALT+TAB key combination to activate a different window.

System keystroke messages are primarily for use by the system rather than by an application. The system uses them to provide its built-in keyboard interface to menus and to allow the user to control which window is active. System keystroke messages are generated when the user types a key in combination with the ALT key, or when the user types and no window has the keyboard focus (for example, when the active application is minimized). In this case, the messages are posted to the message queue attached to the active window.

Nonsystem keystroke messages are for use by application windows; the DefWindowProc function does nothing with them. A window procedure can discard any nonsystem keystroke messages that it does not need.

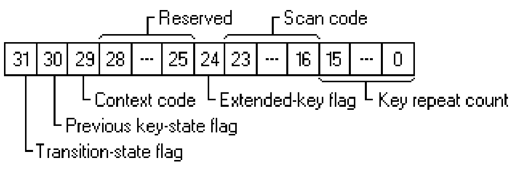
Virtual-Key Codes Described

The **wParam** parameter of a keystroke message contains the virtual-key code of the key that was pressed or released. A window procedure processes or ignores a keystroke message, depending on the value of the virtual-key code.

A typical window procedure processes only a small subset of the keystroke messages that it receives and ignores the rest. For example, a window procedure might process only WM_KEYDOWN keystroke messages, and only those that contain virtual-key codes for the cursor movement keys, shift keys (also called control keys), and function keys. A typical window procedure does not process keystroke messages from character keys. Instead, it uses the TranslateMessage function to convert the message into character messages. For more information about **TranslateMessage** and character messages, see Character Messages.

Keystroke Message Flags

The **lParam** parameter of a keystroke message contains additional information about the keystroke that generated the message. This information includes the repeat count, the scan code, the extended-key flag, the context code, the previous key-state flag, and the transition-state flag. The following illustration shows the locations of these flags and values in the **lParam** parameter.



An application can use the following values to manipulate the keystroke flags.

VALUE	DESCRIPTION
KF_ALTDOWN	Manipulates the ALT key flag, which indicates whether the ALT key is pressed.
KF_DLGMODE	Manipulates the dialog mode flag, which indicates whether a dialog box is active.
KF_EXTENDED	Manipulates the extended key flag.

VALUE	DESCRIPTION
KF_MENUMODE	Manipulates the menu mode flag, which indicates whether a menu is active.
KF_REPEAT	Manipulates the previous key state flag.
KF_UP	Manipulates the transition state flag.

Example code:

```
case WM_KEYDOWN:
case WM_KEYUP:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
{
    WORD vkCode = LOWORD(wParam);                                 // virtual-key code

    BYTE scanCode = LOBYTE(HIWORD(lParam));                       // scan code
    BOOL scanCodeE0 = (HIWORD(lParam) & KF_EXTENDED) == KF_EXTENDED;    // extended-key flag, 1 if scancode
has 0xE0 prefix

    BOOL upFlag = (HIWORD(lParam) & KF_UP) == KF_UP;              // transition-state flag, 1 on keyup
    BOOL repeatFlag = (HIWORD(lParam) & KF_REPEAT) == KF_REPEAT;        // previous key-state flag, 1 on
autorepeat
    WORD repeatCount = LOWORD(lParam);                            // repeat count, > 0 if several
keydown messages was combined into one message

    BOOL altDownFlag = (HIWORD(lParam) & KF_ALTDOWN) == KF_ALTDOWN;     // ALT key was pressed

    BOOL dlgModeFlag = (HIWORD(lParam) & KF_DLGMODE) == KF_DLGMODE;     // dialog box is active
    BOOL menuModeFlag = (HIWORD(lParam) & KF_MENUMODE) == KF_MENUMODE;  // menu is active

    // ...
}
break;
```

Repeat Count

You can check the repeat count to determine whether a keystroke message represents more than one keystroke. The system increments the count when the keyboard generates WM_KEYDOWN or WM_SYSKEYDOWN messages faster than an application can process them. This often occurs when the user holds down a key long enough to start the keyboard's automatic repeat feature. Instead of filling the system message queue with the resulting key-down messages, the system combines the messages into a single key down message and increments the repeat count. Releasing a key cannot start the automatic repeat feature, so the repeat count for WM_KEYUP and WM_SYSKEYUP messages is always set to 1.

Scan Code

The scan code is the value that the keyboard hardware generates when the user presses a key. It is a device-dependent value that identifies the key pressed, as opposed to the character represented by the key. An application typically ignores scan codes. Instead, it uses the device-independent virtual-key codes to interpret keystroke messages.

Extended-Key Flag

The extended-key flag indicates whether the keystroke message originated from one of the additional keys on the enhanced keyboard. The extended keys consist of the ALT and CTRL keys on the right-hand side of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; the NUM LOCK key; the BREAK (CTRL+PAUSE) key; the PRINT SCRN key; and the divide (/) and ENTER keys in the numeric keypad. The extended-key flag is set if the key is an extended key.

If specified, the scan code was preceded by a prefix byte having the value 0xE0 (224).

Context Code

The context code indicates whether the ALT key was down when the keystroke message was generated. The code is 1 if the ALT key was down and 0 if it was up.

Previous Key-State Flag

The previous key-state flag indicates whether the key that generated the keystroke message was previously up or down. It is 1 if the key was previously down and 0 if the key was previously up. You can use this flag to identify keystroke messages generated by the keyboard's automatic repeat feature. This flag is set to 1 for **WM_KEYDOWN** and **WM_SYSKEYDOWN** keystroke messages generated by the automatic repeat feature. It is always set to 1 for **WM_KEYUP** and **WM_SYSKEYUP** messages.

Transition-State Flag

The transition-state flag indicates whether pressing a key or releasing a key generated the keystroke message. This flag is always set to 0 for **WM_KEYDOWN** and **WM_SYSKEYDOWN** messages; it is always set to 1 for **WM_KEYUP** and **WM_SYSKEYUP** messages.

Character Messages

Keystroke messages provide a lot of information about keystrokes, but they do not provide character codes for character keystrokes. To retrieve character codes, an application must include the **TranslateMessage** function in its thread message loop. **TranslateMessage** passes a **WM_KEYDOWN** or **WM_SYSKEYDOWN** message to the keyboard layout. The layout examines the message's virtual-key code and, if it corresponds to a character key, provides the character code equivalent (taking into account the state of the SHIFT and CAPS LOCK keys). It then generates a character message that includes the character code and places the message at the top of the message queue. The next iteration of the message loop removes the character message from the queue and dispatches the message to the appropriate window procedure.

This section covers the following topics:

- Nonsystem Character Messages
- Dead-Character Messages

Nonsystem Character Messages

A window procedure can receive the following character messages: **WM_CHAR**, **WM_DEADCHAR**, **WM_SYSCHAR**, **WM_SYSDEADCHAR**, and **WM_UNICHAR**. The **TranslateMessage** function generates a **WM_CHAR** or **WM_DEADCHAR** message when it processes a **WM_KEYDOWN** message. Similarly, it generates a **WM_SYSCHAR** or **WM_SYSDEADCHAR** message when it processes a **WM_SYSKEYDOWN** message.

An application that processes keyboard input typically ignores all but the **WM_CHAR** and **WM_UNICHAR** messages, passing any other messages to the **DefWindowProc** function. Note that **WM_CHAR** uses 16-bit Unicode Transformation Format (UTF) while **WM_UNICHAR** uses UTF-32. The system uses the **WM_SYSCHAR** and **WM_SYSDEADCHAR** messages to implement menu mnemonics.

The **wParam** parameter of all character messages contains the character code of the character key that was pressed. The value of the character code depends on the window class of the window receiving the message. If the Unicode version of the **RegisterClass** function was used to register the window class, the system provides Unicode characters to all windows of that class. Otherwise, the system provides ASCII character codes. For more information, see Unicode and Character Sets.

The contents of the **lParam** parameter of a character message are identical to the contents of the **lParam** parameter of the key-down message that was translated to produce the character message. For information, see Keystroke Message Flags.

Dead-Character Messages

Some non-English keyboards contain character keys that are not expected to produce characters by themselves. Instead, they are used to add a diacritic to the character produced by the subsequent keystroke. These keys are called *dead keys*. The circumflex key on a German keyboard is an example of a dead key. To enter the character consisting of an "o" with a circumflex, a German user would type the circumflex key followed by the "o" key. The window with the keyboard focus would receive the following sequence of messages:

1. WM_KEYDOWN
2. WM_DEADCHAR
3. WM_KEYUP
4. WM_KEYDOWN
5. WM_CHAR
6. WM_KEYUP

TranslateMessage generates the WM_DEADCHAR message when it processes the WM_KEYDOWN message from a dead key. Although the *wParam* parameter of the **WM_DEADCHAR** message contains the character code of the diacritic for the dead key, an application typically ignores the message. Instead, it processes the WM_CHAR message generated by the subsequent keystroke. The *wParam* parameter of the **WM_CHAR** message contains the character code of the letter with the diacritic. If the subsequent keystroke generates a character that cannot be combined with a diacritic, the system generates two **WM_CHAR** messages. The *wParam* parameter of the first contains the character code of the diacritic; the *wParam* parameter of the second contains the character code of the subsequent character key.

The TranslateMessage function generates the WM_SYSDEADCHAR message when it processes the WM_SYSKEYDOWN message from a system dead key (a dead key that is pressed in combination with the ALT key). An application typically ignores the **WM_SYSDEADCHAR** message.

Key Status

While processing a keyboard message, an application may need to determine the status of another key besides the one that generated the current message. For example, a word-processing application that allows the user to press SHIFT+END to select a block of text must check the status of the SHIFT key whenever it receives a keystroke message from the END key. The application can use the GetKeyState function to determine the status of a virtual key at the time the current message was generated; it can use the GetAsyncKeyState function to retrieve the current status of a virtual key.

The keyboard layout maintains a list of names. The name of a key that produces a single character is the same as the character produced by the key. The name of a noncharacter key such as TAB and ENTER is stored as a character string. An application can retrieve the name of any key from the device driver by calling the GetKeyNameText function.

Keystroke and Character Translations

The system includes several special purpose functions that translate scan codes, character codes, and virtual-key codes provided by various keystroke messages. These functions include MapVirtualKey, ToAscii, ToUnicode, and VkKeyScan.

In addition, Microsoft Rich Edit 3.0 supports the HexToUnicode IME, which allows a user to convert between hexadecimal and Unicode characters by using hot keys. This means that when Microsoft Rich Edit 3.0 is incorporated into an application, the application will inherit the features of the HexToUnicode IME.

Hot-Key Support

A *hot key* is a key combination that generates a **WM_HOTKEY** message, a message the system places at the top of a thread's message queue, bypassing any existing messages in the queue. Applications use hot keys to obtain high-priority keyboard input from the user. For example, by defining a hot key consisting of the CTRL+C key combination, an application can allow the user to cancel a lengthy operation.

To define a hot key, an application calls the **RegisterHotKey** function, specifying the combination of keys that generates the **WM_HOTKEY** message, the handle to the window to receive the message, and the identifier of the hot key. When the user presses the hot key, a **WM_HOTKEY** message is placed in the message queue of the thread that created the window. The *wParam* parameter of the message contains the identifier of the hot key. The application can define multiple hot keys for a thread, but each hot key in the thread must have a unique identifier. Before the application terminates, it should use the **UnregisterHotKey** function to destroy the hot key.

Applications can use a hot key control to make it easy for the user to choose a hot key. Hot key controls are typically used to define a hot key that activates a window; they do not use the **RegisterHotKey** and **UnregisterHotKey** functions. Instead, an application that uses a hot key control typically sends the **WM_SETHOTKEY** message to set the hot key. Whenever the user presses the hot key, the system sends a **WM_SYSCOMMAND** message specifying SC_HOTKEY. For more information about hot key controls, see "Using Hot Key Controls" in Hot Key Controls.

Keyboard Keys for Browsing and Other Functions

Windows provides support for keyboards with special keys for browser functions, media functions, application launching, and power management. The **WM_APPCOMMAND** supports the extra keyboard keys. In addition, the **ShellProc** function is modified to support the extra keyboard keys.

It is unlikely that a child window in a component application will be able to directly implement commands for these extra keyboard keys. So when one of these keys is pressed, **DefWindowProc** will send a **WM_APPCOMMAND** message to a window. **DefWindowProc** will also bubble the **WM_APPCOMMAND** message to its parent window. This is similar to the way context menus are invoked with the right mouse button, which is that **DefWindowProc** sends a **WM_CONTEXTMENU** message on a right button click, and bubbles it to its parent. Additionally, if **DefWindowProc** receives a **WM_APPCOMMAND** message for a top-level window, it will call a shell hook with code **HSHELL_APPCOMMAND**.

Windows also supports the Microsoft IntelliMouse Explorer, which is a mouse with five buttons. The two extra buttons support forward and backward browser navigation. For more information, see XBUTTONs.

Simulating Input

To simulate an uninterrupted series of user input events, use the **SendInput** function. The function accepts three parameters. The first parameter, *cInputs*, indicates the number of input events that will be simulated. The second parameter, *rgInputs*, is an array of **INPUT** structures, each describing a type of input event and additional information about that event. The last parameter, *cbSize*, accepts the size of the **INPUT** structure, in bytes.

The **SendInput** function works by injecting a series of simulated input events into a device's input stream. The effect is similar to calling the **keybd_event** or **mouse_event** function repeatedly, except that the system ensures that no other input events intermingle with the simulated events. When the call completes, the return value indicates the number of input events successfully played. If this value is zero, then input was blocked.

The **SendInput** function does not reset the keyboard's current state. Therefore, if the user has any keys pressed when you call this function, they might interfere with the events that this function generates. If you are concerned about possible interference, check the keyboard's state with the **GetAsyncKeyState** function and correct as necessary.

Languages, Locales, and Keyboard Layouts

A *language* is a natural language, such as English, French, and Japanese. A *sublanguage* is a variant of a natural language that is spoken in a specific geographical region, such as the English sublanguages spoken in the United Kingdom and the United States. Applications use values, called language identifiers, to uniquely identify languages and sublanguages.

Applications typically use *locales* to set the language in which input and output is processed. Setting the locale for the keyboard, for example, affects the character values generated by the keyboard. Setting the locale for the display or printer affects the glyphs displayed or printed. Applications set the locale for a keyboard by loading and using keyboard layouts. They set the locale for a display or printer by selecting a font that supports the specified locale.

A keyboard layout not only specifies the physical position of the keys on the keyboard but also determines the character values generated by pressing those keys. Each layout identifies the current input language and determines which character values are generated by which keys and key combinations.

Every keyboard layout has a corresponding handle that identifies the layout and language. The low word of the handle is a language identifier. The high word is a device handle, specifying the physical layout, or is zero, indicating a default physical layout. The user can associate any input language with a physical layout. For example, an English-speaking user who very occasionally works in French can set the input language of the keyboard to French without changing the physical layout of the keyboard. This means the user can enter text in French using the familiar English layout.

Applications are generally not expected to manipulate input languages directly. Instead, the user sets up language and layout combinations, then switches among them. When the user clicks into text marked with a different language, the application calls the **ActivateKeyboardLayout** function to activate the user's default layout for that language. If the user edits text in a language which is not in the active list, the application can call the **LoadKeyboardLayout** function with the language to get a layout based on that language.

The **ActivateKeyboardLayout** function sets the input language for the current task. The *hkl* parameter can be either the handle to the keyboard layout or a zero-extended language identifier. Keyboard layout handles can be obtained from the **LoadKeyboardLayout** or **GetKeyboardLayoutList** function. The **HKL_NEXT** and **HKL_PREV** values can also be used to select the next or previous keyboard.

The **GetKeyboardLayoutName** function retrieves the name of the active keyboard layout for the calling thread. If an application creates the active layout using the **LoadKeyboardLayout** function, **GetKeyboardLayoutName** retrieves the same string used to create the layout. Otherwise, the string is the primary language identifier corresponding to the locale of the active layout. This means the function may not necessarily differentiate among different layouts with the same primary language, so cannot return specific information about the input language. The **GetKeyboardLayout** function, however, can be used to determine the input language.

The **LoadKeyboardLayout** function loads a keyboard layout and makes the layout available to the user. Applications can make the layout immediately active for the current thread by using the **KLF_ACTIVATE** value. An application can use the **KLF_REORDER** value to reorder the layouts without also specifying the **KLF_ACTIVATE** value. Applications should always use the **KLF_SUBSTITUTE_OK** value when loading keyboard layouts to ensure that the user's preference, if any, is selected.

For multilingual support, the **LoadKeyboardLayout** function provides the **KLF_REPLACELANG** and **KLF_NOTELLSHELL** flags. The **KLF_REPLACELANG** flag directs the function to replace an existing keyboard layout without changing the language. Attempting to replace an existing layout using the same language identifier but without specifying **KLF_REPLACELANG** is an error. The **KLF_NOTELLSHELL** flag prevents the function from notifying the shell when a keyboard layout is added or replaced. This is useful for applications that add multiple layouts in a consecutive series of calls. This flag should be used in all but the last call.

The **UnloadKeyboardLayout** function is restricted in that it cannot unload the system default input language. This ensures that the user always has one layout available for enter text using the same character set as used by the shell and file system.

Using Keyboard Input

4/13/2022 • 10 minutes to read • Edit Online

A window receives keyboard input in the form of keystroke messages and character messages. The message loop attached to the window must include code to translate keystroke messages into the corresponding character messages. If the window displays keyboard input in its client area, it should create and display a caret to indicate the position where the next character will be entered. The following sections describe the code involved in receiving, processing, and displaying keyboard input:

- Processing Keystroke Messages
- Translating Character Messages
- Processing Character Messages
- Using the Caret
- Displaying Keyboard Input

Processing Keystroke Messages

The window procedure of the window that has the keyboard focus receives keystroke messages when the user types at the keyboard. The keystroke messages are **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, and **WM_SYSKEYUP**. A typical window procedure ignores all keystroke messages except **WM_KEYDOWN**. The system posts the **WM_KEYDOWN** message when the user presses a key.

When the window procedure receives the **WM_KEYDOWN** message, it should examine the virtual-key code that accompanies the message to determine how to process the keystroke. The virtual-key code is in the message's *wParam* parameter. Typically, an application processes only keystrokes generated by noncharacter keys, including the function keys, the cursor movement keys, and the special purpose keys such as INS, DEL, HOME, and END.

The following example shows the window procedure framework that a typical application uses to receive and process keystroke messages.

```
        case WM_KEYDOWN:
            switch (wParam)
            {
                case VK_LEFT:

                    // Process the LEFT ARROW key.

                    break;

                case VK_RIGHT:

                    // Process the RIGHT ARROW key.

                    break;

                case VK_UP:

                    // Process the UP ARROW key.

                    break;

                case VK_DOWN:

                    // Process the DOWN ARROW key.

                    break;

                case VK_HOME:

                    // Process the HOME key.

                    break;

                case VK_END:

                    // Process the END key.

                    break;

                case VK_INSERT:

                    // Process the INS key.

                    break;

                case VK_DELETE:

                    // Process the DEL key.

                    break;

                case VK_F2:

                    // Process the F2 key.

                    break;


                // Process other non-character keystrokes.

                default:
                    break;
            }
```

Translating Character Messages

Any thread that receives character input from the user must include the TranslateMessage function in its message loop. This function examines the virtual-key code of a keystroke message and, if the code corresponds to a character, places a character message into the message queue. The character message is removed and dispatched on the next iteration of the message loop; the *wParam* parameter of the message contains the character code.

In general, a thread's message loop should use the TranslateMessage function to translate every message, not just virtual-key messages. Although **TranslateMessage** has no effect on other types of messages, it guarantees that keyboard input is translated correctly. The following example shows how to include the **TranslateMessage** function in a typical thread message loop.

```
MSG msg;
BOOL bRet;

while (( bRet = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0)
{
    if (bRet == -1);
    {
        // handle the error and possibly exit
    }
    else
    {
        if (TranslateAccelerator(hwndMain, haccl, &msg) == 0)
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}
```

Processing Character Messages

A window procedure receives a character message when the TranslateMessage function translates a virtual-key code corresponding to a character key. The character messages are WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, and WM_SYSDEADCHAR. A typical window procedure ignores all character messages except **WM_CHAR**. The **TranslateMessage** function generates a **WM_CHAR** message when the user presses any of the following keys:

- Any character key
- BACKSPACE
- ENTER (carriage return)
- ESC
- SHIFT+ENTER (linefeed)
- TAB

When a window procedure receives the WM_CHAR message, it should examine the character code that accompanies the message to determine how to process the character. The character code is in the message's *wParam* parameter.

The following example shows the window procedure framework that a typical application uses to receive and process character messages.

```
        case WM_CHAR:
            switch (wParam)
            {
                case 0x08:

                    // Process a backspace.

                    break;

                case 0x0A:

                    // Process a linefeed.

                    break;

                case 0x1B:

                    // Process an escape.

                    break;

                case 0x09:

                    // Process a tab.

                    break;

                case 0x0D:

                    // Process a carriage return.

                    break;

                default:

                    // Process displayable characters.

                    break;
            }
```

Using the Caret

A window that receives keyboard input typically displays the characters the user types in the window's client area. A window should use a caret to indicate the position in the client area where the next character will appear. The window should also create and display the caret when it receives the keyboard focus, and hide and destroy the caret when it loses the focus. A window can perform these operations in the processing of the **WM_SETFOCUS** and **WM_KILLFOCUS** messages. For more information about carets, see Carets.

Displaying Keyboard Input

The example in this section shows how an application can receive characters from the keyboard, display them in the client area of a window, and update the position of the caret with each character typed. It also demonstrates how to move the caret in response to the LEFT ARROW, RIGHT ARROW, HOME, and END keystrokes, and shows how to highlight selected text in response to the SHIFT+RIGHT ARROW key combination.

During processing of the **WM_CREATE** message, the window procedure shown in the example allocates a 64K buffer for storing keyboard input. It also retrieves the metrics of the currently loaded font, saving the height and average width of characters in the font. The height and width are used in processing the **WM_SIZE** message to calculate the line length and maximum number of lines, based on the size of the client area.

The window procedure creates and displays the caret when processing the **WM_SETFOCUS** message. It hides and deletes the caret when processing the WM_KILLFOCUS message.

When processing the WM_CHAR message, the window procedure displays characters, stores them in the input buffer, and updates the caret position. The window procedure also converts tab characters to four consecutive space characters. Backspace, linefeed, and escape characters generate a beep, but are not otherwise processed.

The window procedure performs the left, right, end, and home caret movements when processing the WM_KEYDOWN message. While processing the action of the RIGHT ARROW key, the window procedure checks the state of the SHIFT key and, if it is down, selects the character to the right of the caret as the caret is moved.

Note that the following code is written so that it can be compiled either as Unicode or as ANSI. If the source code defines UNICODE, strings are handled as Unicode characters; otherwise, they are handled as ANSI characters.

```
#define BUFSIZE 65535
#define SHIFTED 0x8000

LONG APIENTRY MainWndProc(HWND hwndMain, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;                   // handle to device context
    TEXTMETRIC tm;             // structure for text metrics
    static DWORD dwCharX;      // average width of characters
    static DWORD dwCharY;      // height of characters
    static DWORD dwClientX;    // width of client area
    static DWORD dwClientY;    // height of client area
    static DWORD dwLineLen;    // line length
    static DWORD dwLines;      // text lines in client area
    static int nCaretPosX = 0; // horizontal position of caret
    static int nCaretPosY = 0; // vertical position of caret
    static int nCharWidth = 0; // width of a character
    static int cch = 0;        // characters in buffer
    static int nCurChar = 0;   // index of current character
    static PTCHAR pchInputBuf; // input buffer
    int i, j;                  // loop counters
    int cCR = 0;               // count of carriage returns
    int nCRIndex = 0;          // index of last carriage return
    int nVirtKey;              // virtual-key code
    TCHAR szBuf[128];          // temporary buffer
    TCHAR ch;                  // current character
    PAINTSTRUCT ps;            // required by BeginPaint
    RECT rc;                   // output rectangle for DrawText
    SIZE sz;                   // string dimensions
    COLORREF crPrevText;       // previous text color
    COLORREF crPrevBk;         // previous background color
    size_t * pcch;
    HRESULT hResult;

    switch (uMsg)
    {
        case WM_CREATE:

            // Get the metrics of the current font.

            hdc = GetDC(hwndMain);
            GetTextMetrics(hdc, &tm);
            ReleaseDC(hwndMain, hdc);

            // Save the average character width and height.

            dwCharX = tm.tmAveCharWidth;
            dwCharY = tm.tmHeight;

            // Allocate a buffer to store keyboard input.
            pchInputBuf = (LPTSTR) GlobalAlloc(GPTR,
                BUFSIZE * sizeof(TCHAR));
            return 0;

        case WM_SIZE:

            // Save the new width and height of the client area.

            dwClientX = LOWORD(lParam);
            dwClientY = HIWORD(lParam);

            // Calculate the maximum width of a line and the
            // maximum number of lines in the client area.

            dwLineLen = dwClientX - dwCharX;
            dwLines = dwClientY / dwCharY;
            break;


        case WM_SETFOCUS:

            // Create, position, and display the caret when the
            // window receives the keyboard focus.

            CreateCaret(hwndMain, (HBITMAP) 1, 0, dwCharY);
            SetCaretPos(nCaretPosX, nCaretPosY * dwCharY);
            ShowCaret(hwndMain);
            break;

        case WM_KILLFOCUS:

            // Hide and destroy the caret when the window loses the
            // keyboard focus.

            HideCaret(hwndMain);
            DestroyCaret();
            break;

        case WM_CHAR:
        // check if current location is close enough to the
        // end of the buffer that a buffer overflow may
        // occur. If so, add null and display contents.
    if (cch > BUFSIZE-5)
    {
        pchInputBuf[cch] = 0x00;
        SendMessage(hwndMain, WM_PAINT, 0, 0);
    }
            switch (wParam)
            {
                case 0x08:  // backspace
                case 0x0A:  // linefeed
                case 0x1B:  // escape
                    MessageBeep((UINT) -1);
                    return 0;

                case 0x09:  // tab

                    // Convert tabs to four consecutive spaces.

                    for (i = 0; i < 4; i++)
                        SendMessage(hwndMain, WM_CHAR, 0x20, 0);
                    return 0;

                case 0x0D:  // carriage return

                    // Record the carriage return and position the
                    // caret at the beginning of the new line.

                    pchInputBuf[cch++] = 0x0D;
                    nCaretPosX = 0;
                    nCaretPosY += 1;
                    break;

                default:    // displayable character

                    ch = (TCHAR) wParam;
                    HideCaret(hwndMain);

                    // Retrieve the character's width and output
                    // the character.

                    hdc = GetDC(hwndMain);
                    GetCharWidth32(hdc, (UINT) wParam, (UINT) wParam,
                        &nCharWidth);
                    TextOut(hdc, nCaretPosX, nCaretPosY * dwCharY,
                        &ch, 1);
                    ReleaseDC(hwndMain, hdc);

                    // Store the character in the buffer.

                    pchInputBuf[cch++] = ch;

                    // Calculate the new horizontal position of the
                    // caret. If the position exceeds the maximum,
                    // insert a carriage return and move the caret
                    // to the beginning of the next line.

                    nCaretPosX += nCharWidth;
                    if ((DWORD) nCaretPosX > dwLineLen)
                    {
                        nCaretPosX = 0;
                        pchInputBuf[cch++] = 0x0D;
                        ++nCaretPosY;
                    }
                    nCurChar = cch;
                    ShowCaret(hwndMain);
                    break;
            }
            SetCaretPos(nCaretPosX, nCaretPosY * dwCharY);
            break;

        case WM_KEYDOWN:
            switch (wParam)
            {
                case VK_LEFT:   // LEFT ARROW

                    // The caret can move only to the beginning of
                    // the current line.

                    if (nCaretPosX > 0)
                    {
                        HideCaret(hwndMain);

                        // Retrieve the character to the left of
                        // the caret, calculate the character's
                        // width, then subtract the width from the
                        // current horizontal position of the caret
                        // to obtain the new position.

                        ch = pchInputBuf[--nCurChar];
                        hdc = GetDC(hwndMain);
                        GetCharWidth32(hdc, ch, ch, &nCharWidth);
                        ReleaseDC(hwndMain, hdc);
                        nCaretPosX = max(nCaretPosX - nCharWidth,
                            0);
                        ShowCaret(hwndMain);
                    }
                    break;
                    case VK_RIGHT:  // RIGHT ARROW

                        // Caret moves to the right or, when a carriage
                        // return is encountered, to the beginning of
                        // the next line.

                        if (nCurChar < cch)
                        {
                            HideCaret(hwndMain);

                            // Retrieve the character to the right of
                            // the caret. If it's a carriage return,
                            // position the caret at the beginning of
                            // the next line.

                            ch = pchInputBuf[nCurChar];
                            if (ch == 0x0D)
                            {
                                nCaretPosX = 0;
                                nCaretPosY++;
                            }

                            // If the character isn't a carriage
                            // return, check to see whether the SHIFT
                            // key is down. If it is, invert the text
                            // colors and output the character.

                            else
                            {
                                hdc = GetDC(hwndMain);
                                nVirtKey = GetKeyState(VK_SHIFT);
                                if (nVirtKey & SHIFTED)
                                {
                                    crPrevText = SetTextColor(hdc,
                                        RGB(255, 255, 255));
                                    crPrevBk = SetBkColor(hdc,
                                        RGB(0,0,0));
                                    TextOut(hdc, nCaretPosX,
                                        nCaretPosY * dwCharY,
                                        &ch, 1);
                                    SetTextColor(hdc, crPrevText);
                                    SetBkColor(hdc, crPrevBk);
                                }

                                // Get the width of the character and
                                // calculate the new horizontal
                                // position of the caret.

                                GetCharWidth32(hdc, ch, ch, &nCharWidth);
                                ReleaseDC(hwndMain, hdc);
                                nCaretPosX = nCaretPosX + nCharWidth;
                            }
                            nCurChar++;
                            ShowCaret(hwndMain);
                            break;
                        }
                        break;

                    case VK_UP:     // UP ARROW
                    case VK_DOWN:   // DOWN ARROW
                        MessageBeep((UINT) -1);
                        return 0;

                    case VK_HOME:   // HOME

                        // Set the caret's position to the upper left
                        // corner of the client area.
                    nCaretPosX = nCaretPosY = 0;
                    nCurChar = 0;
                    break;

                case VK_END:    // END

                    // Move the caret to the end of the text.

                    for (i=0; i < cch; i++)
                    {
                        // Count the carriage returns and save the
                        // index of the last one.

                        if (pchInputBuf[i] == 0x0D)
                        {
                            cCR++;
                            nCRIndex = i + 1;
                        }
                    }
                    nCaretPosY = cCR;

                    // Copy all text between the last carriage
                    // return and the end of the keyboard input
                    // buffer to a temporary buffer.

                    for (i = nCRIndex, j = 0; i < cch; i++, j++)
                        szBuf[j] = pchInputBuf[i];
                    szBuf[j] = TEXT('\0');

                    // Retrieve the text extent and use it
                    // to set the horizontal position of the
                    // caret.

                    hdc = GetDC(hwndMain);
                    hResult = StringCchLength(szBuf, 128, pcch);
                    if (FAILED(hResult))
                    {
                    // TODO: write error handler
                    }
                    GetTextExtentPoint32(hdc, szBuf, *pcch,
                        &sz);
                    nCaretPosX = sz.cx;
                    ReleaseDC(hwndMain, hdc);
                    nCurChar = cch;
                    break;

                default:
                    break;
            }
            SetCaretPos(nCaretPosX, nCaretPosY * dwCharY);
            break;

        case WM_PAINT:
            if (cch == 0)       // nothing in input buffer
                break;

            hdc = BeginPaint(hwndMain, &ps);
            HideCaret(hwndMain);

            // Set the clipping rectangle, and then draw the text
            // into it.

            SetRect(&rc, 0, 0, dwLineLen, dwClientY);
            DrawText(hdc, pchInputBuf, -1, &rc, DT_LEFT);

            ShowCaret(hwndMain);
            EndPaint(hwndMain, &ps);
            break;
            // Process other messages.

            case WM_DESTROY:
                PostQuitMessage(0);

                // Free the input buffer.

                GlobalFree((HGLOBAL) pchInputBuf);
                UnregisterHotKey(hwndMain, 0xAAAA);
                break;

            default:
                return DefWindowProc(hwndMain, uMsg, wParam, lParam);
        }
        return NULL;
    }
```

Keyboard Input Reference

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- Keyboard Input Functions
- Keyboard Input Messages
- Keyboard Input Notifications
- Keyboard Input Structures
- Keyboard Input Constants

Keyboard Input Functions

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- ActivateKeyboardLayout
- BlockInput
- EnableWindow
- GetActiveWindow
- GetAsyncKeyState
- GetFocus
- GetKBCodePage
- GetKeyboardLayout
- GetKeyboardLayoutList
- GetKeyboardLayoutName
- GetKeyboardState
- GetKeyboardType
- GetKeyNameText
- GetKeyState
- GetLastInputInfo
- IsWindowEnabled
- keybd_event
- LoadKeyboardLayout
- MapVirtualKey
- MapVirtualKeyEx
- OemKeyScan
- RegisterHotKey
- SendInput
- SetActiveWindow
- SetFocus
- SetKeyboardState
- ToAscii
- ToAsciiEx
- ToUnicode
- ToUnicodeEx
- UnloadKeyboardLayout
- UnregisterHotKey
- VkKeyScan
- VkKeyScanEx

Keyboard Input Messages

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- WM_GETHOTKEY
- WM_SETHOTKEY

WM_GETHOTKEY message

4/13/2022 • 2 minutes to read • Edit Online

Sent to determine the hot key associated with a window.

```
#define WM_GETHOTKEY                    0x0033
```

Parameters

wParam

Not used; must be zero.

lParam

Not used; must be zero.

Return value

The return value is the virtual-key code and modifiers for the hot key, or **NULL** if no hot key is associated with the window. The virtual-key code is in the low byte of the return value and the modifiers are in the high byte. The modifiers can be a combination of the following flags from CommCtrl.h.

RETURN CODE/VALUE	DESCRIPTION
HOTKEYF_ALT 0x04	ALT key
HOTKEYF_CONTROL 0x02	CTRL key
HOTKEYF_EXT 0x08	Extended key
HOTKEYF_SHIFT 0x01	SHIFT key

Remarks

These hot keys are unrelated to the hot keys set by the **RegisterHotKey** function.

Requirements

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

See also

Reference

RegisterHotKey

WM_SETHOTKEY

Conceptual

Keyboard Input

WM_SETHOTKEY message

4/13/2022 • 2 minutes to read • Edit Online

Sent to a window to associate a hot key with the window. When the user presses the hot key, the system activates the window.

```
#define WM_SETHOTKEY                    0x0032
```

Parameters

wParam

The low-order word specifies the virtual-key code to associate with the window.

The high-order word can be one or more of the following values from CommCtrl.h.

Setting *wParam* to **NULL** removes the hot key associated with a window.

VALUE	MEANING
HOTKEYF_ALT 0x04	ALT key
HOTKEYF_CONTROL 0x02	CTRL key
HOTKEYF_EXT 0x08	Extended key
HOTKEYF_SHIFT 0x01	SHIFT key

lParam

This parameter is not used.

Return value

The return value is one of the following.

RETURN VALUE	DESCRIPTION
-1	The function is unsuccessful; the hot key is invalid.

RETURN VALUE	DESCRIPTION
0	The function is unsuccessful; the window is invalid.
1	The function is successful, and no other window has the same hot key.
2	The function is successful, but another window already has the same hot key.

Remarks

A hot key cannot be associated with a child window.

VK_ESCAPE, **VK_SPACE**, and **VK_TAB** are invalid hot keys.

When the user presses the hot key, the system generates a **WM_SYSCOMMAND** message with *wParam* equal to **SC_HOTKEY** and *lParam* equal to the window's handle. If this message is passed on to **DefWindowProc**, the system will bring the window's last active popup (if it exists) or the window itself (if there is no popup window) to the foreground.

A window can only have one hot key. If the window already has a hot key associated with it, the new hot key replaces the old one. If more than one window has the same hot key, the window that is activated by the hot key is random.

These hot keys are unrelated to the hot keys set by **RegisterHotKey**.

Requirements

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

See also

Reference

RegisterHotKey

WM_GETHOTKEY

WM_SYSCOMMAND

Conceptual

Keyboard Input

Keyboard Input Notifications

4/13/2022 • 2 minutes to read • Edit Online

In This Section

WM_ACTIVATE message

4/13/2022 • 2 minutes to read • Edit Online

Sent to both the window being activated and the window being deactivated. If the windows use the same input queue, the message is sent synchronously, first to the window procedure of the top-level window being deactivated, then to the window procedure of the top-level window being activated. If the windows use different input queues, the message is sent asynchronously, so the window is activated immediately.

```
#define WM_ACTIVATE                     0x0006
```

Parameters

wParam

The low-order word specifies whether the window is being activated or deactivated. This parameter can be one of the following values. The high-order word specifies the minimized state of the window being activated or deactivated. A nonzero value indicates the window is minimized.

VALUE	MEANING
WA_ACTIVE 1	Activated by some method other than a mouse click (for example, by a call to the **SetActiveWindow** function or by use of the keyboard interface to select the window).
WA_CLICKACTIVE 2	Activated by a mouse click.
WA_INACTIVE 0	Deactivated.

lParam

A handle to the window being activated or deactivated, depending on the value of the *wParam* parameter. If the low-order word of *wParam* is **WA_INACTIVE**, *lParam* is the handle to the window being activated. If the low-order word of *wParam* is **WA_ACTIVE** or **WA_CLICKACTIVE**, *lParam* is the handle to the window being deactivated. This handle can be **NULL**.

Return value

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

Remarks

If the window is being activated and is not minimized, the **DefWindowProc** function sets the keyboard focus to the window. If the window is activated by a mouse click, it also receives a **WM_MOUSEACTIVATE** message.

Requirements

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

See also

Reference

DefWindowProc

SetActiveWindow

WM_MOUSEACTIVATE

WM_NCACTIVATE

Conceptual

Keyboard Input

WM_APPCOMMAND message

4/13/2022 • 4 minutes to read • Edit Online

Notifies a window that the user generated an application command event, for example, by clicking an application command button using the mouse or typing an application command key on the keyboard.

```
#define WM_APPCOMMAND                   0x0319
```

Parameters

wParam

A handle to the window where the user clicked the button or pressed the key. This can be a child window of the window receiving the message. For more information about processing this message, see the Remarks section.

lParam

Use the following code to get the information contained in the *lParam* parameter.

```
cmd  = GET_APPCOMMAND_LPARAM(lParam);

uDevice = GET_DEVICE_LPARAM(lParam);

dwKeys = GET_KEYSTATE_LPARAM(lParam);
```

The application command is *cmd*, which can be one of the following values.

VALUE	MEANING
APPCOMMAND_BASS_BOOST 20	Toggle the bass boost on and off.
APPCOMMAND_BASS_DOWN 19	Decrease the bass.
APPCOMMAND_BASS_UP 21	Increase the bass.
APPCOMMAND_BROWSER_BACKWARD 1	Navigate backward.
APPCOMMAND_BROWSER_FAVORITES 6	Open favorites.

VALUE	MEANING
APPCOMMAND_BROWSER_FORWARD 2	Navigate forward.
APPCOMMAND_BROWSER_HOME 7	Navigate home.
APPCOMMAND_BROWSER_REFRESH 3	Refresh page.
APPCOMMAND_BROWSER_SEARCH 5	Open search.
APPCOMMAND_BROWSER_STOP 4	Stop download.
APPCOMMAND_CLOSE 31	Close the window (not the application).
APPCOMMAND_COPY 36	Copy the selection.
APPCOMMAND_CORRECTION_LIST 45	Brings up the correction list when a word is incorrectly identified during speech input.
APPCOMMAND_CUT 37	Cut the selection.
APPCOMMAND_DICTATE_OR_COMMAND_CONTROL_TOGGLE 43	Toggles between two modes of speech input: dictation and command/control (giving commands to an application or accessing menus).
APPCOMMAND_FIND 28	Open the **Find** dialog.
APPCOMMAND_FORWARD_MAIL 40	Forward a mail message.

VALUE	MEANING
APPCOMMAND_HELP 27	Open the **Help** dialog.
APPCOMMAND_LAUNCH_APP1 17	Start App1.
APPCOMMAND_LAUNCH_APP2 18	Start App2.
APPCOMMAND_LAUNCH_MAIL 15	Open mail.
APPCOMMAND_LAUNCH_MEDIA_SELECT 16	Go to Media Select mode.
APPCOMMAND_MEDIA_CHANNEL_DOWN 52	Decrement the channel value, for example, for a TV or radio tuner.
APPCOMMAND_MEDIA_CHANNEL_UP 51	Increment the channel value, for example, for a TV or radio tuner.
APPCOMMAND_MEDIA_FAST_FORWARD 49	Increase the speed of stream playback. This can be implemented in many ways, for example, using a fixed speed or toggling through a series of increasing speeds.
APPCOMMAND_MEDIA_NEXTTRACK 11	Go to next track.
APPCOMMAND_MEDIA_PAUSE 47	Pause. If already paused, take no further action. This is a direct PAUSE command that has no state. If there are discrete Play and Pause buttons, applications should take action on this command as well as **APPCOMMAND_MEDIA_PLAY_PAUSE**.
APPCOMMAND_MEDIA_PLAY 46	Begin playing at the current position. If already paused, it will resume. This is a direct PLAY command that has no state. If there are discrete **Play** and **Pause** buttons, applications should take action on this command as well as **APPCOMMAND_MEDIA_PLAY_PAUSE**.
APPCOMMAND_MEDIA_PLAY_PAUSE 14	Play or pause playback. If there are discrete **Play** and **Pause** buttons, applications should take action on this command as well as **APPCOMMAND_MEDIA_PLAY** and **APPCOMMAND_MEDIA_PAUSE**.

VALUE	MEANING
APPCOMMAND_MEDIA_PREVIOUSTRACK 12	Go to previous track.
APPCOMMAND_MEDIA_RECORD 48	Begin recording the current stream.
APPCOMMAND_MEDIA_REWIND 50	Go backward in a stream at a higher rate of speed. This can be implemented in many ways, for example, using a fixed speed or toggling through a series of increasing speeds.
APPCOMMAND_MEDIA_STOP 13	Stop playback.
APPCOMMAND_MIC_ON_OFF_TOGGLE 44	Toggle the microphone.
APPCOMMAND_MICROPHONE_VOLUME_DOWN 25	Decrease microphone volume.
APPCOMMAND_MICROPHONE_VOLUME_MUTE 24	Mute the microphone.
APPCOMMAND_MICROPHONE_VOLUME_UP 26	Increase microphone volume.
APPCOMMAND_NEW 29	Create a new window.
APPCOMMAND_OPEN 30	Open a window.
APPCOMMAND_PASTE 38	Paste
APPCOMMAND_PRINT 33	Print current document.
APPCOMMAND_REDO 35	Redo last action.

VALUE	MEANING
APPCOMMAND_REPLY_TO_MAIL 39	Reply to a mail message.
APPCOMMAND_SAVE 32	Save current document.
APPCOMMAND_SEND_MAIL 41	Send a mail message.
APPCOMMAND_SPELL_CHECK 42	Initiate a spell check.
APPCOMMAND_TREBLE_DOWN 22	Decrease the treble.
APPCOMMAND_TREBLE_UP 23	Increase the treble.
APPCOMMAND_UNDO 34	Undo last action.
APPCOMMAND_VOLUME_DOWN 9	Lower the volume.
APPCOMMAND_VOLUME_MUTE 8	Mute the volume.
APPCOMMAND_VOLUME_UP 10	Raise the volume.

The *uDevice* component indicates the input device that generated the input event, and can be one of the following values.

VALUE	MEANING
FAPPCOMMAND_KEY 0	User pressed a key.

VALUE	MEANING
FAPPCOMMAND_MOUSE 0x8000	User clicked a mouse button.
FAPPCOMMAND_OEM 0x1000	An unidentified hardware source generated the event. It could be a mouse or a keyboard event.

The *dwKeys* component indicates whether various virtual keys are down, and can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

DefWindowProc generates the **WM_APPCOMMAND** message when it processes the WM_XBUTTONUP or WM_NCXBUTTONUP message, or when the user types an application command key.

If a child window does not process this message and instead calls DefWindowProc, **DefWindowProc** will send the message to its parent window. If a top level window does not process this message and instead calls **DefWindowProc**, **DefWindowProc** will call a shell hook with the hook code equal to **HSHELL_APPCOMMAND**.

To get the coordinates of the cursor if the message was generated by a mouse click, the application can call GetMessagePos. An application can test whether the message was generated by the mouse by checking whether *lParam* contains **FAPPCOMMAND_MOUSE**.

Unlike other windows messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called DefWindowProc to process it.

Requirements

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

See also

Reference

DefWindowProc

GET_APPCOMMAND_LPARAM

GET_DEVICE_LPARAM

GET_KEYSTATE_LPARAM

ShellProc

WM_XBUTTONUP

WM_NCXBUTTONUP

Conceptual

Mouse Input

WM_CHAR message

4/13/2022 • 2 minutes to read • Edit Online

Posted to the window with the keyboard focus when a **WM_KEYDOWN** message is translated by the **TranslateMessage** function. The **WM_CHAR** message contains the character code of the key that was pressed.

```
#define WM_CHAR                         0x0102
```

Parameters

wParam

The character code of the key.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	The transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Example

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {

    // ...

    case WM_CHAR:
        OnKeyPress(wParam);
        break;

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

Example from Windows Classic Samples on GitHub.

Remarks

The **WM_CHAR** message uses Unicode Transformation Format (UTF)-16.

There is not necessarily a one-to-one correspondence between keys pressed and character messages generated, and so the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent **WM_KEYDOWN** message that precedes the posting of the **WM_CHAR** message.

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

The **WM_UNICHAR** message is the same as **WM_CHAR**, except it uses UTF-32. It is designed to send or post Unicode characters to ANSI windows, and it can handle Unicode Supplementary Plane characters.

Requirements

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

See also

Reference

TranslateMessage

WM_KEYDOWN

WM_UNICHAR

Conceptual

Keyboard Input

WM_DEADCHAR message

4/13/2022 • 2 minutes to read • Edit Online

Posted to the window with the keyboard focus when a **WM_KEYUP** message is translated by the **TranslateMessage** function. **WM_DEADCHAR** specifies a character code generated by a dead key. A dead key is a key that generates a character, such as the umlaut (double-dot), that is combined with another character to form a composite character. For example, the umlaut-O character ( ) is generated by typing the dead key for the umlaut character, and then typing the O key.

```
#define WM_DEADCHAR                     0x0103
```

Parameters

wParam

The character code generated by the dead key.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	The transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Remarks

The **WM_DEADCHAR** message typically is used by applications to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret.

Because there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent **WM_KEYDOWN** message that precedes the posting of the **WM_DEADCHAR** message.

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

Requirements

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

See also

Reference

TranslateMessage

WM_KEYDOWN

WM_KEYUP

WM_SYSDEADCHAR

Conceptual

Keyboard Input

WM_HOTKEY message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses a hot key registered by the RegisterHotKey function. The message is placed at the top of the message queue associated with the thread that registered the hot key.

```
#define WM_HOTKEY                       0x0312
```

Parameters

wParam

The identifier of the hot key that generated the message. If the message was generated by a system-defined hot key, this parameter will be one of the following values.

VALUE	MEANING
IDHOT_SNAPDESKTOP -2	The "snap desktop" hot key was pressed.
IDHOT_SNAPWINDOW -1	The "snap window" hot key was pressed.

lParam

The low-order word specifies the keys that were to be pressed in combination with the key specified by the high-order word to generate the **WM_HOTKEY** message. This word can be one or more of the following values. The high-order word specifies the virtual key code of the hot key.

VALUE	MEANING
MOD_ALT 0x0001	Either ALT key was held down.
MOD_CONTROL 0x0002	Either CTRL key was held down.
MOD_SHIFT 0x0004	Either SHIFT key was held down.
MOD_WIN 0x0008	Either WINDOWS key was held down. These keys are labeled with the Windows logo. Hotkeys that involve the Windows key are reserved for use by the operating system.

Remarks

WM_HOTKEY is unrelated to the WM_GETHOTKEY and WM_SETHOTKEY hot keys. The **WM_HOTKEY** message is sent for generic hot keys while the **WM_SETHOTKEY** and **WM_GETHOTKEY** messages relate to window activation hot keys.

Requirements

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

See also

Reference

RegisterHotKey

WM_GETHOTKEY

WM_SETHOTKEY

Conceptual

Keyboard Input

WM_KEYDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted to the window with the keyboard focus when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is not pressed.

```
#define WM_KEYDOWN                      0x0100
```

Parameters

wParam

The virtual-key code of the nonsystem key. See Virtual-Key Codes.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown following.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is always 0 for a **WM_KEYDOWN** message.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is zero if the key is up.
31	The transition state. The value is always 0 for a **WM_KEYDOWN** message.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Example

```
LRESULT CALLBACK HostWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_KEYDOWN:
        if (wParam == VK_ESCAPE)
        {
            if (isFullScreen)
            {
                GoPartialScreen();
            }
        }
        break;

    // ...
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Example from Windows Classic Samples on GitHub.

Remarks

If the F10 key is pressed, the **DefWindowProc** function sets an internal flag. When **DefWindowProc** receives the **WM_KEYUP** message, the function checks whether the internal flag is set and, if so, sends a **WM_SYSCOMMAND** message to the top-level window. The **WM_SYSCOMMAND** parameter of the message is set to SC_KEYMENU.

Because of the autorepeat feature, more than one **WM_KEYDOWN** message may be posted before a **WM_KEYUP** message is posted. The previous key state (bit 30) can be used to determine whether the **WM_KEYDOWN** message indicates the first down transition or a repeated down transition.

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards may support the extended-key bit in the *lParam* parameter.

Applications must pass *wParam* to **TranslateMessage** without altering it at all.

Requirements

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

See also

Reference

DefWindowProc

TranslateMessage

WM_CHAR

WM_KEYUP

WM_SYSCOMMAND

Conceptual

Keyboard Input

WM_KEYUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted to the window with the keyboard focus when a nonsystem key is released. A nonsystem key is a key that is pressed when the ALT key is *not* pressed, or a keyboard key that is pressed when a window has the keyboard focus.

```
#define WM_KEYUP                        0x0101
```

Parameters

wParam

The virtual-key code of the nonsystem key. See Virtual-Key Codes.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. The repeat count is always 1 for a **WM_KEYUP** message.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is always 0 for a **WM_KEYUP** message.
30	The previous key state. The value is always 1 for a **WM_KEYUP** message.
31	The transition state. The value is always 1 for a **WM_KEYUP** message.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Remarks

The DefWindowProc function sends a WM_SYSCOMMAND message to the top-level window if the F10 key or the ALT key was released. The *wParam* parameter of the message is set to SC_KEYMENU.

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards may support the extended-key bit in the *lParam* parameter.

Applications must pass *wParam* to TranslateMessage without altering it at all.

Requirements

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

See also

Reference

DefWindowProc

TranslateMessage

WM_KEYDOWN

WM_SYSCOMMAND

Conceptual

Keyboard Input

WM_KILLFOCUS message

4/13/2022 • 2 minutes to read • Edit Online

Sent to a window immediately before it loses the keyboard focus.

```
#define WM_KILLFOCUS                    0x0008
```

Parameters

wParam

A handle to the window that receives the keyboard focus. This parameter can be **NULL**.

lParam

This parameter is not used.

Return value

An application should return zero if it processes this message.

Remarks

If an application is displaying a caret, the caret should be destroyed at this point.

While processing this message, do not make any function calls that display or activate a window. This causes the thread to yield control and can cause the application to stop responding to messages. For more information, see Message Deadlocks.

Requirements

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

See also

Reference

SetFocus

WM_SETFOCUS

Conceptual

Keyboard Input

WM_SETFOCUS message

4/13/2022 • 2 minutes to read • Edit Online

Sent to a window after it has gained the keyboard focus.

```
#define WM_SETFOCUS                     0x0007
```

Parameters

wParam

A handle to the window that has lost the keyboard focus. This parameter can be **NULL**.

lParam

This parameter is not used.

Return value

An application should return zero if it processes this message.

Remarks

To display a caret, an application should call the appropriate caret functions when it receives the **WM_SETFOCUS** message.

Requirements

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

See also

Reference

SetFocus

WM_KILLFOCUS

Conceptual

Keyboard Input

WM_SYSDEADCHAR message

4/13/2022 • 2 minutes to read • Edit Online

Sent to the window with the keyboard focus when a **WM_SYSKEYDOWN** message is translated by the **TranslateMessage** function. **WM_SYSDEADCHAR** specifies the character code of a system dead key that is, a dead key that is pressed while holding down the ALT key.

```
#define WM_SYSDEADCHAR                  0x0107
```

Parameters

wParam

The character code generated by the system dead key that is, a dead key that is pressed while holding down the ALT key.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Remarks

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards may support the extended-key bit in the *lParam* parameter.

Requirements

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

See also

Reference

TranslateMessage

WM_DEADCHAR

WM_SYSKEYDOWN

Conceptual

Keyboard Input

WM_SYSKEYDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted to the window with the keyboard focus when the user presses the F10 key (which activates the menu bar) or holds down the ALT key and then presses another key. It also occurs when no window currently has the keyboard focus; in this case, the **WM_SYSKEYDOWN** message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

```
#define WM_SYSKEYDOWN                   0x0104
```

Parameters

wParam

The virtual-key code of the key being pressed. See Virtual-Key Codes.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is down while the key is pressed; it is 0 if the **WM_SYSKEYDOWN** message is posted to the active window because no window has the keyboard focus.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	The transition state. The value is always 0 for a **WM_SYSKEYDOWN** message.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function examines the specified key and generates a **WM_SYSCOMMAND** message if the key is either TAB or ENTER.

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a character-key message. This allows accelerator keys to be used with the active window even if the active window does not have the keyboard focus.

Because of automatic repeat, more than one **WM_SYSKEYDOWN** message may occur before a **WM_SYSKEYUP** message is sent. The previous key state (bit 30) can be used to determine whether the **WM_SYSKEYDOWN** message indicates the first down transition or a repeated down transition.

For enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards may support the extended-key bit in the *lParam* parameter.

This message is also sent whenever the user presses the F10 key without the ALT key.

Requirements

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

See also

Reference

DefWindowProc

TranslateAccelerator

WM_SYSCOMMAND

WM_SYSKEYUP

Conceptual

Keyboard Input

WM_SYSKEYUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted to the window with the keyboard focus when the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the keyboard focus; in this case, the **WM_SYSKEYUP** message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

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

```
#define WM_SYSKEYUP                     0x0105
```

Parameters

wParam

The virtual-key code of the key being released. See **Virtual-Key Codes**.

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. The repeat count is always one for a **WM_SYSKEYUP** message.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is zero.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is down while the key is released; it is zero if the **WM_SYSKEYUP** message is posted to the active window because no window has the keyboard focus.
30	The previous key state. The value is always 1 for a **WM_SYSKEYUP** message.
31	The transition state. The value is always 1 for a **WM_SYSKEYUP** message.

For more details, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Remarks

The DefWindowProc function sends a WM_SYSCOMMAND message to the top-level window if the F10 key or the ALT key was released. The *wParam* parameter of the message is set to SC_KEYMENU.

When the context code is zero, the message can be passed to the TranslateAccelerator function, which will handle it as though it were a normal key message instead of a character-key message. This allows accelerator keys to be used with the active window even if the active window does not have the keyboard focus.

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards may support the extended-key bit in the *lParam* parameter.

For non-U.S. enhanced 102-key keyboards, the right ALT key is handled as a CTRL+ALT key. The following table shows the sequence of messages that result when the user presses and releases this key.

MESSAGE	VIRTUAL-KEY CODE
WM_KEYDOWN	VK_CONTROL
WM_KEYDOWN	VK_MENU
WM_KEYUP	VK_CONTROL
WM_SYSKEYUP	VK_MENU

Requirements

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

See also

Reference

DefWindowProc

TranslateAccelerator

WM_SYSCOMMAND

WM_SYSKEYDOWN

Conceptual

WM_UNICHAR message

4/13/2022 • 2 minutes to read • Edit Online

The **WM_UNICHAR** message can be used by an application to post input to other windows. This message contains the character code of the key that was pressed. (Test whether a target app can process **WM_UNICHAR** messages by sending the message with *wParam* set to **UNICODE_NOCHAR**.)

```
#define WM_UNICHAR                      0x0109
```

Parameters

wParam

The character code of the key.

If *wParam* is **UNICODE_NOCHAR** and the application processes this message, then return **TRUE**. The **DefWindowProc** function will return **FALSE** (the default).

If *wParam* is not **UNICODE_NOCHAR**, return **FALSE**. The Unicode **DefWindowProc** posts a **WM_CHAR** message with the same parameters and the ANSI **DefWindowProc** function posts either one or two **WM_CHAR** messages with the corresponding ANSI character(s).

lParam

The repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table.

BITS	MEANING
0-15	The repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative.
16-23	The scan code. The value depends on the OEM.
24	Indicates whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0.
25-28	Reserved; do not use.
29	The context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	The previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	The transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For more detail, see Keystroke Message Flags.

Return value

An application should return zero if it processes this message.

Remarks

The **WM_UNICHAR** message is similar to **WM_CHAR**, but it uses Unicode Transformation Format (UTF)-32, whereas **WM_CHAR** uses UTF-16.

This message is designed to send or post Unicode characters to ANSI windows and can handle Unicode Supplementary Plane characters.

Because there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent **WM_KEYDOWN** message that precedes the posting of the **WM_UNICHAR** message.

For enhanced 101- and 102-key keyboards, extended keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

Requirements

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

See also

Reference

TranslateMessage

WM_CHAR

WM_KEYDOWN

Conceptual

Keyboard Input

Keyboard Input Structures

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- HARDWAREINPUT
- INPUT
- KEYBDINPUT
- LASTINPUTINFO
- MOUSEINPUT

Keyboard Input Constants

4/13/2022 • 2 minutes to read • Edit Online

- Virtual-Key Codes

Virtual-Key Codes

4/13/2022 • 5 minutes to read • Edit Online

The following table shows the symbolic constant names, hexadecimal values, and mouse or keyboard equivalents for the virtual-key codes used by the system. The codes are listed in numeric order.

CONSTANT	VALUE	DESCRIPTION
`VK_LBUTTON`	0x01	Left mouse button
`VK_RBUTTON`	0x02	Right mouse button
`VK_CANCEL`	0x03	Control-break processing
`VK_MBUTTON`	0x04	Middle mouse button (three-button mouse)
`VK_XBUTTON1`	0x05	X1 mouse button
`VK_XBUTTON2`	0x06	X2 mouse button
`-`	0x07	Undefined
`VK_BACK`	0x08	BACKSPACE key
`VK_TAB`	0x09	TAB key
`-`	0x0A-0B	Reserved
`VK_CLEAR`	0x0C	CLEAR key
`VK_RETURN`	0x0D	ENTER key
`-`	0x0E-0F	Undefined
`VK_SHIFT`	0x10	SHIFT key
`VK_CONTROL`	0x11	CTRL key
`VK_MENU`	0x12	ALT key
`VK_PAUSE`	0x13	PAUSE key
`VK_CAPITAL`	0x14	CAPS LOCK key
`VK_KANA`	0x15	IME Kana mode

CONSTANT	VALUE	DESCRIPTION
`VK_HANGUEL`	0x15	IME Hanguel mode (maintained for compatibility; use `VK_HANGUL`)
`VK_HANGUL`	0x15	IME Hangul mode
`VK_IME_ON`	0x16	IME On
`VK_JUNJA`	0x17	IME Junja mode
`VK_FINAL`	0x18	IME final mode
`VK_HANJA`	0x19	IME Hanja mode
`VK_KANJI`	0x19	IME Kanji mode
`VK_IME_OFF`	0x1A	IME Off
`VK_ESCAPE`	0x1B	ESC key
`VK_CONVERT`	0x1C	IME convert
`VK_NONCONVERT`	0x1D	IME nonconvert
`VK_ACCEPT`	0x1E	IME accept
`VK_MODECHANGE`	0x1F	IME mode change request
`VK_SPACE`	0x20	SPACEBAR
`VK_PRIOR`	0x21	PAGE UP key
`VK_NEXT`	0x22	PAGE DOWN key
`VK_END`	0x23	END key
`VK_HOME`	0x24	HOME key
`VK_LEFT`	0x25	LEFT ARROW key
`VK_UP`	0x26	UP ARROW key
`VK_RIGHT`	0x27	RIGHT ARROW key
`VK_DOWN`	0x28	DOWN ARROW key
`VK_SELECT`	0x29	SELECT key
`VK_PRINT`	0x2A	PRINT key

CONSTANT	VALUE	DESCRIPTION
`VK_EXECUTE`	0x2B	EXECUTE key
`VK_SNAPSHOT`	0x2C	PRINT SCREEN key
`VK_INSERT`	0x2D	INS key
`VK_DELETE`	0x2E	DEL key
`VK_HELP`	0x2F	HELP key
	0x30	0 key
	0x31	1 key
	0x32	2 key
	0x33	3 key
	0x34	4 key
	0x35	5 key
	0x36	6 key
	0x37	7 key
	0x38	8 key
	0x39	9 key
`-`	0x3A-40	Undefined
	0x41	A key
	0x42	B key
	0x43	C key
	0x44	D key
	0x45	E key
	0x46	F key
	0x47	G key
	0x48	H key
	0x49	I key

CONSTANT	VALUE	DESCRIPTION
	0x4A	J key
	0x4B	K key
	0x4C	L key
	0x4D	M key
	0x4E	N key
	0x4F	O key
	0x50	P key
	0x51	Q key
	0x52	R key
	0x53	S key
	0x54	T key
	0x55	U key
	0x56	V key
	0x57	W key
	0x58	X key
	0x59	Y key
	0x5A	Z key
`VK_LWIN`	0x5B	Left Windows key (Natural keyboard)
`VK_RWIN`	0x5C	Right Windows key (Natural keyboard)
`VK_APPS`	0x5D	Applications key (Natural keyboard)
`-`	0x5E	Reserved
`VK_SLEEP`	0x5F	Computer Sleep key
`VK_NUMPAD0`	0x60	Numeric keypad 0 key
`VK_NUMPAD1`	0x61	Numeric keypad 1 key
`VK_NUMPAD2`	0x62	Numeric keypad 2 key

CONSTANT	VALUE	DESCRIPTION
`VK_NUMPAD3`	0x63	Numeric keypad 3 key
`VK_NUMPAD4`	0x64	Numeric keypad 4 key
`VK_NUMPAD5`	0x65	Numeric keypad 5 key
`VK_NUMPAD6`	0x66	Numeric keypad 6 key
`VK_NUMPAD7`	0x67	Numeric keypad 7 key
`VK_NUMPAD8`	0x68	Numeric keypad 8 key
`VK_NUMPAD9`	0x69	Numeric keypad 9 key
`VK_MULTIPLY`	0x6A	Multiply key
`VK_ADD`	0x6B	Add key
`VK_SEPARATOR`	0x6C	Separator key
`VK_SUBTRACT`	0x6D	Subtract key
`VK_DECIMAL`	0x6E	Decimal key
`VK_DIVIDE`	0x6F	Divide key
`VK_F1`	0x70	F1 key
`VK_F2`	0x71	F2 key
`VK_F3`	0x72	F3 key
`VK_F4`	0x73	F4 key
`VK_F5`	0x74	F5 key
`VK_F6`	0x75	F6 key
`VK_F7`	0x76	F7 key
`VK_F8`	0x77	F8 key
`VK_F9`	0x78	F9 key
`VK_F10`	0x79	F10 key
`VK_F11`	0x7A	F11 key

CONSTANT	VALUE	DESCRIPTION
`VK_F12`	0x7B	F12 key
`VK_F13`	0x7C	F13 key
`VK_F14`	0x7D	F14 key
`VK_F15`	0x7E	F15 key
`VK_F16`	0x7F	F16 key
`VK_F17`	0x80	F17 key
`VK_F18`	0x81	F18 key
`VK_F19`	0x82	F19 key
`VK_F20`	0x83	F20 key
`VK_F21`	0x84	F21 key
`VK_F22`	0x85	F22 key
`VK_F23`	0x86	F23 key
`VK_F24`	0x87	F24 key
`-`	0x88-8F	Unassigned
`VK_NUMLOCK`	0x90	NUM LOCK key
`VK_SCROLL`	0x91	SCROLL LOCK key
	0x92-96	OEM specific
`-`	0x97-9F	Unassigned
`VK_LSHIFT`	0xA0	Left SHIFT key
`VK_RSHIFT`	0xA1	Right SHIFT key
`VK_LCONTROL`	0xA2	Left CONTROL key
`VK_RCONTROL`	0xA3	Right CONTROL key
`VK_LMENU`	0xA4	Left MENU key
`VK_RMENU`	0xA5	Right MENU key

CONSTANT	VALUE	DESCRIPTION
`VK_BROWSER_BACK`	0xA6	Browser Back key
`VK_BROWSER_FORWARD`	0xA7	Browser Forward key
`VK_BROWSER_REFRESH`	0xA8	Browser Refresh key
`VK_BROWSER_STOP`	0xA9	Browser Stop key
`VK_BROWSER_SEARCH`	0xAA	Browser Search key
`VK_BROWSER_FAVORITES`	0xAB	Browser Favorites key
`VK_BROWSER_HOME`	0xAC	Browser Start and Home key
`VK_VOLUME_MUTE`	0xAD	Volume Mute key
`VK_VOLUME_DOWN`	0xAE	Volume Down key
`VK_VOLUME_UP`	0xAF	Volume Up key
`VK_MEDIA_NEXT_TRACK`	0xB0	Next Track key
`VK_MEDIA_PREV_TRACK`	0xB1	Previous Track key
`VK_MEDIA_STOP`	0xB2	Stop Media key
`VK_MEDIA_PLAY_PAUSE`	0xB3	Play/Pause Media key
`VK_LAUNCH_MAIL`	0xB4	Start Mail key
`VK_LAUNCH_MEDIA_SELECT`	0xB5	Select Media key
`VK_LAUNCH_APP1`	0xB6	Start Application 1 key
`VK_LAUNCH_APP2`	0xB7	Start Application 2 key
`-`	0xB8-B9	Reserved
`VK_OEM_1`	0xBA	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ';:' key
`VK_OEM_PLUS`	0xBB	For any country/region, the '+' key
`VK_OEM_COMMA`	0xBC	For any country/region, the ',' key
`VK_OEM_MINUS`	0xBD	For any country/region, the '-' key
`VK_OEM_PERIOD`	0xBE	For any country/region, the '.' key

CONSTANT	VALUE	DESCRIPTION
`VK_OEM_2`	0xBF	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '/?' key
`VK_OEM_3`	0xC0	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '`~' key
`-`	0xC1-D7	Reserved
`-`	0xD8-DA	Unassigned
`VK_OEM_4`	0xDB	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '[{' key
`VK_OEM_5`	0xDC	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '\\|' key
`VK_OEM_6`	0xDD	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ']}' key
`VK_OEM_7`	0xDE	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the 'single-quote/double-quote' key
`VK_OEM_8`	0xDF	Used for miscellaneous characters; it can vary by keyboard.
`-`	0xE0	Reserved
	0xE1	OEM specific
`VK_OEM_102`	0xE2	The `<>` keys on the US standard keyboard, or the `\\|` key on the non-US 102-key keyboard
	0xE3-E4	OEM specific
`VK_PROCESSKEY`	0xE5	IME PROCESS key
	0xE6	OEM specific
`VK_PACKET`	0xE7	Used to pass Unicode characters as if they were keystrokes. The `VK_PACKET` key is the low word of a 32-bit Virtual Key value used for non-keyboard input methods. For more information, see Remark in `KEYBDINPUT`, `SendInput`, `WM_KEYDOWN`, and `WM_KEYUP`

CONSTANT	VALUE	DESCRIPTION
`-`	0xE8	Unassigned
	0xE9-F5	OEM specific
`VK_ATTN`	0xF6	Attn key
`VK_CRSEL`	0xF7	CrSel key
`VK_EXSEL`	0xF8	ExSel key
`VK_EREOF`	0xF9	Erase EOF key
`VK_PLAY`	0xFA	Play key
`VK_ZOOM`	0xFB	Zoom key
`VK_NONAME`	0xFC	Reserved
`VK_PA1`	0xFD	PA1 key
`VK_OEM_CLEAR`	0xFE	Clear key

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h

Mouse Input (Keyboard and Mouse Input)

4/13/2022 • 10 minutes to read • Edit Online

This section describes how the system provides mouse input to your application and how the application receives and processes that input.

In this section

TOPIC	DESCRIPTION
About Mouse Input	This topic discusses mouse input.
Using Mouse Input	This section covers tasks associated with mouse input.
Mouse Input Reference	

Functions

NAME	DESCRIPTION
_TrackMouseEvent	Posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time. This function calls TrackMouseEvent if it exists, otherwise it emulates it.
DragDetect	Captures the mouse and tracks its movement until the user releases the left button, presses the ESC key, or moves the mouse outside the drag rectangle around the specified point. The width and height of the drag rectangle are specified by the **SM_CXDRAG** and **SM_CYDRAG** values returned by the GetSystemMetrics function.
EnableMouseInPointer	Enables the mouse to act as a pointing device.
GetCapture	Retrieves a handle to the window (if any) that has captured the mouse. Only one window at a time can capture the mouse; this window receives mouse input whether or not the cursor is within its borders.
GetDoubleClickTime	Retrieves the current double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second click of a double-click.
GetMouseMovePointsEx	Retrieves a history of up to 64 previous coordinates of the mouse or pen.

NAME	DESCRIPTION
ReleaseCapture	Releases the mouse capture from a window in the current thread and restores normal mouse input processing. A window that has captured the mouse receives all mouse input, regardless of the position of the cursor, except when a mouse button is clicked while the cursor hot spot is in the window of another thread.
SetCapture	Sets the mouse capture to the specified window belonging to the current thread. SetCapture captures mouse input either when the mouse is over the capturing window, or when the mouse button was pressed while the mouse was over the capturing window and the button is still down. Only one window at a time can capture the mouse. If the mouse cursor is over a window created by another thread, the system will direct mouse input to the specified window only if a mouse button is down.
SetDoubleClickTime	Sets the double-click time for the mouse. A double-click is a series of two clicks of a mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.
SwapMouseButton	Reverses or restores the meaning of the left and right mouse buttons.
TrackMouseEvent	Posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

The following function is obsolete.

FUNCTION	DESCRIPTION
mouse_event	Synthesizes mouse motion and button clicks.

Notifications

NAME	DESCRIPTION
WM_CAPTURECHANGED	Sent to the window that is losing the mouse capture.
WM_LBUTTONDBLCLK	Posted when the user double-clicks the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_LBUTTONDOWN	Posted when the user presses the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

NAME	DESCRIPTION
WM_LBUTTONUP	Posted when the user releases the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_MBUTTONDBLCLK	Posted when the user double-clicks the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_MBUTTONDOWN	Posted when the user presses the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_MBUTTONUP	Posted when the user releases the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_MOUSEACTIVATE	Sent when the cursor is in an inactive window and the user presses a mouse button. The parent window receives this message only if the child window passes it to the DefWindowProc function.
WM_MOUSEHOVER	Posted to a window when the cursor hovers over the client area of the window for the period of time specified in a prior call to TrackMouseEvent.
WM_MOUSEHWHEEL	Sent to the focus window when the mouse's horizontal scroll wheel is tilted or rotated. TheDefWindowProcfunction propagates the message to the window's parent. There should be no internal forwarding of the message, because **DefWindowProc** propagates it up the parent chain until it finds a window that processes it.
WM_MOUSELEAVE	Posted to a window when the cursor leaves the client area of the window specified in a prior call to TrackMouseEvent.
WM_MOUSEMOVE	Posted to a window when the cursor moves. If the mouse is not captured, the message is posted to the window that contains the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_MOUSEWHEEL	Sent to the focus window when the mouse wheel is rotated. The DefWindowProc function propagates the message to the window's parent. There should be no internal forwarding of the message, because **DefWindowProc** propagates it up the parent chain until it finds a window that processes it.

NAME	DESCRIPTION
WM_NCHITTEST	Sent to a window in order to determine what part of the window corresponds to a particular screen coordinate. This can happen, for example, when the cursor moves, when a mouse button is pressed or released, or in response to a call to a function such as **WindowFromPoint**. If the mouse is not captured, the message is sent to the window beneath the cursor. Otherwise, the message is sent to the window that has captured the mouse.
WM_NCLBUTTONDBLCLK	Posted when the user double-clicks the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCLBUTTONDOWN	Posted when the user presses the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCLBUTTONUP	Posted when the user releases the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCMBUTTONDBLCLK	Posted when the user double-clicks the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCMBUTTONDOWN	Posted when the user presses the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCMBUTTONUP	Posted when the user releases the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCMOUSEHOVER	Posted to a window when the cursor hovers over the nonclient area of the window for the period of time specified in a prior call toTrackMouseEvent.
WM_NCMOUSELEAVE	Posted to a window when the cursor leaves the nonclient area of the window specified in a prior call toTrackMouseEvent.
WM_NCMOUSEMOVE	Posted to a window when the cursor is moved within the nonclient area of the window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

NAME	DESCRIPTION
WM_NCRBUTTONDBLCLK	Posted when the user double-clicks the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCRBUTTONDOWN	Posted when the user presses the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCRBUTTONUP	Posted when the user releases the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCXBUTTONDBLCLK	Posted when the user double-clicks the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCXBUTTONDOWN	Posted when the user presses the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_NCXBUTTONUP	Posted when the user releases the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.
WM_RBUTTONDBLCLK	Posted when the user double-clicks the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_RBUTTONDOWN	Posted when the user presses the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_RBUTTONUP	Posted when the user releases the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

NAME	DESCRIPTION
WM_XBUTTONDBLCLK	Posted when the user double-clicks the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_XBUTTONDOWN	Posted when the user presses the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.
WM_XBUTTONUP	Posted when the user releases the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

Structures

NAME	DESCRIPTION
MOUSEMOVEPOINT	Contains information about the mouse's location in screen coordinates.
TRACKMOUSEEVENT	Used by the TrackMouseEvent function to track when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

About Mouse Input

4/13/2022 • 18 minutes to read • Edit Online

The mouse is an important, but optional, user-input device for applications. A well-written application should include a mouse interface, but it should not depend solely on the mouse for acquiring user input. The application should provide full keyboard support as well.

An application receives mouse input in the form of messages that are sent or posted to its windows.

This section covers the following topics:

Mouse Cursor

When the user moves the mouse, the system moves a bitmap on the screen called the *mouse cursor*. The mouse cursor contains a single-pixel point called the *hot spot*, a point that the system tracks and recognizes as the position of the cursor. When a mouse event occurs, the window that contains the hot spot typically receives the mouse message resulting from the event. The window need not be active or have the keyboard focus to receive a mouse message.

The system maintains a variable that controls mouse speed—that is, the distance the cursor moves when the user moves the mouse. You can use the **SystemParametersInfo** function with the **SPI_GETMOUSE** or **SPI_SETMOUSE** flag to retrieve or set mouse speed. For more information about mouse cursors, see Cursors.

Mouse Capture

The system typically posts a mouse message to the window that contains the cursor hot spot when a mouse event occurs. An application can change this behavior by using the **SetCapture** function to route mouse messages to a specific window. The window receives all mouse messages until the application calls the **ReleaseCapture** function or specifies another capture window, or until the user clicks a window created by another thread.

When the mouse capture changes, the system sends a **WM_CAPTURECHANGED** message to the window that is losing the mouse capture. The *lParam* parameter of the message specifies a handle to the window that is gaining the mouse capture.

Only the foreground window can capture mouse input. When a background window attempts to capture mouse input, it receives messages only for mouse events that occur when the cursor hot spot is within the visible portion of the window.

Capturing mouse input is useful if a window must receive all mouse input, even when the cursor moves outside the window. For example, an application typically tracks the cursor position after a mouse button down event, following the cursor until a mouse button up event occurs. If an application has not captured mouse input and the user releases the mouse button outside the window, the window does not receive the button-up message.

A thread can use the **GetCapture** function to determine whether one of its windows has captured the mouse. If one of the thread's windows has captured the mouse, **GetCapture** retrieves a handle to the window.

Mouse ClickLock

The Mouse ClickLock accessibility feature enables a user lock down the primary mouse button after a single click. To an application, the button still appears to be pressed down. To unlock the button, an application can send any mouse message or the user can click any mouse button. This feature lets a user do complex mouse combinations more simply. For example, those with certain physical limitations can highlight text, drag objects, or open menus more easily. For more information, see the following flags and the Remarks in **SystemParametersInfo**:

- **SPI_GETMOUSECLICKLOCK**
- **SPI_SETMOUSECLICKLOCK**
- **SPI_GETMOUSECLICKLOCKTIME**
- **SPI_SETMOUSECLICKLOCKTIME**

Mouse Configuration

Although the mouse is an important input device for applications, not every user necessarily has a mouse. An application can determine whether the system includes a mouse by passing the **SM_MOUSEPRESENT** value to the **GetSystemMetrics** function.

Windows supports a mouse having up to three buttons. On a three-button mouse, the buttons are designated as the left, middle, and right buttons. Messages and named constants related to the mouse buttons use the letters L, M, and R to identify the buttons. The button on a single-button mouse is considered to be the left button. Although Windows supports a mouse with multiple buttons, most applications use the left button primarily and the others minimally, if at all.

Applications can also support a mouse wheel. The mouse wheel can be pressed or rotated. When the mouse wheel is pressed, it acts as the middle (third) button, sending normal middle button messages to your application. When it is rotated, a wheel message is sent to your application. For more information, see The Mouse Wheel section.

Applications can support application-command buttons. These buttons, called X buttons, are designed to allow easier access to an Internet browser, electronic mail, and media services. When an X button is pressed, a **WM_APPCOMMAND** message is sent to your application. For more information, see the description in the **WM_APPCOMMAND** message.

An application can determine the number of buttons on the mouse by passing the **SM_CMOUSEBUTTONS** value to the **GetSystemMetrics** function. To configure the mouse for a left-handed user, the application can use the **SwapMouseButton** function to reverse the meaning of the left and right mouse buttons. Passing the **SPI_SETMOUSEBUTTONSWAP** value to the **SystemParametersInfo** function is another way to reverse the meaning of the buttons. Note, however, that the mouse is a shared resource, so reversing the meaning of the buttons affects all applications.

XBUTTONs

Windows supports a mouse with five buttons. In addition to the left, middle, and right buttons there are XBUTTON1 and XBUTTON2, which provide backward and forward navigation when using your browser.

The window manager supports XBUTTON1 and XBUTTON2 through the **WM_XBUTTON*** and **WM_NCXBUTTON*** messages. The HIWORD of the **WPARAM** in these messages contains a flag indicating which X button was pressed. Because these mouse messages also fit between the constants **WM_MOUSEFIRST** and **WM_MOUSELAST**, an application can filter all mouse messages with **GetMessage** or **PeekMessage**.

The following support XBUTTON1 and XBUTTON2:

- WM_APPCOMMAND
- WM_NCXBUTTONDBLCLK
- WM_NCXBUTTONDOWN
- WM_NCXBUTTONUP
- WM_XBUTTONDBLCLK
- WM_XBUTTONDOWN
- WM_XBUTTONUP
- MOUSEHOOKSTRUCTEX

The following APIs were modified to support these buttons:

- mouse_event
- ShellProc
- MSLLHOOKSTRUCT
- MOUSEINPUT
- WM_PARENTNOTIFY

It is unlikely that a child window in a component application will be able to directly implement commands for the XBUTTON1 and XBUTTON2. So **DefWindowProc** sends a **WM_APPCOMMAND** message to a window when an X button is clicked. **DefWindowProc** also sends the **WM_APPCOMMAND** message to its parent window. This is similar to the way context menus are invoked with a right click—**DefWindowProc** sends a **WM_CONTEXTMENU** message to the menu and also sends it to its parent. Additionally, if **DefWindowProc** receives a **WM_APPCOMMAND** message for a top-level window, it calls a shell hook with code HSHELL_APPCOMMAND.

There is support for the keyboards that have extra keys for browser functions, media functions, application launching, and power management. For more information, see Keyboard Keys for Browsing and Other Functions.

Mouse Messages

The mouse generates an input event when the user moves the mouse, or presses or releases a mouse button. The system converts mouse input events into messages and posts them to the appropriate thread's message queue. When mouse messages are posted faster than a thread can process them, the system discards all but the most recent mouse message.

A window receives a mouse message when a mouse event occurs while the cursor is within the borders of the window, or when the window has captured the mouse. Mouse messages are divided into two groups: client area messages and nonclient area messages. Typically, an application processes client area messages and ignores nonclient area messages.

This section covers the following topics:

- Client Area Mouse Messages
- Nonclient Area Mouse Messages

- The WM_NCHITTEST Message

Client Area Mouse Messages

A window receives a client area mouse message when a mouse event occurs within the window's client area. The system posts the WM_MOUSEMOVE message to the window when the user moves the cursor within the client area. It posts one of the following messages when the user presses or releases a mouse button while the cursor is within the client area.

MESSAGE	MEANING
WM_LBUTTONDBLCLK	The left mouse button was double-clicked.
WM_LBUTTONDOWN	The left mouse button was pressed.
WM_LBUTTONUP	The left mouse button was released.
WM_MBUTTONDBLCLK	The middle mouse button was double-clicked.
WM_MBUTTONDOWN	The middle mouse button was pressed.
WM_MBUTTONUP	The middle mouse button was released.
WM_RBUTTONDBLCLK	The right mouse button was double-clicked.
WM_RBUTTONDOWN	The right mouse button was pressed.
WM_RBUTTONUP	The right mouse button was released.
WM_XBUTTONDBLCLK	An X mouse button was double-clicked.
WM_XBUTTONDOWN	An X mouse button was pressed.
WM_XBUTTONUP	An X mouse button was released.

In addition, an application can call the TrackMouseEvent function to have the system send two other messages. It posts the WM_MOUSEHOVER message when the cursor hovers over the client area for a certain time period. It posts the WM_MOUSELEAVE message when the cursor leaves the client area.

Message Parameters

The *lParam* parameter of a client area mouse message indicates the position of the cursor hot spot. The low-order word indicates the x-coordinate of the hot spot, and the high-order word indicates the y-coordinate. The coordinates are specified in client coordinates. In the client coordinate system, all points on the screen are specified relative to the coordinates (0,0) of the upper-left corner of the client area.

The *wParam* parameter contains flags that indicate the status of the other mouse buttons and the CTRL and SHIFT keys at the time of the mouse event. You can check for these flags when mouse-message processing depends on the state of another mouse button or of the CTRL or SHIFT key. The *wParam* parameter can be a combination of the following values.

VALUE	DESCRIPTION
MK_CONTROL	The CTRL key is down.

VALUE	DESCRIPTION
MK_LBUTTON	The left mouse button is down.
MK_MBUTTON	The middle mouse button is down.
MK_RBUTTON	The right mouse button is down.
MK_SHIFT	The SHIFT key is down.
MK_XBUTTON1	The first X button is down.
MK_XBUTTON2	The second X button is down.

Double-Click Messages

The system generates a double-click message when the user clicks a mouse button twice in quick succession. When the user clicks a button, the system establishes a rectangle centered around the cursor hot spot. It also marks the time at which the click occurred. When the user clicks the same button a second time, the system determines whether the hot spot is still within the rectangle and calculates the time elapsed since the first click. If the hot spot is still within the rectangle and the elapsed time does not exceed the double-click time-out value, the system generates a double-click message.

An application can get and set double-click time-out values by using the **GetDoubleClickTime** and **SetDoubleClickTime** functions, respectively. Alternatively, the application can set the double-click–time-out value by using the **SPI_SETDOUBLECLICKTIME** flag with the **SystemParametersInfo** function. It can also set the size of the rectangle that the system uses to detect double-clicks by passing the **SPI_SETDOUBLECLKWIDTH** and **SPI_SETDOUBLECLKHEIGHT** flags to **SystemParametersInfo**. Note, however, that setting the double-click–time-out value and rectangle affects all applications.

An application-defined window does not, by default, receive double-click messages. Because of the system overhead involved in generating double-click messages, these messages are generated only for windows belonging to classes that have the **CS_DBLCLKS** class style. Your application must set this style when registering the window class. For more information, see Window Classes.

A double-click message is always the third message in a four-message series. The first two messages are the button-down and button-up messages generated by the first click. The second click generates the double-click message followed by another button-up message. For example, double-clicking the left mouse button generates the following message sequence:

1. **WM_LBUTTONDOWN**
2. **WM_LBUTTONUP**
3. **WM_LBUTTONDBLCLK**
4. **WM_LBUTTONUP**

Because a window always receives a button-down message before receiving a double-click message, an application typically uses a double-click message to extend a task it began during a button-down message. For example, when the user clicks a color in the color palette of Microsoft Paint, Paint displays the selected color next to the palette. When the user double-clicks a color, Paint displays the color and opens the **Edit Colors** dialog box.

Nonclient Area Mouse Messages

A window receives a nonclient area mouse message when a mouse event occurs in any part of a window except the client area. A window's nonclient area consists of its border, menu bar, title bar, scroll bar, window menu, minimize button, and maximize button.

The system generates nonclient area messages primarily for its own use. For example, the system uses nonclient area messages to change the cursor to a two-headed arrow when the cursor hot spot moves into a window's border. A window must pass nonclient area mouse messages to the DefWindowProc function to take advantage of the built-in mouse interface.

There is a corresponding nonclient area mouse message for each client area mouse message. The names of these messages are similar except that the named constants for the nonclient area messages include the letters NC. For example, moving the cursor in the nonclient area generates a WM_NCMOUSEMOVE message, and pressing the left mouse button while the cursor is in the nonclient area generates a WM_NCLBUTTONDOWN message.

The *lParam* parameter of a nonclient area mouse message is a structure that contains the x- and y-coordinates of the cursor hot spot. Unlike coordinates of client area mouse messages, the coordinates are specified in screen coordinates rather than client coordinates. In the screen coordinate system, all points on the screen are relative to the coordinates (0,0) of the upper-left corner of the screen.

The *wParam* parameter contains a hit-test value, a value that indicates where in the nonclient area the mouse event occurred. The following section explains the purpose of hit-test values.

The WM_NCHITTEST Message

Whenever a mouse event occurs, the system sends a WM_NCHITTEST message to either the window that contains the cursor hot spot or the window that has captured the mouse. The system uses this message to determine whether to send a client area or nonclient area mouse message. An application that must receive mouse movement and mouse button messages must pass the **WM_NCHITTEST** message to the DefWindowProc function.

The *lParam* parameter of the WM_NCHITTEST message contains the screen coordinates of the cursor hot spot. The DefWindowProc function examines the coordinates and returns a hit-test value that indicates the location of the hot spot. The hit-test value can be one of the following values.

VALUE	LOCATION OF HOT SPOT
HTBORDER	In the border of a window that does not have a sizing border.
HTBOTTOM	In the lower-horizontal border of a window.
HTBOTTOMLEFT	In the lower-left corner of a window border.
HTBOTTOMRIGHT	In the lower-right corner of a window border.
HTCAPTION	In a title bar.
HTCLIENT	In a client area.
HTCLOSE	In a **Close** button.
HTERROR	On the screen background or on a dividing line between windows (same as HTNOWHERE, except that the DefWindowProc function produces a system beep to indicate an error).

VALUE	LOCATION OF HOT SPOT
HTGROWBOX	In a size box (same as **HTSIZE**).
HTHELP	In a **Help** button.
HTHSCROLL	In a horizontal scroll bar.
HTLEFT	In the left border of a window.
HTMENU	In a menu.
HTMAXBUTTON	In a **Maximize** button.
HTMINBUTTON	In a **Minimize** button.
HTNOWHERE	On the screen background or on a dividing line between windows.
HTREDUCE	In a **Minimize** button.
HTRIGHT	In the right border of a window.
HTSIZE	In a size box (same as **HTGROWBOX**).
HTSYSMENU	In a **System** menu or in a **Close** button in a child window.
HTTOP	In the upper-horizontal border of a window.
HTTOPLEFT	In the upper-left corner of a window border.
HTTOPRIGHT	In the upper-right corner of a window border.
HTTRANSPARENT	In a window currently covered by another window in the same thread.
HTVSCROLL	In the vertical scroll bar.
HTZOOM	In a **Maximize** button.

If the cursor is in the client area of a window, **DefWindowProc** returns the **HTCLIENT** hit-test value to the window procedure. When the window procedure returns this code to the system, the system converts the screen coordinates of the cursor hot spot to client coordinates, and then posts the appropriate client area mouse message.

The **DefWindowProc** function returns one of the other hit-test values when the cursor hot spot is in a window's nonclient area. When the window procedure returns one of these hit-test values, the system posts a nonclient area mouse message, placing the hit-test value in the message's *wParam* parameter and the cursor coordinates in the *lParam* parameter.

Mouse Sonar

The Mouse Sonar accessibility feature briefly shows several concentric circles around the pointer when the user presses and releases the CTRL key. This feature helps a user locate the mouse pointer on a screen that is cluttered or with resolution set to high, on a poor quality monitor, or for users with impaired vision. For more information, see the following flags in SystemParametersInfo:

SPI_GETMOUSESONAR

SPI_SETMOUSESONAR

Mouse Vanish

The Mouse Vanish accessibility feature hides the pointer when the user is typing. The mouse pointer reappears when the user moves the mouse. This feature keeps the pointer from obscuring the text being typed, for example, in an e-mail or other document. For more information, see the following flags in SystemParametersInfo:

SPI_GETMOUSEVANISH

SPI_SETMOUSEVANISH

The Mouse Wheel

The mouse wheel combines the features of a wheel and a mouse button. The wheel has discrete, evenly-spaced notches. When you rotate the wheel, a wheel message is sent to your application as each notch is encountered. The wheel button can also operate as a normal Windows middle (third) button. Pressing and releasing the mouse wheel sends standard WM_MBUTTONUP and WM_MBUTTONDOWN messages. Double clicking the third button sends the standard WM_MBUTTONDBLCLK message.

The mouse wheel is supported through the WM_MOUSEWHEEL message.

Rotating the mouse sends the WM_MOUSEWHEEL message to the focus window. The DefWindowProc function propagates the message to the window's parent. There should be no internal forwarding of the message, since **DefWindowProc** propagates it up the parent chain until a window that processes it is found.

Determining the Number of Scroll Lines

Applications should use the SystemParametersInfo function to retrieve the number of lines a document scrolls for each scroll operation (wheel notch). To retrieve the number of lines, an application makes the following call:

```
SystemParametersInfo(SPI_GETWHEELSCROLLLINES, 0, pulScrollLines, 0)
```

The variable "pulScrollLines" points to an unsigned integer value that receives the suggested number of lines to scroll when the mouse wheel is rotated without modifier keys:

- If this number is 0, no scrolling should occur.
- If this number is **WHEEL_PAGESCROLL**, a wheel roll should be interpreted as clicking once in the page down or page up regions of the scroll bar.
- If the number of lines to scroll is greater than the number of lines viewable, the scroll operation should also be interpreted as a page down or page up operation.

The default value for the number of scroll lines will be 3. If a user changes the number of scroll lines, by using the Mouse Properties sheet in Control Panel, the operating system broadcasts a WM_SETTINGCHANGE message to all top-level windows with **SPI_SETWHEELSCROLLLINES** specified. When an application receives the **WM_SETTINGCHANGE** message, it can then get the new number of scroll lines by calling:

```
SystemParametersInfo(SPI_GETWHEELSCROLLLINES, 0, pulScrollLines, 0)
```

Controls that Scroll

The table below lists the controls with scrolling functionality (including scroll lines set by the user).

CONTROL	SCROLLING
Edit Control	Vertical and horizontal.
List box Control	Vertical and horizontal.
Combo box	When not dropped down, each scroll retrieves the next or previous item. When dropped down, each scroll forwards the message to the list box, which scrolls accordingly.
CMD (Command line)	Vertical.
Tree View	Vertical and horizontal.
List View	Vertical and horizontal.
Up/down Scrolls	One item at a time.
Trackbar Scrolls	One item at a time.
Microsoft Rich Edit 1.0	Vertical. Note, the Exchange client has its own versions of the list view and tree view controls that do not have wheel support.
Microsoft Rich Edit 2.0	Vertical.

Detecting a Mouse with a Wheel

To determine if a mouse with a wheel is connected, call GetSystemMetrics with **SM_MOUSEWHEELPRESENT**. A return value of **TRUE** indicates that the mouse is connected.

The following example is from the window procedure for a multiline edit control:

```
BOOL ScrollLines(
     PWNDDATA pwndData,   //scrolls the window indicated
     int cLinesToScroll); //number of times

short gcWheelDelta; //wheel delta from roll
PWNDDATA pWndData; //pointer to structure containing info about the window
UINT gucWheelScrollLines=0;//number of lines to scroll on a wheel rotation

gucWheelScrollLines = SystemParametersInfo(SPI_GETWHEELSCROLLLINES,
                             0,
                             pulScrollLines,
                             0);

case WM_MOUSEWHEEL:
    /*
     * Do not handle zoom and datazoom.
     */
    if (wParam & (MK_SHIFT | MK_CONTROL)) {
        goto PassToDefaultWindowProc;
    }

    gcWheelDelta -= (short) HIWORD(wParam);
    if (abs(gcWheelDelta) >= WHEEL_DELTA && gucWheelScrollLines > 0)
    {
        int cLineScroll;

        /*
         * Limit a roll of one (1) WHEEL_DELTA to
         * scroll one (1) page.
         */
        cLineScroll = (int) min(
                (UINT) pWndData->ichLinesOnScreen - 1,
                gucWheelScrollLines);

        if (cLineScroll == 0) {
            cLineScroll++;
        }

        cLineScroll *= (gcWheelDelta / WHEEL_DELTA);
        assert(cLineScroll != 0);

        gcWheelDelta = gcWheelDelta % WHEEL_DELTA;
        return ScrollLines(pWndData, cLineScroll);
    }

    break;
```

Window Activation

When the user clicks an inactive top-level window or the child window of an inactive top-level window, the system sends the WM_MOUSEACTIVATE message (among others) to the top-level or child window. The system sends this message after posting the WM_NCHITTEST message to the window, but before posting the button-down message. When **WM_MOUSEACTIVATE** is passed to the DefWindowProc function, the system activates the top-level window and then posts the button-down message to the top-level or child window.

By processing WM_MOUSEACTIVATE, a window can control whether the top-level window becomes the active window as a result of a mouse click, and whether the window that was clicked receives the subsequent button-down message. It does so by returning one of the following values after processing **WM_MOUSEACTIVATE**.

VALUE	MEANING

VALUE	MEANING
MA_ACTIVATE	Activates the window and does not discard the mouse message.
MA_NOACTIVATE	Does not activate the window and does not discard the mouse message.
MA_ACTIVATEANDEAT	Activates the window and discards the mouse message.
MA_NOACTIVATEANDEAT	Does not activate the window but discards the mouse message.

See also

Taking Advantage of High-Definition Mouse Movement

Using Mouse Input

4/13/2022 • 12 minutes to read • Edit Online

This section covers tasks associated with mouse input.

- Tracking the Mouse Cursor
- Drawing Lines with the Mouse
- Processing a Double Click Message
- Selecting a Line of Text
- Using a Mouse Wheel in a Document with Embedded Objects
- Retrieving the Number of Mouse Wheel Scroll Lines

Tracking the Mouse Cursor

Applications often perform tasks that involve tracking the position of the mouse cursor. Most drawing applications, for example, track the position of the mouse cursor during drawing operations, allowing the user to draw in a window's client area by dragging the mouse. Word-processing applications also track the cursor, enabling the user to select a word or block of text by clicking and dragging the mouse.

Tracking the cursor typically involves processing the **WM_LBUTTONDOWN**, **WM_MOUSEMOVE**, and **WM_LBUTTONUP** messages. A window determines when to begin tracking the cursor by checking the cursor position provided in the *lParam* parameter of the **WM_LBUTTONDOWN** message. For example, a word-processing application would begin tracking the cursor only if the **WM_LBUTTONDOWN** message occurred while the cursor was on a line of text, but not if it was past the end of the document.

A window tracks the position of the cursor by processing the stream of **WM_MOUSEMOVE** messages posted to the window as the mouse moves. Processing the **WM_MOUSEMOVE** message typically involves a repetitive painting or drawing operation in the client area. For example, a drawing application might redraw a line repeatedly as the mouse moves. A window uses the **WM_LBUTTONUP** message as a signal to stop tracking the cursor.

In addition, an application can call the **TrackMouseEvent** function to have the system send other messages that are useful for tracking the cursor. The system posts the **WM_MOUSEHOVER** message when the cursor hovers over the client area for a certain time period. It posts the **WM_MOUSELEAVE** message when the cursor leaves the client area. The **WM_NCMOUSEHOVER** and **WM_NCMOUSELEAVE** messages are the corresponding messages for the nonclient areas.

Drawing Lines with the Mouse

The example in this section demonstrates how to track the mouse cursor. It contains portions of a window procedure that enables the user to draw lines in a window's client area by dragging the mouse.

When the window procedure receives a **WM_LBUTTONDOWN** message, it captures the mouse and saves the coordinates of the cursor, using the coordinates as the starting point of the line. It also uses the **ClipCursor** function to confine the cursor to the client area during the line drawing operation.

During the first **WM_MOUSEMOVE** message, the window procedure draws a line from the starting point to the current position of the cursor. During subsequent **WM_MOUSEMOVE** messages, the window procedure erases the previous line by drawing over it with an inverted pen color. Then it draws a new line from the starting point to the new position of the cursor.

The WM_LBUTTONUP message signals the end of the drawing operation. The window procedure releases the mouse capture and frees the mouse from the client area.

```
LRESULT APIENTRY MainWndProc(HWND hwndMain, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;                       // handle to device context
    RECT rcClient;                 // client area rectangle
    POINT ptClientUL;              // client upper left corner
    POINT ptClientLR;              // client lower right corner
    static POINTS ptsBegin;        // beginning point
    static POINTS ptsEnd;          // new endpoint
    static POINTS ptsPrevEnd;      // previous endpoint
    static BOOL fPrevLine = FALSE; // previous line flag

    switch (uMsg)
    {
       case WM_LBUTTONDOWN:

            // Capture mouse input.

            SetCapture(hwndMain);

            // Retrieve the screen coordinates of the client area,
            // and convert them into client coordinates.

            GetClientRect(hwndMain, &rcClient);
            ptClientUL.x = rcClient.left;
            ptClientUL.y = rcClient.top;

            // Add one to the right and bottom sides, because the
            // coordinates retrieved by GetClientRect do not
            // include the far left and lowermost pixels.

            ptClientLR.x = rcClient.right + 1;
            ptClientLR.y = rcClient.bottom + 1;
            ClientToScreen(hwndMain, &ptClientUL);
            ClientToScreen(hwndMain, &ptClientLR);

            // Copy the client coordinates of the client area
            // to the rcClient structure. Confine the mouse cursor
            // to the client area by passing the rcClient structure
            // to the ClipCursor function.

            SetRect(&rcClient, ptClientUL.x, ptClientUL.y,
                ptClientLR.x, ptClientLR.y);
            ClipCursor(&rcClient);

            // Convert the cursor coordinates into a POINTS
            // structure, which defines the beginning point of the
            // line drawn during a WM_MOUSEMOVE message.

            ptsBegin = MAKEPOINTS(lParam);
            return 0;

        case WM_MOUSEMOVE:

            // When moving the mouse, the user must hold down
            // the left mouse button to draw lines.

            if (wParam & MK_LBUTTON)
            {

                // Retrieve a device context (DC) for the client area.

                hdc = GetDC(hwndMain);

                // The following function ensures that pixels of
                // the previously drawn line are set to white and
                // the previously drawn line are set to white and
                // those of the new line are set to black.

                SetROP2(hdc, R2_NOTXORPEN);

                // If a line was drawn during an earlier WM_MOUSEMOVE
                // message, draw over it. This erases the line by
                // setting the color of its pixels to white.

                if (fPrevLine)
                {
                    MoveToEx(hdc, ptsBegin.x, ptsBegin.y,
                        (LPPOINT) NULL);
                    LineTo(hdc, ptsPrevEnd.x, ptsPrevEnd.y);
                }

                // Convert the current cursor coordinates to a
                // POINTS structure, and then draw a new line.

                ptsEnd = MAKEPOINTS(lParam);
                MoveToEx(hdc, ptsBegin.x, ptsBegin.y, (LPPOINT) NULL);
                LineTo(hdc, ptsEnd.x, ptsEnd.y);

                // Set the previous line flag, save the ending
                // point of the new line, and then release the DC.

                fPrevLine = TRUE;
                ptsPrevEnd = ptsEnd;
                ReleaseDC(hwndMain, hdc);
            }
            break;

        case WM_LBUTTONUP:

            // The user has finished drawing the line. Reset the
            // previous line flag, release the mouse cursor, and
            // release the mouse capture.

            fPrevLine = FALSE;
            ClipCursor(NULL);
            ReleaseCapture();
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        // Process other messages.
```

Processing a Double Click Message

To receive double-click messages, a window must belong to a window class that has the CS_DBLCLKS class style. You set this style when registering the window class, as shown in the following example.

```
BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASS wc;

    wc.style = CS_DBLCLKS | 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_IBEAM);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = "MainMenu";
    wc.lpszClassName = "MainWClass";

    return RegisterClass(&wc);
}
```

A double-click message is always preceded by a button-down message. For this reason, applications typically use a double-click message to extend a task that it began during a button-down message.

Selecting a Line of Text

The example in this section is taken from a simple word-processing application. It includes code that enables the user to set the position of the caret by clicking anywhere on a line of text, and to select (highlight) a line of text by double-clicking anywhere on the line.

```
LRESULT APIENTRY MainWndProc(HWND hwndMain, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;                     // handle to device context
    TEXTMETRIC tm;               // font size data
    int i, j;                    // loop counters
    int cCR = 0;                 // count of carriage returns
    char ch;                     // character from input buffer
    static int nBegLine;         // beginning of selected line
    static int nCurrentLine = 0; // currently selected line
    static int nLastLine = 0;    // last text line
    static int nCaretPosX = 0;   // x-coordinate of caret
    static int cch = 0;          // number of characters entered
    static int nCharWidth = 0;   // exact width of a character
    static char szHilite[128];   // text string to highlight
    static DWORD dwCharX;        // average width of characters
    static DWORD dwLineHeight;   // line height
    static POINTS ptsCursor;     // coordinates of mouse cursor
    static COLORREF crPrevText;  // previous text color
    static COLORREF crPrevBk;    // previous background color
    static PTCHAR pchInputBuf;   // pointer to input buffer
    static BOOL fTextSelected = FALSE; // text-selection flag
    size_t * pcch;
    HRESULT hResult;

    switch (uMsg)
    {
        case WM_CREATE:

            // Get the metrics of the current font.

            hdc = GetDC(hwndMain);
            GetTextMetrics(hdc, &tm);
            ReleaseDC(hwndMain, hdc);

            // Save the average character width and height.

            dwCharX = tm.tmAveCharWidth;
            dwLineHeight = tm.tmHeight;

            // Allocate a buffer to store keyboard input.

            pchInputBuf = (LPSTR) GlobalAlloc(GPTR,
                BUFSIZE * sizeof(TCHAR));

            return 0;

        case WM_CHAR:
            switch (wParam)
            {
                case 0x08:  // backspace
                case 0x0A:  // linefeed
                case 0x1B:  // escape
                    MessageBeep( (UINT) -1);
                    return 0;

                case 0x09:  // tab

                    // Convert tabs to four consecutive spaces.

                    for (i = 0; i < 4; i++)
                        SendMessage(hwndMain, WM_CHAR, 0x20, 0);
                    return 0;

                case 0x0D:  // carriage return

                    // Record the carriage return, and position the
                    // caret at the beginning of the new line.

                    pchInputBuf[cch++] = 0x0D;
                    nCaretPosX = 0;
                    nCurrentLine += 1;
                    break;

                default:    / displayable character

                    ch = (char) wParam;
                    HideCaret(hwndMain);

                    // Retrieve the character's width, and display the
                    // character.

                    hdc = GetDC(hwndMain);
                    GetCharWidth32(hdc, (UINT) wParam, (UINT) wParam,
                        &nCharWidth);
                    TextOut(hdc, nCaretPosX,
                        nCurrentLine * dwLineHeight, &ch, 1);
                    ReleaseDC(hwndMain, hdc);

                    // Store the character in the buffer.

                    pchInputBuf[cch++] = ch;

                    // Calculate the new horizontal position of the
                    // caret. If the new position exceeds the maximum,
                    // insert a carriage return and reposition the
                    // caret at the beginning of the next line.

                    nCaretPosX += nCharWidth;
                    if ((DWORD) nCaretPosX > dwMaxCharX)
                    {
                        nCaretPosX = 0;
                        pchInputBuf[cch++] = 0x0D;
                        ++nCurrentLine;
                    }

                    ShowCaret(hwndMain);
                break;
            }
            SetCaretPos(nCaretPosX, nCurrentLine * dwLineHeight);
            nLastLine = max(nLastLine, nCurrentLine);
            break;

        // Process other messages.

        case WM_LBUTTONDOWN:

            // If a line of text is currently highlighted, redraw
            // the text to remove the highlighting.

            if (fTextSelected)
            {
                hdc = GetDC(hwndMain);
                SetTextColor(hdc, crPrevText);
                SetBkColor(hdc, crPrevBk);
                hResult = StringCchLength(szHilite, 128/sizeof(TCHAR), pcch);
                if (FAILED(hResult))
                {
                // TODO: write error handler
                }
                TextOut(hdc, 0, nCurrentLine * dwLineHeight,
                    szHilite, *pcch);
                ReleaseDC(hwndMain, hdc);
                ShowCaret(hwndMain);
                fTextSelected = FALSE;
            }

            // Save the current mouse-cursor coordinates.

            ptsCursor = MAKEPOINTS(lParam);

            // Determine which line the cursor is on, and save
            // the line number. Do not allow line numbers greater
            // than the number of the last line of text. The
            // line number is later multiplied by the average height
            // of the current font. The result is used to set the
            // y-coordinate of the caret.

            nCurrentLine = min((int)(ptsCursor.y / dwLineHeight),
                nLastLine);

            // Parse the text input buffer to find the first
            // character in the selected line of text. Each
            // line ends with a carriage return, so it is possible
            // to count the carriage returns to find the selected
            // line.

            cCR = 0;
            nBegLine = 0;
            if (nCurrentLine != 0)
            {
                for (i = 0; (i < cch) &&
                        (cCR < nCurrentLine); i++)
                {
                    if (pchInputBuf[i] == 0x0D)
                        ++cCR;
                }
                nBegLine = i;
            }

            // Starting at the beginning of the selected line,
            // measure  the width of each character, summing the
            // width with each character measured. Stop when the
            // sum is greater than the x-coordinate of the cursor.
            // The sum is used to set the x-coordinate of the caret.
            hdc = GetDC(hwndMain);
            nCaretPosX = 0;
            for (i = nBegLine;
                (pchInputBuf[i] != 0x0D) && (i < cch); i++)
            {
                ch = pchInputBuf[i];
                GetCharWidth32(hdc, (int) ch, (int) ch, &nCharWidth);
                if ((nCaretPosX + nCharWidth) > ptsCursor.x) break;
                else nCaretPosX += nCharWidth;
            }
            ReleaseDC(hwndMain, hdc);

            // Set the caret to the user-selected position.

            SetCaretPos(nCaretPosX, nCurrentLine * dwLineHeight);
            break;

        case WM_LBUTTONDBLCLK:

            // Copy the selected line of text to a buffer.

            for (i = nBegLine, j = 0; (pchInputBuf[i] != 0x0D) &&
                    (i < cch); i++)
            {
                szHilite[j++] = pchInputBuf[i];
            }
            szHilite[j] = '\0';

            // Hide the caret, invert the background and foreground
            // colors, and then redraw the selected line.

            HideCaret(hwndMain);
            hdc = GetDC(hwndMain);
            crPrevText = SetTextColor(hdc, RGB(255, 255, 255));
            crPrevBk = SetBkColor(hdc, RGB(0, 0, 0));
            hResult = StringCchLength(szHilite, 128/sizeof(TCHAR), pcch);
            if (FAILED(hResult))
            {
            // TODO: write error handler
            }
            TextOut(hdc, 0, nCurrentLine * dwLineHeight, szHilite, *pcch);
            SetTextColor(hdc, crPrevText);
            SetBkColor(hdc, crPrevBk);
            ReleaseDC(hwndMain, hdc);

            fTextSelected = TRUE;
            break;

            // Process other messages.

        default:
            return DefWindowProc(hwndMain, uMsg, wParam, lParam);
    }
    return NULL;
}
```

Using a Mouse Wheel in a Document with Embedded Objects

This example assumes a Microsoft Word document with various embedded objects:

- A Microsoft Excel spreadsheet
- An embedded list box control that scrolls in response to the wheel
- An embedded text box control that does not respond to the wheel

The MSH_MOUSEWHEEL message is always sent to the main window in Microsoft Word. This is true even if the embedded spreadsheet is active. The following table explains how the MSH_MOUSEWHEEL message is handled according to the focus.

FOCUS IS ON	HANDLING IS AS FOLLOWS
Word document	Word scrolls the document window.
Embedded Excel spreadsheet	Word posts the message to Excel. You must decide if the embedded application should respond to the message or not.
Embedded control	It is up to the application to send the message to an embedded control that has the focus and check the return code to see if the control handled it. If the control did not handle it, then the application should scroll the document window. For example, if the user clicked a list box and then rolled the wheel, that control would scroll in response to a wheel rotation. If the user clicked a text box and then rotated the wheel, the whole document would scroll.

This following example shows how an application might handle the two wheel messages.

```
/*************************************************
* this code deals with MSH_MOUSEWHEEL
*************************************************/
#include "zmouse.h"

//
// Mouse Wheel rotation stuff, only define if we are
// on a version of the OS that does not support
// WM_MOUSEWHEEL messages.
//
#ifndef WM_MOUSEWHEEL
#define WM_MOUSEWHEEL WM_MOUSELAST+1
    // Message ID for IntelliMouse wheel
#endif

UINT uMSH_MOUSEWHEEL = 0;   // Value returned from
                            // RegisterWindowMessage()

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

INT WINAPI WinMain(
         HINSTANCE hInst,
         HINSTANCE hPrevInst,
         LPSTR lpCmdLine,
         INT nCmdShow)
{
    MSG msg;
    BOOL bRet;

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

    //
    // The new IntelliMouse uses a Registered message to transmit
    // wheel rotation info. So register for it!

    uMSH_MOUSEWHEEL =
     RegisterWindowMessage(MSH_MOUSEWHEEL);
    if ( !uMSH_MOUSEWHEEL )
    {
        MessageBox(NULL,"
                    RegisterWindowMessag Failed!",
                    "Error",MB_OK);
        return msg.wParam;
    }

    while (( bRet = GetMessage(&msg, NULL, 0, 0)) != 0)
    {
        if (bRet == -1)
        {
            // handle the error and possibly exit
        }
        else
        {
            if (!TranslateAccelerator(ghwndApp,
                                      ghaccelTable,
                                      &msg))
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
    }

    return msg.wParam;
}

/************************************************
* this code deals with WM_MOUSEWHEEL
*************************************************/
LONG APIENTRY MainWndProc(
    HWND hwnd,
    UINT msg,
    WPARAM wParam,
    LPARAM lParam)
{
    static int nZoom = 0;


    switch (msg)
    {
        //
        // Handle Mouse Wheel messages generated
        // by the operating systems that have built-in
        // support for the WM_MOUSEWHEEL message.
        //

        case WM_MOUSEWHEEL:
            ((short) HIWORD(wParam)< 0) ? nZoom-- : nZoom++;

            //
            // Do other wheel stuff...
            //

            break;

        default:
            //
            // uMSH_MOUSEWHEEL is a message registered by
            // the mswheel dll on versions of Windows that
            // do not support the new message in the OS.

            if( msg == uMSH_MOUSEWHEEL )
            {
               ((int)wParam < 0) ? nZoom-- : nZoom++;

                //
                // Do other wheel stuff...
                // Do other wheel stuff...
                //
                break;
            }

            return DefWindowProc(hwnd,
                                 msg,
                                 wParam,
                                 lParam);
    }

    return 0L;
}
```

Retrieving the Number of Mouse Wheel Scroll Lines

The following code allows an application to retrieve the number of scroll lines using the SystemParametersInfo function.

```
#ifndef SPI_GETWHEELSCROLLLINES
#define SPI_GETWHEELSCROLLLINES   104
#endif

#include "zmouse.h"

/*********************************************************
* FUNCTION: GetNumScrollLines
* Purpose : An OS independent method to retrieve the
*           number of wheel scroll lines
* Params  : none
* Returns : UINT:  Number of scroll lines where WHEEL_PAGESCROLL
*           indicates to scroll a page at a time.
*********************************************************/
UINT GetNumScrollLines(void)
{
   HWND hdlMsWheel;
   UINT ucNumLines=3;  // 3 is the default
   OSVERSIONINFO osversion;
   UINT uiMsh_MsgScrollLines;


   memset(&osversion, 0, sizeof(osversion));
   osversion.dwOSVersionInfoSize =sizeof(osversion);
   GetVersionEx(&osversion);

   if ((osversion.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) ||
       ( (osversion.dwPlatformId == VER_PLATFORM_WIN32_NT) &&
         (osversion.dwMajorVersion < 4) )   )
   {
        hdlMsWheel = FindWindow(MSH_WHEELMODULE_CLASS,
                                MSH_WHEELMODULE_TITLE);
        if (hdlMsWheel)
        {
           uiMsh_MsgScrollLines = RegisterWindowMessage
                                       (MSH_SCROLL_LINES);
           if (uiMsh_MsgScrollLines)
                ucNumLines = (int)SendMessage(hdlMsWheel,
                                    uiMsh_MsgScrollLines,
                                                       0,
                                                       0);
        }
   }
   else if ( (osversion.dwPlatformId ==
                      VER_PLATFORM_WIN32_NT) &&
             (osversion.dwMajorVersion >= 4) )
   {
      SystemParametersInfo(SPI_GETWHEELSCROLLLINES,
                                          0,
                                    &ucNumLines, 0);
   }
   return(ucNumLines);
}
```

Mouse Input Reference

4/13/2022 • 2 minutes to read • Edit Online

The topics contained in this section provide the reference specifications for mouse input.

In this section

TOPIC	DESCRIPTION
Mouse Input Functions	
Mouse Input Macros	
Mouse Input Notifications	
Mouse Input Structures	

Mouse Input Functions

4/13/2022 • 2 minutes to read • Edit Online

In this section

TOPIC	DESCRIPTION
_TrackMouseEvent	Posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time. This function calls TrackMouseEvent if it exists, otherwise it emulates it.
DragDetect	Captures the mouse and tracks its movement until the user releases the left button, presses the ESC key, or moves the mouse outside the drag rectangle around the specified point. The width and height of the drag rectangle are specified by the **SM_CXDRAG** and **SM_CYDRAG** values returned by the GetSystemMetrics function.
GetCapture	Retrieves a handle to the window (if any) that has captured the mouse. Only one window at a time can capture the mouse; this window receives mouse input whether or not the cursor is within its borders.
GetDoubleClickTime	Retrieves the current double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second click of a double-click. The maximum double-click time is 5000 milliseconds.
GetMouseMovePointsEx	Retrieves a history of up to 64 previous coordinates of the mouse or pen.
mouse_event	The mouse_event function synthesizes mouse motion and button clicks. > [!Note] > This function has been superseded. Use SendInput instead.
ReleaseCapture	Releases the mouse capture from a window in the current thread and restores normal mouse input processing. A window that has captured the mouse receives all mouse input, regardless of the position of the cursor, except when a mouse button is clicked while the cursor hot spot is in the window of another thread.
SetCapture	Sets the mouse capture to the specified window belonging to the current thread.

TOPIC	DESCRIPTION
SetDoubleClickTime	Sets the double-click time for the mouse. A double-click is a series of two clicks of a mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.
SwapMouseButton	Reverses or restores the meaning of the left and right mouse buttons.
TrackMouseEvent	Posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time. [!Note] The _TrackMouseEvent function calls TrackMouseEvent if it exists, otherwise **_TrackMouseEvent** emulates **TrackMouseEvent**.

Mouse Input Macros

4/13/2022 • 2 minutes to read • Edit Online

In This Section

Mouse Input Notifications

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- WM_CAPTURECHANGED
- WM_LBUTTONDBLCLK
- WM_LBUTTONDOWN
- WM_LBUTTONUP
- WM_MBUTTONDBLCLK
- WM_MBUTTONDOWN
- WM_MBUTTONUP
- WM_MOUSEACTIVATE
- WM_MOUSEHOVER
- WM_MOUSEHWHEEL
- WM_MOUSELEAVE
- WM_MOUSEMOVE
- WM_MOUSEWHEEL
- WM_NCHITTEST
- WM_NCLBUTTONDBLCLK
- WM_NCLBUTTONDOWN
- WM_NCLBUTTONUP
- WM_NCMBUTTONDBLCLK
- WM_NCMBUTTONDOWN
- WM_NCMBUTTONUP
- WM_NCMOUSEHOVER
- WM_NCMOUSELEAVE
- WM_NCMOUSEMOVE
- WM_NCRBUTTONDBLCLK
- WM_NCRBUTTONDOWN
- WM_NCRBUTTONUP
- WM_NCXBUTTONDBLCLK
- WM_NCXBUTTONDOWN
- WM_NCXBUTTONUP
- WM_RBUTTONDBLCLK
- WM_RBUTTONDOWN
- WM_RBUTTONUP
- WM_XBUTTONDBLCLK
- WM_XBUTTONDOWN
- WM_XBUTTONUP

WM_CAPTURECHANGED message

4/13/2022 • 2 minutes to read • Edit Online

Sent to the window that is losing the mouse capture.

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

```
#define WM_CAPTURECHANGED               0x0215
```

Parameters

wParam

This parameter is not used.

lParam

A handle to the window gaining the mouse capture.

Return value

An application should return zero if it processes this message.

Remarks

A window receives this message even if it calls **ReleaseCapture** itself. An application should not attempt to set the mouse capture in response to this message.

When it receives this message, a window should redraw itself, if necessary, to reflect the new mouse-capture state.

Requirements

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

See also

Reference

ReleaseCapture

SetCapture

Conceptual

WM_LBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_LBUTTONDBLCLK                0x0203
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Only windows that have the **CS_DBLCLKS** style can receive **WM_LBUTTONDBLCLK** messages, which the system generates whenever the user presses, releases, and again presses the left mouse button within the system's double-click time limit. Double-clicking the left mouse button actually generates a sequence of four messages: **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_LBUTTONDBLCLK**, and **WM_LBUTTONUP**.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

GetDoubleClickTime

SetCapture

SetDoubleClickTime

WM_LBUTTONDOWN

WM_LBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_LBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_LBUTTONDOWN                  0x0201
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Example

```
LRESULT CALLBACK WndProc(_In_ HWND hWnd, _In_ UINT msg, _In_ WPARAM wParam, _In_ LPARAM lParam)
{
    POINT pt;

    switch (msg)
    {

    case WM_LBUTTONDOWN:
            {
                pt.x = GET_X_LPARAM(lParam);
                pt.y = GET_Y_LPARAM(lParam);
            }
        }
        break;

    default:
        return DefWindowProc(hWnd, msg, wParam, lParam);
    }
    return 0;
}
```

For more examples see Windows Classic Samples on GitHub.

Remarks

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

To detect that the ALT key was pressed, check whether GetKeyState with **VK_MENU** < 0. Note, this must not be **GetAsyncKeyState**.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

GetKeyState

SetCapture

WM_LBUTTONDBLCLK

WM_LBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_LBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_LBUTTONUP                    0x0202
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the lParam value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the MAKEPOINTS macro to obtain a POINTS structure from the return value. You can also use the GET_X_LPARAM or GET_Y_LPARAM macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and LOWORD and HIWORD treat the coordinates as unsigned quantities.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

SetCapture

WM_LBUTTONDBLCLK

WM_LBUTTONDOWN

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_MBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_MBUTTONDBLCLK                0x0209
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Only windows that have the **CS_DBLCLKS** style can receive **WM_MBUTTONDBLCLK** messages, which the system generates when the user presses, releases, and again presses the middle mouse button within the system's double-click time limit. Double-clicking the middle mouse button actually generates four messages: **WM_MBUTTONDOWN**, **WM_MBUTTONUP**, **WM_MBUTTONDBLCLK**, and **WM_MBUTTONUP** again.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

GetDoubleClickTime

SetCapture

SetDoubleClickTime

WM_MBUTTONDOWN

WM_MBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_MBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_MBUTTONDOWN                  0x0207
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the MAKEPOINTS macro to obtain a POINTS structure from the return value. You can also use the GET_X_LPARAM or GET_Y_LPARAM macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

To detect that the ALT key was pressed, check whether GetKeyState with **VK_MENU** < 0. Note, this must not be GetAsyncKeyState.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

GetKeyState

SetCapture

WM_MBUTTONDBLCLK

WM_MBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_MBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_MBUTTONUP                    0x0208
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Note that when a shortcut menu is present (displayed), coordinates are relative to the screen, not the client area. Because **TrackPopupMenu** is an asynchronous call and the **WM_MBUTTONUP** notification does not have a special flag indicating coordinate derivation, an application cannot tell if the x,y coordinates contained in *lParam* are relative to the screen or the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

SetCapture

WM_MBUTTONDBLCLK

WM_MBUTTONDOWN

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_MOUSEACTIVATE message

4/13/2022 • 2 minutes to read • Edit Online

Sent when the cursor is in an inactive window and the user presses a mouse button. The parent window receives this message only if the child window passes it to the DefWindowProc function.

A window receives this message through its WindowProc function.

```
#define WM_MOUSEACTIVATE                0x0021
```

Parameters

wParam

A handle to the top-level parent window of the window being activated.

lParam

The low-order word specifies the hit-test value returned by the DefWindowProc function as a result of processing the WM_NCHITTEST message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word specifies the identifier of the mouse message generated when the user pressed a mouse button. The mouse message is either discarded or posted to the window, depending on the return value.

Return value

The return value specifies whether the window should be activated and whether the identifier of the mouse message should be discarded. It must be one of the following values.

RETURN CODE/VALUE	DESCRIPTION
MA_ACTIVATE 1	Activates the window, and does not discard the mouse message.
MA_ACTIVATEANDEAT 2	Activates the window, and discards the mouse message.
MA_NOACTIVATE 3	Does not activate the window, and does not discard the mouse message.
MA_NOACTIVATEANDEAT 4	Does not activate the window, but discards the mouse message.

Remarks

The DefWindowProc function passes the message to a child window's parent window before any processing occurs. The parent window determines whether to activate the child window. If it activates the child window, the parent window should return **MA_NOACTIVATE** or **MA_NOACTIVATEANDEAT** to prevent the system from processing the message further.

Requirements

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

See also

Reference

DefWindowProc

HIWORD

LOWORD

WM_NCHITTEST

Conceptual

Mouse Input

WM_MOUSEHOVER message

4/13/2022 • 2 minutes to read • Edit Online

Posted to a window when the cursor hovers over the client area of the window for the period of time specified in a prior call to TrackMouseEvent.

A window receives this message through its WindowProc function.

```
#define WM_MOUSEHOVER                   0x02A1
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is depressed.
MK_LBUTTON 0x0001	The left mouse button is depressed.
MK_MBUTTON 0x0010	The middle mouse button is depressed.
MK_RBUTTON 0x0002	The right mouse button is depressed.
MK_SHIFT 0x0004	The SHIFT key is depressed.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Hover tracking stops when **WM_MOUSEHOVER** is generated. The application must call TrackMouseEvent again if it requires further tracking of mouse hover behavior.

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the MAKEPOINTS macro to obtain a POINTS structure from the return value. You can also use the GET_X_LPARAM or GET_Y_LPARAM macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

SetCapture

TrackMouseEvent

TRACKMOUSEEVENT

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_MOUSEHWHEEL message

4/13/2022 • 2 minutes to read • Edit Online

Sent to the active window when the mouse's horizontal scroll wheel is tilted or rotated. The **DefWindowProc** function propagates the message to the window's parent. There should be no internal forwarding of the message, since **DefWindowProc** propagates it up the parent chain until it finds a window that processes it.

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

```
#define WM_MOUSEHWHEEL                  0x020E
```

Parameters

wParam

The high-order word indicates the distance the wheel is rotated, expressed in multiples or factors of **WHEEL_DELTA**, which is set to 120. A positive value indicates that the wheel was rotated to the right; a negative value indicates that the wheel was rotated to the left.

The low-order word indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.

VALUE	MEANING
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the pointer, relative to the upper-left corner of the screen.

The high-order word specifies the y-coordinate of the pointer, relative to the upper-left corner of the screen.

Return value

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

Remarks

Use the following code to obtain the information in the *wParam* parameter.

```
fwKeys = GET_KEYSTATE_WPARAM(wParam);
zDelta = GET_WHEEL_DELTA_WPARAM(wParam);
```

Use the following code to obtain the horizontal and vertical position.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

The wheel rotation is a multiple of **WHEEL_DELTA**, which is set to 120. This is the threshold for action to be taken, and one such action (for example, scrolling one increment) should occur for each delta.

The delta was set to 120 to allow Microsoft or other vendors to build finer-resolution wheels (for example, a freely-rotating wheel with no notches) to send more messages per rotation, but with a smaller value in each message. To use this feature, you can either add the incoming delta values until **WHEEL_DELTA** is reached (so for a delta-rotation you get the same response), or scroll partial lines in response to more frequent messages. You can also choose your scroll granularity and accumulate deltas until it is reached.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	Winuser.h (include Windowsx.h)

See also

Reference

GET_KEYSTATE_WPARAM

GET_X_LPARAM

GET_Y_LPARAM

GET_WHEEL_DELTA_WPARAM

HIWORD

LOWORD

mouse_event

Conceptual

Mouse Input

Other Resources

GetSystemMetrics

MAKEPOINTS

POINTS

SystemParametersInfo

WM_MOUSELEAVE message

4/13/2022 • 2 minutes to read • Edit Online

Posted to a window when the cursor leaves the client area of the window specified in a prior call to **TrackMouseEvent**.

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

```
#define WM_MOUSELEAVE                   0x02A3
```

Parameters

wParam

This parameter is not used and must be zero.

lParam

This parameter is not used and must be zero.

Return value

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

Remarks

All tracking requested by **TrackMouseEvent** is canceled when this message is generated. The application must call **TrackMouseEvent** when the mouse reenters its window if it requires further tracking of mouse hover behavior.

Requirements

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

See also

Reference

GetCapture

SetCapture

TrackMouseEvent

TRACKMOUSEEVENT

WM_NCMOUSELEAVE

Conceptual

Mouse Input

WM_MOUSEMOVE message

4/13/2022 • 2 minutes to read • Edit Online

Posted to a window when the cursor moves. If the mouse is not captured, the message is posted to the window that contains the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_MOUSEMOVE                    0x0200
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

SetCapture

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_MOUSEWHEEL message

4/13/2022 • 3 minutes to read • Edit Online

Sent to the focus window when the mouse wheel is rotated. The **DefWindowProc** function propagates the message to the window's parent. There should be no internal forwarding of the message, since **DefWindowProc** propagates it up the parent chain until it finds a window that processes it.

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

```
#define WM_MOUSEWHEEL                   0x020A
```

Parameters

wParam

The high-order word indicates the distance the wheel is rotated, expressed in multiples or divisions of **WHEEL_DELTA**, which is 120. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user.

The low-order word indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.

VALUE	MEANING
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the pointer, relative to the upper-left corner of the screen.

The high-order word specifies the y-coordinate of the pointer, relative to the upper-left corner of the screen.

Return value

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

Remarks

Use the following code to get the information in the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM(wParam);
zDelta = GET_WHEEL_DELTA_WPARAM(wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

The wheel rotation will be a multiple of **WHEEL_DELTA**, which is set at 120. This is the threshold for action to be taken, and one such action (for example, scrolling one increment) should occur for each delta.

The delta was set to 120 to allow Microsoft or other vendors to build finer-resolution wheels (a freely-rotating wheel with no notches) to send more messages per rotation, but with a smaller value in each message. To use this feature, you can either add the incoming delta values until **WHEEL_DELTA** is reached (so for a delta-rotation you get the same response), or scroll partial lines in response to the more frequent messages. You can also choose your scroll granularity and accumulate deltas until it is reached.

Note, there is no *fwKeys* for **MSH_MOUSEWHEEL**. Otherwise, the parameters are exactly the same as for **WM_MOUSEWHEEL**.

It is up to the application to forward **MSH_MOUSEWHEEL** to any embedded objects or controls. The application is required to send the message to an active embedded OLE application. It is optional that the application sends it to a wheel-enabled control with focus. If the application does send the message to a control, it can check the return value to see if the message was processed. Controls are required to return a value of **TRUE** if they process the message.

Requirements

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

See also

Reference

GET_KEYSTATE_WPARAM

GET_X_LPARAM

GET_Y_LPARAM

GET_WHEEL_DELTA_WPARAM

HIWORD

LOWORD

mouse_event

Conceptual

Mouse Input

Other Resources

GetSystemMetrics

MAKEPOINTS

POINTS

SystemParametersInfo

WM_NCHITTEST message

4/13/2022 • 3 minutes to read • Edit Online

Sent to a window in order to determine what part of the window corresponds to a particular screen coordinate. This can happen, for example, when the cursor moves, when a mouse button is pressed or released, or in response to a call to a function such as **WindowFromPoint**. If the mouse is not captured, the message is sent to the window beneath the cursor. Otherwise, the message is sent to the window that has captured the mouse.

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

```
#define WM_NCHITTEST                    0x0084
```

Parameters

wParam

This parameter is not used.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the screen.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the screen.

Return value

The return value of the **DefWindowProc** function is one of the following values, indicating the position of the cursor hot spot.

RETURN CODE/VALUE	DESCRIPTION
HTBORDER 18	In the border of a window that does not have a sizing border.
HTBOTTOM 15	In the lower-horizontal border of a resizable window (the user can click the mouse to resize the window vertically).
HTBOTTOMLEFT 16	In the lower-left corner of a border of a resizable window (the user can click the mouse to resize the window diagonally).
HTBOTTOMRIGHT 17	In the lower-right corner of a border of a resizable window (the user can click the mouse to resize the window diagonally).

RETURN CODE/VALUE	DESCRIPTION
HTCAPTION 2	In a title bar.
HTCLIENT 1	In a client area.
HTCLOSE 20	In a **Close** button.
HTERROR -2	On the screen background or on a dividing line between windows (same as **HTNOWHERE**, except that the **DefWindowProc** function produces a system beep to indicate an error).
HTGROWBOX 4	In a size box (same as **HTSIZE**).
HTHELP 21	In a **Help** button.
HTHSCROLL 6	In a horizontal scroll bar.
HTLEFT 10	In the left border of a resizable window (the user can click the mouse to resize the window horizontally).
HTMENU 5	In a menu.
HTMAXBUTTON 9	In a **Maximize** button.
HTMINBUTTON 8	In a **Minimize** button.
HTNOWHERE 0	On the screen background or on a dividing line between windows.

RETURN CODE/VALUE	DESCRIPTION
HTREDUCE 8	In a **Minimize** button.
HTRIGHT 11	In the right border of a resizable window (the user can click the mouse to resize the window horizontally).
HTSIZE 4	In a size box (same as **HTGROWBOX**).
HTSYSMENU 3	In a window menu or in a **Close** button in a child window.
HTTOP 12	In the upper-horizontal border of a window.
HTTOPLEFT 13	In the upper-left corner of a window border.
HTTOPRIGHT 14	In the upper-right corner of a window border.
HTTRANSPARENT -1	In a window currently covered by another window in the same thread (the message will be sent to underlying windows in the same thread until one of them returns a code that is not **HTTRANSPARENT**).
HTVSCROLL 7	In the vertical scroll bar.
HTZOOM 9	In a **Maximize** button.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order `short` of the return value; the y-coordinate is in the high-order `short` (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the MAKEPOINTS macro to obtain a

POINTS structure from the return value. You can also use the GET_X_LPARAM or GET_Y_LPARAM macro to extract the x- or y-coordinate.

> **IMPORTANT**
>
> Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and LOWORD and HIWORD treat the coordinates as unsigned quantities.

Windows Vista: When creating custom frames that include the standard caption buttons, this message should first be passed to the DwmDefWindowProc function. This enables the Desktop Window Manager (DWM) to provide hit-testing for the captions buttons. If **DwmDefWindowProc** does not handle the message, further processing of **WM_NCHITTEST** may be needed.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCLBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCLBUTTONDBLCLK              0x00A3
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

By default, the **DefWindowProc** function tests the specified point to find out the location of the cursor and performs the appropriate action. If appropriate, **DefWindowProc** sends the **WM_SYSCOMMAND** message to the window.

A window need not have the **CS_DBLCLKS** style to receive **WM_NCLBUTTONDBLCLK** messages.

The system generates a **WM_NCLBUTTONDBLCLK** message when the user presses, releases, and again presses the left mouse button within the system's double-click time limit. Double-clicking the left mouse button actually generates four messages: **WM_NCLBUTTONDOWN**, **WM_NCLBUTTONUP**,

WM_NCLBUTTONDBLCLK, and **WM_NCLBUTTONUP** again.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCLBUTTONDOWN

WM_NCLBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCLBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its WindowProc function.

```
#define WM_NCLBUTTONDOWN                0x00A1
```

Parameters

wParam

The hit-test value returned by the DefWindowProc function as a result of processing the WM_NCHITTEST message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A POINTS structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

The DefWindowProc function tests the specified point to find the location of the cursor and performs the appropriate action. If appropriate, **DefWindowProc** sends the WM_SYSCOMMAND message to the window.

You can also use the GET_X_LPARAM and GET_Y_LPARAM macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCLBUTTONDBLCLK

WM_NCLBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCLBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCLBUTTONUP                  0x00A2
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

The **DefWindowProc** function tests the specified point to find out the location of the cursor and performs the appropriate action. If appropriate, **DefWindowProc** sends the **WM_SYSCOMMAND** message to the window.

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCLBUTTONDBLCLK

WM_NCLBUTTONDOWN

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCMBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCMBUTTONDBLCLK              0x00A9
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

A window need not have the **CS_DBLCLKS** style to receive **WM_NCMBUTTONDBLCLK** messages.

The system generates a **WM_NCMBUTTONDBLCLK** message when the user presses, releases, and again presses the middle mouse button within the system's double-click time limit. Double-clicking the middle mouse button actually generates four messages: **WM_NCMBUTTONDOWN**, **WM_NCMBUTTONUP**, **WM_NCMBUTTONDBLCLK**, and **WM_NCMBUTTONUP** again.

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCMBUTTONDOWN

WM_NCMBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCMBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCMBUTTONDOWN                0x00A7
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCMBUTTONDBLCLK

WM_NCMBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCMBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCMBUTTONUP                  0x00A8
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCMBUTTONDBLCLK

WM_NCMBUTTONDOWN

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCMOUSEHOVER message

4/13/2022 • 2 minutes to read • Edit Online

Posted to a window when the cursor hovers over the nonclient area of the window for the period of time specified in a prior call to TrackMouseEvent.

A window receives this message through its WindowProc function.

```
#define WM_NCMOUSEHOVER                 0x02A0
```

Parameters

wParam

The hit-test value returned by the DefWindowProc function as a result of processing the WM_NCHITTEST message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A POINTS structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

Hover tracking stops when this message is generated. The application must call TrackMouseEvent again if it requires further tracking of mouse hover behavior.

You can also use the GET_X_LPARAM and GET_Y_LPARAM macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

TrackMouseEvent

TRACKMOUSEEVENT

WM_NCHITTEST

WM_MOUSEHOVER

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCMOUSELEAVE message

4/13/2022 • 2 minutes to read • Edit Online

Posted to a window when the cursor leaves the nonclient area of the window specified in a prior call to TrackMouseEvent.

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

```
#define WM_NCMOUSELEAVE                 0x02A2
```

Parameters

wParam

This parameter is not used and must be zero.

lParam

This parameter is not used and must be zero.

Return value

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

Remarks

All tracking requested by **TrackMouseEvent** is canceled when this message is generated. The application must call **TrackMouseEvent** when the mouse reenters its window if it requires further tracking of mouse hover behavior.

Requirements

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

See also

Reference

TrackMouseEvent

TRACKMOUSEEVENT

WM_SYSCOMMAND

WM_MOUSELEAVE

Conceptual

Mouse Input

WM_NCMOUSEMOVE message

4/13/2022 • 2 minutes to read • Edit Online

Posted to a window when the cursor is moved within the nonclient area of the window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCMOUSEMOVE                  0x00A0
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

REQUIREMENT	VALUE
Header	Winuser.h (include Windowsx.h)

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCRBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCRBUTTONDBLCLK              0x00A6
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

A window need not have the **CS_DBLCLKS** style to receive **WM_NCRBUTTONDBLCLK** messages.

The system generates a **WM_NCRBUTTONDBLCLK** message when the user presses, releases, and again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four messages: **WM_NCRBUTTONDOWN**, **WM_NCRBUTTONUP**, **WM_NCRBUTTONDBLCLK**, and **WM_NCRBUTTONUP** again.

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCRBUTTONDOWN

WM_NCRBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCRBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCRBUTTONDOWN                0x00A4
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCRBUTTONDBLCLK

WM_NCRBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCRBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCRBUTTONUP                  0x00A5
```

Parameters

wParam

The hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

A **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

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

Remarks

You can also use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y-coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCRBUTTONDBLCLK

WM_NCRBUTTONDOWN

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCXBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

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

```
#define WM_NCXBUTTONDBLCLK              0x00AD
```

Parameters

wParam

The low-order word specifies the hit-test value returned by the **DefWindowProc** function from processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word indicates which button was double-clicked. It can be one of the following values.

VALUE	MEANING
XBUTTON1 0x0001	The first X button was double-clicked..
XBUTTON2 0x0002	The second X button was double-clicked.

lParam

A pointer to a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to get the information in the *wParam* parameter.

```
nHittest = GET_NCHITTEST_WPARAM(wParam);
fwButton = GET_XBUTTON_WPARAM(wParam);
```

You can also use the following code to get the x- and y-coordinates from *lParam*:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

By default, the **DefWindowProc** function tests the specified point to get the position of the cursor and performs the appropriate action. If appropriate, it sends the **WM_SYSCOMMAND** message to the window.

A window need not have the **CS_DBLCLKS** style to receive **WM_NCXBUTTONDBLCLK** messages. The system generates a **WM_NCXBUTTONDBLCLK** message when the user presses, releases, and again presses an X button within the system's double-click time limit. Double-clicking one of these buttons actually generates four messages: **WM_NCXBUTTONDOWN**, **WM_NCXBUTTONUP**, **WM_NCXBUTTONDBLCLK**, and **WM_NCXBUTTONUP** again.

Unlike the **WM_NCLBUTTONDBLCLK**, **WM_NCMBUTTONDBLCLK**, and **WM_NCRBUTTONDBLCLK** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCXBUTTONDOWN

WM_NCXBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCXBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is *not* posted.

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

```
#define WM_NCXBUTTONDOWN                0x00AB
```

Parameters

wParam

The low-order word specifies the hit-test value returned by the **DefWindowProc** function from processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**. The high-order word indicates which button was pressed. It can be one of the following values.

VALUE	MEANING
XBUTTON1 0x0001	The first X button was pressed.
XBUTTON2 0x0002	The second X button was pressed.

lParam

A pointer to a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to get the information in the *wParam* parameter.

```
nHittest = GET_NCHITTEST_WPARAM(wParam);
fwButton = GET_XBUTTON_WPARAM(wParam);
```

You can also use the following code to get the x- and y-coordinates from *lParam*:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

By default, the **DefWindowProc** function tests the specified point to get the position of the cursor and performs the appropriate action. If appropriate, it sends the **WM_SYSCOMMAND** message to the window.

Unlike the **WM_NCLBUTTONDOWN**, **WM_NCMBUTTONDOWN**, and **WM_NCRBUTTONDOWN** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCXBUTTONDBLCLK

WM_NCXBUTTONUP

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_NCXBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is *not* posted.

A window receives this message through its WindowProc function.

```
#define WM_NCXBUTTONUP                  0x00AC
```

Parameters

wParam

The low-order word specifies the hit-test value returned by the DefWindowProc function from processing the WM_NCHITTEST message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word indicates which button was released. It can be one of the following values.

VALUE	MEANING
XBUTTON1 0x0001	The first X button was released.
XBUTTON2 0x0002	The second X button was released.

lParam

A pointer to a POINTS structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to get the information in the *wParam* parameter.

```
nHittest = GET_NCHITTEST_WPARAM(wParam);
fwButton = GET_XBUTTON_WPARAM(wParam);
```

You can also use the following code to get the x- and y-coordinates from *lParam*:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

By default, the **DefWindowProc** function tests the specified point to get the position of the cursor and performs the appropriate action. If appropriate, it sends the **WM_SYSCOMMAND** message to the window.

Unlike the **WM_NCLBUTTONUP**, **WM_NCMBUTTONUP**, and **WM_NCRBUTTONUP** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

Requirements

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

See also

Reference

DefWindowProc

GET_X_LPARAM

GET_Y_LPARAM

WM_NCHITTEST

WM_NCXBUTTONDBLCLK

WM_NCXBUTTONDOWN

WM_SYSCOMMAND

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_RBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_RBUTTONDBLCLK                0x0206
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Only windows that have the **CS_DBLCLKS** style can receive **WM_RBUTTONDBLCLK** messages, which the system generates whenever the user presses, releases, and again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four messages: **WM_RBUTTONDOWN**, **WM_RBUTTONUP**, **WM_RBUTTONDBLCLK**, and **WM_RBUTTONUP** again.

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

GetDoubleClickTime

SetCapture

SetDoubleClickTime

WM_RBUTTONDOWN

WM_RBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_RBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_RBUTTONDOWN                  0x0204
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of

the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

To detect that the ALT key was pressed, check whether **GetKeyState** with **VK_MENU** < 0. Note, this must not be **GetAsyncKeyState**.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

GetKeyState

SetCapture

WM_RBUTTONDBLCLK

WM_RBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_RBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_RBUTTONUP                    0x0205
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

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

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

> **IMPORTANT**
> Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Requirements

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

See also

Reference

GET_X_LPARAM

GET_Y_LPARAM

GetCapture

SetCapture

WM_RBUTTONDBLCLK

WM_RBUTTONDOWN

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_XBUTTONDBLCLK message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user double-clicks the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_XBUTTONDBLCLK                0x020D
```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. It can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

The high-order word indicates which button was double-clicked. It can be one of the following values.

VALUE	MEANING
XBUTTON1 0x0001	The first X button was double-clicked.
XBUTTON2 0x0002	The second X button was double-clicked.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to get the information in the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM (wParam);
fwButton = GET_XBUTTON_WPARAM (wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Only windows that have the **CS_DBLCLKS** style can receive **WM_XBUTTONDBLCLK** messages, which the system generates whenever the user presses, releases, and again presses an X button within the system's double-click time limit. Double-clicking one of these buttons actually generates four messages: **WM_XBUTTONDOWN**, **WM_XBUTTONUP**, **WM_XBUTTONDBLCLK**, and **WM_XBUTTONUP** again.

Unlike the **WM_LBUTTONDBLCLK**, **WM_MBUTTONDBLCLK**, and **WM_RBUTTONDBLCLK** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

Requirements

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

See also

Reference

DefWindowProc

GET_KEYSTATE_WPARAM

GET_X_LPARAM

GET_XBUTTON_WPARAM

GET_Y_LPARAM

GetCapture

GetDoubleClickTime

SetDoubleClickTime

WM_XBUTTONDOWN

WM_XBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_XBUTTONDOWN message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user presses the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_XBUTTONDOWN                  0x020B
```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. It can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

The high-order word indicates which button was clicked. It can be one of the following values.

VALUE	MEANING
XBUTTON1 0x0001	The first X button was clicked.
XBUTTON2 0x0002	The second X button was clicked.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to get the information in the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM (wParam);
fwButton = GET_XBUTTON_WPARAM (wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

IMPORTANT

Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Unlike the **WM_LBUTTONDOWN**, **WM_MBUTTONDOWN**, and **WM_RBUTTONDOWN** messages, an application should return **TRUE** from this message if it processes it. Doing so allows software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

Requirements

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

See also

Reference

GET_KEYSTATE_WPARAM

GET_X_LPARAM

GET_XBUTTON_WPARAM

GET_Y_LPARAM

GetCapture

SetCapture

WM_XBUTTONDBLCLK

WM_XBUTTONUP

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

WM_XBUTTONUP message

4/13/2022 • 2 minutes to read • Edit Online

Posted when the user releases the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

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

```
#define WM_XBUTTONUP                    0x020C
```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. It can be one or more of the following values.

VALUE	MEANING
MK_CONTROL 0x0008	The CTRL key is down.
MK_LBUTTON 0x0001	The left mouse button is down.
MK_MBUTTON 0x0010	The middle mouse button is down.
MK_RBUTTON 0x0002	The right mouse button is down.
MK_SHIFT 0x0004	The SHIFT key is down.
MK_XBUTTON1 0x0020	The first X button is down.
MK_XBUTTON2 0x0040	The second X button is down.

The high-order word indicates which button was released. It can be one of the following values:

VALUE	MEANING
XBUTTON1 0x0001	The first X button was released.
XBUTTON2 0x0002	The second X button was released.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return value

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to get the information in the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM (wParam);
fwButton = GET_XBUTTON_WPARAM (wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

As noted above, the x-coordinate is in the low-order **short** of the return value; the y-coordinate is in the high-order **short** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x- or y-coordinate.

> **IMPORTANT**
>
> Do not use the **LOWORD** or **HIWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

Unlike the **WM_LBUTTONUP**, **WM_MBUTTONUP**, and **WM_RBUTTONUP** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

Requirements

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

See also

Reference

GET_KEYSTATE_WPARAM

GET_X_LPARAM

GET_XBUTTON_WPARAM

GET_Y_LPARAM

GetCapture

SetCapture

WM_XBUTTONDBLCLK

WM_XBUTTONDOWN

Conceptual

Mouse Input

Other Resources

MAKEPOINTS

POINTS

Mouse Input Structures

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- MOUSEMOVEPOINT
- TRACKMOUSEEVENT

Raw Input

4/13/2022 • 2 minutes to read • Edit Online

This section describes how the system provides raw input to your application and how an application receives and processes that input. Raw input is sometimes referred to as generic input.

In This Section

NAME	DESCRIPTION
About Raw Input	Discusses user-input from devices such as joysticks, touch screens, and microphones.
Using Raw Input	Provides sample code for tasks relating to raw input.
Raw Input Reference	Contains the API reference.

Functions

NAME	DESCRIPTION
DefRawInputProc	Calls the default raw input procedure to provide default processing for any raw input messages that an application does not process. This function ensures that every message is processed. **DefRawInputProc** is called with the same parameters received by the window procedure.
GetRawInputBuffer	Performs a buffered read of the raw input data.
GetRawInputData	Gets the raw input from the specified device.
GetRawInputDeviceInfo	Gets information about the raw input device.
GetRawInputDeviceList	Enumerates the raw input devices attached to the system.
GetRegisteredRawInputDevices	Gets the information about the raw input devices for the current application.
RegisterRawInputDevices	Registers the devices that supply the raw input data.

Macros

NAME	DESCRIPTION
GET_RAWINPUT_CODE_WPARAM	Gets the input code from *wParam* in **WM_INPUT**.
NEXTRAWINPUTBLOCK	Gets the location of the next structure in an array of **RAWINPUT** structures.

Notifications

NAME	DESCRIPTION
WM_INPUT	Sent to the window that is getting raw input.
WM_INPUT_DEVICE_CHANGE	Sent to the window that registered to receive raw input.

Structures

NAME	DESCRIPTION
RAWHID	Describes the format of the raw input from a Human Interface Device (HID).
RAWINPUT	Contains the raw input from a device.
RAWINPUTDEVICE	Defines information for the raw input devices.
RAWINPUTDEVICELIST	Contains information about a raw input device.
RAWINPUTHEADER	Contains the header information that is part of the raw input data.
RAWKEYBOARD	Contains information about the state of the keyboard.
RAWMOUSE	Contains information about the state of the mouse.
RID_DEVICE_INFO	Defines the raw input data coming from any device.
RID_DEVICE_INFO_HID	Defines the raw input data coming from the specified HID.
RID_DEVICE_INFO_KEYBOARD	Defines the raw input data coming from the specified keyboard.
RID_DEVICE_INFO_MOUSE	Defines the raw input data coming from the specified mouse.

About Raw Input

4/13/2022 • 4 minutes to read • Edit Online

There are many user-input devices beside the traditional keyboard and mouse. For example, user input can come from a joystick, a touch screen, a microphone, or other devices that allow great flexibility in user input. These devices are collectively known as Human Interface Devices (HIDs). The raw input API provides a stable and robust way for applications to accept raw input from any HID, including the keyboard and mouse.

This section covers the following topics:

Raw Input Model

Previously, the keyboard and mouse typically generated input data. The system interpreted the data coming from these devices in a way that eliminated the device-specific details of the raw information. For example, the keyboard generates the device-specific scan code but the system provides an application with the virtual key code. Besides hiding the details of the raw input, the window manager did not support all the new HIDs. To get input from the unsupported HIDs, an application had to do many things: open the device, manage the shared mode, periodically read the device or set up the I/O completion port, and so forth. The raw input model and the associated APIs were developed to allow simple access to raw input from all input devices, including the keyboard and mouse.

The raw input model is different from the original Windows input model for the keyboard and mouse. In the original input model, an application receives device-independent input in the form of messages that are sent or posted to its windows, such as **WM_CHAR**, **WM_MOUSEMOVE**, and **WM_APPCOMMAND**. In contrast, for raw input an application must register the devices it wants to get data from. Also, the application gets the raw input through the **WM_INPUT** message.

There are several advantages to the raw input model:

- An application does not have to detect or open the input device.
- An application gets the data directly from the device, and processes the data for its needs.
- An application can distinguish the source of the input even if it is from the same type of device. For example, two mouse devices.
- An application manages the data traffic by specifying data from a collection of devices or only specific device types.
- HID devices can be used as they become available in the marketplace, without waiting for new message types or an updated OS to have new commands in **WM_APPCOMMAND**.

Note that **WM_APPCOMMAND** does provide for some HID devices. However, **WM_APPCOMMAND** is a higher level device-independent input event, while **WM_INPUT** sends raw, low level data that is specific to a device.

Registration for Raw Input

By default, no application receives raw input. To receive raw input from a device, an application must register the device.

To register devices, an application first creates an array of **RAWINPUTDEVICE** structures that specify the top level collection (TLC) for the devices it wants. The TLC is defined by a Usage Page (the class of the device) and a Usage ID (the device within the class). For example, to get the keyboard TLC, set UsagePage = 0x01 and UsageID = 0x06. The application calls **RegisterRawInputDevices** to register the devices.

Note that an application can register a device that is not currently attached to the system. When this device is attached, the Windows Manager will automatically send the raw input to the application. To get the list of raw input devices on the system, an application calls **GetRawInputDeviceList**. Using the *hDevice* from this call, an application calls **GetRawInputDeviceInfo** to get the device information.

Through the **dwFlags** member of **RAWINPUTDEVICE**, an application can select the devices to listen to and also those it wants to ignore. For example, an application can ask for input from all telephony devices except for answering machines. For sample code, see Registering for Raw Input.

Note that the mouse and the keyboard are also HIDs, so data from them can come through both the HID message **WM_INPUT** and from traditional messages. An application can select either method by the proper selection of flags in **RAWINPUTDEVICE**.

To get an application's registration status, call **GetRegisteredRawInputDevices** at any time.

Reading Raw Input

An application receives raw input from any HID whose top level collection (TLC) matches a TLC from the registration. When an application receives raw input, its message queue gets a **WM_INPUT** message and the queue status flag **QS_RAWINPUT** is set (**QS_INPUT** also includes this flag). An application can receive data when it is in the foreground and when it is in the background.

There are two ways to read the raw data: the unbuffered (or standard) method and the buffered method. The unbuffered method gets the raw data one **RAWINPUT** structure at a time and is adequate for many HIDs. Here, the application calls **GetMessage** to get the **WM_INPUT** message. Then the application calls **GetRawInputData** using the **RAWINPUT** handle contained in **WM_INPUT**. For an example, see Doing a Standard Read of Raw Input.

In contrast, the buffered method gets an array of **RAWINPUT** structures at a time. This is provided for devices that can produce large amounts of raw input. In this method, the application calls **GetRawInputBuffer** to get an array of **RAWINPUT** structures. Note that the **NEXTRAWINPUTBLOCK** macro is used to traverse an array of **RAWINPUT** structures. For an example, see Doing a Buffered Read of Raw Input.

To interpret the raw input, detailed information about the HIDs is required. An application gets the device information by calling **GetRawInputDeviceInfo** with the device handle. This handle can come either from **WM_INPUT** or from the **hDevice** member of **RAWINPUTHEADER**.

Using Raw Input

4/13/2022 • 2 minutes to read • Edit Online

This section includes sample code for the following purposes:

- Registering for Raw Input
  - Example 1
  - Example 2
- Performing a Standard Read of Raw Input
- Performing a Buffered Read of Raw Input

Registering for Raw Input

Example 1

In this sample, an application specifies the raw input from game controllers (both game pads and joysticks) and all devices off the telephony usage page except answering machines.

```
RAWINPUTDEVICE Rid[4];

Rid[0].usUsagePage = 0x01;          // HID_USAGE_PAGE_GENERIC
Rid[0].usUsage = 0x05;              // HID_USAGE_GENERIC_GAMEPAD
Rid[0].dwFlags = 0;                 // adds game pad
Rid[0].hwndTarget = 0;

Rid[1].usUsagePage = 0x01;          // HID_USAGE_PAGE_GENERIC
Rid[1].usUsage = 0x04;              // HID_USAGE_GENERIC_JOYSTICK
Rid[1].dwFlags = 0;                 // adds joystick
Rid[1].hwndTarget = 0;

Rid[2].usUsagePage = 0x0B;          // HID_USAGE_PAGE_TELEPHONY
Rid[2].usUsage = 0x00;
Rid[2].dwFlags = RIDEV_PAGEONLY;    // adds all devices from telephony page
Rid[2].hwndTarget = 0;

Rid[3].usUsagePage = 0x0B;          // HID_USAGE_PAGE_TELEPHONY
Rid[3].usUsage = 0x02;              // HID_USAGE_TELEPHONY_ANSWERING_MACHINE
Rid[3].dwFlags = RIDEV_EXCLUDE;     // excludes answering machines
Rid[3].hwndTarget = 0;

if (RegisterRawInputDevices(Rid, 4, sizeof(Rid[0])) == FALSE)
{
    //registration failed. Call GetLastError for the cause of the error.
}
```

Example 2

In this sample, an application wants raw input from the keyboard and mouse but wants to ignore legacy keyboard and mouse window messages (which would come from the same keyboard and mouse).

```
RAWINPUTDEVICE Rid[2];

Rid[0].usUsagePage = 0x01;          // HID_USAGE_PAGE_GENERIC
Rid[0].usUsage = 0x02;              // HID_USAGE_GENERIC_MOUSE
Rid[0].dwFlags = RIDEV_NOLEGACY;    // adds mouse and also ignores legacy mouse messages
Rid[0].hwndTarget = 0;

Rid[1].usUsagePage = 0x01;          // HID_USAGE_PAGE_GENERIC
Rid[1].usUsage = 0x06;              // HID_USAGE_GENERIC_KEYBOARD
Rid[1].dwFlags = RIDEV_NOLEGACY;    // adds keyboard and also ignores legacy keyboard messages
Rid[1].hwndTarget = 0;

if (RegisterRawInputDevices(Rid, 2, sizeof(Rid[0])) == FALSE)
{
    //registration failed. Call GetLastError for the cause of the error
}
```

Performing a Standard Read of Raw Input

This sample shows how an application does an unbuffered (or standard) read of raw input from either a keyboard or mouse Human Interface Device (HID) and then prints out various information from the device.

```
case WM_INPUT:
{
    UINT dwSize;

    GetRawInputData((HRAWINPUT)lParam, RID_INPUT, NULL, &dwSize, sizeof(RAWINPUTHEADER));
    LPBYTE lpb = new BYTE[dwSize];
    if (lpb == NULL)
    {
        return 0;
    }

    if (GetRawInputData((HRAWINPUT)lParam, RID_INPUT, lpb, &dwSize, sizeof(RAWINPUTHEADER)) != dwSize)
         OutputDebugString (TEXT("GetRawInputData does not return correct size !\n"));

    RAWINPUT* raw = (RAWINPUT*)lpb;

    if (raw->header.dwType == RIM_TYPEKEYBOARD)
    {
        hResult = StringCchPrintf(szTempOutput, STRSAFE_MAX_CCH,
            TEXT(" Kbd: make=%04x Flags:%04x Reserved:%04x ExtraInformation:%08x, msg=%04x VK=%04x \n"),
            raw->data.keyboard.MakeCode,
            raw->data.keyboard.Flags,
            raw->data.keyboard.Reserved,
            raw->data.keyboard.ExtraInformation,
            raw->data.keyboard.Message,
            raw->data.keyboard.VKey);
        if (FAILED(hResult))
        {
        // TODO: write error handler
        }
        OutputDebugString(szTempOutput);
    }
    else if (raw->header.dwType == RIM_TYPEMOUSE)
    {
        hResult = StringCchPrintf(szTempOutput, STRSAFE_MAX_CCH,
            TEXT("Mouse: usFlags=%04x ulButtons=%04x usButtonFlags=%04x usButtonData=%04x ulRawButtons=%04x
lLastX=%04x lLastY=%04x ulExtraInformation=%04x\r\n"),
            raw->data.mouse.usFlags,
            raw->data.mouse.ulButtons,
            raw->data.mouse.usButtonFlags,
            raw->data.mouse.usButtonData,
            raw->data.mouse.ulRawButtons,
            raw->data.mouse.lLastX,
            raw->data.mouse.lLastY,
            raw->data.mouse.ulExtraInformation);

        if (FAILED(hResult))
        {
        // TODO: write error handler
        }
        OutputDebugString(szTempOutput);
    }

    delete[] lpb;
    return 0;
}
```

Performing a Buffered Read of Raw Input

This sample shows how an application does a buffered read of raw input from a generic HID.

```
case MSG_GETRIBUFFER: // Private message
    {
    UINT cbSize;
    Sleep(1000);

    VERIFY(GetRawInputBuffer(NULL, &cbSize, sizeof(RAWINPUTHEADER)) == 0);
    cbSize *= 16; // up to 16 messages
    Log(_T("Allocating %d bytes"), cbSize);
    PRAWINPUT pRawInput = (PRAWINPUT)malloc(cbSize);
    if (pRawInput == NULL)
    {
        Log(_T("Not enough memory"));
        return;
    }

    for (;;)
    {
        UINT cbSizeT = cbSize;
        UINT nInput = GetRawInputBuffer(pRawInput, &cbSizeT, sizeof(RAWINPUTHEADER));
        Log(_T("nInput = %d"), nInput);
        if (nInput == 0)
        {
            break;
        }

        ASSERT(nInput > 0);
        PRAWINPUT* paRawInput = (PRAWINPUT*)malloc(sizeof(PRAWINPUT) * nInput);
        if (paRawInput == NULL)
        {
            Log(_T("paRawInput NULL"));
            break;
        }

        PRAWINPUT pri = pRawInput;
        for (UINT i = 0; i < nInput; ++i)
        {
            Log(_T(" input[%d] = @%p"), i, pri);
            paRawInput[i] = pri;
            pri = NEXTRAWINPUTBLOCK(pri);
        }

        free(paRawInput);
    }
    free(pRawInput);
}
```

Raw Input Reference

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- Raw Input Functions
- Raw Input Macros
- Raw Input Notifications
- Raw Input Structures

Raw Input Functions

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- DefRawInputProc
- GetRawInputBuffer
- GetRawInputData
- GetRawInputDeviceInfo
- GetRawInputDeviceList
- GetRegisteredRawInputDevices
- RegisterRawInputDevices

Raw Input Macros

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- GET_RAWINPUT_CODE_WPARAM
- NEXTRAWINPUTBLOCK

Raw Input Notifications

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- WM_INPUT
- WM_INPUT_DEVICE_CHANGE

WM_INPUT message

4/13/2022 • 2 minutes to read • Edit Online

Sent to the window that is getting raw input.

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

```
#define WM_INPUT 0x00FF
```

Parameters

wParam

The input code. Use **GET_RAWINPUT_CODE_WPARAM** macro to get the value.

Can be one of the following values:

VALUE	MEANING
RIM_INPUT 0	Input occurred while the application was in the foreground. The application must call **DefWindowProc** so the system can perform cleanup.
RIM_INPUTSINK 1	Input occurred while the application was not in the foreground.

lParam

A **HRAWINPUT** handle to the **RAWINPUT** structure that contains the raw input from the device. To get the raw data, use this handle in the call to **GetRawInputData**.

Return value

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

Remarks

Raw input is available only when the application calls **RegisterRawInputDevices** with valid device specifications.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]

REQUIREMENT	VALUE
Header	Winuser.h (include Windows.h)

See also

Reference

GetRawInputData

RegisterRawInputDevices

RAWINPUT

GET_RAWINPUT_CODE_WPARAM

Conceptual

Raw Input

WM_INPUT_DEVICE_CHANGE message

4/13/2022 • 2 minutes to read • Edit Online

Description

Sent to the window that registered to receive raw input.

Raw input notifications are available only after the application calls RegisterRawInputDevices with RIDEV_DEVNOTIFY flag.

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

```
#define WM_INPUT_DEVICE_CHANGE          0x00FE
```

Parameters

wParam

Type: **WPARAM**

This parameter can be one of the following values.

VALUE	MEANING
GIDC_ARRIVAL 1	A new device has been added to the system. You can call GetRawInputDeviceInfo to get more information regarding the device.
GIDC_REMOVAL 2	A device has been removed from the system.

lParam

Type: **LPARAM**

The **HANDLE** to the raw input device.

Return value

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

See also

Conceptual

Raw Input

Reference

RegisterRawInputDevices

RAWINPUTDEVICE structure

Raw Input Structures

4/13/2022 • 2 minutes to read • Edit Online

In This Section

- RAWHID
- RAWINPUT
- RAWINPUTDEVICE
- RAWINPUTDEVICELIST
- RAWINPUTHEADER
- RAWKEYBOARD
- RAWMOUSE
- RID_DEVICE_INFO
- RID_DEVICE_INFO_HID
- RID_DEVICE_INFO_KEYBOARD
- RID_DEVICE_INFO_MOUSE