MOUSE

The mouse is a pointing device with one or more buttons that allows users to interact with computers.



Despite the emergence of alternative input devices like touchscreens and light pens, the mouse remains the primary input device for desktop computers.



This is due to its versatility, ease of use, and affordability.

Windows 98 supports mice with one, two, or three buttons. It can also use joysticks or light pens to mimic mouse input. In the early days of Windows, applications were designed to work with one-button mice, as many users didn't have two-button mice.



However, two-button mice have become the standard, and most applications now utilize the second button for various functions, such as invoking context menus or performing special drag operations.

Determining Mouse Presence and Button Count

To determine if a mouse is present, you can use the GetSystemMetrics function with the SM\_MOUSEPRESENT parameter.

However, this function always returns TRUE in Windows 98, regardless of whether a mouse is attached. To get accurate information, use this function in Microsoft Windows NT.

To determine the number of buttons on the installed mouse, use the GetSystemMetrics function with the SM\_CMOUSEBUTTONS parameter.

This function should also return 0 if a mouse is not installed. However, in Windows 98, the function returns 2 if a mouse is not installed.

Left-Handed Mouse Users

Left-handed users can switch the mouse buttons using the Windows Control Panel. While an application can determine this by calling GetSystemMetrics with the SM\_SWAPBUTTON parameter, this is usually unnecessary. The button triggered by the index finger is considered the left button, even if it's physically on the right side of the mouse.

Setting Mouse Parameters

You can set other mouse parameters, such as the double-click speed, using the SystemParametersInfo function. This function allows you to set or obtain various mouse-related settings from within your Windows application.

Fun facts:

* The mouse cursor is a small bitmapped picture that moves on the display as the user moves the mouse.
* The hot spot is the single-pixel point on the cursor that indicates the precise location on the display.
* Windows supports several predefined mouse cursors, such as IDC\_ARROW, IDC\_CROSS, and IDC\_WAIT.
* Programmers can also design their own custom cursors.
* The default cursor for a particular window is specified when defining the window class structure.
* Common mouse actions include clicking, double-clicking, and dragging.
* On a three-button mouse, the buttons are called the left button, the middle button, and the right button.
* On a two-button mouse, there is only a left button and a right button.
* The single button on a one-button mouse is a left button.
* The plural of "mouse" is a matter of debate, with both "mice" and "mouses" being considered acceptable.
* The Microsoft Manual of Style for Technical Publications recommends avoiding the plural "mice" and using "mouse devices" instead.

Overview

Client-area mouse messages are notifications sent by Windows to a window's procedure when mouse events occur within the window's client area. These messages provide information about the mouse's position, button state, and modifier keys.

Mouse Messages vs. Keyboard Messages

Unlike keyboard messages, which are only sent to the window that has the input focus, mouse messages are sent to any window that the mouse cursor passes over or clicks within, regardless of whether the window is active or has the input focus. This allows windows to respond to mouse interactions even when they are not in the foreground.

Types of Mouse Messages

Windows defines 21 mouse messages, but only 10 of them relate to the client area. These messages can be categorized into three types:

* Mouse movement: The WM\_MOUSEMOVE message is sent when the mouse cursor moves within the client area.
* Button press/release: When a mouse button is pressed or released within the client area, the window procedure receives one of the messages shown in the table in the text.
* Double-click: Double-click messages are sent only if the window class has been defined to receive them.

Extracting Mouse Position

The value of lParam in the client-area mouse messages contains the position of the mouse cursor. The low word is the x-coordinate, and the high word is the y-coordinate, both relative to the upper-left corner of the client area. These values can be extracted using the LOWORD and HIWORD macros.

Extracting Mouse Button State and Modifier Keys

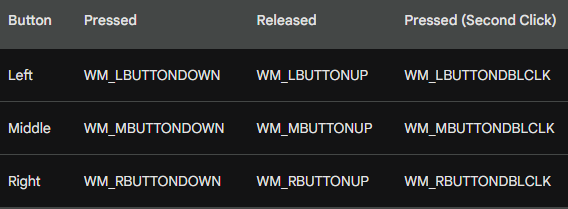
The value of wParam in the client-area mouse messages indicates the state of the mouse buttons and the Shift and Ctrl keys. These states can be tested using the bit masks defined in the WINUSER.H header file, which have the prefix "MK" for "mouse key".

WM\_LBUTTONDOWN Message and Active Window

Clicking the left mouse button in the client area of an inactive window causes Windows to make the clicked window active and then send the WM\_LBUTTONDOWN message to the window procedure. This allows the window to respond to the click even if it was not previously active.

WM\_LBUTTONDOWN and WM\_LBUTTONUP Messages

A window procedure may receive a WM\_LBUTTONDOWN message without a corresponding WM\_LBUTTONUP message, or vice versa. This can happen if the mouse button is pressed or released outside the window's client area.



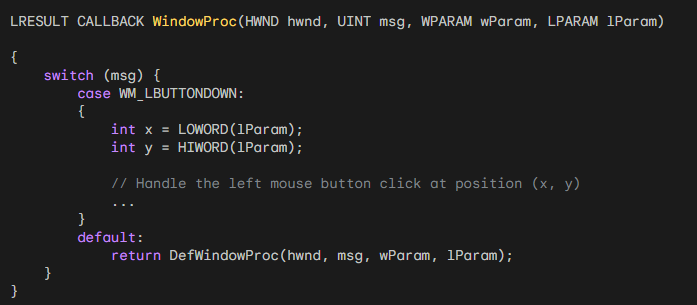
Mouse Capture

A window procedure can capture the mouse and continue to receive mouse messages even when the mouse is outside the window's client area. This is useful for operations that require continuous mouse tracking, such as drawing or dragging.

System Modal Message Boxes and Dialog Boxes

When a system modal message box or dialog box is on the display, no other program can receive mouse messages. These modal boxes prevent switching to another window while they are active.

Here's an example of how to handle the WM\_LBUTTONDOWN message in C code using WinAPI:



This code snippet defines a window procedure function called WindowProc that handles the WM\_LBUTTONDOWN message. When the left mouse button is pressed within the window's client area, the function extracts the mouse coordinates (x, y) from the lParam parameter and performs the corresponding action.

Connect folder in Chapter 7 has the code. Here’s the video for it’s working…



The CONNECT program is a simple mouse-driven demo program that allows users to connect dots on the screen. The program processes three mouse messages:

* WM\_LBUTTONDOWN: Clears the client area.
* WM\_MOUSEMOVE: If the left mouse button is down, draws a black dot on the client area at the mouse position and saves the coordinates.
* WM\_LBUTTONUP: Connects every dot shown in the client area to every other dot.

Notes:

* The program uses three GDI function calls: SetPixel, MoveToEx, and LineTo.
* The program stores a maximum of 1000 points.
* The program switches to an hourglass cursor while processing the WM\_PAINT message.
* The program calls ShowCursor twice to change the cursor visibility.
* The term "tracking" refers to the way programs handle mouse movement.

Additional Points

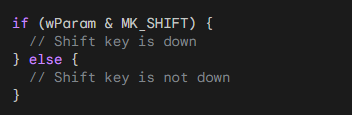
* The CONNECT program works best for a curved pattern of a few dots.
* If you move the mouse cursor out of the client area before releasing the button, CONNECT does not connect the dots.
* You can continue a design after releasing the button outside the client area by pressing the left button again while the mouse is outside the client area and then moving the mouse back inside.
* The CONNECT program might take some time to draw the lines, depending on your hardware.
* Because CONNECT is a preemptive multitasking environment, you can switch to other programs while the program is busy.

Processing Shift Keys with wParam

The CONNECT program utilizes the wParam value to determine the state of the Shift keys when handling the WM\_MOUSEMOVE message. This value is obtained from the mouse message and provides information about the mouse button presses and the Shift and Ctrl keys.

To check if the Shift key is pressed, you can perform a bitwise AND operation between wParam and MK\_SHIFT. The MK\_SHIFT constant represents the state of the Shift key. If the result of the operation is non-zero (TRUE), then the Shift key is down.

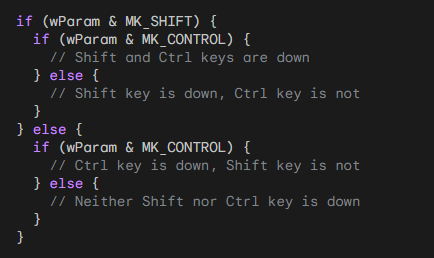
*Here's an example of how to check if the Shift key is pressed:*



Advanced Shift Key Handling

You can also use wParam to check for specific combinations of keys, such as Shift and Ctrl together. For instance, if you need to differentiate between Shift, Ctrl, and both Shift and Ctrl being pressed, you can use nested if-else statements to handle each case separately.

*Here's an example of how to check for Shift and Ctrl key combinations:*

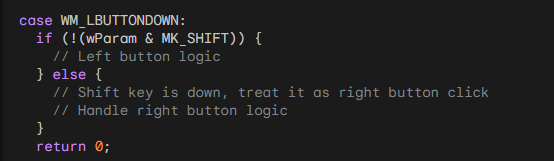


Emulating Right Button Click with Shift and Left Button

If you want to support both left and right mouse buttons in your program and accommodate users with a one-button mouse, you can make the Shift key in combination with the left button act like the right button.

This can be done by checking for the Shift key state in the WM\_LBUTTONDOWN message handler and then handling it accordingly.

*Here's an example of how to emulate the right button click:*



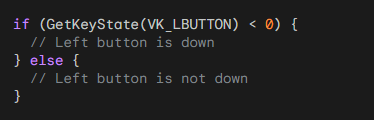
Using GetKeyState for Mouse and Key States

The GetKeyState function can also be used to retrieve the state of the mouse buttons or shift keys using the virtual key codes VK\_LBUTTON, VK\_RBUTTON, VK\_MBUTTON, VK\_SHIFT, and VK\_CONTROL. If the value returned from GetKeyState is negative, the corresponding button or key is down.

Unlike wParam, GetKeyState provides the current state of the mouse buttons or keys, even if they were pressed in a previous message. This allows you to check the state of a button or key at any point during message processing.

However, it's important to note that GetKeyState should not be used to wait for a button or key press. Instead, you should rely on message-based processing and handle button or key presses within the respective message handlers.

Here's an example of how to check the state of the left button using GetKeyState:



In summary, processing Shift keys and mouse button states in Windows applications can be achieved using both wParam and GetKeyState. wParam provides information about the state of the buttons and keys within the current message, while GetKeyState provides the current state of the buttons and keys, regardless of the current message.

Understanding Mouse Double-Clicks

A mouse double-click is a common interaction technique in Windows applications. It involves quickly clicking the mouse button twice in close proximity, typically within a specified time interval called the "double-click speed." The default double-click speed is set by the system, but users can modify it through the Control Panel.

Handling Double-Click Messages

To enable your window procedure to receive double-click messages, you must include the CS\_DBLCLKS flag in the window class style when registering the window class.

This flag instructs the system to send WM\_LBUTTONDBLCLK messages to your window procedure instead of generating separate WM\_LBUTTONDOWN messages for each click.

Default Double-Click Behavior

If you include CS\_DBLCLKS in the window class style, the window procedure receives the following messages for a double-click:

WM\_LBUTTONDOWN: This message indicates the first click of the double-click.

WM\_LBUTTONUP: This message indicates the release of the mouse button after the first click.

WM\_LBUTTONDBLCLK: This message replaces the second WM\_LBUTTONDOWN message and signals that a double-click has occurred.

WM\_LBUTTONUP: This message indicates the release of the mouse button after the double-click.

Processing Double-Clicks

When implementing double-click logic, it's often advantageous for the first click to perform the same action as a single click.

This allows users to perform the single-click action without worrying about accidentally triggering a double-click. The second click (the WM\_LBUTTONDBLCLK message) can then perform an additional action.

For instance, consider how double-clicks are handled in Windows Explorer. A single click selects a file, highlighting it with a reverse-video bar.

A double-click performs two actions: it selects the file like a single click, and it also directs Explorer to open the file. This design is straightforward and user-friendly.

Complex Double-Click Logic

Handling double-clicks becomes more complex if the first click does not perform the same action as a single click.

In such cases, the window procedure needs to track click events and distinguish between single clicks and double-clicks based on the time interval between clicks.

This can involve using the GetMessageTime function to obtain the relative times of WM\_LBUTTONDOWN messages.

Understanding Nonclient-Area Mouse Messages

Windows applications receive mouse messages when the user interacts with the mouse within the window's client area.

However, if the mouse interaction occurs within the window's nonclient area, which includes the title bar, menu, and window scroll bars, Windows sends a different set of messages called nonclient-area mouse messages.

Purpose of Nonclient-Area Mouse Messages

Nonclient-area mouse messages are primarily used for system functions, such as resizing windows, minimizing or maximizing windows, and dragging windows around the screen.

Typically, you don't need to directly process these messages in your application's window procedure. Instead, you can pass them to DefWindowProc to allow Windows to handle the default system behavior.

Similarity to System Keyboard Messages

Nonclient-area mouse messages share similarities with the system keyboard messages WM\_SYSKEYDOWN, WM\_SYSKEYUP, and WM\_SYSCHAR. These messages are also handled by DefWindowProc to perform system-level actions in response to keyboard events.

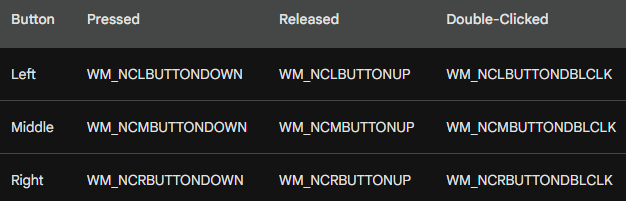
Nonclient-Area Mouse Messages vs. Client-Area Mouse Messages

The nonclient-area mouse messages closely parallel the client-area mouse messages. However, they are distinguished by the prefix "NC" in their message identifiers.

For instance, when the mouse moves within a nonclient area, the window procedure receives WM\_NCMOUSEMOVE, which corresponds to WM\_MOUSEMOVE for client-area mouse movements.

Nonclient-Area Mouse Messages for Mouse Buttons

The nonclient-area mouse messages for mouse button presses and releases follow a similar pattern:



wParam and lParam Parameters for Nonclient-Area Mouse Messages

The wParam and lParam parameters for nonclient-area mouse messages differ slightly from those for client-area mouse messages.

The wParam parameter indicates the specific nonclient area where the mouse interaction occurred. It is set to one of the identifiers defined in the WINUSER.H header file, starting with the prefix "HT" (for "hit-test").

The lParam parameter contains the screen coordinates of the mouse position, with the:

x-coordinate in the low word and the y-coordinate in the high word.

These coordinates are based on the entire screen, not just the window's client area.

Converting Screen Coordinates to Client-Area Coordinates

To convert screen coordinates to client-area coordinates and vice versa, you can use the Windows functions ScreenToClient and ClientToScreen. These functions take a POINT structure as input and modify its coordinates accordingly.

Summary

Nonclient-area mouse messages are used for system-level interactions with a window's nonclient area. They are typically handled by DefWindowProc to maintain the default system behavior. Understanding these messages and their parameters is essential for creating applications that interact seamlessly with the Windows environment.