

Keyboard and Mouse Input

Article • 08/23/2019

The following sections describe methods of capturing user input.

In This Section

 Expand table

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

Article • 08/04/2022

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

In This Section

[+] Expand table

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

[+] Expand table

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.

Name	Description
GetKeyboardLayout	Retrieves the active input locale identifier (formerly called the keyboard layout) for the specified thread. If the <i>idThread</i> parameter is zero, the input locale identifier for the active thread is returned.
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.

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

[+] [Expand table](#)

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.

Function	Description
VkKeyScan	Translates a character to the corresponding virtual-key code and shift state for the current keyboard.

Messages

[+] [Expand table](#)

Name	Description
WM_GETHOTKEY	Determines the hot key associated with a window.
WM_SETHOTKEY	Associates a hot key with the window. When the user presses the hot key, the system activates the window.

Notifications

[+] [Expand table](#)

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 (ö) 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

Name	Description
	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.
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 <i>lParam</i> 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 <i>lParam</i> 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

[+] [Expand table](#)

Name	Description
HARDWAREINPUT	Contains information about a simulated message generated by an input

Name	Description
	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

[\[+\] Expand table](#)

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.

See also

- [About Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Keyboard Input Overview

Article • 09/18/2023

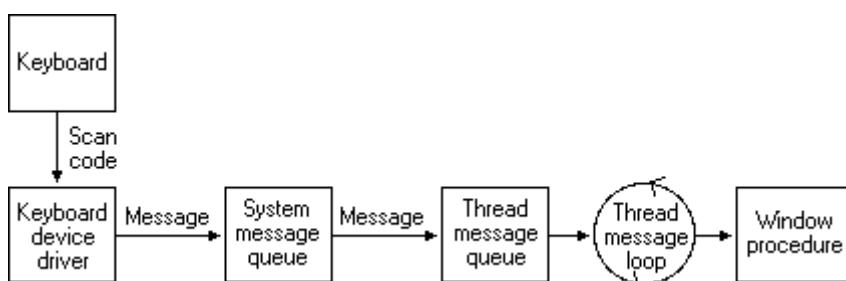
Applications should accept user input from the keyboard as well as from the mouse. An application receives keyboard input in the form of messages posted to its windows.

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* keystroke 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.

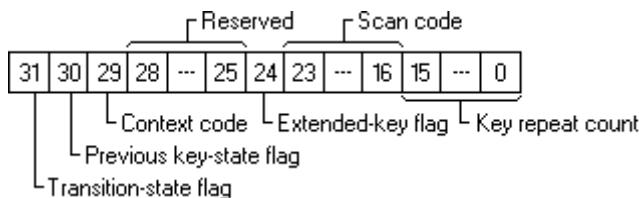
Virtual-Key Codes Described

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 get the keystroke flags from high-order word of the **lParam**.

[+] [Expand table](#)

Value	Description
KF_EXTENDED 0x0100	Manipulates the extended key flag .
KF_DLGMODE 0x0800	Manipulates the dialog mode flag, which indicates whether a dialog box is active.
KF_MENUMODE 0x1000	Manipulates the menu mode flag, which indicates whether a menu is active.
KF_ALTDOWN 0x2000	Manipulates the context code flag .

Value	Description
KF_REPEAT 0x4000	Manipulates the previous key state flag .
KF_UP 0x8000	Manipulates the transition state flag .

Example code:

C++

```

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

    WORD keyFlags = HIWORD(lParam);

    WORD scanCode = LOBYTE(keyFlags);                             // scan code
    BOOL isExtendedKey = (keyFlags & KF_EXTENDED) == KF_EXTENDED; // extended-key flag, 1 if scancode has 0xE0 prefix

    if (isExtendedKey)
        scanCode = MAKEWORD(scanCode, 0xE0);

    BOOL wasKeyDown = (keyFlags & 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 isKeyReleased = (keyFlags & KF_UP) == KF_UP;             // transition-state flag, 1 on keyup

    // if we want to distinguish these keys:
    switch (vkCode)
    {
        case VK_SHIFT:   // converts to VK_LSHIFT or VK_RSHIFT
        case VK_CONTROL: // converts to VK_LCONTROL or VK_RCONTROL
        case VK_MENU:     // converts to VK_LMENU or VK_RMENU
            vkCode = LOWORD(MapVirtualKeyW(scanCode, MAPVK_VSC_TO_VK_EX));
            break;
    }

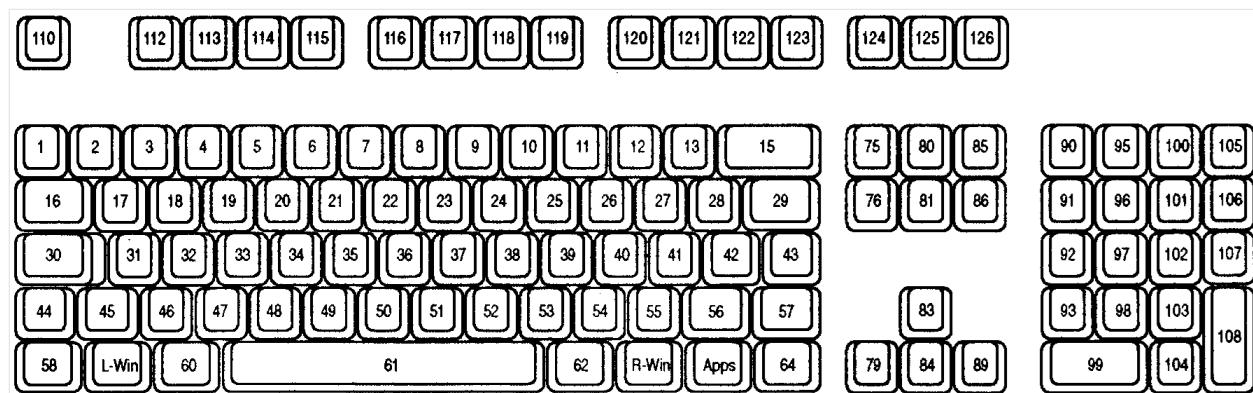
    // ...
}
break;

```

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 Codes



The scan code is the value that the system generates when the user presses a key. It is a value that identifies the key pressed regardless of the active [keyboard layout](#), as opposed to the character represented by the key. An application typically ignores scan codes. Instead, it uses the virtual-key codes to interpret keystroke messages.

Modern keyboards are using [Human Interface Devices \(HID\)](#) specification to communicate with a computer. [Keyboard driver](#) converts reported HID Usage values sent from the keyboard to scan codes and passes them on to applications.

ⓘ Note

While virtual key codes are typically more useful to desktop applications, scan codes might be required in specific cases when you need to know which key is pressed regardless of the current [keyboard layout](#). For example, the WASD (W is up, A is left, S is down, and D is right) key bindings for games, which ensure a consistent key formation across [US QWERTY](#) or [French AZERTY](#) keyboard layouts.

The following table lists the set of Scan Codes as presently recognized by Windows. HID Usage Page/HID Usage ID/HID Usage Name values reference the [HID Usage Tables](#) document. The Key Location values reference the preceding keyboard image.

The Scan 1 Make code is delivered in

[WM_KEYDOWN](#)/[WM_KEYUP](#)/[WM_SYSKEYDOWN](#)/[WM_SYSKEYUP](#) and [WM_INPUT](#) messages.

 Expand table

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
Generic Desktop	System Power Down	0x0001	0x0081	0xE05E	
Generic Desktop	System Sleep	0x0001	0x0082	0xE05F	
Generic Desktop	System Wake Up	0x0001	0x0083	0xE063	
Keyboard/Keypad	ErrorRollOver	0x0007	0x0001	0x00FF	
Keyboard/Keypad	Keyboard A	0x0007	0x0004	0x001E	31
Keyboard/Keypad	Keyboard B	0x0007	0x0005	0x0030	50
Keyboard/Keypad	Keyboard C	0x0007	0x0006	0x002E	48
Keyboard/Keypad	Keyboard D	0x0007	0x0007	0x0020	33
Keyboard/Keypad	Keyboard E	0x0007	0x0008	0x0012	19
Keyboard/Keypad	Keyboard F	0x0007	0x0009	0x0021	34
Keyboard/Keypad	Keyboard G	0x0007	0x000A	0x0022	35
Keyboard/Keypad	Keyboard H	0x0007	0x000B	0x0023	36
Keyboard/Keypad	Keyboard I	0x0007	0x000C	0x0017	24
Keyboard/Keypad	Keyboard J	0x0007	0x000D	0x0024	37
Keyboard/Keypad	Keyboard K	0x0007	0x000E	0x0025	38
Keyboard/Keypad	Keyboard L	0x0007	0x000F	0x0026	39
Keyboard/Keypad	Keyboard M	0x0007	0x0010	0x0032	52
Keyboard/Keypad	Keyboard N	0x0007	0x0011	0x0031	51
Keyboard/Keypad	Keyboard O	0x0007	0x0012	0x0018	25

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
Keyboard/Keypad	Keyboard P	0x0007	0x0013	0x0019	26
Keyboard/Keypad	Keyboard Q	0x0007	0x0014	0x0010	17
Keyboard/Keypad	Keyboard R	0x0007	0x0015	0x0013	20
Keyboard/Keypad	Keyboard S	0x0007	0x0016	0x001F	32
Keyboard/Keypad	Keyboard T	0x0007	0x0017	0x0014	21
Keyboard/Keypad	Keyboard U	0x0007	0x0018	0x0016	23
Keyboard/Keypad	Keyboard V	0x0007	0x0019	0x002F	49
Keyboard/Keypad	Keyboard W	0x0007	0x001A	0x0011	18
Keyboard/Keypad	Keyboard X	0x0007	0x001B	0x002D	47
Keyboard/Keypad	Keyboard Y	0x0007	0x001C	0x0015	22
Keyboard/Keypad	Keyboard Z	0x0007	0x001D	0x002C	46
Keyboard/Keypad	Keyboard 1 and Bang	0x0007	0x001E	0x0002	2
Keyboard/Keypad	Keyboard 2 and At	0x0007	0x001F	0x0003	3
Keyboard/Keypad	Keyboard 3 And Hash	0x0007	0x0020	0x0004	4
Keyboard/Keypad	Keyboard 4 and Dollar	0x0007	0x0021	0x0005	5
Keyboard/Keypad	Keyboard 5 and Percent	0x0007	0x0022	0x0006	6
Keyboard/Keypad	Keyboard 6 and Caret	0x0007	0x0023	0x0007	7
Keyboard/Keypad	Keyboard 7 and Ampersand	0x0007	0x0024	0x0008	8
Keyboard/Keypad	Keyboard 8 and Star	0x0007	0x0025	0x0009	9
Keyboard/Keypad	Keyboard 9 and Left Bracket	0x0007	0x0026	0x000A	10
Keyboard/Keypad	Keyboard 0 and Right Bracket	0x0007	0x0027	0x000B	11
Keyboard/Keypad	Keyboard Return Enter	0x0007	0x0028	0x001C	43
Keyboard/Keypad	Keyboard Escape	0x0007	0x0029	0x0001	110

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
Keyboard/Keypad	Keyboard Delete	0x0007	0x002A	0x000E	15
Keyboard/Keypad	Keyboard Tab	0x0007	0x002B	0x000F	16
Keyboard/Keypad	Keyboard Spacebar	0x0007	0x002C	0x0039	61
Keyboard/Keypad	Keyboard Dash and Underscore	0x0007	0x002D	0x000C	12
Keyboard/Keypad	Keyboard Equals and Plus	0x0007	0x002E	0x000D	13
Keyboard/Keypad	Keyboard Left Brace	0x0007	0x002F	0x001A	27
Keyboard/Keypad	Keyboard Right Brace	0x0007	0x0030	0x001B	28
Keyboard/Keypad	Keyboard Pipe and Slash	0x0007	0x0031	0x002B	29
Keyboard/Keypad	Keyboard Non-US	0x0007	0x0032	0x002B	42
Keyboard/Keypad	Keyboard SemiColon and Colon	0x0007	0x0033	0x0027	40
Keyboard/Keypad	Keyboard Apostrophe and Double Quotation Mark	0x0007	0x0034	0x0028	41
Keyboard/Keypad	Keyboard Grave Accent and Tilde	0x0007	0x0035	0x0029	1
Keyboard/Keypad	Keyboard Comma	0x0007	0x0036	0x0033	53
Keyboard/Keypad	Keyboard Period	0x0007	0x0037	0x0034	54
Keyboard/Keypad	Keyboard QuestionMark	0x0007	0x0038	0x0035	55
Keyboard/Keypad	Keyboard Caps Lock	0x0007	0x0039	0x003A	30
Keyboard/Keypad	Keyboard F1	0x0007	0x003A	0x003B	112
Keyboard/Keypad	Keyboard F2	0x0007	0x003B	0x003C	113
Keyboard/Keypad	Keyboard F3	0x0007	0x003C	0x003D	114
Keyboard/Keypad	Keyboard F4	0x0007	0x003D	0x003E	115
Keyboard/Keypad	Keyboard F5	0x0007	0x003E	0x003F	116

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
Keyboard/Keypad	Keyboard F6	0x0007	0x003F	0x0040	117
Keyboard/Keypad	Keyboard F7	0x0007	0x0040	0x0041	118
Keyboard/Keypad	Keyboard F8	0x0007	0x0041	0x0042	119
Keyboard/Keypad	Keyboard F9	0x0007	0x0042	0x0043	120
Keyboard/Keypad	Keyboard F10	0x0007	0x0043	0x0044	121
Keyboard/Keypad	Keyboard F11	0x0007	0x0044	0x0057	122
Keyboard/Keypad	Keyboard F12	0x0007	0x0045	0x0058	123
Keyboard/Keypad	Keyboard PrintScreen	0x0007	0x0046	0xE037 0x0054 *Note 1	124
Keyboard/Keypad	Keyboard Scroll Lock	0x0007	0x0047	0x0046	125
Keyboard/Keypad	Keyboard Pause	0x0007	0x0048	0xE11D45 0xE046 *Note 2 0x0045 *Note 3	126
Keyboard/Keypad	Keyboard Insert	0x0007	0x0049	0xE052	75
Keyboard/Keypad	Keyboard Home	0x0007	0x004A	0xE047	80
Keyboard/Keypad	Keyboard PageUp	0x0007	0x004B	0xE049	85
Keyboard/Keypad	Keyboard Delete Forward	0x0007	0x004C	0xE053	76
Keyboard/Keypad	Keyboard End	0x0007	0x004D	0xE04F	81
Keyboard/Keypad	Keyboard PageDown	0x0007	0x004E	0xE051	86
Keyboard/Keypad	Keyboard RightArrow	0x0007	0x004F	0xE04D	89
Keyboard/Keypad	Keyboard LeftArrow	0x0007	0x0050	0xE04B	79
Keyboard/Keypad	Keyboard DownArrow	0x0007	0x0051	0xE050	84
Keyboard/Keypad	Keyboard UpArrow	0x0007	0x0052	0xE048	83
Keyboard/Keypad	Keypad Num Lock and Clear	0x0007	0x0053	0x0045 0xE045	90

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
*Note 3					
Keyboard/Keypad	Keypad Forward Slash	0x0007	0x0054	0xE035	95
Keyboard/Keypad	Keypad Star	0x0007	0x0055	0x0037	100
Keyboard/Keypad	Keypad Dash	0x0007	0x0056	0x004A	105
Keyboard/Keypad	Keypad Plus	0x0007	0x0057	0x004E	106
Keyboard/Keypad	Keypad ENTER	0x0007	0x0058	0xE01C	108
Keyboard/Keypad	Keypad 1 and End	0x0007	0x0059	0x004F	93
Keyboard/Keypad	Keypad 2 and Down Arrow	0x0007	0x005A	0x0050	98
Keyboard/Keypad	Keypad 3 and PageDn	0x0007	0x005B	0x0051	103
Keyboard/Keypad	Keypad 4 and Left Arrow	0x0007	0x005C	0x004B	92
Keyboard/Keypad	Keypad 5	0x0007	0x005D	0x004C	97
Keyboard/Keypad	Keypad 6 and Right Arrow	0x0007	0x005E	0x004D	102
Keyboard/Keypad	Keypad 7 and Home	0x0007	0x005F	0x0047	91
Keyboard/Keypad	Keypad 8 and Up Arrow	0x0007	0x0060	0x0048	96
Keyboard/Keypad	Keypad 9 and PageUp	0x0007	0x0061	0x0049	101
Keyboard/Keypad	Keypad 0 and Insert	0x0007	0x0062	0x0052	99
Keyboard/Keypad	Keypad Period	0x0007	0x0063	0x0053	104
Keyboard/Keypad	Keyboard Non-US Slash Bar	0x0007	0x0064	0x0056	45
Keyboard/Keypad	Keyboard Application	0x0007	0x0065	0xE05D	129
Keyboard/Keypad	Keyboard Power	0x0007	0x0066	0xE05E	
Keyboard/Keypad	Keypad Equals	0x0007	0x0067	0x0059	
Keyboard/Keypad	Keyboard F13	0x0007	0x0068	0x0064	
Keyboard/Keypad	Keyboard F14	0x0007	0x0069	0x0065	

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
Keyboard/Keypad	Keyboard F15	0x0007	0x006A	0x0066	
Keyboard/Keypad	Keyboard F16	0x0007	0x006B	0x0067	
Keyboard/Keypad	Keyboard F17	0x0007	0x006C	0x0068	
Keyboard/Keypad	Keyboard F18	0x0007	0x006D	0x0069	
Keyboard/Keypad	Keyboard F19	0x0007	0x006E	0x006A	
Keyboard/Keypad	Keyboard F20	0x0007	0x006F	0x006B	
Keyboard/Keypad	Keyboard F21	0x0007	0x0070	0x006C	
Keyboard/Keypad	Keyboard F22	0x0007	0x0071	0x006D	
Keyboard/Keypad	Keyboard F23	0x0007	0x0072	0x006E	
Keyboard/Keypad	Keyboard F24	0x0007	0x0073	0x0076	
Keyboard/Keypad	Keypad Comma	0x0007	0x0085	0x007E	107 *Note 4
Keyboard/Keypad	Keyboard International1	0x0007	0x0087	0x0073	56 *Note 4, 5
Keyboard/Keypad	Keyboard International2	0x0007	0x0088	0x0070	133 *Note 5
Keyboard/Keypad	Keyboard International3	0x0007	0x0089	0x007D	14 *Note 5
Keyboard/Keypad	Keyboard International4	0x0007	0x008A	0x0079	132 *Note 5
Keyboard/Keypad	Keyboard International5	0x0007	0x008B	0x007B	131 *Note 5
Keyboard/Keypad	Keyboard International6	0x0007	0x008C	0x005C	
Keyboard/Keypad	Keyboard LANG1	0x0007	0x0090 *Note 6 0x00F2 *Note 3, 6	0x0072 *Note 6 0x00F2 *Note 3, 6	
Keyboard/Keypad	Keyboard LANG2	0x0007	0x0091 *Note 6	0x0071 *Note 6	

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
					0x00F1 *Note 3, 6
Keyboard/Keypad	Keyboard LANG3	0x0007	0x0092	0x0078	
Keyboard/Keypad	Keyboard LANG4	0x0007	0x0093	0x0077	
Keyboard/Keypad	Keyboard LANG5	0x0007	0x0094	0x0076	
Keyboard/Keypad	Keyboard LeftControl	0x0007	0x00E0	0x001D	58
Keyboard/Keypad	Keyboard LeftShift	0x0007	0x00E1	0x002A	44
Keyboard/Keypad	Keyboard LeftAlt	0x0007	0x00E2	0x0038	60
Keyboard/Keypad	Keyboard Left GUI	0x0007	0x00E3	0xE05B	127
Keyboard/Keypad	Keyboard RightControl	0x0007	0x00E4	0xE01D	64
Keyboard/Keypad	Keyboard RightShift	0x0007	0x00E5	0x0036	57
Keyboard/Keypad	Keyboard RightAlt	0x0007	0x00E6	0xE038	62
Keyboard/Keypad	Keyboard Right GUI	0x0007	0x00E7	0xE05C	128
Consumer	Scan Next Track	0x000C	0x00B5	0xE019	
Consumer	Scan Previous Track	0x000C	0x00B6	0xE010	
Consumer	Stop	0x000C	0x00B7	0xE024	
Consumer	Play/Pause	0x000C	0x00CD	0xE022	
Consumer	Mute	0x000C	0x00E2	0xE020	
Consumer	Volume Increment	0x000C	0x00E9	0xE030	
Consumer	Volume Decrement	0x000C	0x00EA	0xE02E	
Consumer	AL Consumer Control Configuration	0x000C	0x0183	0xE06D	
Consumer	AL Email Reader	0x000C	0x018A	0xE06C	
Consumer	AL Calculator	0x000C	0x0192	0xE021	
Consumer	AL Local Machine Browser	0x000C	0x0194	0xE06B	
Consumer	AC Search	0x000C	0x0221	0xE065	

HID Usage Page Name	HID Usage Name	HID Usage Page	HID Usage ID	Scan 1 Make	Key Location
Consumer	AC Home	0x000C	0x0223	0xE032	
Consumer	AC Back	0x000C	0x0224	0xE06A	
Consumer	AC Forward	0x000C	0x0225	0xE069	
Consumer	AC Stop	0x000C	0x0226	0xE068	
Consumer	AC Refresh	0x000C	0x0227	0xE067	
Consumer	AC Bookmarks	0x000C	0x022A	0xE066	

Notes:

1. *SysRq* key scan code is emitted on *Alt+Print screen* keystroke
2. *Break* key scan code is emitted on *Ctrl+Pause* keystroke
3. As seen in [legacy keyboard messages](#)
4. The key is present on Brazilian keyboards
5. The key is present on Japanese keyboards
6. The scan code is emitted in key release event only

Extended-Key Flag

The extended-key flag indicates whether the keystroke message originated from one of the additional keys on the Enhanced 101/102-key keyboard. The extended keys consist of the *Alt* and *Ctrl* keys on the right-hand side of the keyboard; the *Insert**, *Delete**, *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 screen* key; and the *Divide* (/) and *Enter* keys on the numeric keypad. The right-hand *Shift* key is not considered an extended-key, it has a separate scan code instead.

If specified, the scan code consists of a sequence of two bytes, where the first byte has a value of 0xE0.

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 UTF-16 (16-bit Unicode Transformation Format) or ANSI character set while [WM_UNICHAR](#) always uses UTF-32 (32-bit Unicode Transformation Format). 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 ANSI character codes. For more information, see [Registering Window Classes](#) and [Use UTF-8 code pages in Windows apps](#).

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.

Some keys are considered toggle keys that change the state of the keyboard layout. Toggle keys usually include *Caps Lock* (**VK_CAPITAL**), *Num Lock* (**VK_NUMLOCK**), and *Scroll Lock* (**VK_SCROLL**) keys. Most keyboards have corresponding LED indicators for these keys.

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 keyboard layout 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* keystroke, 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 [XBUTTONNs](#).

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_NOTESELLSHELL** 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_NOTESELLSHELL** 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 entering text using the same character set as used by the shell and file system.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Using Keyboard Input

Article • 11/29/2022

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.

C++

```
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;

    default:
        // Process other non-character keystrokes.
        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.

C++

```
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.

C++

```
case WM_CHAR:
    switch (wParam)
    {
        case 0x08: // or '\b'
            // Process a backspace.
            break;

        case 0x09: // or '\t'
            // Process a tab.
            break;

        case 0x0A: // or '\n'
            // Process a linefeed.
            break;

        case 0x0D:
            // Process a carriage return.
            break;

        case 0x1B:
            // Process an escape.
            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 [Caret](#).

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.

C++

```
#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 nCRIIndex = 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;
}

```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Keyboard Input Reference

Article • 04/27/2021

In This Section

- [Keyboard Input Functions](#)
- [Keyboard Input Messages](#)
- [Keyboard Input Notifications](#)
- [Keyboard Input Structures](#)
- [Keyboard Input Constants](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Keyboard Input Functions

Article • 04/27/2021

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

ActivateKeyboardLayout function (winuser.h)

Article 10/13/2021

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.

Syntax

C++

```
HKL ActivateKeyboardLayout(
    [in] HKL hkl,
    [in] UINT Flags
);
```

Parameters

[in] hkl

Type: **HKL**

Input locale identifier to be activated.

The input locale identifier must have been loaded by a previous call to the [LoadKeyboardLayout](#) function. This parameter must be either the handle to a keyboard layout or one of the following values.

[] [Expand table](#)

Value	Meaning
HKL_NEXT 1	Selects the next locale identifier in the circular list of loaded locale identifiers maintained by the system.
HKL_PREV 0	Selects the previous locale identifier in the circular list of loaded locale identifiers maintained by the system.

[in] Flags

Type: **UINT**

Specifies how the input locale identifier is to be activated. This parameter can be one of the following values.

[+] Expand table

Value	Meaning
KLF_REORDER 0x00000008	If this bit is set, the system's circular list of loaded locale identifiers is reordered by moving the locale identifier to the head of the list. If this bit is not set, the list is rotated without a change of order. For example, if a user had an English locale identifier active, as well as having French, German, and Spanish locale identifiers loaded (in that order), then activating the German locale identifier with the KLF_REORDER bit set would produce the following order: German, English, French, Spanish. Activating the German locale identifier without the KLF_REORDER bit set would produce the following order: German, Spanish, English, French. If less than three locale identifiers are loaded, the value of this flag is irrelevant.
KLF_RESET 0x40000000	If set but KLF_SHIFTLOCK is not set, the Caps Lock state is turned off by pressing the Caps Lock key again. If set and KLF_SHIFTLOCK is also set, the Caps Lock state is turned off by pressing either SHIFT key. These two methods are mutually exclusive, and the setting persists as part of the User's profile in the registry.
KLF_SETFORPROCESS 0x00000100	Activates the specified locale identifier for the entire process and sends the WM_INPUTLANGCHANGE message to the current thread's focus or active window.
KLF_SHIFTLOCK 0x00010000	This is used with KLF_RESET . See KLF_RESET for an explanation.
KLF_UNLOADPREVIOUS	This flag is unsupported. Use the UnloadKeyboardLayout function instead.

Return value

Type: **HKL**

The return value is of type **HKL**. If the function succeeds, the return value is the previous input locale identifier. Otherwise, it is zero.

To get extended error information, use the [GetLastError](#) function.

Remarks

This function only affects the layout for the current process or thread.

This function is not restricted to keyboard layouts. The *hkl* parameter is actually an input locale identifier. This is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input. Several input locale identifiers can be loaded at any one time, but only one is active at a time. Loading multiple input locale identifiers makes it possible to rapidly switch between them.

When multiple IMEs are allowed for each locale, passing an input locale identifier in which the high word (the device handle) is zero activates the first IME in the list belonging to the locale.

The **KLF_RESET** and **KLF_SHIFTLOCK** flags alter the method by which the Caps Lock state is turned off. By default, the Caps Lock state is turned off by hitting the Caps Lock key again. If only **KLF_RESET** is set, the default state is reestablished. If **KLF_RESET** and **KLF_SHIFTLOCK** are set, the Caps Lock state is turned off by pressing either Caps Lock key. This feature is used to conform to local keyboard behavior standards as well as for personal preferences.

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[GetKeyboardLayoutName](#)

[Keyboard Input](#)

[LoadKeyboardLayout](#)

Reference

[UnloadKeyboardLayout](#)

Feedback

Was this page helpful?

 Yes

 No

BlockInput function (winuser.h)

Article 02/22/2024

Blocks keyboard and mouse input events from reaching applications.

Syntax

C++

```
BOOL BlockInput(  
    [in] BOOL fBlockIt  
);
```

Parameters

[in] fBlockIt

Type: **BOOL**

The function's purpose. If this parameter is **TRUE**, keyboard and mouse input events are blocked. If this parameter is **FALSE**, keyboard and mouse events are unblocked. Note that only the thread that blocked input can successfully unblock input.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

When input is blocked, real physical input from the mouse or keyboard will not affect the input queue's synchronous key state (reported by [GetKeyState](#) and [GetKeyboardState](#)), nor will it affect the asynchronous key state (reported by [GetAsyncKeyState](#)). However, the thread that is blocking input can affect both of these key states by calling [SendInput](#). No other thread can do this.

The system will unblock input in the following cases:

- The thread that blocked input unexpectedly exits without calling `BlockInput` with `fBlock` set to `FALSE`. In this case, the system cleans up properly and re-enables input.
- The user presses **CTRL+ALT+DEL** or the system invokes the **Hard System Error** modal message box (for example, when a program faults or a device fails).

Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[GetAsyncKeyState](#)

[GetKeyState](#)

[GetKeyboardState](#)

[Keyboard Input](#)

Reference

[SendInput](#)

Feedback

Was this page helpful?

 Yes

 No

EnableWindow function (winuser.h)

Article 10/13/2021

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.

Syntax

C++

```
BOOL EnableWindow(
    [in] HWND hWnd,
    [in] BOOL bEnable
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be enabled or disabled.

[in] bEnable

Type: **BOOL**

Indicates whether to enable or disable the window. If this parameter is **TRUE**, the window is enabled. If the parameter is **FALSE**, the window is disabled.

Return value

Type: **BOOL**

If the window was previously disabled, the return value is nonzero.

If the window was not previously disabled, the return value is zero.

Remarks

If the window is being disabled, the system sends a [WM_CANCELMODE](#) message. If the enabled state of a window is changing, the system sends a [WM_ENABLE](#) message after the [WM_CANCELMODE](#) message. (These messages are sent before [EnableWindow](#) returns.) If a window is already disabled, its child windows are implicitly disabled, although they are not sent a [WM_ENABLE](#) message.

A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the application must enable the main window before destroying the dialog box. Otherwise, another window will receive the keyboard focus and be activated. If a child window is disabled, it is ignored when the system tries to determine which window should receive mouse messages.

By default, a window is enabled when it is created. To create a window that is initially disabled, an application can specify the [WS_DISABLED](#) style in the [CreateWindow](#) or [CreateWindowEx](#) function. After a window has been created, an application can use [EnableWindow](#) to enable or disable the window.

An application can use this function to enable or disable a control in a dialog box. A disabled control cannot receive the keyboard focus, nor can a user gain access to it.

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[CreateWindow](#)

[CreateWindowEx](#)

[IsWindowEnabled](#)

[Keyboard Input](#)

Reference

[WM_ENABLE](#)

Feedback

Was this page helpful?

 Yes

 No

GetActiveWindow function (winuser.h)

Article02/22/2024

Retrieves the window handle to the active window attached to the calling thread's message queue.

Syntax

C++

```
HWND GetActiveWindow();
```

Return value

Type: **HWND**

The return value is the handle to the active window attached to the calling thread's message queue. Otherwise, the return value is **NULL**.

Remarks

To get the handle to the foreground window, you can use [GetForegroundWindow](#).

To get the window handle to the active window in the message queue for another thread, use [GetGUIThreadInfo](#).

Requirements

[] Expand table

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

Requirement	Value
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetForegroundWindow](#)

[GetGUIThreadInfo](#)

[Keyboard Input](#)

Reference

[SetActiveWindow](#)

Feedback

Was this page helpful?

 Yes

 No

GetAsyncKeyState function (winuser.h)

Article 08/04/2022

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](#).

Syntax

C++

```
SHORT GetAsyncKeyState(  
    [in] int vKey  
);
```

Parameters

[in] vKey

Type: int

The virtual-key code. For more information, see [Virtual Key Codes](#).

You can use left- and right-distinguishing constants to specify certain keys. See the Remarks section for further information.

Return value

Type: SHORT

If the function succeeds, the return value specifies whether the key was pressed since the last call to [GetAsyncKeyState](#), and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after the previous call to [GetAsyncKeyState](#). However, you should not rely on this last behavior; for more information, see the Remarks.

The return value is zero for the following cases:

- The current desktop is not the active desktop
- The foreground thread belongs to another process and the desktop does not allow the hook or the journal record.

Remarks

The **GetAsyncKeyState** function works with mouse buttons. However, it checks on the state of the physical mouse buttons, not on the logical mouse buttons that the physical buttons are mapped to. For example, the call **GetAsyncKeyState(VK_LBUTTON)** always returns the state of the left physical mouse button, regardless of whether it is mapped to the left or right logical mouse button. You can determine the system's current mapping of physical mouse buttons to logical mouse buttons by calling

```
GetSystemMetrics(SM_SWAPBUTTON).
```

which returns TRUE if the mouse buttons have been swapped.

Although the least significant bit of the return value indicates whether the key has been pressed since the last query, due to the preemptive multitasking nature of Windows, another application can call **GetAsyncKeyState** and receive the "recently pressed" bit instead of your application. The behavior of the least significant bit of the return value is retained strictly for compatibility with 16-bit Windows applications (which are non-preemptive) and should not be relied upon.

You can use the virtual-key code constants **VK_SHIFT**, **VK_CONTROL**, and **VK_MENU** as values for the *vKey* parameter. This gives the state of the SHIFT, CTRL, or ALT keys without distinguishing between left and right.

You can use the following virtual-key code constants as values for *vKey* to distinguish between the left and right instances of those keys.

[+] Expand table

Code	Meaning
VK_LSHIFT	Left-shift key.
VK_RSHIFT	Right-shift key.
VK_LCONTROL	Left-control key.
VK_RCONTROL	Right-control key.
VK_LMENU	Left-menu key.
VK_RMENU	Right-menu key.

These left- and right-distinguishing constants are only available when you call the [GetKeyboardState](#), [SetKeyboardState](#), [GetAsyncKeyState](#), [GetKeyState](#), and

MapVirtualKey functions.

Example

C++

```
while (GetMessage(&msg, nullptr, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    switch (msg.message)
    {
        case WM_KEYDOWN:
            if ((.GetAsyncKeyState(VK_ESCAPE) & 0x01) && bRunning)
            {
                Stop();
            }
            break;
    }
}
```

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

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
 - [GetKeyState](#)
 - [GetKeyboardState](#)
 - [GetSystemMetrics](#)
 - [MapVirtualKey](#)
 - [SetKeyboardState](#)
 - [Keyboard Input](#)
-

Feedback

Was this page helpful?

 Yes

 No

GetFocus function (winuser.h)

Article02/22/2024

Retrieves the handle to the window that has the keyboard focus, if the window is attached to the calling thread's message queue.

Syntax

C++

```
HWND GetFocus();
```

Return value

Type: **HWND**

The return value is the handle to the window with the keyboard focus. If the calling thread's message queue does not have an associated window with the keyboard focus, the return value is **NULL**.

Remarks

GetFocus returns the window with the keyboard focus for the current thread's message queue. If **GetFocus** returns **NULL**, another thread's queue may be attached to a window that has the keyboard focus.

Use the [GetForegroundWindow](#) function to retrieve the handle to the window with which the user is currently working. You can associate your thread's message queue with the windows owned by another thread by using the [AttachThreadInput](#) function.

To get the window with the keyboard focus on the foreground queue or the queue of another thread, use the [GetGUIThreadInfo](#) function.

Examples

For an example, see "Creating a Combo Box Toolbar" in [Using Combo Boxes](#).

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[AttachThreadInput](#)

[Conceptual](#)

[GetForegroundWindow](#)

[GetGUIThreadInfo](#)

[Keyboard Input](#)

[Other Resources](#)

[Reference](#)

[SetFocus](#)

[WM_KILLFOCUS](#)

[WM_SETFOCUS](#)

Feedback

Was this page helpful?

 Yes

 No

GetKBCodePage function (winuser.h)

Article02/22/2024

Retrieves the current code page.

Note This function is provided only for compatibility with 16-bit versions of Windows. Applications should use the [GetOEMCP](#) function to retrieve the OEM code-page identifier for the system.

Syntax

C++

```
UINT GetKBCodePage();
```

Return value

Type: [UINT](#)

The return value is an OEM code-page identifier, or it is the default identifier if the registry value is not readable. For a list of OEM code-page identifiers, see [Code Page Identifiers](#).

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib

Requirement	Value
DLL	User32.dll

See also

Conceptual

[GetACP](#)

[GetOEMCP](#)

[Keyboard Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

GetKeyboardLayout function (winuser.h)

Article 02/22/2024

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

Syntax

C++

```
HKL GetKeyboardLayout(
    [in] DWORD idThread
);
```

Parameters

[in] idThread

Type: **DWORD**

The identifier of the thread to query, or 0 for the current thread.

Return value

Type: **HKL**

The return value is the input locale identifier for the thread. The low word contains a [Language Identifier](#) for the input language and the high word contains a device handle to the physical layout of the keyboard.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

Since the keyboard layout can be dynamically changed, applications that cache information about the current keyboard layout should process the [WM_INPUTLANGCHANGE](#) message to be informed of changes in the input language.

To get the KLID (keyboard layout ID) of the currently active HKL, call the [GetKeyboardLayoutName](#).

Beginning in Windows 8: The preferred method to retrieve the language associated with the current keyboard layout or input method is a call to [Windows.Globalization.Language.CurrentInputMethodLanguageTag](#). If your app passes language tags from [CurrentInputMethodLanguageTag](#) to any [National Language Support](#) functions, it must first convert the tags by calling [ResolveLocaleName](#).

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[ActivateKeyboardLayout](#)

[Conceptual](#)

[CreateThread](#)

[Keyboard Input](#)

[LoadKeyboardLayout](#)

[Other Resources](#)

[Reference](#)

[WM_INPUTLANGCHANGE](#)

Feedback

Was this page helpful?

 Yes

 No

GetKeyboardLayoutList function (winuser.h)

Article 02/22/2024

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.

Syntax

C++

```
int GetKeyboardLayoutList(
    [in] int nBuff,
    [out] HKL *lpList
);
```

Parameters

[in] nBuff

Type: **int**

The maximum number of handles that the buffer can hold.

[out] lpList

Type: **HKL***

A pointer to the buffer that receives the array of input locale identifiers.

Return value

Type: **int**

If the function succeeds, the return value is the number of input locale identifiers copied to the buffer or, if *nBuff* is zero, the return value is the size, in array elements, of the buffer needed to receive all current input locale identifiers.

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

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

Beginning in Windows 8: The preferred method to retrieve the language associated with the current keyboard layout or input method is a call to [Windows.Globalization.Language.CurrentInputMethodLanguageTag](#). If your app passes language tags from [CurrentInputMethodLanguageTag](#) to any [National Language Support](#) functions, it must first convert the tags by calling [ResolveLocaleName](#).

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[GetKeyboardLayout](#)

[Keyboard Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

GetKeyboardLayoutNameA function (winuser.h)

Article 02/22/2024

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

Syntax

C++

```
BOOL GetKeyboardLayoutNameA(  
    [out] LPSTR pwszKLID  
) ;
```

Parameters

[out] pwszKLID

Type: **LPTSTR**

The buffer (of at least **KL_NAMELENGTH** characters in length) that receives the name of the input locale identifier, including the terminating null character. This will be a copy of the string provided to the [LoadKeyboardLayout](#) function, unless layout substitution took place.

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

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

Beginning in Windows 8: The preferred method to retrieve the language associated with the current keyboard layout or input method is a call to [Windows.Globalization.Language.CurrentInputMethodLanguageTag](#). If your app passes language tags from [CurrentInputMethodLanguageTag](#) to any [National Language Support](#) functions, it must first convert the tags by calling [ResolveLocaleName](#).

 **Note**

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

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[ActivateKeyboardLayout](#)

[Conceptual](#)

Keyboard Input

[LoadKeyboardLayout](#)

Reference

[UnloadKeyboardLayout](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GetKeyboardState function (winuser.h)

Article02/22/2024

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

Syntax

C++

```
BOOL GetKeyboardState(  
    [out] PBYTE lpKeyState  
) ;
```

Parameters

[out] lpKeyState

Type: **PBYTE**

The 256-byte array that receives the status data for each virtual key.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

An application can call this function to retrieve the current status of all the virtual keys. The status changes as a thread removes keyboard messages from its message queue. The status does not change as keyboard messages are posted to the thread's message queue, nor does it change as keyboard messages are posted to or retrieved from message queues of other threads. (Exception: Threads that are connected through [AttachThreadInput](#) share the same keyboard state.)

When the function returns, each member of the array pointed to by the *lpKeyState* parameter contains status data for a virtual key. If the high-order bit is 1, the key is down; otherwise, it is up. If the key is a toggle key, for example CAPS LOCK, then the low-order bit is 1 when the key is toggled and is 0 if the key is untoggled. The low-order bit is meaningless for non-toggle keys. A toggle key is said to be toggled when it is turned on. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

To retrieve status information for an individual key, use the [GetKeyState](#) function. To retrieve the current state for an individual key regardless of whether the corresponding keyboard message has been retrieved from the message queue, use the [GetAsyncKeyState](#) function.

An application can use the virtual-key code constants **VK_SHIFT**, **VK_CONTROL** and **VK_MENU** as indices into the array pointed to by *lpKeyState*. This gives the status of the SHIFT, CTRL, or ALT keys without distinguishing between left and right. An application can also use the following virtual-key code constants as indices to distinguish between the left and right instances of those keys:

[+] Expand table

VK_LSHIFT
VK_RSHIFT
VK_LCONTROL
VK_RCONTROL
VK_LMENU
VK_RMENU

These left- and right-distinguishing constants are available to an application only through the [GetKeyboardState](#), [SetKeyboardState](#), [GetAsyncKeyState](#), [GetKeyState](#), and [MapVirtualKey](#) functions.

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-rawinput-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

- [GetAsyncKeyState](#)
- [GetKeyState](#)
- [GetKeyboardState](#)
- [GetSystemMetrics](#)
- [MapVirtualKey](#)
- [SetKeyboardState](#)
- [Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

GetKeyboardType function (winuser.h)

Article02/22/2024

Retrieves information about the current keyboard.

Syntax

C++

```
int GetKeyboardType(
    [in] int nTypeFlag
);
```

Parameters

[in] *nTypeFlag*

Type: **int**

The type of keyboard information to be retrieved. This parameter can be one of the following values.

[+] Expand table

Value	Meaning
0	Keyboard type
1	Keyboard subtype
2	The number of function keys on the keyboard

Return value

Type: **int**

If the function succeeds, the return value specifies the requested information.

If the function fails and *nTypeFlag* is not 1, the return value is 0; 0 is a valid return value when *nTypeFlag* is 1 (keyboard subtype). To get extended error information, call [GetLastError](#).

Remarks

Valid keyboard types are:

[+] Expand table

Value	Description
0x4	Enhanced 101- or 102-key keyboards (and compatibles)
0x7	Japanese Keyboard
0x8	Korean Keyboard
0x51	Unknown type or HID keyboard

Keyboard subtypes are original equipment manufacturer (OEM)-dependent values.

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Keyboard Input Functions](#)

Feedback

Was this page helpful?

 Yes

 No

GetKeyNameTextA function (winuser.h)

Article 06/26/2024

Retrieves a string that represents the name of a key.

Syntax

C++

```
int GetKeyNameTextA(
    [in] LONG lParam,
    [out] LPSTR lpString,
    [in] int cchSize
);
```

Parameters

[in] *lParam*

Type: **LONG**

The second parameter of the keyboard message (such as [WM_KEYDOWN](#)) to be processed. The function interprets the following bit positions in the *lParam*.

[\[+\] Expand table](#)

Bits	Meaning
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	"Do not care" bit. The application calling this function sets this bit to indicate that the function should not distinguish between left and right CTRL and SHIFT keys, for example.

For more detail, see [Keystroke Message Flags](#).

[out] *lpString*

Type: **LPTSTR**

The buffer that will receive the key name.

[in] cchSize

Type: int

The maximum length, in characters, of the key name, including the terminating null character. (This parameter should be equal to the size of the buffer pointed to by the *lpString* parameter.)

Return value

Type: int

If the function succeeds, a null-terminated string is copied into the specified buffer, and the return value is the length of the string, in characters, not counting the terminating null character.

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

Remarks

The format of the key-name string depends on the current keyboard layout.

The keyboard layout maintains a list of names in the form of character strings for keys with names longer than a single character. The key name is translated according to the [currently active keyboard layout](#), therefore the function might return different results for different [keyboard layouts](#).

The name of a character key is the character itself. The names of dead keys are spelled out in full.

Character keys that are mapped to the 'A'..'Z' [virtual-key codes](#) are translated to the <U+0041 LATIN CAPITAL LETTER A>..<U+005A LATIN CAPITAL LETTER Z> characters regardless of current keyboard layout. In this case, use the [ToUnicode](#) or [ToUnicodeEx](#) methods to get the character for the corresponding key press.

This method might not work properly with some [keyboard layouts](#) that produce multiple characters (such as ligatures) or supplementary Unicode characters on a single key press.

The winuser.h header defines GetKeyNameText as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE

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

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Keyboard Input](#)

[Keyboard Layouts](#)

[Keyboard Layout Samples](#)

[ToUnicode](#)

[ToUnicodeEx](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GetKeyState function (winuser.h)

Article 08/04/2022

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).

Syntax

C++

```
SHORT GetKeyState(  
    [in] int nVirtKey  
);
```

Parameters

[in] nVirtKey

Type: int

A virtual key. If the desired virtual key is a letter or digit (A through Z, a through z, or 0 through 9), *nVirtKey* must be set to the ASCII value of that character. For other keys, it must be a virtual-key code.

If a non-English keyboard layout is used, virtual keys with values in the range ASCII A through Z and 0 through 9 are used to specify most of the character keys. For example, for the German keyboard layout, the virtual key of value ASCII O (0x4F) refers to the "o" key, whereas VK_OEM_1 refers to the "o with umlaut" key.

Return value

Type: SHORT

The return value specifies the status of the specified virtual key, as follows:

- If the high-order bit is 1, the key is down; otherwise, it is up.
- If the low-order bit is 1, the key is toggled. A key, such as the CAPS LOCK key, is toggled if it is turned on. The key is off and untoggled if the low-order bit is 0. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

Remarks

The key status returned from this function changes as a thread reads key messages from its message queue. The status does not reflect the interrupt-level state associated with the hardware. Use the [GetAsyncKeyState](#) function to retrieve that information.

An application calls [GetKeyState](#) in response to a keyboard-input message. This function retrieves the state of the key when the input message was generated.

To retrieve state information for all the virtual keys, use the [GetKeyboardState](#) function.

An application can use the [virtual key code](#) constants **VK_SHIFT**, **VK_CONTROL**, and **VK_MENU** as values for the *nVirtKey* parameter. This gives the status of the SHIFT, CTRL, or ALT keys without distinguishing between left and right. An application can also use the following virtual-key code constants as values for *nVirtKey* to distinguish between the left and right instances of those keys:

VK_LSHIFT **VK_RSHIFT** **VK_LCONTROL** **VK_RCONTROL** **VK_LMENU** **VK_RMENU** These left- and right-distinguishing constants are available to an application only through the [GetKeyboardState](#), [SetKeyboardState](#), [GetAsyncKeyState](#), [GetKeyState](#), and [MapVirtualKey](#) functions.

Examples

For an example, see [Displaying Keyboard Input](#).

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
 - [GetKeyState](#)
 - [GetKeyboardState](#)
 - [MapVirtualKey](#)
 - [SetKeyboardState](#)
 - [Keyboard Input](#)
-

Feedback

Was this page helpful?

 Yes

 No

GetLastInputInfo function (winuser.h)

Article 02/22/2024

Retrieves the time of the last input event.

Syntax

C++

```
BOOL GetLastInputInfo(  
    [out] PLASTINPUTINFO plii  
)
```

Parameters

[out] plii

Type: [PLASTINPUTINFO](#)

A pointer to a [LASTINPUTINFO](#) structure that receives the time of the last input event.

Return value

Type: [BOOL](#)

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

This function is useful for input idle detection. However, [GetLastInputInfo](#) does not provide system-wide user input information across all running sessions. Rather, [GetLastInputInfo](#) provides session-specific user input information for only the session that invoked the function.

The tick count when the last input event was received (see [LASTINPUTINFO](#)) is not guaranteed to be incremental. In some cases, the value might be less than the tick count of a prior event. For example, this can be caused by a timing gap between the raw input

thread and the desktop thread or an event raised by [SendInput](#), which supplies its own tick count.

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[Keyboard Input](#)

[LASTINPUTINFO](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

IsWindowEnabled function (winuser.h)

Article 02/22/2024

Determines whether the specified window is enabled for mouse and keyboard input.

Syntax

C++

```
BOOL IsWindowEnabled(  
    [in] HWND hWnd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window to be tested.

Return value

Type: **BOOL**

If the window is enabled, the return value is nonzero.

If the window is not enabled, the return value is zero.

Remarks

A child window receives input only if it is both enabled and visible.

Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[EnableWindow](#)

[IsWindowVisible](#)

[Keyboard Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

keybd_event function (winuser.h)

Article02/22/2024

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.

Note This function has been superseded. Use [SendInput](#) instead.

Syntax

C++

```
void keybd_event(
    [in] BYTE      bVk,
    [in] BYTE      bScan,
    [in] DWORD     dwFlags,
    [in] ULONG_PTR dwExtraInfo
);
```

Parameters

[in] bVk

Type: **BYTE**

A virtual-key code. The code must be a value in the range 1 to 254. For a complete list, see [Virtual Key Codes](#).

[in] bScan

Type: **BYTE**

A hardware scan code for the key.

[in] dwFlags

Type: **DWORD**

Controls various aspects of function operation. This parameter can be one or more of the following values.

Value	Meaning
KEYEVENTF_EXTENDEDKEY 0x0001	If specified, the scan code was preceded by a prefix byte having the value 0xE0 (224).
KEYEVENTF_KEYUP 0x0002	If specified, the key is being released. If not specified, the key is being depressed.

[in] dwExtraInfo

Type: **ULONG_PTR**

An additional value associated with the key stroke.

Return value

None

Remarks

An application can simulate a press of the PRINTSCRN key in order to obtain a screen snapshot and save it to the clipboard. To do this, call **keybd_event** with the *bVk* parameter set to **VK_SNAPSHOT**.

Examples

The following sample program toggles the NUM LOCK light by using **keybd_event** with a virtual key of **VK_NUMLOCK**. It takes a Boolean value that indicates whether the light should be turned off (**FALSE**) or on (**TRUE**). The same technique can be used for the CAPS LOCK key (**VK_CAPITAL**) and the SCROLL LOCK key (**VK_SCROLL**).

```
#include <windows.h>

void SetNumLock( BOOL bState )
{
    BYTE keyState[256];

    GetKeyboardState((LPBYTE)&keyState);
    if( (bState && !(keyState[VK_NUMLOCK] & 1)) ||
        (!bState && (keyState[VK_NUMLOCK] & 1)) )

```

```

{
    // Simulate a key press
    keybd_event( VK_NUMLOCK,
                 0x45,
                 KEYEVENTF_EXTENDEDKEY | 0,
                 0 );

    // Simulate a key release
    keybd_event( VK_NUMLOCK,
                 0x45,
                 KEYEVENTF_EXTENDEDKEY | KEYEVENTF_KEYUP,
                 0 );
}

void main()
{
    SetNumLock( TRUE );
}

```

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
- [GetKeyState](#)
- [GetKeyboardState](#)
- [GetSystemMetrics](#)
- [MapVirtualKey](#)
- [SetKeyboardState](#)

- Keyboard Input
-

Feedback

Was this page helpful?

 Yes

 No

LoadKeyboardLayoutA function (winuser.h)

Article03/11/2023

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

Prior to Windows 8: 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.

Beginning in Windows 8: The input locale identifier is loaded for the entire system. This function has no effect if the current process does not own the window with keyboard focus.

Syntax

C++

```
HKL LoadKeyboardLayoutA(
    [in] LPCSTR pwszKLID,
    [in] UINT   Flags
);
```

Parameters

[in] pwszKLID

Type: [LPCTSTR](#)

The name of the input locale identifier to load. This name is a string composed of the hexadecimal value of the [Language Identifier](#) (low word) and a device identifier (high word). For example, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named "00000409". Variants of U.S. English layout (such as the Dvorak layout) are named "00010409", "00020409", and so on.

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

[in] Flags

Type: [UINT](#)

Specifies how the input locale identifier is to be loaded. This parameter can be one or more of the following values.

[+] Expand table

Value	Meaning
KLF_ACTIVATE 0x00000001	<p>Prior to Windows 8: If the specified input locale identifier is not already loaded, the function loads and activates the input locale identifier for the current thread.</p> <p>Beginning in Windows 8: If the specified input locale identifier is not already loaded, the function loads and activates the input locale identifier for the system.</p>
KLF_NOTESELLHELL 0x00000080	<p>Prior to Windows 8: Prevents a ShellProc hook procedure from receiving an HSHELL_LANGUAGE hook code when the new input locale identifier is loaded. This value is typically used when an application loads multiple input locale identifiers one after another. Applying this value to all but the last input locale identifier delays the shell's processing until all input locale identifiers have been added.</p> <p>Beginning in Windows 8: In this scenario, the last input locale identifier is set for the entire system.</p>
KLF_REORDER 0x00000008	<p>Prior to Windows 8: Moves the specified input locale identifier to the head of the input locale identifier list, making that locale identifier the active locale identifier for the current thread. This value reorders the input locale identifier list even if KLF_ACTIVATE is not provided.</p> <p>Beginning in Windows 8: Moves the specified input locale identifier to the head of the input locale identifier list, making that locale identifier the active locale identifier for the system. This value reorders the input locale identifier list even if KLF_ACTIVATE is not provided.</p>
KLF_REPLACELANG 0x00000010	<p>If the new input locale identifier has the same language identifier as a current input locale identifier, the new input locale identifier replaces the current one as the input locale identifier for that language. If this value is not provided and the input locale identifiers have the same language identifiers, the current input locale identifier is not replaced and the function returns NULL.</p>
KLF_SUBSTITUTE_OK 0x00000002	<p>Substitutes the specified input locale identifier with another locale preferred by the user. The system starts with this flag set, and it is recommended that your</p>

application always use this flag. The substitution occurs only if the registry key **HKEY_CURRENT_USER\Keyboard Layout\Substitutes** explicitly defines a substitution locale. For example, if the key includes the value name "00000409" with value "00010409", loading the US layout ("00000409") causes the United States-Dvorak layout ("00010409") to be loaded instead. The system uses **KLF_SUBSTITUTE_OK** when booting, and it is recommended that all applications use this value when loading input locale identifiers to ensure that the user's preference is selected.

KLF_SETFORPROCESS 0x00000100	<p>Prior to Windows 8: This flag is valid only with KLF_ACTIVATE. Activates the specified input locale identifier for the entire process and sends the WM_INPUTLANGCHANGE message to the current thread's Focus or Active window. Typically, LoadKeyboardLayout activates an input locale identifier only for the current thread.</p> <p>Beginning in Windows 8: This flag is not used. LoadKeyboardLayout always activates an input locale identifier for the entire system if the current process owns the window with keyboard focus.</p>
KLF_UNLOADPREVIOUS	This flag is unsupported. Use the UnloadKeyboardLayout function instead.

Return value

Type: **HKL**

If the function succeeds, the return value is the input locale identifier corresponding to the name specified in *pwszKLID*. If no matching locale is available, the return value is the default language of the system.

If the function fails, the return value is NULL. This can occur if the layout library is loaded from the application directory.

To get extended error information, call [GetLastError](#).

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

An application can and will typically load the default input locale identifier or IME for a language and can do so by specifying only a string version of the language identifier. If an application wants to load a specific locale or IME, it should read the registry to determine the specific input locale identifier to pass to **LoadKeyboardLayout**. In this case, a request to activate the default input locale identifier for a locale will activate the first matching one. A specific IME should be activated using an explicit input locale identifier returned from [GetKeyboardLayout](#) or [LoadKeyboardLayout](#).

Prior to Windows 8: This function only affects the layout for the current process or thread.

Beginning in Windows 8: This function affects the layout for the entire system.

 **Note**

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

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[ActivateKeyboardLayout](#)

Conceptual

[GetKeyboardLayoutName](#)

[Keyboard Input](#)

[MAKELANGID](#)

Other Resources

Reference

[UnloadKeyboardLayout](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

MapVirtualKeyA function (winuser.h)

Article 02/09/2023

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

Syntax

C++

```
UINT MapVirtualKeyA(
    [in] UINT uCode,
    [in] UINT uMapType
);
```

Parameters

[in] *uCode*

Type: **UINT**

The [virtual key code](#) or scan code for a key. How this value is interpreted depends on the value of the *uMapType* parameter.

[in] *uMapType*

Type: **UINT**

The translation to be performed. The value of this parameter depends on the value of the *uCode* parameter.

[+] Expand table

Value	Meaning
MAPVK_VK_TO_VSC 0	The <i>uCode</i> parameter is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left- and right-hand keys, the left-hand scan code is returned. If there is no translation, the function returns 0.
MAPVK_VSC_TO_VK 1	The <i>uCode</i> parameter is a scan code and is translated into a virtual-key code that does not distinguish between left- and right-hand keys. If there is no translation, the function returns 0.

Value	Meaning
	Windows Vista and later: the high byte of the <i>uCode</i> value can contain either 0xe0 or 0xe1 to specify the extended scan code.
MAPVK_VK_TO_CHAR 2	The <i>uCode</i> parameter is a virtual-key code and is translated into an unshifted character value in the low order word of the return value. Dead keys (diacritics) are indicated by setting the top bit of the return value. If there is no translation, the function returns 0. See Remarks.
MAPVK_VSC_TO_VK_EX 3	<p>The <i>uCode</i> parameter is a scan code and is translated into a virtual-key code that distinguishes between left- and right-hand keys. If there is no translation, the function returns 0.</p> <p>Windows Vista and later: the high byte of the <i>uCode</i> value can contain either 0xe0 or 0xe1 to specify the extended scan code.</p>
MAPVK_VK_TO_VSC_EX 4	Windows Vista and later: The <i>uCode</i> parameter is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left- and right-hand keys, the left-hand scan code is returned. If the scan code is an extended scan code, the high byte of the returned value will contain either 0xe0 or 0xe1 to specify the extended scan code. If there is no translation, the function returns 0.

Return value

Type: **UINT**

The return value is either a scan code, a virtual-key code, or a character value, depending on the value of *uCode* and *uMapType*. If there is no translation, the return value is zero.

Remarks

To specify a handle to the keyboard layout to use for translating the specified code, use the [MapVirtualKeyEx](#) function.

An application can use **MapVirtualKey** to translate scan codes to the virtual-key code constants **VK_SHIFT**, **VK_CONTROL**, and **VK_MENU**, and vice versa. These translations do not distinguish between the left and right instances of the SHIFT, CTRL, or ALT keys.

An application can get the scan code corresponding to the left or right instance of one of these keys by calling **MapVirtualKey** with *uCode* set to one of the following virtual-key code constants:

- **VK_LSHIFT**

- VK_RSHIFT
- VK_LCONTROL
- VK_RCONTROL
- VK_LMENU
- VK_RMENU

These left- and right-distinguishing constants are available to an application only through the [GetKeyboardState](#), [SetKeyboardState](#), [GetAsyncKeyState](#), [GetKeyState](#), [MapVirtualKey](#), and [MapVirtualKeyEx](#) functions. For list complete table of virtual key codes, see [Virtual Key Codes](#).

In **MAPVK_VK_TO_CHAR** mode [virtual-key codes](#), the 'A'..'Z' keys are translated to upper-case 'A'..'Z' characters regardless of current keyboard layout. If you want to translate a virtual-key code to the corresponding character, use the [ToAscii](#) function.

 **Note**

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

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
 - [GetKeyState](#)
 - [GetKeyboardState](#)
 - [GetSystemMetrics](#)
 - [MapVirtualKey](#)
 - [SetKeyboardState](#)
 - [Keyboard Input](#)
 - [Keyboard Input Overview](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

MapVirtualKeyExA function (winuser.h)

Article02/21/2024

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

Syntax

C++

```
UINT MapVirtualKeyExA(
    [in]             UINT uCode,
    [in]             UINT uMapType,
    [in, out, optional] HKL dwhkl
);
```

Parameters

[in] `uCode`

Type: `UINT`

The [virtual key code](#) or scan code for a key. How this value is interpreted depends on the value of the `uMapType` parameter.

[in] `uMapType`

Type: `UINT`

The translation to perform. The value of this parameter depends on the value of the `uCode` parameter.

[+] Expand table

Value	Meaning
<code>MAPVK_VK_TO_VSC</code> 0	The <code>uCode</code> parameter is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left- and right-hand keys, the left-hand scan code is returned. If there is no translation, the function returns 0.

Value	Meaning
MAPVK_VSC_TO_VK 1	The <i>uCode</i> parameter is a scan code and is translated into a virtual-key code that does not distinguish between left- and right-hand keys. If there is no translation, the function returns 0.
	Windows Vista and later: the high byte of the <i>uCode</i> value can contain either 0xe0 or 0xe1 to specify the extended scan code.
MAPVK_VK_TO_CHAR 2	The <i>uCode</i> parameter is a virtual-key code and is translated into an unshifted character value in the low order word of the return value. Dead keys (diacritics) are indicated by setting the top bit of the return value. If there is no translation, the function returns 0. See Remarks.
MAPVK_VSC_TO_VK_EX 3	The <i>uCode</i> parameter is a scan code and is translated into a virtual-key code that distinguishes between left- and right-hand keys. If there is no translation, the function returns 0.
	Windows Vista and later: the high byte of the <i>uCode</i> value can contain either 0xe0 or 0xe1 to specify the extended scan code.
MAPVK_VK_TO_VSC_EX 4	Windows Vista and later: The <i>uCode</i> parameter is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left- and right-hand keys, the left-hand scan code is returned. If the scan code is an extended scan code, the high byte of the returned value will contain either 0xe0 or 0xe1 to specify the extended scan code. If there is no translation, the function returns 0.

[in, out, optional] dwhkl

Type: HKL

Input locale identifier to use for translating the specified code. This parameter can be any input locale identifier previously returned by the [LoadKeyboardLayout](#) function.

Return value

Type: UINT

The return value is either a scan code, a virtual-key code, or a character value, depending on the value of *uCode* and *uMapType*. If there is no translation, the return value is zero.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form

of input.

An application can use **MapVirtualKeyEx** to translate scan codes to the virtual-key code constants **VK_SHIFT**, **VK_CONTROL**, and **VK_MENU**, and vice versa. These translations do not distinguish between the left and right instances of the SHIFT, CTRL, or ALT keys.

An application can get the scan code corresponding to the left or right instance of one of these keys by calling **MapVirtualKeyEx** with *uCode* set to one of the following virtual-key code constants:

- **VK_LSHIFT**
- **VK_RSHIFT**
- **VK_LCONTROL**
- **VK_RCONTROL**
- **VK_LMENU**
- **VK_RMENU**

These left- and right-distinguishing constants are available to an application only through the [GetKeyboardState](#), [SetKeyboardState](#), [GetAsyncKeyState](#), [GetKeyState](#), [MapVirtualKey](#), and [MapVirtualKeyEx](#) functions. For list complete table of virtual key codes, see [Virtual Key Codes](#).

In **MAPVK_VK_TO_CHAR** mode [virtual-key codes](#), the 'A'..'Z' keys are translated to upper-case 'A'..'Z' characters regardless of current keyboard layout. If you want to translate a virtual-key code to the corresponding character, use the [ToAscii](#) function.

Note

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

Requirements

 Expand table

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

Requirement	Value
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
- [GetKeyState](#)
- [GetKeyboardState](#)
- [GetSystemMetrics](#)
- [MapVirtualKey](#)
- [SetKeyboardState](#)
- [LoadKeyboardLayout](#)
- [Keyboard Input](#)
- [Keyboard Input Overview](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

OemKeyScan function (winuser.h)

Article02/22/2024

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.

Syntax

C++

```
DWORD OemKeyScan(
    [in] WORD wOemChar
);
```

Parameters

[in] wOemChar

Type: **WORD**

The ASCII value of the OEM character.

Return value

Type: **DWORD**

The low-order word of the return value contains the scan code of the OEM character, and the high-order word contains the shift state, which can be a combination of the following bits.

[] Expand table

Bit	Description
1	Either SHIFT key is pressed.
2	Either CTRL key is pressed.
4	Either ALT key is pressed.
8	The Hankaku key is pressed.

16	Reserved (defined by the keyboard layout driver).
32	Reserved (defined by the keyboard layout driver).

If the character cannot be produced by a single keystroke using the current keyboard layout, the return value is –1.

Remarks

This function does not provide translations for characters that require CTRL+ALT or dead keys. Characters not translated by this function must be copied by simulating input using the ALT+ keypad mechanism. The NUMLOCK key must be off.

This function does not provide translations for characters that cannot be typed with one keystroke using the current keyboard layout, such as characters with diacritics requiring dead keys. Characters not translated by this function may be simulated using the ALT+ keypad mechanism. The NUMLOCK key must be on.

This function is implemented using the [VkKeyScan](#) function.

Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[Keyboard Input](#)

[VkKeyScan](#)

Feedback

Was this page helpful?

 Yes

 No

RegisterHotKey function (winuser.h)

Article06/12/2024

Defines a system-wide hot key.

Syntax

C++

```
BOOL RegisterHotKey(
    [in, optional] HWND hWnd,
    [in]           int id,
    [in]           UINT fsModifiers,
    [in]           UINT vk
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window that will receive [WM_HOTKEY](#) messages generated by the hot key. If this parameter is **NULL**, **WM_HOTKEY** messages are posted to the message queue of the calling thread and must be processed in the message loop.

[in] id

Type: **int**

The identifier of the hot key. If the *hWnd* parameter is **NULL**, then the hot key is associated with the current thread rather than with a particular window. If a hot key already exists with the same *hWnd* and *id* parameters, see Remarks for the action taken.

[in] fsModifiers

Type: **UINT**

The keys that must be pressed in combination with the key specified by the *vk* parameter in order to generate the [WM_HOTKEY](#) message. The *fsModifiers* parameter can be a combination of the following values.

Value	Meaning
MOD_ALT 0x0001	Either ALT key must be held down.
MOD_CONTROL 0x0002	Either CTRL key must be held down.
MOD_NOREPEAT 0x4000	Changes the hotkey behavior so that the keyboard auto-repeat does not yield multiple hotkey notifications. Windows Vista: This flag is not supported.
MOD_SHIFT 0x0004	Either SHIFT key must be held down.
MOD_WIN 0x0008	Either WINDOWS key must be held down. These keys are labeled with the Windows logo. Keyboard shortcuts that involve the WINDOWS key are reserved for use by the operating system.

[in] vk**Type: [UINT](#)**

The virtual-key code of the hot key. See [Virtual Key Codes](#).

Return value

Type: [BOOL](#)

If the function succeeds, the return value is nonzero.

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

This function fails if you try to associate a hot key with a window created by another thread.

Typically, [RegisterHotKey](#) also fails if the keystrokes specified for the hot key have already been registered for another hot key. However, some pre-existing, default hotkeys registered by the OS (such as PrintScreen, which launches the Snipping tool) may be overridden by another hot key registration when one of the app's windows is in the foreground.

Remarks

When a key is pressed, the system looks for a match against all hot keys. Upon finding a match, the system posts the [WM_HOTKEY](#) message to the message queue of the window with which the hot key is associated. If the hot key is not associated with a window, then the [WM_HOTKEY](#) message is posted to the thread associated with the hot key.

If a hot key already exists with the same *hWnd* and *id* parameters, it is maintained along with the new hot key. The application must explicitly call [UnregisterHotKey](#) to unregister the old hot key.

The F12 key is reserved for use by the debugger at all times, so it should not be registered as a hot key. Even when you are not debugging an application, F12 is reserved in case a kernel-mode debugger or a just-in-time debugger is resident.

An application must specify an *id* value in the range 0x0000 through 0xBFFF. A shared DLL must specify a value in the range 0xC000 through 0xFFFF (the range returned by the [GlobalAddAtom](#) function). To avoid conflicts with hot-key identifiers defined by other shared DLLs, a DLL should use the [GlobalAddAtom](#) function to obtain the hot-key identifier.

**Windows Server 2003: **If a hot key already exists with the same *hWnd* and *id* parameters, it is replaced by the new hot key.

Examples

The following example shows how to use the [RegisterHotKey](#) function with the **MOD_NOREPEAT** flag.

In this example, the hotkey 'ALT+b' is registered for the main thread. When the hotkey is pressed, the thread will receive a [WM_HOTKEY](#) message, which will get picked up in the [GetMessage](#) call. Because this example uses **MOD_ALT** with the **MOD_NOREPEAT** value for *fsModifiers*, the thread will only receive another [WM_HOTKEY](#) message when the 'b' key is released and then pressed again while the 'ALT' key is being pressed down.

C++

```
#include "stdafx.h"

int _cdecl _tmain (
    int argc,
    TCHAR *argv[])
{
    if (RegisterHotKey(
        NULL,
        1,
```

```

        MOD_ALT | MOD_NOREPEAT,
        0x42)) //0x42 is 'b'
    {
        _tprintf(_T("Hotkey 'ALT+b' registered, using MOD_NOREPEAT
flag\n"));
    }

    MSG msg = {0};
    while (GetMessage(&msg, NULL, 0, 0) != 0)
    {
        if (msg.message == WM_HOTKEY)
        {
            _tprintf(_T("WM_HOTKEY received\n"));
        }
    }

    return 0;
}

```

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[GlobalAddAtom](#)

[Keyboard Input](#)

[Register hotkey for the current app \(CSRegisterHotkey\)](#) ↗

[Register hotkey for the current app \(CppRegisterHotkey\)](#) ↗

[UnregisterHotKey](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

SendInput function (winuser.h)

Article 10/13/2021

Synthesizes keystrokes, mouse motions, and button clicks.

Syntax

C++

```
UINT SendInput(
    [in] UINT    cInputs,
    [in] LPINPUT pInputs,
    [in] int     cbSize
);
```

Parameters

[in] cInputs

Type: **UINT**

The number of structures in the *pInputs* array.

[in] pInputs

Type: **LPINPUT**

An array of **INPUT** structures. Each structure represents an event to be inserted into the keyboard or mouse input stream.

[in] cbSize

Type: **int**

The size, in bytes, of an **INPUT** structure. If *cbSize* is not the size of an **INPUT** structure, the function fails.

Return value

Type: **UINT**

The function returns the number of events that it successfully inserted into the keyboard or mouse input stream. If the function returns zero, the input was already blocked by another thread. To get extended error information, call [GetLastError](#).

This function fails when it is blocked by UIPI. Note that neither [GetLastError](#) nor the return value will indicate the failure was caused by UIPI blocking.

Remarks

This function is subject to UIPI. Applications are permitted to inject input only into applications that are at an equal or lesser integrity level.

The **SendInput** function inserts the events in the **INPUT** structures serially into the keyboard or mouse input stream. These events are not interspersed with other keyboard or mouse input events inserted either by the user (with the keyboard or mouse) or by calls to [keybd_event](#), [mouse_event](#), or other calls to **SendInput**.

This function does not reset the keyboard's current state. Any keys that are already pressed when the function is called might interfere with the events that this function generates. To avoid this problem, check the keyboard's state with the [GetAsyncKeyState](#) function and correct as necessary.

Because the touch keyboard uses the surrogate macros defined in winnls.h to send input to the system, a listener on the keyboard event hook must decode input originating from the touch keyboard. For more information, see [Surrogates and Supplementary Characters](#).

An accessibility application can use **SendInput** to inject keystrokes corresponding to application launch shortcut keys that are handled by the shell. This functionality is not guaranteed to work for other types of applications.

Example

C++

```
*****  
//  
// Sends Win + D to toggle to the desktop  
//  
*****  
void ShowDesktop()  
{  
    OutputString(L"Sending 'Win-D'\r\n");  
    INPUT inputs[4] = {};
```

```

ZeroMemory(inputs, sizeof(inputs));

inputs[0].type = INPUT_KEYBOARD;
inputs[0].ki.wVk = VK_LWIN;

inputs[1].type = INPUT_KEYBOARD;
inputs[1].ki.wVk = 'D';

inputs[2].type = INPUT_KEYBOARD;
inputs[2].ki.wVk = 'D';
inputs[2].ki.dwFlags = KEYEVENTF_KEYUP;

inputs[3].type = INPUT_KEYBOARD;
inputs[3].ki.wVk = VK_LWIN;
inputs[3].ki.dwFlags = KEYEVENTF_KEYUP;

UINT uSent = SendInput(ARRAYSIZE(inputs), inputs, sizeof(INPUT));
if (uSent != ARRAYSIZE(inputs))
{
    OutputString(L"SendInput failed: 0x%x\n",
HRESULT_FROM_WIN32(GetLastError()));
}
}

```

Requirements

[\[\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[GetAsyncKeyState](#)

[INPUT](#)

[Keyboard Input](#)

[Reference](#)

[Surrogates and Supplementary Characters](#)

[keybd_event](#)

[mouse_event](#)

Feedback

Was this page helpful?

 Yes

 No

SetActiveWindow function (winuser.h)

Article 02/22/2024

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

Syntax

C++

```
HWND SetActiveWindow(  
    [in] HWND hWnd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the top-level window to be activated.

Return value

Type: **HWND**

If the function succeeds, the return value is the handle to the window that was previously active.

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

Remarks

The **SetActiveWindow** function activates a window, but not if the application is in the background. The window will be brought into the foreground (top of [Z-Order](#)) if its application is in the foreground when the system activates the window.

If the window identified by the *hWnd* parameter was created by the calling thread, the active window status of the calling thread is set to *hWnd*. Otherwise, the active window

status of the calling thread is set to **NULL**.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetActiveWindow](#)

[Keyboard Input](#)

Reference

[SetForegroundWindow](#)

[WM_ACTIVATE](#)

Feedback

Was this page helpful?

 Yes

 No

SetFocus function (winuser.h)

Article 02/22/2024

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

Syntax

C++

```
HWND SetFocus(  
    [in, optional] HWND hWnd  
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window that will receive the keyboard input. If this parameter is NULL, keystrokes are ignored.

Return value

Type: **HWND**

If the function succeeds, the return value is the handle to the window that previously had the keyboard focus. If the *hWnd* parameter is invalid or the window is not attached to the calling thread's message queue, the return value is NULL. To get extended error information, call [GetLastError function](#).

Extended error ERROR_INVALID_PARAMETER (0x57) means that window is in disabled state.

Remarks

This function sends a [WM_KILLFOCUS](#) message to the window that loses the keyboard focus and a [WM_SETFOCUS](#) message to the window that receives the keyboard focus. It

also activates either the window that receives the focus or the parent of the window that receives the focus.

If a window is active but does not have the focus, any key pressed produces the [WM_SYSCHAR](#), [WM_SYSKEYDOWN](#), or [WM_SYSKEYUP](#) message. If the VK_MENU key is also pressed, bit 30 of the *IParam* parameter of the message is set. Otherwise, the messages produced do not have this bit set.

By using the [AttachThreadInput function](#), a thread can attach its input processing to another thread. This allows a thread to call SetFocus to set the keyboard focus to a window attached to another thread's message queue.

Examples

For an example, see [Initializing a Dialog Box](#).

Requirements

[] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-window-l1-1-4 (introduced in Windows 10, version 10.0.14393)

See also

[AttachThreadInput function](#), [GetFocus function](#), [WM_KILLFOCUS](#), [WM_SETFOCUS](#), [WM_SYSCHAR](#), [WM_SYSKEYDOWN](#), [WM_SYSKEYUP](#), [Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

SetKeyboardState function (winuser.h)

Article02/22/2024

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

Syntax

C++

```
BOOL SetKeyboardState(  
    [in] LPBYTE lpKeyState  
) ;
```

Parameters

[in] lpKeyState

Type: **LPBYTE**

A pointer to a 256-byte array that contains keyboard key states.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

Because the **SetKeyboardState** function alters the input state of the calling thread and not the global input state of the system, an application cannot use **SetKeyboardState** to set the NUM LOCK, CAPS LOCK, or SCROLL LOCK (or the Japanese KANA) indicator lights on the keyboard. These can be set or cleared using [SendInput](#) to simulate keystrokes.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
- [GetKeyState](#)
- [GetKeyboardState](#)
- [GetSystemMetrics](#)
- [MapVirtualKey](#)
- [SetKeyboardState](#)
- [SendInput](#)
- [keybd_event](#)
- [Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

ToAscii function (winuser.h)

Article 05/19/2023

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.

ⓘ Note

This method may not work properly with some **keyboard layouts** that may produce multiple characters (i.e. ligatures) and/or supplementary Unicode characters on a single key press. It is highly recommended to use the [ToUnicode](#) or [ToUnicodeEx](#) methods that handles such cases properly.

Syntax

C++

```
int ToAscii(
    [in]          UINT      uVirtKey,
    [in]          UINT      uScanCode,
    [in, optional] const BYTE *lpKeyState,
    [out]         LPWORD    lpChar,
    [in]          UINT      uFlags
);
```

Parameters

[in] `uVirtKey`

Type: **UINT**

The virtual-key code to be translated. See [Virtual-Key Codes](#).

[in] `uScanCode`

Type: **UINT**

The hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

[in, optional] lpKeyState

Type: **const BYTE***

A pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant. The toggle state of the NUM LOCK and SCROLL LOCK keys is ignored.

[out] lpChar

Type: **LPWORD**

A pointer to the buffer that receives the translated character (or two characters packed into a single **WORD** value, where the low-order byte contains the first character and the high-order byte contains the second character).

[in] uFlags

Type: **UINT**

This parameter must be 1 if a menu is active, or 0 otherwise.

Return value

Type: **int**

The return value is one of the following values.

[+] Expand table

Return value	Description
0	The specified virtual key has no translation for the current state of the keyboard.
1	One character was copied to the buffer.
2	Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic)

stored in the keyboard layout cannot be composed with the specified virtual key to form a single character.

Remarks

The parameters supplied to the **ToAscii** function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, **ToAscii** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *uScanCode* parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+ *number key* combinations.

Although NUM LOCK is a toggle key that affects keyboard behavior, **ToAscii** ignores the toggle setting (the low bit) of *lpKeyState* (VK_NUMLOCK) because the *uVirtKey* parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0 - VK_NUMPAD9).

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[Keyboard Input](#)

[OemKeyScan](#)

Reference

[ToAsciiEx](#)

[ToUnicode](#)

[VkKeyScan](#)

Feedback

Was this page helpful?

 Yes

 No

ToAsciiEx function (winuser.h)

Article 02/22/2024

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.

ⓘ Note

This method may not work properly with some **keyboard layouts** that may produce multiple characters (i.e. ligatures) and/or supplementary Unicode characters on a single key press. It is highly recommended to use the **ToUnicode** or **ToUnicodeEx** methods that handles such cases properly.

Syntax

C++

```
int ToAsciiEx(
    [in]          UINT      uVirtKey,
    [in]          UINT      uScanCode,
    [in, optional] const BYTE *lpKeyState,
    [out]         LPWORD    lpChar,
    [in]          UINT      uFlags,
    [in, optional] HKL      dwhkl
);
```

Parameters

[in] uVirtKey

Type: **UINT**

The virtual-key code to be translated. See [Virtual-Key Codes](#).

[in] uScanCode

Type: **UINT**

The hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

[in, optional] lpKeyState

Type: **const BYTE***

A pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant. The toggle state of the NUM LOCK and SCOLL LOCK keys is ignored.

[out] lpChar

Type: **LPWORD**

A pointer to the buffer that receives the translated character (or two characters packed into a single **WORD** value, where the low-order byte contains the first character and the high-order byte contains the second character).

[in] uFlags

Type: **UINT**

This parameter must be 1 if a menu is active, zero otherwise.

[in, optional] dwhkl

Type: **HKL**

Input locale identifier to use to translate the code. This parameter can be any input locale identifier previously returned by the [LoadKeyboardLayout](#) function.

Return value

Type: **int**

The return value is one of the following values.

[+] Expand table

Return value	Description
0	The specified virtual key has no translation for the current state of the keyboard.

1	One character was copied to the buffer.
2	Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

The parameters supplied to the **ToAsciiEx** function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, **ToAsciiEx** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *uScanCode* parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+number key combinations.

Although NUM LOCK is a toggle key that affects keyboard behavior, **ToAsciiEx** ignores the toggle setting (the low bit) of *lpKeyState* (**VK_NUMLOCK**) because the *uVirtKey* parameter alone is sufficient to distinguish the cursor movement keys (**VK_HOME**, **VK_INSERT**, and so on) from the numeric keys (**VK_DECIMAL**, **VK_NUMPAD0** - **VK_NUMPAD9**).

Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[Keyboard Input](#)

[LoadKeyboardLayout](#)

[MapVirtualKeyEx](#)

Reference

[ToUnicodeEx](#)

[VkKeyScan](#)

Feedback

Was this page helpful?

 Yes

 No

ToUnicode function (winuser.h)

Article 02/02/2023

Translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.

Syntax

C++

```
int ToUnicode(
    [in]             UINT      wVirtKey,
    [in]             UINT      wScanCode,
    [in, optional]   const BYTE *lpKeyState,
    [out]            LPWSTR    pwszBuff,
    [in]             int       cchBuff,
    [in]             UINT      wFlags
);
```

Parameters

[in] wVirtKey

Type: **UINT**

The virtual-key code to be translated. See [Virtual-Key Codes](#).

[in] wScanCode

Type: **UINT**

The hardware [scan code](#) of the key to be translated. The high-order bit of this value is set if the key is up.

[in, optional] lpKeyState

Type: **const BYTE***

A pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key.

If the high-order bit of a byte is set, the key is down. The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant.

The toggle state of the NUM LOCK and SCROLL LOCK keys is ignored. See [GetKeyboardState](#) for more info.

[out] `pwszBuff`

Type: **LPWSTR**

The buffer that receives the translated character or characters as array of UTF-16 code units. This buffer may be returned without being null-terminated even though the variable name suggests that it is null-terminated. You can use the return value of this method to determine how many characters were written.

[in] `cchBuff`

Type: **int**

The size, in characters, of the buffer pointed to by the *pwszBuff* parameter.

[in] `wFlags`

Type: **UINT**

The behavior of the function.

If bit 0 is set, a menu is active. In this mode **Alt+Numeric keypad** key combinations are not handled.

If bit 2 is set, keyboard state is not changed (Windows 10, version 1607 and newer)

All other bits (through 31) are reserved.

Return value

Type: **int**

The function returns one of the following values.

 Expand table

Return value	Description
<code>value < 0</code>	The specified virtual key is a dead key character (accent or diacritic). This value is returned regardless of the keyboard layout, even if several characters have been typed and are stored in the keyboard state. If possible, even with Unicode keyboard layouts, the function has written a spacing version of the dead-key character to the

	buffer specified by <i>pwszBuff</i> . For example, the function writes the character ACUTE ACCENT (U+00B4), rather than the character COMBINING ACUTE ACCENT (U+0301).
0	The specified virtual key has no translation for the current state of the keyboard. Nothing was written to the buffer specified by <i>pwszBuff</i> .
<i>value</i> > 0	One or more UTF-16 code units were written to the buffer specified by <i>pwszBuff</i> . Returned <i>pwszBuff</i> may contain more characters than the return value specifies. When this happens, any extra characters are invalid and should be ignored.

Remarks

To specify a handle to the keyboard layout to use to translate the specified code, use the [ToUnicodeEx](#) function.

Some keyboard layouts may return several characters and/or supplementary characters as [surrogate pairs](#) in *pwszBuff*. If a dead key character (accent or diacritic) stored in the keyboard layout could not be combined with the specified virtual key to form a single character then the previous entered dead character can be combined with the current character.

The parameters supplied to the [ToUnicodeEx](#) function might not be sufficient to translate the virtual-key code because a previous [dead key](#) is stored in the keyboard layout.

Typically, [ToUnicode](#) performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *wScanCode* parameter can be used to distinguish between a key press and a key release (for example for ALT+numpad key entry).

As [ToUnicode](#) translates the virtual-key code, it also changes the state of the kernel-mode keyboard buffer. This state-change affects dead keys, ligatures, [Alt+Numeric keypad](#) key entry, and so on. It might also cause undesired side-effects if used in conjunction with [TranslateMessage](#) (which also changes the state of the kernel-mode keyboard buffer).

Requirements

[] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[Keyboard Input](#)

Reference

[ToAscii](#)

[ToUnicodeEx](#)

[VkKeyScan](#)

Feedback

Was this page helpful?

 Yes

 No

ToUnicodeEx function (winuser.h)

Article 02/02/2023

Translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.

Syntax

C++

```
int ToUnicodeEx(
    [in]          UINT      wVirtKey,
    [in]          UINT      wScanCode,
    [in]          const BYTE *lpKeyState,
    [out]         LPWSTR    pwszBuff,
    [in]          int       cchBuff,
    [in]          UINT      wFlags,
    [in, optional] HKL      dwhkl
);
```

Parameters

[in] **wVirtKey**

Type: **UINT**

The virtual-key code to be translated. See [Virtual-Key Codes](#).

[in] **wScanCode**

Type: **UINT**

The hardware [scan code](#) of the key to be translated. The high-order bit of this value is set if the key is up.

[in] **lpKeyState**

Type: **const BYTE***

A pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key.

If the high-order bit of a byte is set, the key is down. The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant.

The toggle state of the NUM LOCK and SCROLL LOCK keys is ignored. See [GetKeyboardState](#) for more info.

[out] pwszBuff

Type: **LPWSTR**

The buffer that receives the translated character or characters as array of UTF-16 code units. This buffer may be returned without being null-terminated even though the variable name suggests that it is null-terminated. You can use the return value of this method to determine how many characters were written.

[in] cchBuff

Type: **int**

The size, in characters, of the buffer pointed to by the *pwszBuff* parameter.

[in] wFlags

Type: **UINT**

The behavior of the function.

If bit 0 is set, a menu is active. In this mode **Alt+Numeric keypad** key combinations are not handled.

If bit 1 is set, **ToUnicodeEx** will translate scancodes marked as key break events in addition to its usual treatment of key make events.

If bit 2 is set, keyboard state is not changed (Windows 10, version 1607 and newer)

All other bits (through 31) are reserved.

[in, optional] dwhkl

Type: **HKL**

The input locale identifier used to translate the specified code. This parameter can be any input locale identifier previously returned by the [LoadKeyboardLayout](#) function.

Return value

Type: **int**

The function returns one of the following values.

Return value	Description
<i>value</i> < 0	The specified virtual key is a dead key character (accent or diacritic). This value is returned regardless of the keyboard layout, even if several characters have been typed and are stored in the keyboard state. If possible, even with Unicode keyboard layouts, the function has written a spacing version of the dead-key character to the buffer specified by <i>pwszBuff</i> . For example, the function writes the character ACUTE ACCENT (U+00B4), rather than the character COMBINING ACUTE ACCENT (U+0301).
0	The specified virtual key has no translation for the current state of the keyboard. Nothing was written to the buffer specified by <i>pwszBuff</i> .
<i>value</i> > 0	One or more UTF-16 code units were written to the buffer specified by <i>pwszBuff</i> . Returned <i>pwszBuff</i> may contain more characters than the return value specifies. When this happens, any extra characters are invalid and should be ignored.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

Some keyboard layouts may return several characters and/or supplementary characters as [surrogate pairs](#) in *pwszBuff*. If dead key character (accent or diacritic) stored in the keyboard layout could not be combined with the specified virtual key to form a single character then the previous entered dead character can be combined with the current character.

The parameters supplied to the **ToUnicodeEx** function might not be sufficient to translate the virtual-key code because a previous [dead key](#) is stored in the keyboard layout.

Typically, **ToUnicodeEx** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *wScanCode* parameter can be used to distinguish between a key press and a key release (for example for ALT+numpad key entry).

As `ToUnicodeEx` translates the virtual-key code, it also changes the state of the kernel-mode keyboard buffer. This state-change affects dead keys, ligatures, **Alt+Numeric keypad** key entry, and so on. It might also cause undesired side-effects if used in conjunction with `TranslateMessage` (which also changes the state of the kernel-mode keyboard buffer).

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[Keyboard Input](#)

[LoadKeyboardLayout](#)

Reference

[ToAsciiEx](#)

[VkKeyScan](#)

Feedback

Was this page helpful?

 Yes

 No

UnloadKeyboardLayout function (winuser.h)

Article02/22/2024

Unloads an input locale identifier (formerly called a keyboard layout).

Syntax

C++

```
BOOL UnloadKeyboardLayout(  
    [in] HKL hkl  
);
```

Parameters

[in] hkl

Type: **HKL**

The input locale identifier to be unloaded.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function can fail for the following reasons:

- An invalid input locale identifier was passed.
- The input locale identifier was preloaded.
- The input locale identifier is in use.

To get extended error information, call [GetLastError](#).

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

UnloadKeyboardLayout cannot unload the system default input locale identifier if it is the only keyboard layout loaded. You must first load another input locale identifier before unloading the default input locale identifier.

Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[ActivateKeyboardLayout](#)

Conceptual

[GetKeyboardLayoutName](#)

[Keyboard Input](#)

[LoadKeyboardLayout](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

UnregisterHotKey function (winuser.h)

Article02/22/2024

Frees a hot key previously registered by the calling thread.

Syntax

C++

```
BOOL UnregisterHotKey(
    [in, optional] HWND hWnd,
    [in]           int   id
);
```

Parameters

[in, optional] hWnd

Type: **HWND**

A handle to the window associated with the hot key to be freed. This parameter should be **NULL** if the hot key is not associated with a window.

[in] id

Type: **int**

The identifier of the hot key to be freed.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[Keyboard Input](#)

[Reference](#)

[RegisterHotKey](#)

[WM_HOTKEY](#)

Feedback

Was this page helpful?

 Yes

 No

VkKeyScanA function (winuser.h)

Article02/22/2024

[This function has been superseded by the [VkKeyScanEx](#) function. You can still use [VkKeyScan](#), however, if you do not need to specify a keyboard layout.]

Translates a character to the corresponding virtual-key code and shift state for the current keyboard.

Syntax

C++

```
SHORT VkKeyScanA(  
    [in] CHAR ch  
)
```

Parameters

[in] ch

Type: **TCHAR**

The character to be translated into a virtual-key code.

Return value

Type: **SHORT**

If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which can be a combination of the following flag bits.

[] [Expand table](#)

Return value	Description
1	Either SHIFT key is pressed.
2	Either CTRL key is pressed.
4	Either ALT key is pressed.

8	The Hankaku key is pressed
16	Reserved (defined by the keyboard layout driver).
32	Reserved (defined by the keyboard layout driver).

If the function finds no key that translates to the passed character code, both the low-order and high-order bytes contain –1.

Remarks

For keyboard layouts that use the right-hand ALT key as a shift key (for example, the French keyboard layout), the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL+ALT.

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to translate characters into keystrokes from the main keyboard section only. For example, the character "7" is translated into VK_7, not VK_NUMPAD7.

`VkKeyScan` is used by applications that send characters by using the [WM_KEYUP](#) and [WM_KEYDOWN](#) messages.

ⓘ Note

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

Requirements

[+] [Expand table](#)

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

Requirement	Value
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
- [GetKeyNameText](#)
- [GetKeyState](#)
- [GetKeyboardState](#)
- [Keyboard Input](#)
- [SetKeyboardState](#)
- [VkKeyScanEx](#)
- [WM_KEYDOWN](#)
- [WM_KEYUP](#)
- [Keyboard Input](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

VkKeyScanExA function (winuser.h)

Article 02/09/2023

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.

Syntax

C++

```
SHORT VkKeyScanExA(  
    [in] CHAR ch,  
    [in] HKL dwhkl  
)
```

Parameters

[in] ch

Type: **TCHAR**

The character to be translated into a virtual-key code.

[in] dwhkl

Type: **HKL**

Input locale identifier used to translate the character. This parameter can be any input locale identifier previously returned by the [LoadKeyboardLayout](#) function.

Return value

Type: **SHORT**

If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which can be a combination of the following flag bits.

[] Expand table

Return value	Description
1	Either SHIFT key is pressed.
2	Either CTRL key is pressed.
4	Either ALT key is pressed.
8	The Hankaku key is pressed
16	Reserved (defined by the keyboard layout driver).
32	Reserved (defined by the keyboard layout driver).

If the function finds no key that translates to the passed character code, both the low-order and high-order bytes contain –1.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an Input Method Editor (IME), or any other form of input.

For keyboard layouts that use the right-hand ALT key as a shift key (for example, the French keyboard layout), the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL+ALT.

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to translate characters into keystrokes from the main keyboard section only. For example, the character "7" is translated into VK_7, not VK_NUMPAD7.

`VkKeyScanEx` is used by applications that send characters by using the [WM_KEYUP](#) and [WM_KEYDOWN](#) messages.

ⓘ Note

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

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

- [GetAsyncKeyState](#)
- [GetKeyNameText](#)
- [GetKeyState](#)
- [GetKeyboardState](#)
- [LoadKeyboardLayout](#)
- [SetKeyboardState](#)
- [ToAsciiEx](#)
- [Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Keyboard Input Messages

Article • 03/11/2025

In This Section

- [WM_GETHOTKEY](#)
- [WM_SETHOTKEY](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_GETHOTKEY message

Article • 12/11/2020

Sent to determine the hot key associated with a window.

C++

```
#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](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SETHOTKEY message

Article • 08/19/2024

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

C++

```
#define WM_SETHOTKEY 0x0032
```

Parameters

wParam

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

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

[Expand table](#)

Value	Meaning
HOTKEYF_ALT 0x04	ALT key
HOTKEYF_CONTROL 0x02	CTRL key
HOTKEYF_EXT 0x08	Extended key
HOTKEYF_SHIFT 0x01	SHIFT key

The high-order word of *wParam* is ignored.

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

lParam

This parameter is not used.

Return value

The return value is one of the following.

[+] Expand table

Return value	Description
-1	The function is unsuccessful; the hot key is invalid.
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

[+] Expand table

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

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

See also

Reference

[RegisterHotKey](#)

[WM_GETHOTKEY](#)

[WM_SYSCOMMAND](#)

Conceptual

[Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

Keyboard Input Notifications

Article • 04/27/2021

In This Section

- [WM_ACTIVATE](#)
- [WM_APPCOMMAND](#)
- [WM_CHAR](#)
- [WM_DEADCHAR](#)
- [WM_HOTKEY](#)
- [WM_KEYDOWN](#)
- [WM_KEYUP](#)
- [WM_KILLFOCUS](#)
- [WM_SETFOCUS](#)
- [WM_SYSDEADCHAR](#)
- [WM_SYSKEYDOWN](#)
- [WM_SYSKEYUP](#)
- [WM_UNICHAR](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_ACTIVATE message

Article • 12/11/2020

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.

C++

```
#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](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_APPCOMMAND message

Article • 12/11/2020

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.

C++

```
#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.

[] Expand table

Value	Meaning
APPCOMMAND_BASS_BOOST 20	Toggle the bass boost on and off.
APPCOMMAND_BASS_DOWN 19	Decrease the bass.
APPCOMMAND_BASS_UP	Increase the bass.

Value	Meaning
21	
APPCOMMAND_BROWSER_BACKWARD 1	Navigate backward.
APPCOMMAND_BROWSER_FAVORITES 6	Open favorites.
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.
APPCOMMAND_HELP 27	Open the Help dialog.

Value	Meaning
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

Value	Meaning
	APPCOMMAND_MEDIA_PLAY and APPCOMMAND_MEDIA_PAUSE.
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.
APPCOMMAND_REPLY_TO_MAIL 39	Reply to a mail message.
APPCOMMAND_SAVE	Save current document.

Value	Meaning
32	
APPCOMMAND_SEND_MAIL	Send a mail message.
41	
APPCOMMAND SPELL_CHECK	Initiate a spell check.
42	
APPCOMMAND_TREBLE_DOWN	Decrease the treble.
22	
APPCOMMAND_TREBLE_UP	Increase the treble.
23	
APPCOMMAND_UNDO	Undo last action.
34	
APPCOMMAND_VOLUME_DOWN	Lower the volume.
9	
APPCOMMAND_VOLUME_MUTE	Mute the volume.
8	
APPCOMMAND_VOLUME_UP	Raise the volume.
10	

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

[\[+\] Expand table](#)

Value	Meaning
FAPPCOMMAND_KEY 0	User pressed a key.
FAPPCOMMAND_MOUSE 0x8000	User clicked a mouse button.
FAPPCOMMAND_OEM 0x1000	An unidentified hardware source generated the event. It could be a mouse or a keyboard event.

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

[\[+\] Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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

[+] Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

WM_CHAR message

Article • 08/04/2022

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.

C++

```
#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.

Bits	Meaning
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

C++

```
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 UTF-16 (16-bit Unicode Transformation Format) code units in its **wParam** if the Unicode version of the [RegisterClass](#) function was used to register the window class. Otherwise, the system provides characters in the current process code page, which can be set to UTF-8 in Windows Version 1903 (May 2019 Update) and newer. For more information, see [Registering Window Classes](#) and [Use UTF-8 code pages in Windows apps](#).

Starting with Windows Vista, **WM_CHAR** message can send UTF-16 surrogate pairs to Unicode windows. Use the [IS_HIGH_SURROGATE](#), [IS_LOW_SURROGATE](#), and [IS_SURROGATE_PAIR](#) macros to detect such cases, if necessary.

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

- [TranslateMessage](#)
- [WM_KEYDOWN](#)
- [WM_UNICHAR](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful? [!\[\]\(ccbc6214dd8c4e9e4470e37f519c5f2f_img.jpg\) Yes](#) [!\[\]\(f7f3a42e4948be35c746344634895fd9_img.jpg\) No](#)

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

WM_DEADCHAR message

Article • 08/04/2022

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 (ö) is generated by typing the dead key for the umlaut character, and then typing the O key.

C++

```
#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.

Bits	Meaning
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

- [TranslateMessage](#)
 - [WM_KEYDOWN](#)
 - [WM_KEYUP](#)
 - [WM_SYSDEADCHAR](#)
 - [Keyboard Input](#)
 - [About Keyboard Input](#)
-

Feedback

Was this page helpful?

 Yes

 No

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

WM_HOTKEY message

Article • 12/11/2020

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.

C++

```
#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.

Value	Meaning
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](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_KEYDOWN message

Article • 08/04/2022

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.

C++

```
#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

C++

```
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

- [DefWindowProc](#)
- [TranslateMessage](#)
- [WM_CHAR](#)
- [WM_KEYUP](#)
- [WM_SYSCOMMAND](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful?



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

WM_KEYUP message

Article • 08/04/2022

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.

C++

```
#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

- [DefWindowProc](#)
- [TranslateMessage](#)
- [WM_KEYDOWN](#)
- [WM_SYSCOMMAND](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback



Was this page helpful? Yes No

Get help at Microsoft Q&A

WM_KILLFOCUS message

Article • 12/11/2020

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

C++

```
#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](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SETFOCUS message

Article • 12/11/2020

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

C++

```
#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](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SYSDEADCHAR message

Article • 08/04/2022

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.

C++

```
#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.

Bits	Meaning
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

- [TranslateMessage](#)
- [WM_DEADCHAR](#)
- [WM_SYSKEYDOWN](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback



Was this page helpful? [!\[\]\(8bb2ea2c00da452047a65937b2c4f7fe_img.jpg\) Yes](#) [!\[\]\(a8849ef700f039e5268c724b5a54a3c9_img.jpg\) No](#)

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

WM_SYSKEYDOWN message

Article • 08/04/2022

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.

C++

```
#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.

Bits	Meaning
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]

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

See also

- [DefWindowProc](#)
- [TranslateAccelerator](#)
- [WM_SYSCOMMAND](#)
- [WM_SYSKEYUP](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

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

WM_SYSKEYUP message

Article • 08/04/2022

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.

C++

```
#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.

[+] [Expand 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.

Bits	Meaning
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.

[\[+\] Expand table](#)

Message	Virtual-key code
WM_KEYDOWN	VK_CONTROL
WM_KEYDOWN	VK_MENU

Message	Virtual-key code
WM_KEYUP	VK_CONTROL
WM_SYSKEYUP	VK_MENU

Requirements

 Expand table

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

- [DefWindowProc](#)
- [TranslateAccelerator](#)
- [WM_SYSCOMMAND](#)
- [WM_SYSKEYDOWN](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful?



Yes



No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_UNICHAR message

Article • 08/04/2022

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**.)

C++

```
#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.

Bits	Meaning
25-	Reserved; do not use.
28	
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
-------------	-------

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

- [TranslateMessage](#)
- [WM_CHAR](#)
- [WM_KEYDOWN](#)
- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

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

Keyboard Input Structures

Article • 04/27/2021

In This Section

- [HARDWAREINPUT](#)
 - [INPUT](#)
 - [KEYBDINPUT](#)
 - [LASTINPUTINFO](#)
 - [MOUSEINPUT](#)
-

Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

HARDWAREINPUT structure (winuser.h)

Article 02/22/2024

Contains information about a simulated message generated by an input device other than a keyboard or mouse.

Syntax

C++

```
typedef struct tagHARDWAREINPUT {
    DWORD uMsg;
    WORD wParamL;
    WORD wParamH;
} HARDWAREINPUT, *PHARDWAREINPUT, *LPHARDWAREINPUT;
```

Members

uMsg

Type: **DWORD**

The message generated by the input hardware.

wParamL

Type: **WORD**

The low-order word of the *lParam* parameter for **uMsg**.

wParamH

Type: **WORD**

The high-order word of the *lParam* parameter for **uMsg**.

Requirements

[+] Expand table

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

[Conceptual](#)

[INPUT](#)

[Keyboard Input](#)

[Reference](#)

[SendInput](#)

Feedback

Was this page helpful?

 Yes

 No

INPUT structure (winuser.h)

Article 02/22/2024

Used by [SendInput](#) to store information for synthesizing input events such as keystrokes, mouse movement, and mouse clicks.

Syntax

C++

```
typedef struct tagINPUT {
    DWORD type;
    union {
        MOUSEINPUT     mi;
        KEYBDINPUT    ki;
        HARDWAREINPUT hi;
    } DUMMYUNIONNAME;
} INPUT, *PINPUT, *LPIINPUT;
```

Members

type

Type: **DWORD**

The type of the input event. This member can be one of the following values.

[Expand table](#)

Value	Meaning
INPUT_MOUSE 0	The event is a mouse event. Use the mi structure of the union.
INPUT_KEYBOARD 1	The event is a keyboard event. Use the ki structure of the union.
INPUT_HARDWARE 2	The event is a hardware event. Use the hi structure of the union.

DUMMYUNIONNAME

DUMMYUNIONNAME.mi

Type: [MOUSEINPUT](#)

The information about a simulated mouse event.

DUMMYUNIONNAME.ki

Type: [KEYBDINPUT](#)

The information about a simulated keyboard event.

DUMMYUNIONNAME.hi

Type: [HARDWAREINPUT](#)

The information about a simulated hardware event.

Remarks

`INPUT_KEYBOARD` supports nonkeyboard input methods, such as handwriting recognition or voice recognition, as if it were text input by using the `KEYEVENTF_UNICODE` flag. For more information, see the remarks section of [KEYBDINPUT](#).

Requirements

[+] [Expand table](#)

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

[Conceptual](#)

[GetMessageExtraInfo](#)

[HARDWAREINPUT](#)

[KEYBDINPUT](#)

[Keyboard Input](#)

[MOUSEINPUT](#)

[Reference](#)

[SendInput](#)

[keybd_event](#)

[mouse_event](#)

Feedback

Was this page helpful?

 Yes

 No

KEYBDINPUT structure (winuser.h)

Article 05/19/2023

Contains information about a simulated keyboard event.

Syntax

C++

```
typedef struct tagKEYBDINPUT {
    WORD      wVk;
    WORD      wScan;
    DWORD     dwFlags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
} KEYBDINPUT, *PKEYBDINPUT, *LPKEYBDINPUT;
```

Members

wVk

Type: **WORD**

A [virtual-key code](#). The code must be a value in the range 1 to 254. If the **dwFlags** member specifies **KEYEVENTF_UNICODE**, **wVk** must be 0.

wScan

Type: **WORD**

A hardware scan code for the key. If **dwFlags** specifies **KEYEVENTF_UNICODE**, **wScan** specifies a Unicode character which is to be sent to the foreground application.

dwFlags

Type: **DWORD**

Specifies various aspects of a keystroke. This member can be certain combinations of the following values.

[] [Expand table](#)

Value	Meaning
-------	---------

KEYEVENTF_EXTENDEDKEY 0x0001	If specified, the wScan scan code consists of a sequence of two bytes, where the first byte has a value of 0xE0. See Extended-Key Flag for more info.
KEYEVENTF_KEYUP 0x0002	If specified, the key is being released. If not specified, the key is being pressed.
KEYEVENTF_SCANCODE 0x0008	If specified, wScan identifies the key and wVk is ignored.
KEYEVENTF_UNICODE 0x0004	If specified, the system synthesizes a VK_PACKET keystroke. The wVk parameter must be zero. This flag can only be combined with the KEYEVENTF_KEYUP flag. For more information, see the Remarks section.

time

Type: **DWORD**

The time stamp for the event, in milliseconds. If this parameter is zero, the system will provide its own time stamp.

dwExtraInfo

Type: **ULONG_PTR**

An additional value associated with the keystroke. Use the [GetMessageExtraInfo](#) function to obtain this information.

Remarks

INPUT_KEYBOARD supports nonkeyboard-input methods—such as handwriting recognition or voice recognition—as if it were text input by using the **KEYEVENTF_UNICODE** flag. If **KEYEVENTF_UNICODE** is specified, [SendInput](#) sends a [WM_KEYDOWN](#) or [WM_KEYUP](#) message to the foreground thread's message queue with **wParam** equal to **VK_PACKET**. Once [GetMessage](#) or [PeekMessage](#) obtains this message, passing the message to [TranslateMessage](#) posts a [WM_CHAR](#) message with the Unicode character originally specified by **wScan**. This Unicode character will automatically be converted to the appropriate ANSI value if it is posted to an ANSI window.

Set the **KEYEVENTF_SCANCODE** flag to define keyboard input in terms of the scan code. This is useful for simulating a physical keystroke regardless of which keyboard is currently being used. You can also pass the **KEYEVENTF_EXTENDEDKEY** flag if the scan code is an extended key. The virtual key value of a key can change depending on the

current keyboard layout or what other keys were pressed, but the scan code will always be the same.

Requirements

 Expand table

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

- [GetMessageExtraInfo](#)
- [INPUT](#)
- [SendInput](#)
- [Keyboard Input](#)

Feedback

Was this page helpful?



Yes



No

LASTINPUTINFO structure (winuser.h)

Article 02/22/2024

Contains the time of the last input.

Syntax

C++

```
typedef struct tagLASTINPUTINFO {
    UINT cbSize;
    DWORD dwTime;
} LASTINPUTINFO, *PLASTINPUTINFO;
```

Members

`cbSize`

Type: `UINT`

The size of the structure, in bytes. This member must be set to `sizeof(LASTINPUTINFO)`.

`dwTime`

Type: `DWORD`

The tick count when the last input event was received.

Remarks

This function is useful for input idle detection. For more information on tick counts, see [GetTickCount](#).

Requirements

[+] [Expand table](#)

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 Windows.h)

See also

Conceptual

[GetLastInputInfo](#)

[GetTickCount](#)

[Keyboard Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

MOUSEINPUT structure (winuser.h)

Article 04/27/2021

Contains information about a simulated mouse event.

Syntax

C++

```
typedef struct tagMOUSEINPUT {
    LONG      dx;
    LONG      dy;
    DWORD     mouseData;
    DWORD     dwFlags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
} MOUSEINPUT, *PMOUSEINPUT, *LPMOUSEINPUT;
```

Members

dx

Type: **LONG**

The absolute position of the mouse, or the amount of motion since the last mouse event was generated, depending on the value of the **dwFlags** member. Absolute data is specified as the x coordinate of the mouse; relative data is specified as the number of pixels moved.

dy

Type: **LONG**

The absolute position of the mouse, or the amount of motion since the last mouse event was generated, depending on the value of the **dwFlags** member. Absolute data is specified as the y coordinate of the mouse; relative data is specified as the number of pixels moved.

mouseData

Type: **DWORD**

If **dwFlags** contains **MOUSEEVENTF_WHEEL**, then **mouseData** specifies the amount of wheel movement. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. One wheel click is defined as **WHEEL_DELTA**, which is 120.

Windows Vista: If *dwFlags* contains **MOUSEEVENTF_HWHEEL**, then *dwData* specifies the amount of wheel movement. A positive value indicates that the wheel was rotated to the right; a negative value indicates that the wheel was rotated to the left. One wheel click is defined as **WHEEL_DELTA**, which is 120.

If **dwFlags** does not contain **MOUSEEVENTF_WHEEL**, **MOUSEEVENTF_XDOWN**, or **MOUSEEVENTF_XUP**, then **mouseData** should be zero.

If **dwFlags** contains **MOUSEEVENTF_XDOWN** or **MOUSEEVENTF_XUP**, then **mouseData** specifies which X buttons were pressed or released. This value may be any combination of the following flags.

[] [Expand table](#)

Value	Meaning
XBUTTON1 0x0001	Set if the first X button is pressed or released.
XBUTTON2 0x0002	Set if the second X button is pressed or released.

dwFlags

Type: **DWORD**

A set of bit flags that specify various aspects of mouse motion and button clicks. The bits in this member can be any reasonable combination of the following values.

The bit flags that specify mouse button status are set to indicate changes in status, not ongoing conditions. For example, if the left mouse button is pressed and held down, **MOUSEEVENTF_LEFTDOWN** is set when the left button is first pressed, but not for subsequent motions. Similarly **MOUSEEVENTF_LEFTUP** is set only when the button is first released.

You cannot specify both the **MOUSEEVENTF_WHEEL** flag and either **MOUSEEVENTF_XDOWN** or **MOUSEEVENTF_XUP** flags simultaneously in the **dwFlags** parameter, because they both require use of the **mouseData** field.

[Expand table](#)

Value	Meaning
MOUSEEVENTF_MOVE 0x0001	Movement occurred.
MOUSEEVENTF_LEFTDOWN 0x0002	The left button was pressed.
MOUSEEVENTF_LEFTUP 0x0004	The left button was released.
MOUSEEVENTF_RIGHTDOWN 0x0008	The right button was pressed.
MOUSEEVENTF_RIGHTUP 0x0010	The right button was released.
MOUSEEVENTF_MIDDLEDOWN 0x0020	The middle button was pressed.
MOUSEEVENTF_MIDDLEUP 0x0040	The middle button was released.
MOUSEEVENTF_XDOWN 0x0080	An X button was pressed.
MOUSEEVENTF_XUP 0x0100	An X button was released.
MOUSEEVENTF_WHEEL 0x0800	The wheel was moved, if the mouse has a wheel. The amount of movement is specified in mouseData .
MOUSEEVENTF_HWHEEL 0x1000	The wheel was moved horizontally, if the mouse has a wheel. The amount of movement is specified in mouseData . Windows XP/2000: This value is not supported.
MOUSEEVENTF_MOVE_NOCOALESCE 0x2000	The WM_MOUSEMOVE messages will not be coalesced. The default behavior is to coalesce WM_MOUSEMOVE messages. Windows XP/2000: This value is not supported.
MOUSEEVENTF_VIRTUALDESK 0x4000	Maps coordinates to the entire desktop. Must be used with MOUSEEVENTF_ABSOLUTE .
MOUSEEVENTF_ABSOLUTE 0x8000	The dx and dy members contain normalized absolute coordinates. If the flag is not set, dx and dy contain relative data (the change in position since the last reported position). This flag can be set, or not set, regardless of what kind of mouse or other pointing

Value	Meaning
	device, if any, is connected to the system. For further information about relative mouse motion, see the following Remarks section.

`time`

Type: **DWORD**

The time stamp for the event, in milliseconds. If this parameter is 0, the system will provide its own time stamp.

`dwExtraInfo`

Type: **ULONG_PTR**

An additional value associated with the mouse event. An application calls [GetMessageExtraInfo](#) to obtain this extra information.

Remarks

If the mouse has moved, indicated by **MOUSEEVENTF_MOVE**, **dx** and **dy** specify information about that movement. The information is specified as absolute or relative integer values.

If **MOUSEEVENTF_ABSOLUTE** value is specified, **dx** and **dy** contain normalized absolute coordinates between 0 and 65,535. The event procedure maps these coordinates onto the display surface. Coordinate (0,0) maps onto the upper-left corner of the display surface; coordinate (65535,65535) maps onto the lower-right corner. In a multimonitor system, the coordinates map to the primary monitor.

If **MOUSEEVENTF_VIRTUALDESK** is specified, the coordinates map to the entire virtual desktop.

If the **MOUSEEVENTF_ABSOLUTE** value is not specified, **dx** and **dy** specify movement relative to the previous mouse event (the last reported position). Positive values mean the mouse moved right (or down); negative values mean the mouse moved left (or up).

Relative mouse motion is subject to the effects of the mouse speed and the two-mouse threshold values. A user sets these three values with the **Pointer Speed** slider of the Control Panel's **Mouse Properties** sheet. You can obtain and set these values using the [SystemParametersInfo](#) function.

The system applies two tests to the specified relative mouse movement. If the specified distance along either the x or y axis is greater than the first mouse threshold value, and the mouse speed is not zero, the system doubles the distance. If the specified distance along either the x or y axis is greater than the second mouse threshold value, and the mouse speed is equal to two, the system doubles the distance that resulted from applying the first threshold test. It is thus possible for the system to multiply specified relative mouse movement along the x or y axis by up to four times.

Requirements

 Expand table

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

Conceptual

[GetMessageExtraInfo](#)

[INPUT](#)

[Keyboard Input](#)

Reference

[SendInput](#)

Feedback

Was this page helpful?

 Yes

 No

Keyboard Input Constants

Article • 03/11/2025

- [Virtual-Key Codes](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Virtual-Key Codes

Article • 03/14/2025

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.

[\[+\] Expand table](#)

Constant	Value	Description
<code>VK_LBUTTON</code>	0x01	Left mouse button
<code>VK_RBUTTON</code>	0x02	Right mouse button
<code>VK_CANCEL</code>	0x03	Control-break processing
<code>VK_MBUTTON</code>	0x04	Middle mouse button
<code>VK_XBUTTON1</code>	0x05	X1 mouse button
<code>VK_XBUTTON2</code>	0x06	X2 mouse button
	0x07	Reserved
<code>VK_BACK</code>	0x08	Backspace key
<code>VK_TAB</code>	0x09	Tab key
	0x0A-0B	Reserved
<code>VK_CLEAR</code>	0x0C	Clear key
<code>VK_RETURN</code>	0x0D	Enter key
	0x0E-0F	Unassigned
<code>VK_SHIFT</code>	0x10	Shift key
<code>VK_CONTROL</code>	0x11	Ctrl key
<code>VK_MENU</code>	0x12	Alt key
<code>VK_PAUSE</code>	0x13	Pause key
<code>VK_CAPITAL</code>	0x14	Caps lock key

Constant	Value	Description
<code>VK_KANA</code>	0x15	IME Kana mode
<code>VK_HANGUL</code>	0x15	IME Hangul mode
<code>VK ime_on</code>	0x16	IME On
<code>VK_junja</code>	0x17	IME Junja mode
<code>VK_final</code>	0x18	IME final mode
<code>VK_hanja</code>	0x19	IME Hanja mode
<code>VK_kanji</code>	0x19	IME Kanji mode
<code>VK_IME_OFF</code>	0x1A	IME Off
<code>VK_ESCAPE</code>	0x1B	Esc key
<code>VK_CONVERT</code>	0x1C	IME convert
<code>VK_NONCONVERT</code>	0x1D	IME nonconvert
<code>VK_ACCEPT</code>	0x1E	IME accept
<code>VK_MODECHANGE</code>	0x1F	IME mode change request
<code>VK_SPACE</code>	0x20	Spacebar key
<code>VK_PRIOR</code>	0x21	Page up key
<code>VK_NEXT</code>	0x22	Page down key
<code>VK_END</code>	0x23	End key
<code>VK_HOME</code>	0x24	Home key
<code>VK_LEFT</code>	0x25	Left arrow key
<code>VK_UP</code>	0x26	Up arrow key
<code>VK_RIGHT</code>	0x27	Right arrow key
<code>VK_DOWN</code>	0x28	Down arrow key
<code>VK_SELECT</code>	0x29	Select key
<code>VK_PRINT</code>	0x2A	Print key
<code>VK_EXECUTE</code>	0x2B	Execute key
<code>VK_SNAPSHOT</code>	0x2C	Print screen key

Constant	Value	Description
VK_INSERT	0x2D	Insert key
VK_DELETE	0x2E	Delete key
VK_HELP	0x2F	Help key
`0`	0x30	0 key
`1`	0x31	1 key
`2`	0x32	2 key
`3`	0x33	3 key
`4`	0x34	4 key
`5`	0x35	5 key
`6`	0x36	6 key
`7`	0x37	7 key
`8`	0x38	8 key
`9`	0x39	9 key
	0x3A- 40	Undefined
`A`	0x41	A key
`B`	0x42	B key
`C`	0x43	C key
`D`	0x44	D key
`E`	0x45	E key
`F`	0x46	F key
`G`	0x47	G key
`H`	0x48	H key
`I`	0x49	I key
`J`	0x4A	J key
`K`	0x4B	K key

Constant	Value	Description
`L`	0x4C	L key
`M`	0x4D	M key
`N`	0x4E	N key
`O`	0x4F	O key
`P`	0x50	P key
`Q`	0x51	Q key
`R`	0x52	R key
`S`	0x53	S key
`T`	0x54	T key
`U`	0x55	U key
`V`	0x56	V key
`W`	0x57	W key
`X`	0x58	X key
`Y`	0x59	Y key
`Z`	0x5A	Z key
<code>VK_LWIN</code>	0x5B	Left Windows logo key
<code>VK_RWIN</code>	0x5C	Right Windows logo key
<code>VK_APPS</code>	0x5D	Application key
	0x5E	Reserved
<code>VK_SLEEP</code>	0x5F	Computer Sleep key
<code>VK_NUMPAD0</code>	0x60	Numeric keypad 0 key
<code>VK_NUMPAD1</code>	0x61	Numeric keypad 1 key
<code>VK_NUMPAD2</code>	0x62	Numeric keypad 2 key
<code>VK_NUMPAD3</code>	0x63	Numeric keypad 3 key
<code>VK_NUMPAD4</code>	0x64	Numeric keypad 4 key
<code>VK_NUMPAD5</code>	0x65	Numeric keypad 5 key

Constant	Value	Description
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
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

Constant	Value	Description
<code>VK_F17</code>	0x80	F17 key
<code>VK_F18</code>	0x81	F18 key
<code>VK_F19</code>	0x82	F19 key
<code>VK_F20</code>	0x83	F20 key
<code>VK_F21</code>	0x84	F21 key
<code>VK_F22</code>	0x85	F22 key
<code>VK_F23</code>	0x86	F23 key
<code>VK_F24</code>	0x87	F24 key
	0x88- 8F	Reserved
<code>VK_NUMLOCK</code>	0x90	Num lock key
<code>VK_SCROLL</code>	0x91	Scroll lock key
	0x92- 96	OEM specific
	0x97- 9F	Unassigned
<code>VK_LSHIFT</code>	0xA0	Left Shift key
<code>VK_RSHIFT</code>	0xA1	Right Shift key
<code>VK_LCONTROL</code>	0xA2	Left Ctrl key
<code>VK_RCONTROL</code>	0xA3	Right Ctrl key
<code>VK_LMENU</code>	0xA4	Left Alt key
<code>VK_RMENU</code>	0xA5	Right Alt key
<code>VK_BROWSER_BACK</code>	0xA6	Browser Back key
<code>VK_BROWSER_FORWARD</code>	0xA7	Browser Forward key
<code>VK_BROWSER_REFRESH</code>	0xA8	Browser Refresh key
<code>VK_BROWSER_STOP</code>	0xA9	Browser Stop key
<code>VK_BROWSER_SEARCH</code>	0xAA	Browser Search key

Constant	Value	Description
<code>VK_BROWSER_FAVORITES</code>	0xAB	Browser Favorites key
<code>VK_BROWSER_HOME</code>	0xAC	Browser Start and Home key
<code>VK_VOLUME_MUTE</code>	0xAD	Volume Mute key
<code>VK_VOLUME_DOWN</code>	0xAE	Volume Down key
<code>VK_VOLUME_UP</code>	0xAF	Volume Up key
<code>VK_MEDIA_NEXT_TRACK</code>	0xB0	Next Track key
<code>VK_MEDIA_PREV_TRACK</code>	0xB1	Previous Track key
<code>VK_MEDIA_STOP</code>	0xB2	Stop Media key
<code>VK_MEDIA_PLAY_PAUSE</code>	0xB3	Play/Pause Media key
<code>VK_LAUNCH_MAIL</code>	0xB4	Start Mail key
<code>VK_LAUNCH_MEDIA_SELECT</code>	0xB5	Select Media key
<code>VK_LAUNCH_APP1</code>	0xB6	Start Application 1 key
<code>VK_LAUNCH_APP2</code>	0xB7	Start Application 2 key
	0xB8- B9	Reserved
<code>VK_OEM_1</code>	0xBA	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ;: key
<code>VK_OEM_PLUS</code>	0xBB	For any country/region, the + key
<code>VK_OEM_COMMAS</code>	0xBC	For any country/region, the , key
<code>VK_OEM_MINUS</code>	0xBD	For any country/region, the - key
<code>VK_OEM_PERIOD</code>	0xBE	For any country/region, the . key
<code>VK_OEM_2</code>	0xBF	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the /? key
<code>VK_OEM_3</code>	0xC0	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the `~ key
	0xC1- DA	Reserved
<code>VK_OEM_4</code>	0xDB	Used for miscellaneous characters; it can vary by keyboard.

Constant	Value	Description
		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 ' " 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
	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

Constant	Value	Description
VK_NONAME	0xFC	Reserved
VK_PA1	0xFD	PA1 key
VK_OEM_CLEAR	0xFE	Clear key

Remarks

Do not rely on the K_LWIN (0x5B) + VK_F17 (0x80) keys to permanently switch a setting. At shutdown, the system uses these keys to reset various settings, which could include those set by your app.

Requirements

[\[+\] Expand table](#)

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

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Mouse Input

Article • 01/22/2025

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

In this section

[+] Expand table

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

[+] Expand table

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.
BlockInput	Blocks keyboard and mouse input events from reaching applications.
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.
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.
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

Name	Description
	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.
IsWindowEnabled	Determines whether the specified window is enabled for mouse and keyboard input.
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.
SendInput	Synthesizes keystrokes, mouse motions, and button clicks.
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.

[Expand table](#)

Function	Description
mouse_event	Synthesizes mouse motion and button clicks.

Notifications

[\[\] Expand table](#)

Name	Description
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_CAPTURECHANGED	Sent to the window that is losing the mouse capture.
WM_LBUTTONDOWNDBLCLK	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.
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_MBUTTONDOWNDBLCLK	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,

Name	Description
	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_MOUSEWHEEL	Sent to the focus 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, 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.
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_NCLBUTTONDOWNBLCLK	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.

Name	Description
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 to TrackMouseEvent .
WM_NCMOUSELEAVE	Posted to a window when the cursor leaves the nonclient area of the window specified in a prior call to TrackMouseEvent .
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.
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 either XBUTTON1 or XBUTTON2 while the cursor is in the nonclient area of a window. This message

Name	Description
	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 either XBUTTON1 or XBUTTON2 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 either XBUTTON1 or XBUTTON2 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_RBUTTONDOWNDBLCLK	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.
WM_XBUTTONDOWNDBLCLK	Posted when the user double-clicks either XBUTTON1 or XBUTTON2 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 either XBUTTON1 or XBUTTON2 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 either XBUTTON1 or XBUTTON2 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

[] [Expand table](#)

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.
LASTINPUTINFO	Contains the time of the last input.
MOUSEINPUT	Contains information about a simulated mouse event.
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.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Mouse Input Overview

Article • 08/04/2022

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](#)
- [Mouse Capture](#)
- [Mouse ClickLock](#)
- [Mouse Configuration](#)
- [XBUTTONNs](#)
- [Mouse Messages](#)
 - [Client Area Mouse Messages](#)
 - [Nonclient Area Mouse Messages](#)
 - [The WM_NCHITTEST Message](#)
- [Mouse Sonar](#)
- [Mouse Vanish](#)
- [The Mouse Wheel](#)
- [Window Activation](#)

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 either XBUTTON1 or XBUTTON2, are designed to allow easier access to an Internet browser, electronic mail, and media services. When either XBUTTON1 or XBUTTON2 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 mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web 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 XBUTTON 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_XBUTTONDOWNS](#)
- [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 either XBUTTON1 or XBUTTON2 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.

[Expand table](#)

Message	Meaning
WM_LBUTTONDOWNDBLCLK	The left mouse button was double-clicked.
WM_LBUTTONDOWN	The left mouse button was pressed.
WM_BUTTONUP	The left mouse button was released.
WM_MBUTTONDOWNDBLCLK	The middle mouse button was double-clicked.
WM_MBUTTONDOWN	The middle mouse button was pressed.
WM_MBUTTONUP	The middle mouse button was released.
WM_RBUTTONDOWNDBLCLK	The right mouse button was double-clicked.
WM_RBUTTONDOWN	The right mouse button was pressed.

Message	Meaning
WM_RBUTTONUP	The right mouse button was released.
WM_XBUTTONDOWNDBLCLK	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.

[] [Expand table](#)

Value	Description
MK_CONTROL	The CTRL key is down.
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 XBUTTON1 is down.

Value	Description
MK_XBUTTON2	The XBUTTON2 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_LBUTTONDOWNDBLCLK](#)
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.

[+] [Expand table](#)

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).
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).

Value	Location of hot spot
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_MBUTTONDOWN](#) and [WM_MBUTTONUP](#) messages. Double clicking the third button sends the standard [WM_MBUTTONDOWNDBLCLK](#) 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).

[+] Expand table

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(
    PWND pwndData,    //scrolls the window indicated
    int cLinesToScroll); //number of times

short gcWheelDelta; //wheel delta from roll
PWND 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](#).

[\[+\] Expand table](#)

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Using Mouse Input

Article • 08/19/2020

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 *IParam* 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
        // 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_LBUTTONDOWN:

    // 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_DBCLKS** 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_DBCLKS | 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 nBeginLine;         // 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_LBUTTONDOWNCLK:

        // 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.

[\[+\] Expand table](#)

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,"
                    RegisterWindowMessage 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...
            //
            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.


```
    return(ucNumLines);  
}
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Mouse Input Reference

Article • 08/23/2019

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

In this section

 Expand table

Topic	Description
Mouse Input Functions	
Mouse Input Macros	
Mouse Input Notifications	
Mouse Input Structures	

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Mouse Input Functions

Article • 08/18/2023

In this section

[+] Expand table

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	<p>The mouse_event function synthesizes mouse motion and button clicks.</p> <p>Note: This function has been superseded. Use SendInput instead.</p>
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 .

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

_TrackMouseEvent function (commctrl.h)

Article02/22/2024

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.

Syntax

C++

```
BOOL _TrackMouseEvent(
    [in, out] LPTRACKMOUSEEVENT lpEventTrack
);
```

Parameters

[in, out] lpEventTrack

Type: [LPTRACKMOUSEEVENT](#)

A pointer to a [TRACKMOUSEEVENT](#) structure that contains tracking information.

Return value

Type: [BOOL](#)

If the function succeeds, the return value is nonzero.

If the function fails, return value is zero.

Requirements

[Expand table](#)

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

Requirement	Value
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	commctrl.h
Library	Comctl32.lib
DLL	Comctl32.dll

See also

[Conceptual](#)

[Mouse Input](#)

[Other Resources](#)

[Reference](#)

[SystemParametersInfo](#)

[TRACKMOUSEEVENT](#)

[TrackMouseEvent](#)

Feedback

Was this page helpful?

 Yes

 No

DragDetect function (winuser.h)

Article11/19/2022

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.

Syntax

C++

```
BOOL DragDetect(
    [in] HWND hwnd,
    [in] POINT pt
);
```

Parameters

[in] hwnd

Type: **HWND**

A handle to the window receiving mouse input.

[in] pt

Type: **POINT**

Initial position of the mouse, in screen coordinates. The function determines the coordinates of the drag rectangle by using this point.

Return value

Type: **BOOL**

If the user moved the mouse outside of the drag rectangle while holding down the left button, the return value is nonzero.

If the user did not move the mouse outside of the drag rectangle while holding down the left button, the return value is zero.

Remarks

The system metrics for the drag rectangle are configurable, allowing for larger or smaller drag rectangles.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[GetSystemMetrics](#)

[Mouse Input](#)

[POINT](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

EnterReaderModeHelper function

Article • 04/19/2023

Enables the middle mouse button to toggle the presence of a scrolling compass cursor on a scrollable window.

Syntax

C++

```
LONG EnterReaderModeHelper(
    HANDLE hwnd
);
```

Parameters

hwnd

A handle to the window for which the scrolling compass cursor is enabled.

Return value

A value evaluating to FALSE if the passed window handle is invalid; otherwise, TRUE.

Remarks

This function is not defined in an SDK header and must be declared by the caller. This function is exported from user32.dll.

Requirements

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows 10
DLL	user32.dll

See also

Feedback

Was this page helpful?



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

GetCapture function (winuser.h)

Article02/22/2024

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.

Syntax

C++

```
HWND GetCapture();
```

Return value

Type: **HWND**

The return value is a handle to the capture window associated with the current thread. If no window in the thread has captured the mouse, the return value is **NULL**.

Remarks

A **NULL** return value means the current thread has not captured the mouse. However, it is possible that another thread or process has captured the mouse.

To get a handle to the capture window on another thread, use the [GetGUIThreadInfo](#) function.

Requirements

[] Expand table

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

Requirement	Value
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-mouse-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[GetGUIThreadInfo](#)

[Mouse Input](#)

Reference

[ReleaseCapture](#)

[SetCapture](#)

Feedback

Was this page helpful?

 Yes

 No

GetDoubleClickTime function (winuser.h)

Article02/22/2024

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.

Syntax

C++

```
UINT GetDoubleClickTime();
```

Return value

Type: **UINT**

The return value specifies the current double-click time, in milliseconds. The maximum return value is 5000 milliseconds.

Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

Requirement	Value
API set	ext-ms-win-ntuser-mouse-l1-1-0 (introduced in Windows 8)

See also

Conceptual

[Mouse Input](#)

Reference

[SetDoubleClickTime](#)

Feedback

Was this page helpful?

 Yes

 No

GetMouseMovePointsEx function (winuser.h)

Article 10/13/2021

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

Syntax

C++

```
int GetMouseMovePointsEx(
    [in]  UINT          cbSize,
    [in]  LPMOUSEMOVEPOINT lppt,
    [out] LPMOUSEMOVEPOINT lpptBuf,
    [in]  int           nBufPoints,
    [in]  DWORD         resolution
);
```

Parameters

[in] cbSize

Type: **UINT**

The size, in bytes, of the [MOUSEMOVEPOINT](#) structure.

[in] lppt

Type: **LPMOUSEMOVEPOINT**

A pointer to a [MOUSEMOVEPOINT](#) structure containing valid mouse coordinates (in screen coordinates). It may also contain a time stamp.

The **GetMouseMovePointsEx** function searches for the point in the mouse coordinates history. If the function finds the point, it returns the last *nBufPoints* prior to and including the supplied point.

If your application supplies a time stamp, the **GetMouseMovePointsEx** function will use it to differentiate between two equal points that were recorded at different times.

An application should call this function using the mouse coordinates received from the [WM_MOUSEMOVE](#) message and convert them to screen coordinates.

[out] lppBuf

Type: **LPMOUSEMOVEPOINT**

A pointer to a buffer that will receive the points. It should be at least $cbSize * nBufPoints$ in size.

[in] nBufPoints

Type: **int**

The number of points to be retrieved.

[in] resolution

Type: **DWORD**

The resolution desired. This parameter can be one of the following values.

 Expand table

Value	Meaning
GMMP_USE_DISPLAY_POINTS 1	Retrieves the points using the display resolution.
GMMP_USE_HIGH_RESOLUTION_POINTS 2	Retrieves high resolution points. Points can range from zero to 65,535 (0xFFFF) in both x- and y-coordinates. This is the resolution provided by absolute coordinate pointing devices such as drawing tablets.

Return value

Type: **int**

If the function succeeds, the return value is the number of points in the buffer. Otherwise, the function returns –1. For extended error information, your application can call [GetLastError](#).

Remarks

The system retains the last 64 mouse coordinates and their time stamps. If your application supplies a mouse coordinate to **GetMouseMovePointsEx** and the coordinate exists in the system's mouse coordinate history, the function retrieves the specified

number of coordinates from the systems' history. You can also supply a time stamp, which will be used to differentiate between identical points in the history.

The **GetMouseMovePointsEx** function will return points that eventually were dispatched not only to the calling thread but also to other threads.

GetMouseMovePointsEx may fail or return erroneous values in the following cases:

- If negative coordinates are passed in the **MOUSEMOVEPOINT** structure.
- If **GetMouseMovePointsEx** retrieves a coordinate with a negative value.

These situations can occur if multiple monitors are present. To correct this, first call **GetSystemMetrics** to get the following values:

- **SM_XVIRTUALSCREEN**,
- **SM_YVIRTUALSCREEN**,
- **SM_CXVIRTUALSCREEN**, and
- **SM_CYVIRTUALSCREEN**.

Then, for each point that is returned from **GetMouseMovePointsEx**, perform the following transform:

```
int nVirtualWidth = GetSystemMetrics(SM_CXVIRTUALSCREEN) ;
int nVirtualHeight = GetSystemMetrics(SM_CYVIRTUALSCREEN) ;
int nVirtualLeft = GetSystemMetrics(SM_XVIRTUALSCREEN) ;
int nVirtualTop = GetSystemMetrics(SM_YVIRTUALSCREEN) ;
int cpt = 0 ;
int mode = GMMP_USE_DISPLAY_POINTS ;

MOUSEMOVEPOINT mp_in ;
MOUSEMOVEPOINT mp_out[64] ;

ZeroMemory(&mp_in, sizeof(mp_in)) ;
mp_in.x = pt.x & 0x0000FFFF ;//Ensure that this number will pass through.
mp_in.y = pt.y & 0x0000FFFF ;
cpt = GetMouseMovePointsEx(&mp_in, &mp_out, 64, mode) ;

for (int i = 0; i < cpt; i++)
{
    switch(mode)
    {
        case GMMP_USE_DISPLAY_POINTS:
            if (mp_out[i].x > 32767)
                mp_out[i].x -= 65536 ;
            if (mp_out[i].y > 32767)
                mp_out[i].y -= 65536 ;
            break ;
        case GMMP_USE_HIGH_RESOLUTION_POINTS:
    }
}
```

```

        mp_out[i].x = ((mp_out[i].x * (nVirtualWidth - 1)) - (nVirtualLeft *
65536)) / nVirtualWidth ;
        mp_out[i].y = ((mp_out[i].y * (nVirtualHeight - 1)) - (nVirtualTop *
65536)) / nVirtualHeight ;
        break ;
    }
}

```

Requirements

[\[\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[MOUSEMOVEPOINT](#)

[Mouse Input](#)

[Reference](#)

Feedback

Was this page helpful?

[Yes](#)

[No](#)

mouse_event function (winuser.h)

Article 10/13/2021

The **mouse_event** function synthesizes mouse motion and button clicks.

Note This function has been superseded. Use **SendInput** instead.

Syntax

C++

```
void mouse_event(
    [in] DWORD      dwFlags,
    [in] DWORD      dx,
    [in] DWORD      dy,
    [in] DWORD      dwData,
    [in] ULONG_PTR  dwExtraInfo
);
```

Parameters

[in] dwFlags

Type: **DWORD**

Controls various aspects of mouse motion and button clicking. This parameter can be certain combinations of the following values.

[] Expand table

Value	Meaning
MOUSEEVENTF_ABSOLUTE 0x8000	The <i>dx</i> and <i>dy</i> parameters contain normalized absolute coordinates. If not set, those parameters contain relative data: the change in position since the last reported position. This flag can be set, or not set, regardless of what kind of mouse or mouse-like device, if any, is connected to the system. For further information about relative mouse motion, see the following Remarks section.

MOUSEEVENTF_LEFTDOWN 0x0002	The left button is down.
MOUSEEVENTF_LEFTUP 0x0004	The left button is up.
MOUSEEVENTF_MIDDLEDOWN 0x0020	The middle button is down.
MOUSEEVENTF_MIDDLEUP 0x0040	The middle button is up.
MOUSEEVENTF_MOVE 0x0001	Movement occurred.
MOUSEEVENTF_RIGHTDOWN 0x0008	The right button is down.
MOUSEEVENTF_RIGHTUP 0x0010	The right button is up.
MOUSEEVENTF_WHEEL 0x0800	The wheel has been moved, if the mouse has a wheel. The amount of movement is specified in <i>dwData</i>
MOUSEEVENTF_XDOWN 0x0080	An X button was pressed.
MOUSEEVENTF_XUP 0x0100	An X button was released.
MOUSEEVENTF_WHEEL 0x0800	The wheel button is rotated.
MOUSEEVENTF_HWHEEL 0x01000	The wheel button is tilted.

The values that specify mouse button status are set to indicate changes in status, not ongoing conditions. For example, if the left mouse button is pressed and held down, **MOUSEEVENTF_LEFTDOWN** is set when the left button is first pressed, but not for subsequent motions. Similarly, **MOUSEEVENTF_LEFTUP** is set only when the button is first released.

You cannot specify both **MOUSEEVENTF_WHEEL** and either **MOUSEEVENTF_XDOWN** or **MOUSEEVENTF_XUP** simultaneously in the *dwFlags* parameter, because they both require use of the *dwData* field.

[in] dx

Type: **DWORD**

The mouse's absolute position along the x-axis or its amount of motion since the last mouse event was generated, depending on the setting of **MOUSEEVENTF_ABSOLUTE**. Absolute data is specified as the mouse's actual x-coordinate; relative data is specified as the number of mickeys moved. A *mickey* is the amount that a mouse has to move for it to report that it has moved.

[in] *dy*

Type: **DWORD**

The mouse's absolute position along the y-axis or its amount of motion since the last mouse event was generated, depending on the setting of **MOUSEEVENTF_ABSOLUTE**. Absolute data is specified as the mouse's actual y-coordinate; relative data is specified as the number of mickeys moved.

[in] *dwData*

Type: **DWORD**

If *dwFlags* contains **MOUSEEVENTF_WHEEL**, then *dwData* specifies the amount of wheel movement. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. One wheel click is defined as **WHEEL_DELTA**, which is 120.

If *dwFlags* contains **MOUSEEVENTF_HWHEEL**, then *dwData* specifies the amount of wheel movement. A positive value indicates that the wheel was tilted to the right; a negative value indicates that the wheel was tilted to the left.

If *dwFlags* contains **MOUSEEVENTF_XDOWN** or **MOUSEEVENTF_XUP**, then *dwData* specifies which X buttons were pressed or released. This value may be any combination of the following flags.

If *dwFlags* is not **MOUSEEVENTF_WHEEL**, **MOUSEEVENTF_XDOWN**, or **MOUSEEVENTF_XUP**, then *dwData* should be zero.

 Expand table

Value	Meaning
XBUTTON1 0x0001	Set if the first X button was pressed or released.
XBUTTON2	Set if the second X button was pressed or released.

[in] dwExtraInfo

Type: **ULONG_PTR**

An additional value associated with the mouse event. An application calls [GetMessageExtraInfo](#) to obtain this extra information.

Return value

None

Remarks

If the mouse has moved, indicated by **MOUSEEVENTF_MOVE** being set, *dx* and *dy* hold information about that motion. The information is specified as absolute or relative integer values.

If **MOUSEEVENTF_ABSOLUTE** value is specified, *dx* and *dy* contain normalized absolute coordinates between 0 and 65,535. The event procedure maps these coordinates onto the display surface. Coordinate (0,0) maps onto the upper-left corner of the display surface, (65535,65535) maps onto the lower-right corner.

If the **MOUSEEVENTF_ABSOLUTE** value is not specified, *dx* and *dy* specify relative motions from when the last mouse event was generated (the last reported position). Positive values mean the mouse moved right (or down); negative values mean the mouse moved left (or up).

Relative mouse motion is subject to the settings for mouse speed and acceleration level. An end user sets these values using the Mouse application in Control Panel. An application obtains and sets these values with the [SystemParametersInfo](#) function.

The system applies two tests to the specified relative mouse motion when applying acceleration. If the specified distance along either the x or y axis is greater than the first mouse threshold value, and the mouse acceleration level is not zero, the operating system doubles the distance. If the specified distance along either the x- or y-axis is greater than the second mouse threshold value, and the mouse acceleration level is equal to two, the operating system doubles the distance that resulted from applying the first threshold test. It is thus possible for the operating system to multiply relatively-specified mouse motion along the x- or y-axis by up to four times.

Once acceleration has been applied, the system scales the resultant value by the desired mouse speed. Mouse speed can range from 1 (slowest) to 20 (fastest) and represents how much the pointer moves based on the distance the mouse moves. The default value is 10, which results in no additional modification to the mouse motion.

The **mouse_event** function is used to synthesize mouse events by applications that need to do so. It is also used by applications that need to obtain more information from the mouse than its position and button state. For example, if a tablet manufacturer wants to pass pen-based information to its own applications, it can write a DLL that communicates directly to the tablet hardware, obtains the extra information, and saves it in a queue. The DLL then calls **mouse_event** with the standard button and x/y position data, along with, in the *dwExtraInfo* parameter, some pointer or index to the queued extra information. When the application needs the extra information, it calls the DLL with the pointer or index stored in *dwExtraInfo*, and the DLL returns the extra information.

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

Conceptual

[GetMessageExtraInfo](#)

[Mouse Input](#)

Other Resources

Feedback

Was this page helpful?

 Yes

 No

ReleaseCapture function (winuser.h)

Article02/22/2024

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.

Syntax

C++

```
BOOL ReleaseCapture();
```

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

An application calls this function after calling the [SetCapture](#) function.

Examples

For an example, see [Drawing Lines with the Mouse](#).

Requirements

[+] [Expand table](#)

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

Requirement	Value
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-mouse-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetCapture](#)

[Mouse Input](#)

[Reference](#)

[SetCapture](#)

[WM_CAPTURECHANGED](#)

Feedback

Was this page helpful?

 Yes

 No

SetCapture function (winuser.h)

Article02/22/2024

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.

Syntax

C++

```
HWND SetCapture(  
    [in] HWND hWnd  
);
```

Parameters

[in] hWnd

Type: **HWND**

A handle to the window in the current thread that is to capture the mouse.

Return value

Type: **HWND**

The return value is a handle to the window that had previously captured the mouse. If there is no such window, the return value is **NULL**.

Remarks

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

foreground window has captured the mouse, the user can still click another window, bringing it to the foreground.

When the window no longer requires all mouse input, the thread that created the window should call the [ReleaseCapture](#) function to release the mouse.

This function cannot be used to capture mouse input meant for another process.

When the mouse is captured, menu hotkeys and other keyboard accelerators do not work.

Examples

For an example, see [Drawing Lines with the Mouse](#).

Requirements

[] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-mouse-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[GetCapture](#)

[Mouse Input](#)

[Reference](#)

[ReleaseCapture](#)

Feedback

Was this page helpful?

 Yes

 No

SetDoubleClickTime function (winuser.h)

Article02/22/2024

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.

Syntax

C++

```
BOOL SetDoubleClickTime(  
    [in] UINT unnamedParam1  
) ;
```

Parameters

[in] unnamedParam1

Type: **UINT**

The number of milliseconds that may occur between the first and second clicks of a double-click. If this parameter is set to 0, the system uses the default double-click time of 500 milliseconds. If this parameter value is greater than 5000 milliseconds, the system sets the value to 5000 milliseconds.

Return value

Type: **BOOL**

If the function succeeds, the return value is nonzero.

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

Remarks

The **SetDoubleClickTime** function alters the double-click time for all windows in the system.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-mouse-l1-1-1 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetDoubleClickTime](#)

[Mouse Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

SwapMouseButton function (winuser.h)

Article 02/22/2024

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

Syntax

C++

```
BOOL SwapMouseButton(  
    [in] BOOL fSwap  
);
```

Parameters

[in] fSwap

Type: **BOOL**

If this parameter is **TRUE**, the left button generates right-button messages and the right button generates left-button messages. If this parameter is **FALSE**, the buttons are restored to their original meanings.

Return value

Type: **BOOL**

If the meaning of the mouse buttons was reversed previously, before the function was called, the return value is nonzero.

If the meaning of the mouse buttons was not reversed, the return value is zero.

Remarks

Button swapping is provided as a convenience to people who use the mouse with their left hands. The **SwapMouseButton** function is usually called by Control Panel only. Although an application is free to call the function, the mouse is a shared resource and reversing the meaning of its buttons affects all applications.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[Mouse Input](#)

[Reference](#)

[SetDoubleClickTime](#)

Feedback

Was this page helpful?

 Yes

 No

TrackMouseEvent function (winuser.h)

Article10/13/2021

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`.

Syntax

C++

```
BOOL TrackMouseEvent(  
    [in, out] LPTRACKMOUSEEVENT lpEventTrack  
);
```

Parameters

[in, out] lpEventTrack

Type: `LPTRACKMOUSEEVENT`

A pointer to a `TRACKMOUSEEVENT` structure that contains tracking information.

Return value

Type: `BOOL`

If the function succeeds, the return value is nonzero .

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

Remarks

The mouse pointer is considered to be hovering when it stays within a specified rectangle for a specified period of time. Call [SystemParametersInfo](#). and use the values

SPI_GETMOUSEHOVERWIDTH, **SPI_GETMOUSEHOVERHEIGHT**, and **SPI_GETMOUSEHOVERTIME** to retrieve the size of the rectangle and the time.

The function can post the following messages.

[+] Expand table

Message	Description
WM_NCMOUSEOVER	The same meaning as WM_MOUSEOVER except this is for the nonclient area of the window.
WM_NCMOUSELEAVE	The same meaning as WM_MOUSELEAVE except this is for the nonclient area of the window.
WM_MOUSEOVER	The mouse hovered over the client area of the window for the period of time specified in a prior call to TrackMouseEvent . Hover tracking stops when this message is generated. The application must call TrackMouseEvent again if it requires further tracking of mouse hover behavior.
WM_MOUSELEAVE	The mouse left the client area of the window specified in a prior call to TrackMouseEvent . 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

[+] Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-mouse-l1-1-0 (introduced in Windows 8)

See also

[Conceptual](#)

[Mouse Input](#)

[Other Resources](#)

[Reference](#)

[SystemParametersInfo](#)

[TRACKMOUSEEVENT](#)

[_TrackMouseEvent](#)

Feedback

Was this page helpful?

 Yes

 No

Mouse Input Macros

Article • 04/27/2021

In This Section

- [GET_APPCOMMAND_LPARAM](#)
- [GET_DEVICE_LPARAM](#)
- [GET_FLAGS_LPARAM](#)
- [GET_KEYSTATE_LPARAM](#)
- [GET_KEYSTATE_WPARAM](#)
- [GET_MOUSEORKEY_LPARAM](#)
- [GET_NCHITTEST_WPARAM](#)
- [GET_WHEEL_DELTA_WPARAM](#)
- [GET_XBUTTON_WPARAM](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GET_APPCOMMAND_LPARAM macro (winuser.h)

Article02/22/2024

Retrieves the application command from the specified **LPARAM** value.

Syntax

C++

```
void GET_APPCOMMAND_LPARAM(  
    LPARAM  
);
```

Parameters

lParam

The value to be converted.

Return value

None

Requirements

[+] Expand table

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

See also

Feedback

Was this page helpful?

 Yes

 No

GET_DEVICE_LPARAM macro (winuser.h)

Article02/22/2024

Retrieves the input device type from the specified LPARAM value.

Syntax

C++

```
void GET_DEVICE_LPARAM(  
    lParam  
);
```

Parameters

lParam

The value to be converted.

Return value

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

[+] Expand table

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

Return value

None

Remarks

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

Requirements

 [Expand table](#)

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

See also

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

Feedback

Was this page helpful?

 Yes

 No

GET_FLAGS_LPARAM macro (winuser.h)

Article02/22/2024

Retrieves the state of certain virtual keys from the specified LPARAM value.

Syntax

C++

```
void GET_FLAGS_LPARAM(  
    lParam  
);
```

Parameters

lParam

The value to be converted.

Return value

None

Remarks

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

Requirements

[] Expand table

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

See also

Conceptual

[GET_KEYSTATE_LPARAM](#)

[Mouse Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

GET_KEYSTATE_LPARAM macro (winuser.h)

Article 02/22/2024

Retrieves the state of certain virtual keys from the specified LPARAM value.

Syntax

C++

```
void GET_KEYSTATE_LPARAM(  
    lParam  
);
```

Parameters

lParam

The value to be converted.

Return value

None

Remarks

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

Requirements

[+] Expand table

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

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

See also

Conceptual

[GET_FLAGS_LPARAM](#)

[Mouse Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

GET_KEYSTATE_WPARAM macro (winuser.h)

Article02/22/2024

Retrieves the state of certain virtual keys from the specified **WPARAM** value.

Syntax

C++

```
void GET_KEYSTATE_WPARAM(  
    wParam  
);
```

Parameters

wParam

The value to be converted.

Return value

None

Requirements

[+] Expand table

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

See also

Feedback

Was this page helpful?

 Yes

 No

GET_MOUSEORKEY_LPARAM macro

Article04/20/2016

Retrieves the input device type from the specified **LPARAM** value.

Syntax

c++

```
WORD GET_MOUSEORKEY_LPARAM(  
    LPARAM lParam  
)
```

Parameters

- *lParam*
The value to be converted.

Return value

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

[] Expand table

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

Remarks

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

Requirements

 [Expand table](#)

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

[GET_DEVICE_LPARAM ↗](#)

Conceptual

[Mouse Input ↗](#)

GET_NCHITTEST_WPARAM macro (winuser.h)

Article02/22/2024

Retrieves the hit-test value from the specified **WPARAM** value.

Syntax

C++

```
void GET_NCHITTEST_WPARAM(  
    wParam  
);
```

Parameters

wParam

The value to be converted.

Return value

None

Requirements

[+] Expand table

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

See also

Feedback

Was this page helpful?

 Yes

 No

GET_WHEEL_DELTA_WPARAM macro (winuser.h)

Article02/22/2024

Retrieves the wheel-delta value from the specified **WPARAM** value.

Syntax

C++

```
void GET_WHEEL_DELTA_WPARAM(  
    wParam  
);
```

Parameters

wParam

The value to be converted.

Return value

None

Requirements

[+] Expand table

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

See also

Feedback

Was this page helpful?

 Yes

 No

GET_XBUTTON_WPARAM macro (winuser.h)

Article02/22/2024

Retrieves the state of certain buttons from the specified **WPARAM** value.

Syntax

C++

```
void GET_XBUTTON_WPARAM(  
    wParam  
);
```

Parameters

wParam

The value to be converted.

Return value

None

Requirements

[+] Expand table

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

See also

Feedback

Was this page helpful?

 Yes

 No

Mouse Input Notifications

Article • 04/27/2021

In This Section

- [WM_CAPTURECHANGED](#)
- [WM_LBUTTONDOWNDBLCLK](#)
- [WM_LBUTTONDOWN](#)
- [WM_LBUTTONUP](#)
- [WM_MBUTTONDOWNDBLCLK](#)
- [WM_MBUTTONDOWN](#)
- [WM_MBUTTONUP](#)
- [WM_MOUSEACTIVATE](#)
- [WM_MOUSEHOVER](#)
- [WM_MOUSEWHEEL](#)
- [WM_MOUSELEAVE](#)
- [WM_MOUSEMOVE](#)
- [WM_MOUSEWHEEL](#)
- [WM_NCHITTEST](#)
- [WM_NCLBUTTONDOWNDBLCLK](#)
- [WM_NCLBUTTONDOWN](#)
- [WM_NCLBUTTONUP](#)
- [WM_NCMBUTTONDOWNDBLCLK](#)
- [WM_NCMBUTTONDOWN](#)
- [WM_NCMBUTTONUP](#)
- [WM_NCMOUSEHOVER](#)
- [WM_NCMOUSELEAVE](#)
- [WM_NCMOUSEMOVE](#)
- [WM_NCRBUTTONDOWNDBLCLK](#)
- [WM_NCRBUTTONDOWN](#)
- [WM_NCRBUTTONUP](#)
- [WM_NCXBUTTONDOWNDBLCLK](#)
- [WM_NCXBUTTONDOWN](#)
- [WM_NCXBUTTONUP](#)
- [WM_RBUTTONDOWNDBLCLK](#)
- [WM_RBUTTONDOWN](#)
- [WM_RBUTTONUP](#)
- [WM_XBUTTONDOWNDBLCLK](#)
- [WM_XBUTTONDOWN](#)

- WM_XBUTTONUP
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_CAPTURECHANGED message

Article • 12/11/2020

Sent to the window that is losing the mouse capture.

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

C++

```
#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

[] [Expand table](#)

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 Windows.h)

See also

Reference

[ReleaseCapture](#)

[SetCapture](#)

Conceptual

[Mouse Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_LBUTTONDOWNDBLCLK message

Article • 11/19/2022

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.

C++

```
#define WM_LBUTTONDOWNDBLCLK 0x0203
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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 **HIGHWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y-coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

Only windows that have the **CS_DBLCLKS** style can receive **WM_LBUTTONDOWNDBLCLK** 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_LBUTTONDOWNDBLCLK](#), and [WM_LBUTTONUP](#).

Requirements

[Expand table](#)

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_LBUTTONDOWN message

Article • 04/18/2023

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.

C++

```
#define WM_LBUTTONDOWN 0x0201
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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

C++

```
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

 Expand table

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_LBUTTONDOWNDBLCLK](#)

[WM_LBUTTONUP](#)

[Conceptual](#)

[Mouse Input](#)

[Other Resources](#)

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_LBUTTONDOWN message

Article • 01/22/2025

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.

C++

```
#define WM_LBUTTONDOWN 0x0202
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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

 Expand table

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_LBUTTONDOWNDBLCLK](#)

[WM_LBUTTONDOWN](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MBUTTONDOWNDBLCLK message

Article • 01/22/2025

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.

C++

```
#define WM_MBUTTONDOWNDBLCLK 0x0209
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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_MBUTTONDOWNDBLCLK** 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_BUTTONUP](#), [WM_MBUTTONDOWNDBLCLK](#), and [WM_BUTTONUP](#) again.

Requirements

[Expand table](#)

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_BUTTONUP](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MBUTTONDOWN message

Article • 11/19/2022

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.

C++

```
#define WM_MBUTTONDOWN 0x0207
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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_MBUTTONDOWNDBLCLK](#)

[WM_MBUTTONUP](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MBUTTONDOWN message

Article • 01/22/2025

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.

C++

```
#define WM_MBUTTONDOWN 0x0208
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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_MBUTTONDOWN](#) 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

 Expand table

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_MBUTTONDOWNDBLCLK](#)

[WM_MBUTTONDOWN](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MOUSEACTIVATE message

Article • 12/11/2020

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.

C++

```
#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.

[+] Expand table

Return code/value	Description
MA_ACTIVATE 1	Activates the window, and does not discard the mouse message.

Return code/value	Description
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

[Expand table](#)

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MOUSEHOVER message

Article • 11/19/2022

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.

C++

```
#define WM_MOUSEHOVER 0x02A1
```

Parameters

wParam

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

[+] Expand table

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MOUSEWHEEL message

Article • 11/19/2022

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.

C++

```
#define WM_MOUSEWHEEL 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.

[\[+\] Expand table](#)

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.

Value	Meaning
MK_XBUTTON1 0x0020	The XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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

 Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MOUSELEAVE message

Article • 12/11/2020

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.

C++

```
#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

[+] [Expand table](#)

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 Windows.h)

See also

Reference

[GetCapture](#)

[SetCapture](#)

[TrackMouseEvent](#)

[TRACKMOUSEEVENT](#)

[WM_NCMOUSELEAVE](#)

Conceptual

[Mouse Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | Get help at Microsoft Q&A

WM_MOUSEMOVE message

Article • 01/22/2025

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.

C++

```
#define WM_MOUSEMOVE 0x0200
```

Parameters

wParam

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

[\[+\] Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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

[+] Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_MOUSEWHEEL message

Article • 11/19/2022

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.

C++

```
#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.

[+] [Expand table](#)

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.

Value	Meaning
MK_XBUTTON1 0x0020	The XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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 **HIGHWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y-coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

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

 Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCHITTEST message

Article • 11/19/2022

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.

C++

```
#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.

[+] Expand table

Return code/value	Description
HTBORDER	In the border of a window that does not have a sizing border.

Return code/value	Description
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).
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.
HTREDUCE 8	In a Minimize button.

Return code/value	Description
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

 Expand table

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

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCLBUTTONDOWNDBLCLK message

Article • 11/19/2022

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.

C++

```
#define WM_NCLBUTTONDOWNDBLCLK 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_NCLBUTTONDOWNDBLCLK** messages.

The system generates a **WM_NCLBUTTONDOWNDBLCLK** 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_NCLBUTTONDOWNDBLCLK](#), and [WM_NCLBUTTONUP](#) again.

Requirements

[\[+\] Expand table](#)

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](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCLBUTTONDOWN message

Article • 11/19/2022

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.

C++

```
#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

[+] Expand table

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_NCLBUTTONDOWNDBLCLK](#)

[WM_NCLBUTTONUP](#)

[WM_SYSCOMMAND](#)

Conceptual

[Mouse Input](#)

[Other Resources](#)

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCLBUTTONUP message

Article • 11/19/2022

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.

C++

```
#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

[Expand table](#)

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_NCLBUTTONDOWNDBLCLK](#)

[WM_NCLBUTTONDOWN](#)

[WM_SYSCOMMAND](#)

[Conceptual](#)

[Mouse Input](#)

[Other Resources](#)

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

[!\[\]\(533f4c010795fda3595ccd2f4e32280e_img.jpg\) Yes](#)

[!\[\]\(58f50b136ff3c4d90e225ffd062ad35c_img.jpg\) No](#)

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCMBUTTONONDBLCLK message

Article • 11/19/2022

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.

C++

```
#define WM_NCMBUTTONONDBLCLK 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_NCMBUTTONONDBLCLK](#) messages.

The system generates a [WM_NCMBUTTONONDBLCLK](#) 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_NCMBUTTONONDBLCLK](#), 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

 Expand table

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](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCMBUTTONDOWN message

Article • 11/19/2022

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.

C++

```
#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

[\[+\] Expand table](#)

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_NCMBUTTONDBLCLK](#)

[WM_NCMBUTTONUP](#)

[WM_SYSCOMMAND](#)

Conceptual

[Mouse Input](#)

Other Resources

MAKEPOINTS

POINTS

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

WM_NCMBUTTONUP message

Article • 11/19/2022

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.

C++

```
#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

[\[+\] Expand table](#)

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_NCMBUTTONDBLCLK](#)

[WM_NCMBUTTONDOWN](#)

[WM_SYSCOMMAND](#)

Conceptual

[Mouse Input](#)

Other Resources

MAKEPOINTS

POINTS

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

WM_NCMOUSEHOVER message

Article • 11/19/2022

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.

C++

```
#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

 Expand table

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](#)

[TrackMouseEvent](#)

[TRACKMOUSEEVENT](#)

[WM_NCHITTEST](#)

[WM_MOUSEHOVER](#)

Conceptual

[Mouse Input](#)

Other Resources

MAKEPOINTS

POINTS

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | Get help at Microsoft Q&A

WM_NCMOUSELEAVE message

Article • 12/11/2020

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.

C++

```
#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

[+] [Expand table](#)

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 Windows.h)

See also

Reference

[TrackMouseEvent](#)

[TRACKMOUSEEVENT](#)

[WM_SYSCOMMAND](#)

[WM_MOUSELEAVE](#)

Conceptual

[Mouse Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCMOUSEMOVE message

Article • 11/19/2022

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.

C++

```
#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

 Expand table

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_SYSCOMMAND](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCRBUTTONDOWNDBLCLK message

Article • 11/19/2022

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.

C++

```
#define WM_NCRBUTTONDOWNDBLCLK 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_NCRBUTTONDOWNDBLCLK](#) messages.

The system generates a [WM_NCRBUTTONDOWNDBLCLK](#) 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_NCRBUTTONDOWNDBLCLK](#), 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

 Expand table

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](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCRBUTTONDOWN message

Article • 11/19/2022

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.

C++

```
#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

[\[+\] Expand table](#)

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_NCRBUTTONDOWNDBLCLK](#)

[WM_NCRBUTTONUP](#)

[WM_SYSCOMMAND](#)

Conceptual

[Mouse Input](#)

Other Resources

MAKEPOINTS

POINTS

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCRBUTTONUP message

Article • 11/19/2022

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.

C++

```
#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

[\[+\] Expand table](#)

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_NCRBUTTONDOWNDBLCLK](#)

[WM_NCRBUTTONDOWN](#)

[WM_SYSCOMMAND](#)

Conceptual

[Mouse Input](#)

Other Resources

MAKEPOINTS

POINTS

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCXBUTTONONDBLCLK message

Article • 01/22/2025

Posted when the user double-clicks either XBUTTON1 or XBUTTON2 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.

C++

```
#define WM_NCXBUTTONONDBLCLK 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.

[+] [Expand table](#)

Value	Meaning
XBUTTON1 0x0001	The XBUTTON1 was double-clicked..
XBUTTON2 0x0002	The second XBUTTON2 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

Windows supports mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web browser.

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

```
nHitTest = GET_NCHITTEST_WPARAM(wParam);  
fButton = 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_NCXBUTTONONDBLCLK** messages. The system generates a **WM_NCXBUTTONONDBLCLK** message when the user presses, releases, and again presses an XBUTTON 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_NCLBUTTONDOWNDBLCLK](#), [WM_NCMBUTTONDBLCLK](#), and [WM_NCRBUTTONDOWNDBLCLK](#) 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

[+] [Expand table](#)

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

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_NCXBUTTONDOWN message

Article • 01/22/2025

Posted when the user presses either XBUTTON1 or XBUTTON2 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.

C++

```
#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.

[+] [Expand table](#)

Value	Meaning
XBUTTON1 0x0001	The XBUTTON1 was pressed.
XBUTTON2 0x0002	The XBUTTON2 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

Windows supports mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web browser.

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 **HIGHWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y- coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

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

 Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

WM_NCXBUTTONUP message

Article • 01/22/2025

Posted when the user releases either XBUTTON1 or XBUTTON2 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.

C++

```
#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.

[+] [Expand table](#)

Value	Meaning
XBUTTON1 0x0001	The XBUTTON1 was released.
XBUTTON2 0x0002	The XBUTTON2 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

Windows supports mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web browser.

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

```
nHitTest = GET_NCHITTEST_WPARAM(wParam);  
fButton = 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

[Expand table](#)

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_RBUTTONDOWNDBLCLK message

Article • 01/22/2025

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.

C++

```
#define WM_RBUTTONDOWNDBLCLK 0x0206
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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_RBUTTONDOWNDBLCLK** 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_RBUTTONDOWNDBLCLK**, 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

[+] Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_RBUTTONDOWN message

Article • 01/22/2025

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.

C++

```
#define WM_RBUTTONDOWN 0x0204
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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_RBUTTONDOWNDBLCLK](#)

[WM_RBUTTONUP](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_RBUTTONDOWN message

Article • 11/19/2022

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.

C++

```
#define WM_RBUTTONDOWN 0x0205
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 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

[+] [Expand table](#)

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_RBUTTONDOWNDBLCLK](#)

[WM_RBUTTONDOWN](#)

Conceptual

[Mouse Input](#)

Other Resources

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_XBUTTONDOWNDBLCLK message

Article • 01/22/2025

Posted when the user double-clicks either XBUTTON1 or XBUTTON2 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.

C++

```
#define WM_XBUTTONDOWNDBLCLK 0x020D
```

Parameters

wParam

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

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 is down.

The high-order word indicates which button was double-clicked. It can be one of the following values.

[+] Expand table

Value	Meaning
XBUTTON1 0x0001	The XBUTTON1 was double-clicked.
XBUTTON2 0x0002	The XBUTTON2 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

Windows supports mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web browser.

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 **HIGHWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y-coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

Only windows that have the **CS_DBLCLKS** style can receive **WM_XBUTTONDOWNDBLCLK** messages, which the system generates whenever the user presses, releases, and again presses either XBUTTON1 or XBUTTON2 within the system's double-click time limit. Double-clicking one of these buttons actually generates four messages: **WM_XBUTTONDOWN**, **WM_XBUTTONUP**, **WM_XBUTTONDOWNDBLCLK**, and **WM_XBUTTONUP** again.

Unlike the **WM_LBUTTONDOWNDBLCLK**, **WM_MBUTTONDOWNDBLCLK**, and **WM_RBUTTONDOWNDBLCLK** 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

 Expand table

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_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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_XBUTTONDOWN message

Article • 01/22/2025

Posted when the user presses either XBUTTON1 or XBUTTON2 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.

C++

```
#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.

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 is down.

The high-order word indicates which button was clicked. It can be one of the following values.

[+] Expand table

Value	Meaning
XBUTTON1 0x0001	The XBUTTON1 was clicked.
XBUTTON2 0x0002	The XBUTTON2 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

Windows supports mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web browser.

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 **HIGHWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y-coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

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

[+] Expand table

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_XBUTTONDOWNDBLCLK](#)

[WM_XBUTTONUP](#)

[Conceptual](#)

[Mouse Input](#)

[Other Resources](#)

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_XBUTTONUP message

Article • 01/22/2025

Posted when the user releases either XBUTTON1 or XBUTTON2 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.

C++

```
#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.

[+] [Expand table](#)

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 XBUTTON1 is down.
MK_XBUTTON2 0x0040	The XBUTTON2 is down.

The high-order word indicates which button was released. It can be one of the following values:

[Expand table](#)

Value	Meaning
XBUTTON1 0x0001	The XBUTTON1 was released.
XBUTTON2 0x0002	The XBUTTON2 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

Windows supports mice with up to five buttons: left, middle, and right, plus two additional buttons called XBUTTON1 and XBUTTON2. The XBUTTON1 and XBUTTON2 buttons are often located on the sides of the mouse, near the base. These extra buttons are not present on all mice. If present, the XBUTTON1 and XBUTTON2 buttons are often mapped to an application function, such as forward and backward navigation in a Web browser.

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 **HIGHWORD** macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitors can have negative x- and y-coordinates, and **LOWORD** and **HIGHWORD** treat the coordinates as unsigned quantities.

Unlike the **WM_LBUTTONDOWN**, **WM_MBUTTONDOWN**, and **WM_RBUTTONDOWN** 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

[+] Expand table

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_XBUTTONDOWNDBLCLK](#)

[WM_XBUTTONDOWN](#)

[Conceptual](#)

[Mouse Input](#)

[Other Resources](#)

[MAKEPOINTS](#)

[POINTS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Mouse Input Structures

Article • 03/11/2025

In This Section

- [MOUSEMOVEPOINT](#)
- [TRACKMOUSEEVENT](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

MOUSEMOVEPOINT structure (winuser.h)

Article 02/22/2024

Contains information about the mouse's location in screen coordinates.

Syntax

C++

```
typedef struct tagMOUSEMOVEPOINT {
    int          x;
    int          y;
    DWORD        time;
    ULONG_PTR    dwExtraInfo;
} MOUSEMOVEPOINT, *PMOUSEMOVEPOINT, *LPMOUSEMOVEPOINT;
```

Members

x

Type: **int**

The x-coordinate of the mouse.

y

Type: **int**

The y-coordinate of the mouse.

time

Type: **DWORD**

The time stamp of the mouse coordinate.

dwExtraInfo

Type: **ULONG_PTR**

Additional information associated with this coordinate.

Requirements

 Expand table

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

Conceptual

[GetMouseMovePointsEx](#)

[Mouse Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

TRACKMOUSEEVENT structure (winuser.h)

Article04/02/2021

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.

Syntax

C++

```
typedef struct tagTRACKMOUSEEVENT {
    DWORD cbSize;
    DWORD dwFlags;
    HWND  hwndTrack;
    DWORD dwHoverTime;
} TRACKMOUSEEVENT, *LPTRACKMOUSEEVENT;
```

Members

`cbSize`

Type: **DWORD**

The size of the **TRACKMOUSEEVENT** structure, in bytes.

`dwFlags`

Type: **DWORD**

The services requested. This member can be a combination of the following values.

[+] Expand table

Value	Meaning
TME_CANCEL 0x80000000	The caller wants to cancel a prior tracking request. The caller should also specify the type of tracking that it wants to cancel. For example, to cancel hover tracking, the caller must pass the TME_CANCEL and TME_HOVER flags.

TME_HOVER 0x00000001	The caller wants hover notification. Notification is delivered as a WM_MOUSEHOVER message. If the caller requests hover tracking while hover tracking is already active, the hover timer will be reset. This flag is ignored if the mouse pointer is not over the specified window or area.
TME_LEAVE 0x00000002	The caller wants leave notification. Notification is delivered as a WM_MOUSELEAVE message. If the mouse is not over the specified window or area, a leave notification is generated immediately and no further tracking is performed.
TME_NONCLIENT 0x00000010	The caller wants hover and leave notification for the nonclient areas. Notification is delivered as WM_NCMOUSEHOVER and WM_NCMOUSELEAVE messages.
TME_QUERY 0x40000000	The function fills in the structure instead of treating it as a tracking request. The structure is filled such that had that structure been passed to TrackMouseEvent , it would generate the current tracking. The only anomaly is that the hover time-out returned is always the actual time-out and not HOVER_DEFAULT , if HOVER_DEFAULT was specified during the original TrackMouseEvent request.

hwndTrack

Type: **HWND**

A handle to the window to track.

dwHoverTime

Type: **DWORD**

The hover time-out (if **TME_HOVER** was specified in **dwFlags**), in milliseconds. Can be **HOVER_DEFAULT**, which means to use the system default hover time-out.

Remarks

The system default hover time-out is initially the menu drop-down time, which is 400 milliseconds. You can call [SystemParametersInfo](#) and use [SPI_GETMOUSEHOVERTIME](#) to retrieve the default hover time-out.

The system default hover rectangle is the same as the double-click rectangle. You can call [SystemParametersInfo](#) and use **SPI_GETMOUSEHOVERWIDTH** and **SPI_GETMOUSEHOVERHEIGHT** to retrieve the size of the rectangle within which the mouse pointer has to stay for [TrackMouseEvent](#) to generate a **WM_MOUSEOVER** message.

Requirements

[] Expand table

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

[Mouse Input](#)

Feedback

Was this page helpful?



Raw Input

Article • 08/19/2020

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

[+] Expand table

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

[+] Expand table

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

[+] Expand table

Name	Description
GET_RAWINPUT_CODE_WPARAM	Gets the input code from <i>wParam</i> in WM_INPUT .
NEXTRAWINPUTBLOCK	Gets the location of the next structure in an array of RAWINPUT structures.

Notifications

[+] Expand table

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

[+] Expand table

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.

Name	Description
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.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Raw Input Overview

Article • 08/04/2022

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](#)
- [Registration for Raw Input](#)
- [Reading Raw Input](#)

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](#).

See also

- [Keyboard Input](#)
- [About Keyboard Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Using Raw Input

Article • 03/28/2021

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.

C++

```
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;              // adds all devices from telephony page
Rid[2].dwFlags = RIDEV_PAGEONLY;
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;
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).

C++

```
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.

C++

```
case WM_INPUT:  
{  
    UINT dwSize;  
  
    GetRawInputData((HRAWINPUT)lParam, RID_INPUT, NULL, &dwSize,  
    sizeof(RAWINPUTHEADER));  
    LPBYTE lpb = new BYTE[dwSize];  
  
    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.

C++

```
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);
}
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Raw Input Reference

Article • 04/27/2021

In This Section

- [Raw Input Functions](#)
- [Raw Input Macros](#)
- [Raw Input Notifications](#)
- [Raw Input Structures](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Raw Input Functions

Article • 04/27/2021

In This Section

- [DefRawInputProc](#)
- [GetRawInputBuffer](#)
- [GetRawInputData](#)
- [GetRawInputDeviceInfo](#)
- [GetRawInputDeviceList](#)
- [GetRegisteredRawInputDevices](#)
- [RegisterRawInputDevices](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

DefRawInputProc function (winuser.h)

Article 02/22/2024

Unlike [DefWindowProcA](#) and [DefWindowProcW](#), this function doesn't do any processing.

DefRawInputProc only checks whether **cbSizeHeader**'s value corresponds to the expected size of [RAWINPUTHEADER](#).

Syntax

C++

```
LRESULT DefRawInputProc(
    [in] PRAWINPUT *paRawInput,
    [in] INT         nInput,
    [in] UINT        cbSizeHeader
);
```

Parameters

[in] paRawInput

Type: **PRAWINPUT***

Ignored.

[in] nInput

Type: **INT**

Ignored.

[in] cbSizeHeader

Type: **UINT**

The size, in bytes, of the [RAWINPUTHEADER](#) structure.

Return value

Type: **LRESULT**

If successful, the function returns 0. Otherwise it returns -1.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-rawinput-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[RAWINPUT](#)

[RAWINPUTHEADER](#)

[Raw Input](#)

Feedback

Was this page helpful?

 Yes

 No

GetRawInputBuffer function (winuser.h)

Article 08/04/2022

Performs a buffered read of the raw input messages data found in the calling thread's message queue.

Syntax

C++

```
UINT GetRawInputBuffer(
    [out, optional] PRAWINPUT pData,
    [in, out]        PUINT     pcbSize,
    [in]             UINT      cbSizeHeader
);
```

Parameters

[out, optional] pData

Type: **PRAWINPUT**

A pointer to a buffer of **RAWINPUT** structures that contain the raw input data. Buffer should be aligned on a pointer boundary, which is a **DWORD** on 32-bit architectures and a **QWORD** on 64-bit architectures.

If **NULL**, size of the first raw input message data (minimum required buffer), in bytes, is returned in ***pcbSize**.

[in, out] pcbSize

Type: **PUINT**

The size, in bytes, of the provided **RAWINPUT** buffer.

[in] cbSizeHeader

Type: **UINT**

The size, in bytes, of the **RAWINPUTHEADER** structure.

Return value

Type: **UINT**

If *pData* is **NULL** and the function is successful, the return value is zero. If *pData* is not **NULL** and the function is successful, the return value is the number of **RAWINPUT** structures written to *pData*.

If an error occurs, the return value is **(UINT)-1**. Call [GetLastError](#) for the error code.

Remarks

When an application receives raw input, its message queue gets a **WM_INPUT** message and the queue status flag **QS_RAWINPUT** is set.

Using **GetRawInputBuffer**, the raw input data is read in the array of variable size **RAWINPUT** structures and corresponding **WM_INPUT** messages are removed from the calling thread's message queue. You can call this method several times with buffer that cannot fit all message's data until all raw input messages have been read.

The **NEXTRAWINPUTBLOCK** macro allows an application to traverse an array of **RAWINPUT** structures.

If all raw input messages have been successfully read from message queue then **QS_RAWINPUT** flag is cleared from the calling thread's message queue status.

ⓘ Note

WOW64: To get the correct size of the raw input buffer, do not use **pcbSize*, use **pcbSize * 8* instead. To ensure **GetRawInputBuffer** behaves properly on WOW64, you must align the **RAWINPUT** structure by 8 bytes. The following code shows how to align **RAWINPUT** for WOW64.

C#

```
[StructLayout(LayoutKind.Explicit)]
internal struct RAWINPUT
{
    [FieldOffset(0)]
    public RAWINPUTHEADER header;

    [FieldOffset(16+8)]
    public RAWMOUSE mouse;

    [FieldOffset(16+8)]
    public RAWKEYBOARD keyboard;
```

```
[FieldOffset(16+8)]  
public RAWHID hid;  
}
```

Requirements

[] Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[GetMessage](#)

[NEXTRAWINPUTBLOCK](#)

[RAWINPUT](#)

[RAWINPUTHEADER](#)

[Raw Input](#)

Reference

[Raw Input Overview](#)

Feedback

Was this page helpful?

 Yes

 No

GetRawInputData function (winuser.h)

Article 10/13/2021

Retrieves the raw input from the specified device.

Syntax

C++

```
UINT GetRawInputData(
    [in]          HRAWINPUT hRawInput,
    [in]          UINT      uiCommand,
    [out, optional] LPVOID   pData,
    [in, out]       PUINT    pcbSize,
    [in]          UINT      cbSizeHeader
);
```

Parameters

[in] hRawInput

Type: **HRAWINPUT**

A handle to the [RAWINPUT](#) structure. This comes from the *IParam* in [WM_INPUT](#).

[in] uiCommand

Type: **UINT**

The command flag. This parameter can be one of the following values.

[+] [Expand table](#)

Value	Meaning
RID_HEADER 0x10000005	Get the header information from the RAWINPUT structure.
RID_INPUT 0x10000003	Get the raw data from the RAWINPUT structure.

[out, optional] pData

Type: **LPVOID**

A pointer to the data that comes from the [RAWINPUT](#) structure. This depends on the value of *uiCommand*. If *pData* is **NULL**, the required size of the buffer is returned in **pcbSize*.

[in, out] *pcbSize*

Type: **PUINT**

The size, in bytes, of the data in *pData*.

[in] *cbSizeHeader*

Type: **UINT**

The size, in bytes, of the [RAWINPUTHEADER](#) structure.

Return value

Type: **UINT**

If *pData* is **NULL** and the function is successful, the return value is 0. If *pData* is not **NULL** and the function is successful, the return value is the number of bytes copied into *pData*.

If there is an error, the return value is **(UINT)-1**.

Remarks

[GetRawInputData](#) gets the raw input one [RAWINPUT](#) structure at a time. In contrast, [GetRawInputBuffer](#) gets an array of [RAWINPUT](#) structures.

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows

Requirement	Value
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-rawinput-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetRawInputBuffer](#)

[RAWINPUT](#)

[RAWINPUTHEADER](#)

[Raw Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

GetRawInputDeviceInfoA function (winuser.h)

Article 02/22/2024

Retrieves information about the raw input device.

Syntax

C++

```
UINT GetRawInputDeviceInfoA(
    [in, optional]     HANDLE hDevice,
    [in]                UINT   uiCommand,
    [in, out, optional] LPVOID pData,
    [in, out]           PUINT  pcbSize
);
```

Parameters

[in, optional] `hDevice`

Type: **HANDLE**

A handle to the raw input device. This comes from the `hDevice` member of `RAWINPUTHEADER` or from `GetRawInputDeviceList`.

[in] `uiCommand`

Type: **UINT**

Specifies what data will be returned in `pData`. This parameter can be one of the following values.

[+] Expand table

Value	Meaning
<code>RIDI_PREPARSEDDATA</code> 0x20000005	<code>pData</code> is a <code>PHIDP_PREPARSED_DATA</code> pointer to a buffer for a top-level collection's prepared data.
<code>RIDI_DEVICENAME</code> 0x20000007	<code>pData</code> points to a string that contains the device interface name.

If this device is [opened with Shared Access Mode](#) then you can call [CreateFile](#) with this name to open a HID collection and use returned handle for calling [ReadFile](#) to read input reports and [WriteFile](#) to send output reports.

For more information, see [Opening HID Collections](#) and [Handling HID Reports](#).

For this *uiCommand* only, the value in *pcbSize* is the character count (not the byte count).

RIDI_DEVICEINFO
0x2000000b

pData points to an [RID_DEVICE_INFO](#) structure.

[in, out, optional] *pData*

Type: **LPVOID**

A pointer to a buffer that contains the information specified by *uiCommand*.

If *uiCommand* is **RIDI_DEVICEINFO**, set the *cbSize* member of [RID_DEVICE_INFO](#) to `sizeof(RID_DEVICE_INFO)` before calling [GetRawDeviceInfo](#).

[in, out] *pcbSize*

Type: **PUINT**

The size, in bytes, of the data in *pData*.

Return value

Type: **UINT**

If successful, this function returns a non-negative number indicating the number of bytes copied to *pData*.

If *pData* is not large enough for the data, the function returns -1. If *pData* is **NULL**, the function returns a value of zero. In both of these cases, *pcbSize* is set to the minimum size required for the *pData* buffer.

Call [GetLastError](#) to identify any other errors.

Remarks

① Note

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

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-rawinput-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

[Conceptual](#)

[RAWINPUTHEADER](#)

[RID_DEVICE_INFO](#)

[Raw Input](#)

[Reference](#)

[WM_INPUT](#)

[Top-Level Collections](#)

[Prepared Data](#)

[PHIDP_PREPARED_DATA](#)

[Opening HID collections](#)

[Handling HID Reports](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GetRawInputDeviceList function (winuser.h)

Article 05/20/2022

Enumerates the raw input devices attached to the system.

Syntax

C++

```
UINT GetRawInputDeviceList(
    [out, optional] PRAWINPUTDEVICELIST pRawInputDeviceList,
    [in, out]        PUINT             puiNumDevices,
    [in]              UINT              cbSize
);
```

Parameters

[out, optional] pRawInputDeviceList

Type: **PRAWINPUTDEVICELIST**

An array of **RAWINPUTDEVICELIST** structures for the devices attached to the system. If **NULL**, the number of devices are returned in **puiNumDevices*.

[in, out] puiNumDevices

Type: **PUINT**

If *pRawInputDeviceList* is **NULL**, the function populates this variable with the number of devices attached to the system; otherwise, this variable specifies the number of **RAWINPUTDEVICELIST** structures that can be contained in the buffer to which *pRawInputDeviceList* points. If this value is less than the number of devices attached to the system, the function returns the actual number of devices in this variable and fails with **ERROR_INSUFFICIENT_BUFFER**. If this value is greater than or equal to the number of devices attached to the system, then the value is unchanged, and the number of devices is reported as the return value.

[in] cbSize

Type: **UINT**

The size of a [RAWINPUTDEVICELIST](#) structure, in bytes.

Return value

Type: **UINT**

If the function is successful, the return value is the number of devices stored in the buffer pointed to by *pRawInputDeviceList*.

On any other error, the function returns (**UINT**) -1 and [GetLastError](#) returns the error indication.

Remarks

The devices returned from this function are the mouse, the keyboard, and other Human Interface Device (HID) devices.

To get more detailed information about the attached devices, call [GetRawInputDeviceInfo](#) using the hDevice from [RAWINPUTDEVICELIST](#).

Examples

The following sample code shows a typical call to [GetRawInputDeviceList](#):

C++

```
UINT nDevices;
PRAWINPUTDEVICELIST pRawInputDeviceList = NULL;
while (true) {
    if (GetRawInputDeviceList(NULL, &nDevices, sizeof(RAWINPUTDEVICELIST))
!= 0) { Error(); }
    if (nDevices == 0) { break; }
    if ((pRawInputDeviceList = malloc(sizeof(RAWINPUTDEVICELIST) *
nDevices)) == NULL) { Error(); }
    nDevices = GetRawInputDeviceList(pRawInputDeviceList, &nDevices,
sizeof(RAWINPUTDEVICELIST));
    if (nDevices == (UINT)-1) {
        if (GetLastError() != ERROR_INSUFFICIENT_BUFFER) { Error(); }
        // Devices were added.
        free(pRawInputDeviceList);
        continue;
    }
    break;
}
// do the job...
```

```
// after the job, free the RAWINPUTDEVICELIST  
free(pRawInputDeviceList);
```

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-rawinput-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[GetRawInputDeviceInfo](#)

[RAWINPUTDEVICELIST](#)

[Raw Input](#)

Reference

Feedback

Was this page helpful?

👍 Yes

👎 No

GetRegisteredRawInputDevices function (winuser.h)

Article 02/22/2024

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

Syntax

C++

```
UINT GetRegisteredRawInputDevices(
    [out, optional] PRAWINPUTDEVICE pRawInputDevices,
    [in, out]        PUINT          puiNumDevices,
    [in]             UINT           cbSize
);
```

Parameters

[out, optional] pRawInputDevices

Type: **PRAWINPUTDEVICE**

An array of **RAWINPUTDEVICE** structures for the application.

[in, out] puiNumDevices

Type: **PUINT**

The number of **RAWINPUTDEVICE** structures in **pRawInputDevices*.

[in] cbSize

Type: **UINT**

The size, in bytes, of a **RAWINPUTDEVICE** structure.

Return value

Type: **UINT**

If successful, the function returns a non-negative number that is the number of **RAWINPUTDEVICE** structures written to the buffer.

If the *pRawInputDevices* buffer is too small or **NULL**, the function sets the last error as **ERROR_INSUFFICIENT_BUFFER**, returns -1, and sets *puiNumDevices* to the required number of devices. If the function fails for any other reason, it returns -1. For more details, call [GetLastError](#).

Remarks

To receive raw input from a device, an application must register it by using [RegisterRawInputDevices](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

[Conceptual](#)

[RAWINPUTDEVICE](#)

[Raw Input](#)

[Reference](#)

[RegisterRawInputDevices](#)

Feedback

Was this page helpful?

 Yes

 No

RegisterRawInputDevices function (winuser.h)

Article 10/13/2021

Registers the devices that supply the raw input data.

Syntax

C++

```
BOOL RegisterRawInputDevices(
    [in] PCRAWINPUTDEVICE pRawInputDevices,
    [in] UINT             uiNumDevices,
    [in] UINT             cbSize
);
```

Parameters

[in] `pRawInputDevices`

Type: **PCRAWINPUTDEVICE**

An array of **RAWINPUTDEVICE** structures that represent the devices that supply the raw input.

[in] `uiNumDevices`

Type: **UINT**

The number of **RAWINPUTDEVICE** structures pointed to by *pRawInputDevices*.

[in] `cbSize`

Type: **UINT**

The size, in bytes, of a **RAWINPUTDEVICE** structure.

Return value

Type: **BOOL**

TRUE if the function succeeds; otherwise, **FALSE**. If the function fails, call [GetLastError](#) for more information.

Remarks

To receive [WM_INPUT](#) messages, an application must first register the raw input devices using [RegisterRawInputDevices](#). By default, an application does not receive raw input.

To receive [WM_INPUT_DEVICE_CHANGE](#) messages, an application must specify the [RIDEV_DEVNOTIFY](#) flag for each device class that is specified by the [usUsagePage](#) and [usUsage](#) fields of the [RAWINPUTDEVICE](#) structure . By default, an application does not receive [WM_INPUT_DEVICE_CHANGE](#) notifications for raw input device arrival and removal.

If a [RAWINPUTDEVICE](#) structure has the [RIDEV_REMOVE](#) flag set and the [hwndTarget](#) parameter is not set to [NULL](#), then parameter validation will fail.

Only one window per raw input device class may be registered to receive raw input within a process (the window passed in the last call to [RegisterRawInputDevices](#)). Because of this, [RegisterRawInputDevices](#) should not be used from a library, as it may interfere with any raw input processing logic already present in applications that load it.

Requirements

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-rawinput-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

Conceptual

[RAWINPUTDEVICE](#)

[Raw Input](#)

Reference

[WM_INPUT](#)

Feedback

Was this page helpful?



Raw Input Macros

Article • 04/27/2021

In This Section

- [GET_RAWINPUT_CODE_WPARAM](#)
- [NEXTRAWINPUTBLOCK](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GET_RAWINPUT_CODE_WPARAM macro (winuser.h)

Article02/22/2024

Retrieves the input code from *wParam* in [WM_INPUT](#) message.

Syntax

C++

```
void GET_RAWINPUT_CODE_WPARAM(  
    wParam  
);
```

Parameters

wParam

wParam from [WM_INPUT](#) message.

Return value

Input code value. Can be one of the following:

[+] Expand table

Value	Meaning
RIM_INPUT 0	Input occurred while the application was in the foreground.
RIM_INPUTSINK 1	Input occurred while the application was not in the foreground.

Requirements

[+] Expand table

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

Requirement	Value
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

See also

Conceptual

[RAWINPUT](#)

[Raw Input](#)

Reference

[WM_INPUT](#)

Feedback

Was this page helpful?

 Yes

 No

NEXTRAWINPUTBLOCK macro (winuser.h)

Article 02/22/2024

Retrieves the location of the next structure in an array of [RAWINPUT](#) structures.

Syntax

C++

```
void NEXTRAWINPUTBLOCK(
    ptr
);
```

Parameters

ptr

A pointer to a structure in an array of [RAWINPUT](#) structures.

Return value

None

Remarks

This macro is called repeatedly to traverse an array of [RAWINPUT](#) structures.

Requirements

[] [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows

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

See also

Conceptual

[RAWINPUT](#)

[Raw Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

Raw Input Notifications

Article • 03/11/2025

In This Section

- [WM_INPUT](#)
- [WM_INPUT_DEVICE_CHANGE](#)

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_INPUT message

Article • 12/11/2020

Sent to the window that is getting raw input.

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

C++

```
#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:

[+] [Expand table](#)

Value	Meaning
RIM_INPUT	Input occurred while the application was in the foreground.
0	The application must call DefWindowProc so the system can perform cleanup.
RIM_INPUTSINK	Input occurred while the application was not in the foreground.
1	

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

 Expand table

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

[GetRawInputData](#)

[RegisterRawInputDevices](#)

[RAWINPUT](#)

[GET_RAWINPUT_CODE_WPARAM](#)

Conceptual

[Raw Input](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

WM_INPUT_DEVICE_CHANGE message

Article • 09/10/2020

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.

C++

```
#define WM_INPUT_DEVICE_CHANGE      0x00FE
```

Parameters

wParam

Type: [WPARAM](#)

This parameter can be one of the following values.

[+] Expand table

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Raw Input Structures

Article • 04/27/2021

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](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RAWHID structure (winuser.h)

Article 02/22/2024

Describes the format of the raw input from a Human Interface Device (HID).

Syntax

C++

```
typedef struct tagRAWHID {
    DWORD dwSizeHid;
    DWORD dwCount;
    BYTE bRawData[1];
} RAWHID, *PRAWHID, *LPRAWHID;
```

Members

`dwSizeHid`

Type: **DWORD**

The size, in bytes, of each HID input in **bRawData**.

`dwCount`

Type: **DWORD**

The number of HID inputs in **bRawData**.

`bRawData[1]`

Type: **BYTE[1]**

The raw input data, as an array of bytes.

Remarks

Each [WM_INPUT](#) can indicate several inputs, but all of the inputs come from the same HID. The size of the **bRawData** array is **dwSizeHid * dwCount**.

For more information, see [Interpreting HID Reports](#).

Requirements

 Expand table

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

Conceptual

[RAWINPUT](#)

[Raw Input](#)

[Introduction to Human Interface Devices \(HID\)](#)

Reference

[WM_INPUT](#)

[Interpreting HID Reports](#)

Feedback

Was this page helpful?

 Yes

 No

RAWINPUT structure (winuser.h)

Article 02/22/2024

Contains the raw input from a device.

Syntax

C++

```
typedef struct tagRAWINPUT {
    RAWINPUTHEADER header;
    union {
        RAWMOUSE     mouse;
        RAWKEYBOARD keyboard;
        RAWHID       hid;
    } data;
} RAWINPUT, *PRAWINPUT, *LPRAWINPUT;
```

Members

header

Type: [RAWINPUTHEADER](#)

The raw input data.

data

data.mouse

Type: [RAWMOUSE](#)

If the data comes from a mouse, this is the raw input data.

data.keyboard

Type: [RAWKEYBOARD](#)

If the data comes from a keyboard, this is the raw input data.

data.hid

Type: [RAWHID](#)

If the data comes from an HID, this is the raw input data.

Remarks

The handle to this structure is passed in the *lParam* parameter of [WM_INPUT](#).

To get detailed information -- such as the header and the content of the raw input -- call [GetRawInputData](#).

To read the **RAWINPUT** in the message loop as a buffered read, call [GetRawInputBuffer](#).

To get device specific information, call [GetRawInputDeviceInfo](#) with the *hDevice* from [RAWINPUTHEADER](#).

Raw input is available only when the application calls [RegisterRawInputDevices](#) with valid device specifications.

Requirements

[+] [Expand table](#)

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

[Conceptual](#)

[GetRawInputBuffer](#)

[GetRawInputData](#)

[RAWHID](#)

[RAWINPUTHEADER](#)

[RAWKEYBOARD](#)

[RAWMOUSE](#)

Raw Input

Reference

Feedback

Was this page helpful?

 Yes

 No

RAWINPUTDEVICE structure (winuser.h)

Article04/02/2021

Defines information for the raw input devices.

Syntax

C++

```
typedef struct tagRAWINPUTDEVICE {
    USHORT usUsagePage;
    USHORT usUsage;
    DWORD dwFlags;
    HWND hwndTarget;
} RAWINPUTDEVICE, *PRAWINPUTDEVICE, *LPRAWINPUTDEVICE;
```

Members

`usUsagePage`

Type: **USHORT**

Top level collection Usage page for the raw input device. See [HID Clients Supported in Windows](#) for details on possible values.

`usUsage`

Type: **USHORT**

Top level collection Usage ID for the raw input device. See [HID Clients Supported in Windows](#) for details on possible values.

`dwFlags`

Type: **DWORD**

Mode flag that specifies how to interpret the information provided by `usUsagePage` and `usUsage`. It can be zero (the default) or one of the following values. By default, the operating system sends raw input from devices with the specified top level collection (TLC) to the registered application as long as it has the window focus.

[+] Expand table

Value	Meaning
RIDEV_REMOVE 0x00000001	If set, this removes the top level collection from the inclusion list. This tells the operating system to stop reading from a device which matches the top level collection.
RIDEV_EXCLUDE 0x00000010	If set, this specifies the top level collections to exclude when reading a complete usage page. This flag only affects a TLC whose usage page is already specified with RIDEV_PAGEONLY .
RIDEV_PAGEONLY 0x00000020	If set, this specifies all devices whose top level collection is from the specified usUsagePage . Note that usUsage must be zero. To exclude a particular top level collection, use RIDEV_EXCLUDE .
RIDEV_NOLEGACY 0x00000030	If set, this prevents any devices specified by usUsagePage or usUsage from generating legacy messages . This is only for the mouse and keyboard. See Remarks.
RIDEV_INPUTSINK 0x00000100	If set, this enables the caller to receive the input even when the caller is not in the foreground. Note that hwndTarget must be specified.
RIDEV_CAPTUREMOUSE 0x00000200	If set, the mouse button click does not activate the other window. RIDEV_CAPTUREMOUSE can be specified only if RIDEV_NOLEGACY is specified for a mouse device.
RIDEV_NOHOTKEYS 0x00000200	If set, the application-defined keyboard device hotkeys are not handled. However, the system hotkeys; for example, ALT+TAB and CTRL+ALT+DEL, are still handled. By default, all keyboard hotkeys are handled. RIDEV_NOHOTKEYS can be specified even if RIDEV_NOLEGACY is not specified and hwndTarget is NULL.
RIDEV_APPKEYS 0x00000400	If set, the application command keys are handled. RIDEV_APPKEYS can be specified only if RIDEV_NOLEGACY is specified for a keyboard device.
RIDEV_EXINPUTSINK 0x00001000	If set, this enables the caller to receive input in the background only if the foreground application does not process it. In other words, if the foreground application is not registered for raw input, then the background application that is registered will receive the input. Windows XP: This flag is not supported until Windows Vista
RIDEV_DEVNOTIFY 0x00002000	If set, this enables the caller to receive WM_INPUT_DEVICE_CHANGE notifications for device

arrival and device removal.

Windows XP: This flag is not supported until Windows Vista

hwndTarget

Type: **HWND**

A handle to the target window. If **NULL**, raw input events follow the keyboard focus to ensure only the focused application window receives the events.

Remarks

If **RIDEV_NOLEGACY** is set for a mouse or a keyboard, the system does not generate any legacy message for that device for the application. For example, if the mouse TLC is set with **RIDEV_NOLEGACY**, **WM_LBUTTONDOWN** and [related legacy mouse messages](#) are not generated. Likewise, if the keyboard TLC is set with **RIDEV_NOLEGACY**, **WM_KEYDOWN** and [related legacy keyboard messages](#) are not generated.

If **RIDEV_REMOVE** is set and the **hwndTarget** member is not set to **NULL**, then [RegisterRawInputDevices](#) function will fail.

Requirements

[+] [Expand table](#)

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

[Conceptual](#)

[GetRegisteredRawInputDevices](#)

[Raw Input](#)

[Introduction to Human Interface Devices \(HID\)](#)

HID Clients Supported in Windows

[HID USB homepage ↗](#)

Reference

[RegisterRawInputDevices](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RAWINPUTDEVICELIST structure (winuser.h)

Article 02/22/2024

Contains information about a raw input device.

Syntax

C++

```
typedef struct tagRAWINPUTDEVICELIST {  
    HANDLE hDevice;  
    DWORD dwType;  
} RAWINPUTDEVICELIST, *PRAWINPUTDEVICELIST;
```

Members

`hDevice`

Type: **HANDLE**

A handle to the raw input device.

`dwType`

Type: **DWORD**

The type of device. This can be one of the following values.

[\[\] Expand table](#)

Value	Meaning
RIM_TYPEHID 2	The device is an HID that is not a keyboard and not a mouse.
RIM_TYPEKEYBOARD 1	The device is a keyboard.
RIM_TYPEMOUSE 0	The device is a mouse.

Requirements

 Expand table

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

Conceptual

[GetRawInputDeviceList](#)

[Raw Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

RAWINPUTHEADER structure (winuser.h)

Article 02/22/2024

Contains the header information that is part of the raw input data.

Syntax

C++

```
typedef struct tagRAWINPUTHEADER {
    DWORD  dwType;
    DWORD  dwSize;
    HANDLE hDevice;
    WPARAM wParam;
} RAWINPUTHEADER, *PRAWINPUTHEADER, *LPRAWINPUTHEADER;
```

Members

`dwType`

Type: **DWORD**

The type of raw input. It can be one of the following values:

[+] Expand table

Value	Meaning
RIM_TYPEMOUSE 0	Raw input comes from the mouse.
RIM_TYPEKEYBOARD 1	Raw input comes from the keyboard.
RIM_TYPEHID 2	Raw input comes from some device that is not a keyboard or a mouse.

`dwSize`

Type: **DWORD**

The size, in bytes, of the entire input packet of data. This includes [RAWINPUT](#) plus possible extra input reports in the [RAWHID](#) variable length array.

`hDevice`

Type: **HANDLE**

A handle to the device generating the raw input data.

wParam

Type: **WPARAM**

The value passed in the *wParam* parameter of the [WM_INPUT](#) message.

Remarks

To get more information on the device, use **hDevice** in a call to [GetRawInputDeviceInfo](#). **hDevice** can be zero if an input is received from a precision touchpad.

Requirements

[+] [Expand table](#)

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

[Conceptual](#)

[GetRawInputDeviceInfo](#)

[RAWINPUT structure](#)

[RAWKEYBOARD structure](#)

[RAWMOUSE structure](#)

[RAWHID structure](#)

[Raw Input](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

RAWKEYBOARD structure (winuser.h)

Article 06/26/2024

Contains information about the state of the keyboard.

Syntax

C++

```
typedef struct tagRAWKEYBOARD {
    USHORT MakeCode;
    USHORT Flags;
    USHORT Reserved;
    USHORT VKey;
    UINT   Message;
    ULONG  ExtraInformation;
} RAWKEYBOARD, *PRAWKEYBOARD, *LPRAWKEYBOARD;
```

Members

MakeCode

Type: **USHORT**

Specifies the [scan code](#) associated with a key press. See Remarks.

Flags

Type: **USHORT**

Flags for scan code information. It can be one or more of the following:

[+] Expand table

Value	Meaning
RI_KEY_MAKE 0	The key is down.
RI_KEY_BREAK 1	The key is up.
RI_KEY_E0 2	The scan code has the E0 prefix.
RI_KEY_E1 4	The scan code has the E1 prefix.

Reserved

Type: **USHORT**

Reserved; must be zero.

VKey

Type: **USHORT**

The corresponding legacy virtual-key code.

Message

Type: **UINT**

The corresponding legacy keyboard window message, for example [WM_KEYDOWN](#), [WM_SYSKEYDOWN](#), and so forth.

ExtraInformation

Type: **ULONG**

The device-specific additional information for the event.

Remarks

KEYBOARD_OVERRUN_MAKE_CODE is a special **MakeCode** value sent when an invalid or unrecognizable combination of keys is pressed or the number of keys pressed exceeds the limit for this keyboard.

C++

```
case WM_INPUT:  
{  
    UINT dwSize = sizeof(RAWINPUT);  
    static BYTE lpb[sizeof(RAWINPUT)];  
  
    GetRawInputData((HRAWINPUT)lParam, RID_INPUT, lpb, &dwSize,  
    sizeof(RAWINPUTHEADER));  
  
    RAWINPUT* raw = (RAWINPUT*)lpb;  
  
    if (raw->header.dwType == RIM_TYPEKEYBOARD)  
    {  
        RAWKEYBOARD& keyboard = raw->data.keyboard;  
        WORD scanCode = 0;  
        BOOL keyUp = keyboard.Flags & RI_KEY_BREAK;
```

```

        // Ignore key overrun state and keys not mapped to any virtual key
        code
        if (keyboard.MakeCode == KEYBOARD_OVERRUN_MAKE_CODE || keyboard.VKey
>= UCHAR_MAX)
            return 0;

        if (keyboard.MakeCode)
        {
            // Compose the full scan code value with its extended byte
            scanCode = MAKEWORD(keyboard.MakeCode & 0x7f, ((keyboard.Flags &
RI_KEY_E0) ? 0xe0 : ((keyboard.Flags & RI_KEY_E1) ? 0xe1 : 0x00)));
        }
        else
        {
            // Scan code value may be empty for some buttons (for example
            multimedia buttons)
            // Try to get the scan code from the virtual key code
            scanCode = LOWORD(MapVirtualKey(keyboard.VKey,
MAPVK_VK_TO_VSC_EX));
        }

        // Get the key name for debug output
        TCHAR keyNameBuffer[MAX_PATH] = {};
        GetKeyNameText((LONG)MAKELPARAM(0, (HIBYTE(scanCode) ? KF_EXTENDED :
0x00) | LOBYTE(scanCode)), keyNameBuffer, MAX_PATH);

        // Debug output
        TCHAR printBuffer[MAX_PATH] = {};
        StringCchPrintf(printBuffer, MAX_PATH, TEXT("Keyboard: scanCode=%04x
keyName=%s\r\n"), scanCode, keyNameBuffer);
        OutputDebugString(printBuffer);
    }
    ...

    return 0;
}

```

Requirements

[\[+\] Expand table](#)

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

- [GetRawInputDeviceInfo](#)
 - [RAWINPUT](#)
 - [Raw Input](#)
 - [Keyboard and mouse HID client drivers](#)
 - [KEYBOARD_INPUT_DATA structure](#)
 - [Keyboard Input](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RAWMOUSE structure (winuser.h)

Article 08/04/2022

Contains information about the state of the mouse.

Syntax

C++

```
typedef struct tagRAWMOUSE {
    USHORT usFlags;
    union {
        ULONG ulButtons;
        struct {
            USHORT usButtonFlags;
            USHORT usButtonData;
        } DUMMYSTRUCTNAME;
    } DUMMYUNIONNAME;
    ULONG ulRawButtons;
    LONG lLastX;
    LONG lLastY;
    ULONG ulExtraInformation;
} RAWMOUSE, *PRAWMOUSE, *LPRAWMOUSE;
```

Members

usFlags

Type: **USHORT**

The mouse state. This member can be any reasonable combination of the following.

[+] Expand table

Value	Meaning
MOUSE_MOVE_RELATIVE 0x00	Mouse movement data is relative to the last mouse position. For further information about mouse motion, see the following Remarks section.
MOUSE_MOVE_ABSOLUTE 0x01	Mouse movement data is based on absolute position. For further information about mouse motion, see the following Remarks section.

Value	Meaning
MOUSE_VIRTUAL_DESKTOP 0x02	Mouse coordinates are mapped to the virtual desktop (for a multiple monitor system). For further information about mouse motion, see the following Remarks section.
MOUSE_ATTRIBUTES_CHANGED 0x04	Mouse attributes changed; application needs to query the mouse attributes.
MOUSE_MOVE_NO_COALESCE 0x08	This mouse movement event was not coalesced. Mouse movement events can be coalesced by default. Windows XP/2000: This value is not supported.

DUMMYUNIONNAME

DUMMYUNIONNAME.ulButtons

Type: **ULONG**

Reserved.

DUMMYUNIONNAME.DUMMYSTRUCTNAME

DUMMYUNIONNAME.DUMMYSTRUCTNAME.usButtonFlags

Type: **USHORT**

The transition state of the mouse buttons. This member can be one or more of the following values.

[Expand table](#)

Value	Meaning
RI_MOUSE_BUTTON_1_DOWN RI_MOUSE_LEFT_BUTTON_DOWN 0x0001	Left button changed to down.
RI_MOUSE_BUTTON_1_UP RI_MOUSE_LEFT_BUTTON_UP 0x0002	Left button changed to up.
RI_MOUSE_BUTTON_2_DOWN RI_MOUSE_RIGHT_BUTTON_DOWN 0x0004	Right button changed to down.
RI_MOUSE_BUTTON_2_UP RI_MOUSE_RIGHT_BUTTON_UP 0x0008	Right button changed to up.

Value	Meaning
RI_MOUSE_BUTTON_3_DOWN RI_MOUSE_MIDDLE_BUTTON_DOWN 0x0010	Middle button changed to down.
RI_MOUSE_BUTTON_3_UP RI_MOUSE_MIDDLE_BUTTON_UP 0x0020	Middle button changed to up.
RI_MOUSE_BUTTON_4_DOWN 0x0040	XBUTTON1 changed to down.
RI_MOUSE_BUTTON_4_UP 0x0080	XBUTTON1 changed to up.
RI_MOUSE_BUTTON_5_DOWN 0x0100	XBUTTON2 changed to down.
RI_MOUSE_BUTTON_5_UP 0x0200	XBUTTON2 changed to up.
RI_MOUSE_WHEEL 0x0400	<p>Raw input comes from a mouse wheel. The wheel delta is stored in usButtonData. 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. For further information see the following Remarks section.</p>
RI_MOUSE_HWHEEL 0x0800	<p>Raw input comes from a horizontal mouse wheel. The wheel delta is stored in usButtonData. A positive value indicates that the wheel was rotated to the right; a negative value indicates that the wheel was rotated to the left. For further information see the following Remarks section.</p> <p>Windows XP/2000: This value is not supported.</p>

DUMMYUNIONNAME.DUMMYSTRUCTNAME.usButtonData

Type: **USHORT**

If **usButtonFlags** has **RI_MOUSE_WHEEL** or **RI_MOUSE_HWHEEL**, this member specifies the distance the wheel is rotated. For further information see the following Remarks section.

ulRawButtons

Type: **ULONG**

The raw state of the mouse buttons. The Win32 subsystem does not use this member.

ILastX

Type: **LONG**

The motion in the X direction. This is signed relative motion or absolute motion, depending on the value of **usFlags**.

ILastY

Type: **LONG**

The motion in the Y direction. This is signed relative motion or absolute motion, depending on the value of **usFlags**.

ulExtraInformation

Type: **ULONG**

Additional device-specific information for the event. See [Distinguishing Pen Input from Mouse and Touch](#) for more info.

Remarks

If the mouse has moved, indicated by **MOUSE_MOVE_RELATIVE** or **MOUSE_MOVE_ABSOLUTE**, **ILastX** and **ILastY** specify information about that movement. The information is specified as relative or absolute integer values.

If **MOUSE_MOVE_RELATIVE** value is specified, **ILastX** and **ILastY** specify movement relative to the previous mouse event (the last reported position). Positive values mean the mouse moved right (or down); negative values mean the mouse moved left (or up).

If **MOUSE_MOVE_ABSOLUTE** value is specified, **ILastX** and **ILastY** contain normalized absolute coordinates between 0 and 65,535. Coordinate (0,0) maps onto the upper-left corner of the display surface; coordinate (65535,65535) maps onto the lower-right corner. In a multimonitor system, the coordinates map to the primary monitor.

If **MOUSE_VIRTUAL_DESKTOP** is specified in addition to **MOUSE_MOVE_ABSOLUTE**, the coordinates map to the entire virtual desktop.

C++

```
case WM_INPUT:  
{  
    UINT dwSize = sizeof(RAWINPUT);
```

```

static BYTE lpb[sizeof(RAWINPUT)];
GetRawInputData((HRAWINPUT)lParam, RID_INPUT, lpb, &dwSize,
sizeof(RAWINPUTHEADER));

RAWINPUT* raw = (RAWINPUT*)lpb;

if (raw->header.dwType == RIM_TYPEMOUSE)
{
    RAWMOUSE& mouse = raw->data.mouse;

    if (mouse.usFlags & MOUSE_MOVE_ABSOLUTE)
    {
        RECT rect;
        if (mouse.usFlags & MOUSE_VIRTUAL_DESKTOP)
        {
            rect.left = GetSystemMetrics(SM_XVIRTUALSCREEN);
            rect.top = GetSystemMetrics(SM_YVIRTUALSCREEN);
            rect.right = GetSystemMetrics(SM_CXVIRTUALSCREEN);
            rect.bottom = GetSystemMetrics(SM_CYVIRTUALSCREEN);
        }
        else
        {
            rect.left = 0;
            rect.top = 0;
            rect.right = GetSystemMetrics(SM_CXSCREEN);
            rect.bottom = GetSystemMetrics(SM_CYSCREEN);
        }

        int absoluteX = MulDiv(mouse.lLastX, rect.right, USHRT_MAX) +
rect.left;
        int absoluteY = MulDiv(mouse.lLastY, rect.bottom, USHRT_MAX) +
rect.top;
        ...
    }
    else if (mouse.lLastX != 0 || mouse.lLastY != 0)
    {
        int relativeX = mouse.lLastX;
        int relativeY = mouse.lLastY;
        ...
    }
    ...
}

return 0;
}

```

In contrast to legacy [WM_MOUSEMOVE](#) window messages Raw Input mouse events is not subject to the effects of the mouse speed set in the Control Panel's **Mouse Properties** sheet. See [Mouse Input Overview](#) for details.

If mouse wheel is moved, indicated by **RI_MOUSE_WHEEL** or **RI_MOUSE_HWHEEL** in **usButtonFlags**, then **usButtonData** contains a signed **short** value that specifies the distance the wheel is rotated.

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.

The application could also retrieve the current lines-to-scroll and characters-to-scroll user setting by using the [SystemParametersInfo](#) API with **SPI_GETWHEELSCROLLLINES** or **SPI_GETWHEELSCROLLCHARS** parameter.

Here is example of such wheel handling code:

C++

```
RAWMOUSE& mouse = raw->data.mouse;

if ((mouse.usButtonFlags & RI_MOUSE_WHEEL) || (mouse.usButtonFlags &
RI_MOUSE_HWHEEL))
{
    short wheelDelta = (short)mouse.usButtonData;
    float scrollDelta = (float)wheelDelta / WHEEL_DELTA;

    if (mouse.usButtonFlags & RI_MOUSE_HWHEEL) // Horizontal
    {
        unsigned long scrollChars = 1; // 1 is the default
        SystemParametersInfo(SPI_GETWHEELSCROLLCHARS, 0, &scrollChars, 0);
        scrollDelta *= scrollChars;
        ...
    }
    else // Vertical
    {
        unsigned long scrollLines = 3; // 3 is the default
        SystemParametersInfo(SPI_GETWHEELSCROLLLINES, 0, &scrollLines, 0);
        if (scrollLines != WHEEL_PAGESCROLL)
            scrollDelta *= scrollLines;
        ...
    }
}
```

Requirements

[Expand table](#)

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

Conceptual

[GetRawInputDeviceInfo](#)

[RAWINPUT](#)

[Raw Input](#)

Reference

[MOUSEINPUT structure](#)

[SendInput function](#)

[MOUSE_INPUT_DATA structure](#)

[Mouse Input Overview \(legacy\)](#)

[Mouse Input Notifications \(legacy\)](#)

[SystemParametersInfo](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

RID_DEVICE_INFO structure (winuser.h)

Article 02/22/2024

Defines the raw input data coming from any device.

Syntax

C++

```
typedef struct tagRID_DEVICE_INFO {
    DWORD cbSize;
    DWORD dwType;
    union {
        RID_DEVICE_INFO_MOUSE     mouse;
        RID_DEVICE_INFO_KEYBOARD keyboard;
        RID_DEVICE_INFO_HID       hid;
    } DUMMYUNIONNAME;
} RID_DEVICE_INFO, *PRID_DEVICE_INFO, *LPRID_DEVICE_INFO;
```

Members

`cbSize`

Type: **DWORD**

The size, in bytes, of the **RID_DEVICE_INFO** structure.

`dwType`

Type: **DWORD**

The type of raw input data. This member can be one of the following values.

[] [Expand table](#)

Value	Meaning
RIM_TYPEMOUSE 0	Data comes from a mouse.
RIM_TYPEKEYBOARD 1	Data comes from a keyboard.
RIM_TYPEHID	Data comes from an HID that is not a keyboard or a

DUMMYUNIONNAME

DUMMYUNIONNAME.mouse

Type: [RID_DEVICE_INFO_MOUSE](#)

If dwType is RIM_TYPEMOUSE, this is the [RID_DEVICE_INFO_MOUSE](#) structure that defines the mouse.

DUMMYUNIONNAME.keyboard

Type: [RID_DEVICE_INFO_KEYBOARD](#)

If dwType is RIM_TYPEKEYBOARD, this is the [RID_DEVICE_INFO_KEYBOARD](#) structure that defines the keyboard.

DUMMYUNIONNAME.hid

Type: [RID_DEVICE_INFO_HID](#)

If dwType is RIM_TYPEHID, this is the [RID_DEVICE_INFO_HID](#) structure that defines the HID device.

Requirements

 [Expand table](#)

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

[Conceptual](#)

[GetRawInputDeviceInfo](#)

[RID_DEVICE_INFO_HID](#)

[RID_DEVICE_INFO_KEYBOARD](#)

[RID_DEVICE_INFO_MOUSE](#)

[Raw Input](#)

[Reference](#)

Feedback

Was this page helpful?

 Yes

 No

RID_DEVICE_INFO_HID structure (winuser.h)

Article 02/22/2024

Defines the raw input data coming from the specified Human Interface Device (HID).

Syntax

C++

```
typedef struct tagRID_DEVICE_INFO_HID {
    DWORD dwVendorId;
    DWORD dwProductId;
    DWORD dwVersionNumber;
    USHORT usUsagePage;
    USHORT usUsage;
} RID_DEVICE_INFO_HID, *PRID_DEVICE_INFO_HID;
```

Members

`dwVendorId`

Type: **DWORD**

The vendor identifier for the HID.

`dwProductId`

Type: **DWORD**

The product identifier for the HID.

`dwVersionNumber`

Type: **DWORD**

The version number for the HID.

`usUsagePage`

Type: **USHORT**

The top-level collection Usage Page for the device.

usUsage

Type: USHORT

The top-level collection Usage for the device.

Requirements

 Expand table

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

Conceptual

[RID_DEVICE_INFO](#)

[Raw Input](#)

Reference

Feedback

Was this page helpful?

 Yes

 No

RID_DEVICE_INFO_KEYBOARD structure (winuser.h)

Article 02/22/2024

Defines the raw input data coming from the specified keyboard.

Syntax

C++

```
typedef struct tagRID_DEVICE_INFO_KEYBOARD {
    DWORD dwType;
    DWORD dwSubType;
    DWORD dwKeyboardMode;
    DWORD dwNumberOfFunctionKeys;
    DWORD dwNumberOfIndicators;
    DWORD dwNumberOfKeysTotal;
} RID_DEVICE_INFO_KEYBOARD, *PRID_DEVICE_INFO_KEYBOARD;
```

Members

dwType

Type: **DWORD**

The type of keyboard. See [Remarks](#).

[] [Expand table](#)

Value	Description
0x4	Enhanced 101- or 102-key keyboards (and compatibles)
0x7	Japanese Keyboard
0x8	Korean Keyboard
0x51	Unknown type or HID keyboard

dwSubType

Type: **DWORD**

The vendor-specific subtype of the keyboard. See [Remarks](#).

`dwKeyboardMode`

Type: **DWORD**

The scan code mode. Usually 1, which means that *Scan Code Set 1* is used. See [Keyboard Scan Code Specification](#).

`dwNumberOfFunctionKeys`

Type: **DWORD**

The number of function keys on the keyboard.

`dwNumberOfIndicators`

Type: **DWORD**

The number of LED indicators on the keyboard.

`dwNumberOfKeysTotal`

Type: **DWORD**

The total number of keys on the keyboard.

Remarks

For information about keyboard types, subtypes, scan code modes, and related keyboard layouts, see the documentation in *kbd.h*, *ntdd8042.h* and *ntddkbd.h* headers in Windows SDK, and the [Keyboard Layout Samples](#).

Requirements

[] [Expand table](#)

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

[Conceptual](#)

[RID_DEVICE_INFO](#)

[Raw Input](#)

[Reference](#)

[KEYBOARD_ATTRIBUTES structure](#)

Feedback

Was this page helpful?

 Yes

 No

RID_DEVICE_INFO_MOUSE structure (winuser.h)

Article 02/22/2024

Defines the raw input data coming from the specified mouse.

Syntax

C++

```
typedef struct tagRID_DEVICE_INFO_MOUSE {
    DWORD dwId;
    DWORD dwNumberOfButtons;
    DWORD dwSampleRate;
    BOOL fHasHorizontalWheel;
} RID_DEVICE_INFO_MOUSE, *PRID_DEVICE_INFO_MOUSE;
```

Members

dwId

Type: **DWORD**

The bitfield of the mouse device identification properties:

[+] Expand table

Value	ntddmou.h constant	Description
0x0080	MOUSE_HID_HARDWARE	HID mouse
0x0100	WHEELMOUSE_HID_HARDWARE	HID wheel mouse
0x8000	HORIZONTAL_WHEEL_PRESENT	Mouse with horizontal wheel

dwNumberOfButtons

Type: **DWORD**

The number of buttons for the mouse.

dwSampleRate

Type: **DWORD**

The number of data points per second. This information may not be applicable for every mouse device.

`fHasHorizontalWheel`

Type: **BOOL**

TRUE if the mouse has a wheel for horizontal scrolling; otherwise, **FALSE**.

Windows XP: This member is only supported starting with Windows Vista.

Remarks

For the mouse, the Usage Page is 1 and the Usage is 2.

Requirements

[] [Expand table](#)

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

[Conceptual](#)

[RID_DEVICE_INFO](#)

[Raw Input](#)

[Reference](#)

Feedback

[Was this page helpful?](#)

 Yes

 No