KEYBOARD

The keyboard and mouse are the two primary means of user input in Microsoft Windows 98. While the mouse has become increasingly dominant in modern applications, the keyboard remains an essential component of personal computers.

The keyboard's history traces back to the first Remington typewriter in 1874.

Early computer programmers interacted with mainframes using keyboards to punch holes in Hollerith cards or enter commands on dumb terminals.

Personal computers have expanded the keyboard's functionality with function keys, cursor positioning keys, and numeric keypads. However, the fundamental principles of typing remain unchanged.

Keyboard Basics

Windows programs receive keyboard input through messages that convey information about keystrokes.

While there are eight different keyboard messages, your program can safely ignore most of them.

Additionally, the information provided in these messages often exceeds what your program needs.

Therefore, effectively handling keyboard input involves identifying and processing only the relevant messages.

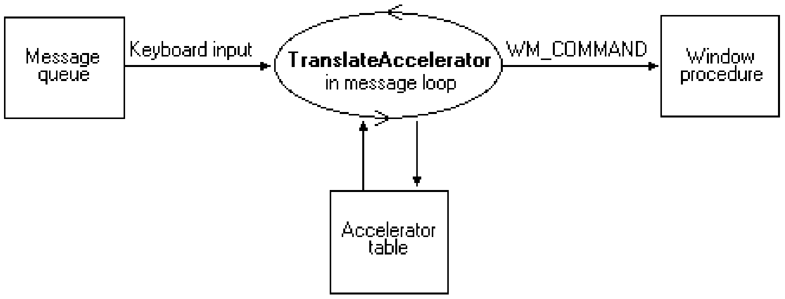
Ignoring Keyboard Input

Your program doesn't need to respond to every keyboard message it receives, as Windows handles many keyboard functions by default.

These functions typically involve the Alt key and relate to system operations.

Although your program can monitor these keystrokes, it can also rely on Windows notifications to learn about their effects.

Keyboard accelerators, which combine the Ctrl key with a function or letter key, activate common menu items.



These accelerators are defined in a program's resource script and translated by Windows into menu command messages. Your program doesn't need to perform this translation itself.

Windows manages the keyboard interface for dialog boxes and sends messages to your program regarding the outcome of keystrokes.

Edit controls within dialog boxes allow users to enter text, but Windows handles the logic for these controls and provides your program with the final contents once editing is complete.

Multiline edit controls can function as rudimentary text editors, and Windows provides a rich-text edit control for editing and displaying formatted text.

Child window controls can also be used to process keyboard and mouse input, sending higher-level information to the parent window. By utilizing child window controls, your program may not need to directly handle keyboard messages.

Who's Got the Focus?

In the realm of personal computers, the keyboard is a shared resource among all running applications under Windows. This includes multiple windows within a single application.

Recall that the MSG structure employed by programs to retrieve messages from the message queue contains an hwnd field.

This field identifies the handle of the window designated to receive the message.

The DispatchMessage function within the message loop routes the message to the window procedure associated with the intended recipient window.

When a key is pressed, only one window procedure receives the corresponding keyboard message, which includes a handle to the receiving window.

The window that receives a particular keyboard event is the one with the input focus. Input focus is closely linked to the concept of the active window.

The window with input focus is either the active window itself or a descendant of the active window. This encompasses child windows, grandchild windows, and so on.

Identifying the active window is typically straightforward. It always falls under the category of a top-level window, meaning its parent window handle is NULL.

If the active window possesses a title bar, Windows highlights it. In the absence of a title bar, if the active window employs a dialog frame (commonly seen in dialog boxes), Windows highlights the frame instead.

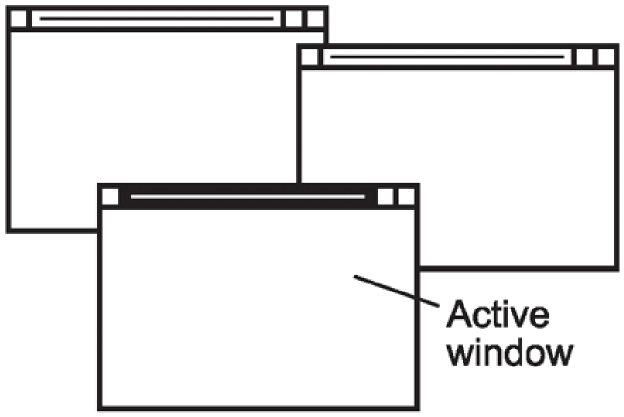


FIGURE 2. Active window

If the active window happens to be minimized, Windows distinguishes it in the taskbar by presenting it as a sunken button.

When child windows exist within the active window, the input focus can reside either in the active window itself or one of its descendants.

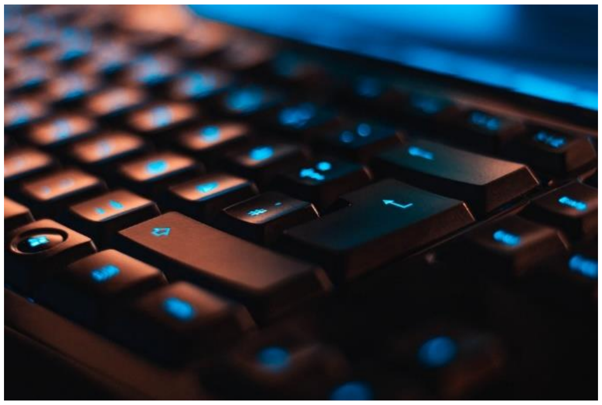
Common child windows include controls such as push buttons, radio buttons, checkboxes, scrollbars, edit boxes, and list boxes, often found in dialog boxes.

Child windows never assume the role of active windows. A child window can only acquire input focus if it belongs within the active window's lineage.

Child window controls typically indicate their possession of input focus by displaying a blinking caret or a dotted line.

Occasionally, no window has the input focus. This occurs when all programs are minimized.

Despite this, Windows continues to send keyboard messages to the active window, but these messages differ in form from those sent to active windows in a non-minimized state.



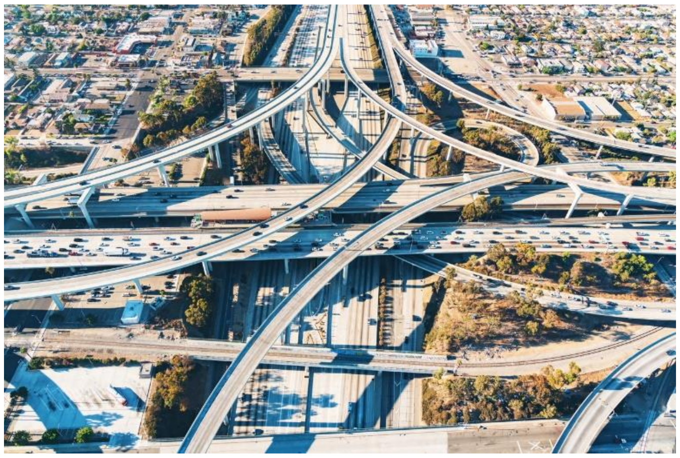
A window procedure can determine whether it holds the input focus by intercepting WM_SETFOCUS and WM_KILLFOCUS messages.

WM_SETFOCUS signifies that the window is receiving the input focus, while WM_KILLFOCUS indicates that the window is relinquishing the input focus.

These messages will be discussed in more detail later in this chapter.

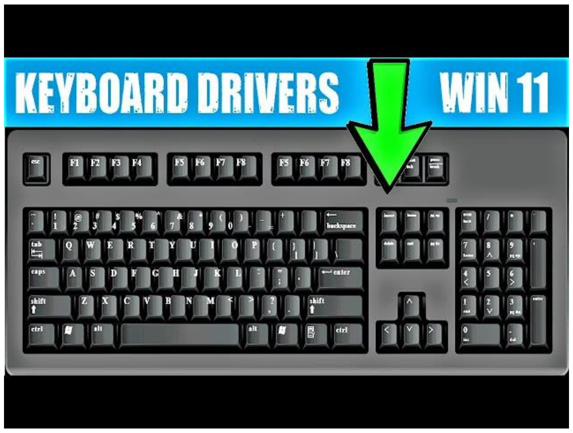
Here is a more detailed explanation of the system message queue and its role in synchronizing keyboard input:

Imagine Windows as a bustling city with a complex network of roads and a multitude of vehicles vying for space. When you type on your keyboard, it's akin to a sudden influx of cars entering the city, each with a specific destination.

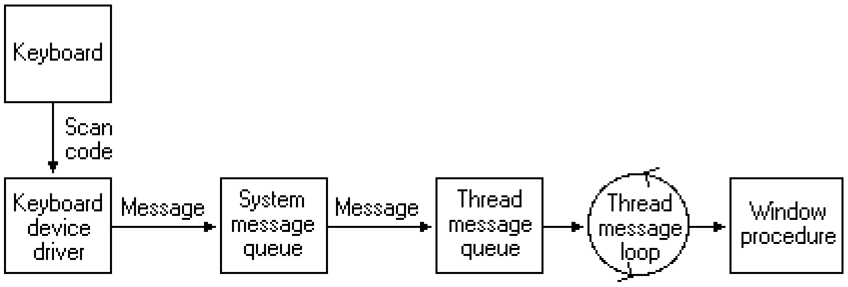


Windows, acting as the city's traffic management system, doesn't simply let all these vehicles flood the streets at once. Instead, it employs a designated holding area, analogous to the system message queue, to temporarily park the incoming vehicles until the traffic flow is smooth and organized.

The system message queue serves as a crucial intermediary between the keyboard device driver and your applications. It acts as a buffer, preventing a surge of keyboard messages from overwhelming your applications and causing chaos.



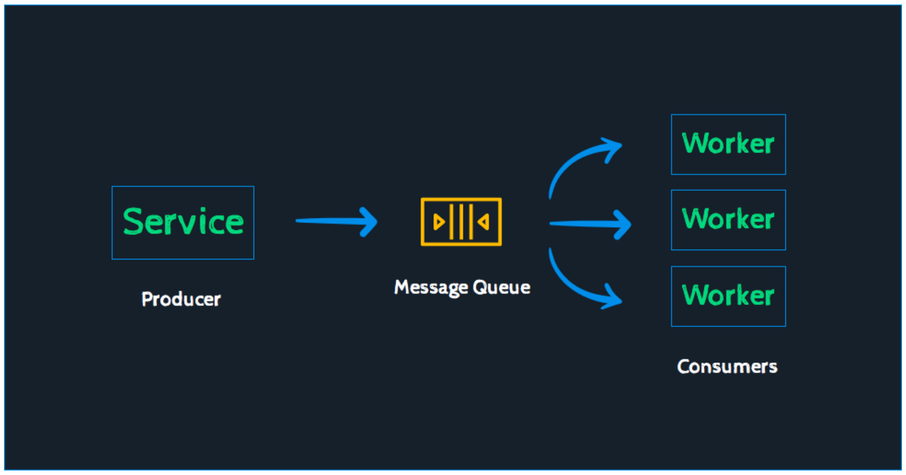
Windows meticulously manages this queue, ensuring that each message is processed sequentially and routed to the appropriate application only when the previous message has been handled.



This synchronized approach is essential for several reasons. Firstly, it prevents keystrokes from being lost or misdirected.

If Windows were to send all keyboard messages directly to applications without any control, keystrokes could potentially end up in the wrong window, leading to confusion and frustration.

Secondly, the system message queue maintains the integrity of the input focus. When you switch focus between windows, the queue ensures that subsequent keystrokes are directed to the newly active window, preventing them from lingering in the queue and being processed by the previously focused window.



Thirdly, the queue allows Windows to prioritize certain types of keyboard messages. For instance, system-level hotkeys, such as those used to control volume or open the Start menu, are processed immediately, ensuring that these critical actions are not delayed by the regular flow of keyboard input.

In essence, the system message queue functions as a diligent traffic controller, regulating the flow of keyboard input and ensuring that your keystrokes reach their intended destinations in a timely and orderly manner. It's an unsung hero of the Windows operating system, quietly maintaining order amidst the chaos of user input.

KEYSTROKES vs CHARACTERS

The messages that an application receives from Windows regarding keyboard events differentiate between keystrokes and characters. This distinction stems from the dual nature of the keyboard.

On one hand, the keyboard can be viewed as a collection of physical keys. Each key, like the "A" key, has a specific label and generates a corresponding signal upon activation. Pressing and releasing a key are both considered keystrokes.

On the other hand, the keyboard serves as an input device that produces displayable characters or control characters.

The "A" key, for instance, can generate various characters depending on the state of the modifier keys (Ctrl, Shift, and Caps Lock).

Typically, the "A" key produces a lowercase "a." However, if the Shift key is held or Caps Lock is enabled, it generates an uppercase "A."

If the Ctrl key is pressed, it produces a Ctrl+A character, which carries a specific meaning in ASCII and may function as a keyboard shortcut in Windows.

In certain scenarios, a keystroke may be preceded by a dead key or a combination of modifier keys (Shift, Ctrl, or Alt). These combinations can generate characters with accent marks, such as **à, á, â, ã, Ä, or Å.**

For keystroke combinations that result in displayable characters, Windows sends both keystroke and character messages to the program.

However, some keys, such as the modifier keys, function keys, cursor movement keys, and special keys like Insert and Delete, do not generate characters. For these keys, Windows only sends keystroke messages.

Keystroke Messages

Keystroke messages provide information about the physical key that was pressed or released. They include the following details:

- *The type of event (key down or key up).*
- *The virtual key code, which uniquely identifies the key.*
- *The scan code, which represents the physical location of the key on the keyboard.*
- *The state of the modifier keys (Ctrl, Shift, Alt).*
- *The repeat count, indicating the number of times the key has been pressed in rapid succession.*

Character Messages

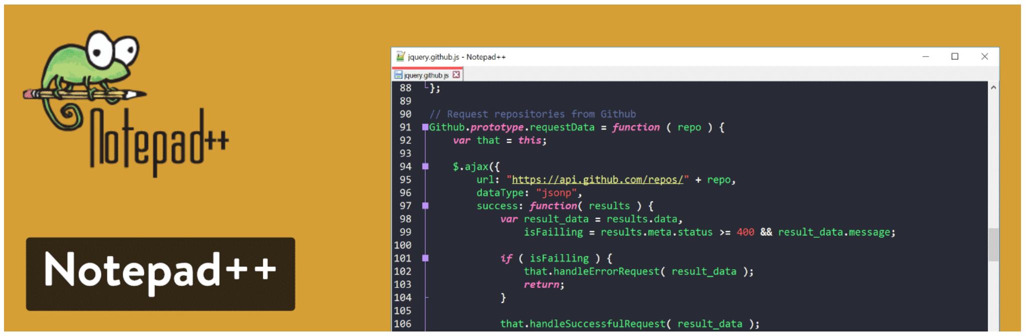
Character messages convey information about the character that was generated by the keystroke. They include the following details:

- *The Unicode character code.*
- *The virtual key code corresponding to the key that generated the character.*
- *The state of the modifier keys (Ctrl, Shift, Alt).*

Applications and Keystroke/Character Messages

Programs can handle both keystroke and character messages based on their specific needs.

For instance, a text editor would primarily be interested in character messages to process and display the entered text.



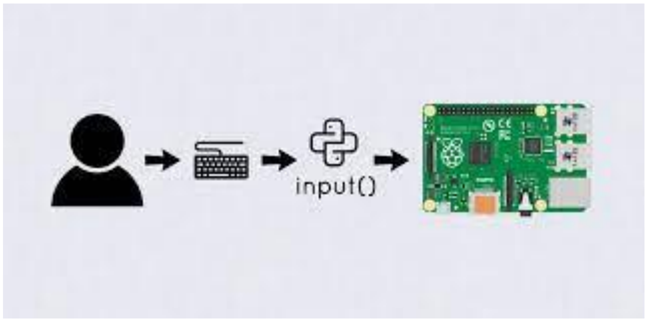
```
};

// Request repositories from Github
Github.prototype.requestData = function ( repo ) {
    var that = this;

    $.ajax({
        url: "https://api.github.com/repos/" + repo,
        dataType: "jsonp",
        success: function( results ) {
            var result_data = results.data,
                isFailling = results.meta.status >= 400 && result_data.message;

            if ( isFailling ) {
                that.handleErrorRequest( result_data );
                return;
            }

            that.handleSuccessfulRequest( result_data );
```

A game, on the other hand, might rely heavily on keystroke messages to detect and respond to user actions.

The distinction between keystrokes and characters allows programs to handle keyboard input in a more granular and versatile manner. By understanding the nuances of these two concepts, developers can create applications that are responsive, efficient, and user-friendly.

Keystroke Messages: Capturing User Input

When you interact with a computer, your keystrokes are the primary means of conveying your intentions and commands. Windows, the operating system that powers most personal computers, plays a crucial role in translating these keystrokes into meaningful actions.



Keystroke messages are the fundamental building blocks of this communication process. They act as messengers, carrying information about each key press and release from the keyboard to the corresponding application window.

Keystroke Types: System and Nonsystem

Windows distinguishes between two types of keystrokes: system keystrokes and nonsystem keystrokes. This distinction is based on the role of the keys involved and how they affect the overall system operation.

System Keystrokes:

System keystrokes are associated with keys that perform system-wide functions, such as the Ctrl key, Alt key, and Windows key. These keys often act in combination with other keys to invoke system commands, open menus, or control various aspects of the operating system.

When a system keystroke occurs, Windows generates a pair of messages: WM_SYSKEYDOWN and WM_SYSKEYUP.

The WM_SYSKEYDOWN message indicates that the key has been pressed, while WM_SYSKEYUP signals that the key has been released.

Nonsystem Keystrokes:

Nonsystem keystrokes, on the other hand, are associated with keys that directly input characters into the active application. These include the alphanumeric keys, punctuation keys, and various symbols.

When a nonsystem keystroke occurs, Windows generates another pair of messages: WM_KEYDOWN and WM_KEYUP. These messages follow the same pattern as system keystroke messages, indicating the key press and release events.

Message Sequencing and Autorepeat Handling

Under normal circumstances, keystroke messages occur in pairs, with a WM_KEYDOWN message preceding a WM_KEYUP message. However, there's an exception to this pattern: autorepeat functionality.

Autorepeat, also known as typematic action, is a feature that continuously generates keystroke messages while a key is held down. This allows for rapid input of characters without the need to repeatedly press the same key.

When autorepeat is activated, Windows sends a series of WM_KEYDOWN messages (or WM_SYSKEYDOWN messages for system keys) as the key is held.

Once the key is released, a single WM_KEYUP message (or WM_SYSKEYUP message for system keys) is generated.

Timestamping: Capturing the Temporal Context

Just as each keystroke has a physical location and character representation, it also exists within a specific time frame. To capture this temporal context, Windows assigns timestamps to each keystroke message.

The GetMessageTime function allows applications to retrieve the relative time at which a key was pressed or released. This information can be used for various purposes, such as tracking typing speed, analyzing user behavior, or implementing time-based keyboard shortcuts.

Speed
ACCURACY
TypeRacer
Learn typing through racing games with friends
MULTIPLAYER | SINGLEPLAYER | PLAY WITH FRIENDS

Conclusion: Keystroke Messages – The Language of User Input

Keystroke messages serve as the foundation for capturing and processing user input in Windows applications. They provide a detailed record of each key press and release, allowing applications to respond to user actions and translate them into meaningful operations. The distinction between system and nonsystem keystrokes, along with autorepeat handling and timestamping, further enhances the versatility of keystroke messages, making them an essential part of the Windows user interface.

DELVING INTO THE WORLD OF KEYSTROKES

When you interact with your computer, the keyboard plays a crucial role in conveying your commands and intentions.

The operating system, in this case Windows, acts as an intermediary, translating these keystrokes into meaningful actions.

At the heart of this process are keystroke messages, the messengers that carry information about each key press and release from the keyboard to the corresponding application window.

Handling Keystroke Messages: Applications vs. Windows

Windows takes care of processing system keystrokes, handling the Alt key logic and generating menu options or system commands.

Applications generally ignore WM_SYSKEYDOWN and WM_SYSKEYUP messages, passing them to DefWindowProc, the default window procedure provided by Windows.

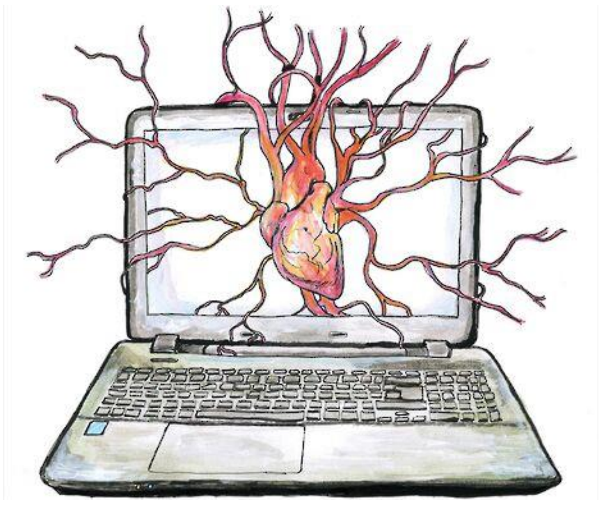
However, applications can choose to trap system keystroke messages if they need to perform specific actions or override the default behavior.

In such cases, they should pass the processed messages back to DefWindowProc to ensure that Windows can still handle them for their intended purposes.

For nonsystem keystrokes, applications have complete control over how they are handled. They can choose to process these messages to capture and interpret the input characters, or they can discard them if the input is not relevant to the application's functionality.

Exploring the Power of Window Procedures

The window procedure serves as the heart of a window's event handling, acting as a control center for processing messages and responding to user interactions. Through the window procedure, applications can selectively handle or discard different types of messages, including keystroke messages.



By modifying the window procedure, applications can gain granular control over how they respond to user input. This allows them to customize their behavior, implement custom shortcuts, or even disable certain system-level operations.

The Role of Virtual Key Codes and lParam

For all four keystroke messages, wParam contains a virtual key code, which uniquely identifies the key being pressed or released. This code provides a consistent way to refer to keys across different keyboard layouts and configurations.

lParam, on the other hand, contains additional information pertaining to the keystroke, such as the state of the modifier keys (Ctrl, Alt, Shift) and the repeat count for autorepeat functionality.

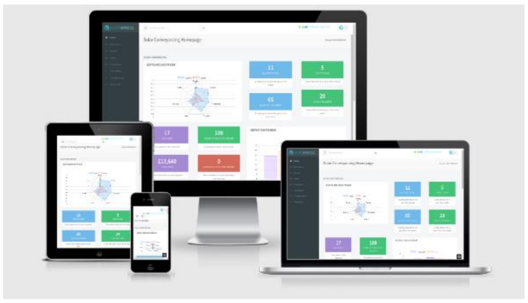
VIRTUAL KEY CODES: BRIDGING THE GAP BETWEEN HARDWARE AND SOFTWARE

In the realm of computer input, virtual key codes play a crucial role in translating physical key presses into meaningful actions within applications. They serve as a bridge between the hardware, represented by the keyboard, and the software, represented by the operating system and applications.

Virtual vs. Real: Unraveling the Terminology

The term "virtual" often evokes the notion of something intangible or existing only in the mind. While this may seem counterintuitive for key codes, which are associated with physical keys, the distinction lies in their abstraction from the keyboard's physical layout.

In contrast to scan codes, which represent the physical location of keys on the keyboard, virtual key codes provide a device-independent representation of keys. This means that they remain consistent regardless of the keyboard's layout or manufacturer.



The Utility of Virtual Key Codes for Windows

The adoption of virtual key codes in Windows stems from several advantages:

Device Independence: Virtual key codes enable Windows to handle input consistently across various keyboard layouts and manufacturers, eliminating the need for device-specific code.

Future-Proofing: By decoupling key codes from the physical keyboard layout, Windows can accommodate new keyboard designs and input methods without affecting existing applications.

Accessibility: Virtual key codes allow for alternative input methods, such as assistive technologies, to interact with Windows applications without requiring specific knowledge of keyboard layouts.

Commonly Used Virtual Key Codes

Many virtual key codes correspond to frequently used keys on the keyboard. These include:

- VK_BACK: Backspace key
- VK_TAB: Tab key
- VK_CLEAR: Numeric keypad 5 with Num Lock off
- VK_RETURN: Enter key
- VK_SHIFT: Shift key
- VK_CONTROL: Ctrl key
- VK_MENU: Alt key
- VK_ESCAPE: Esc key
- VK_SPACE: Spacebar

Virtual Key Codes and Windows Applications

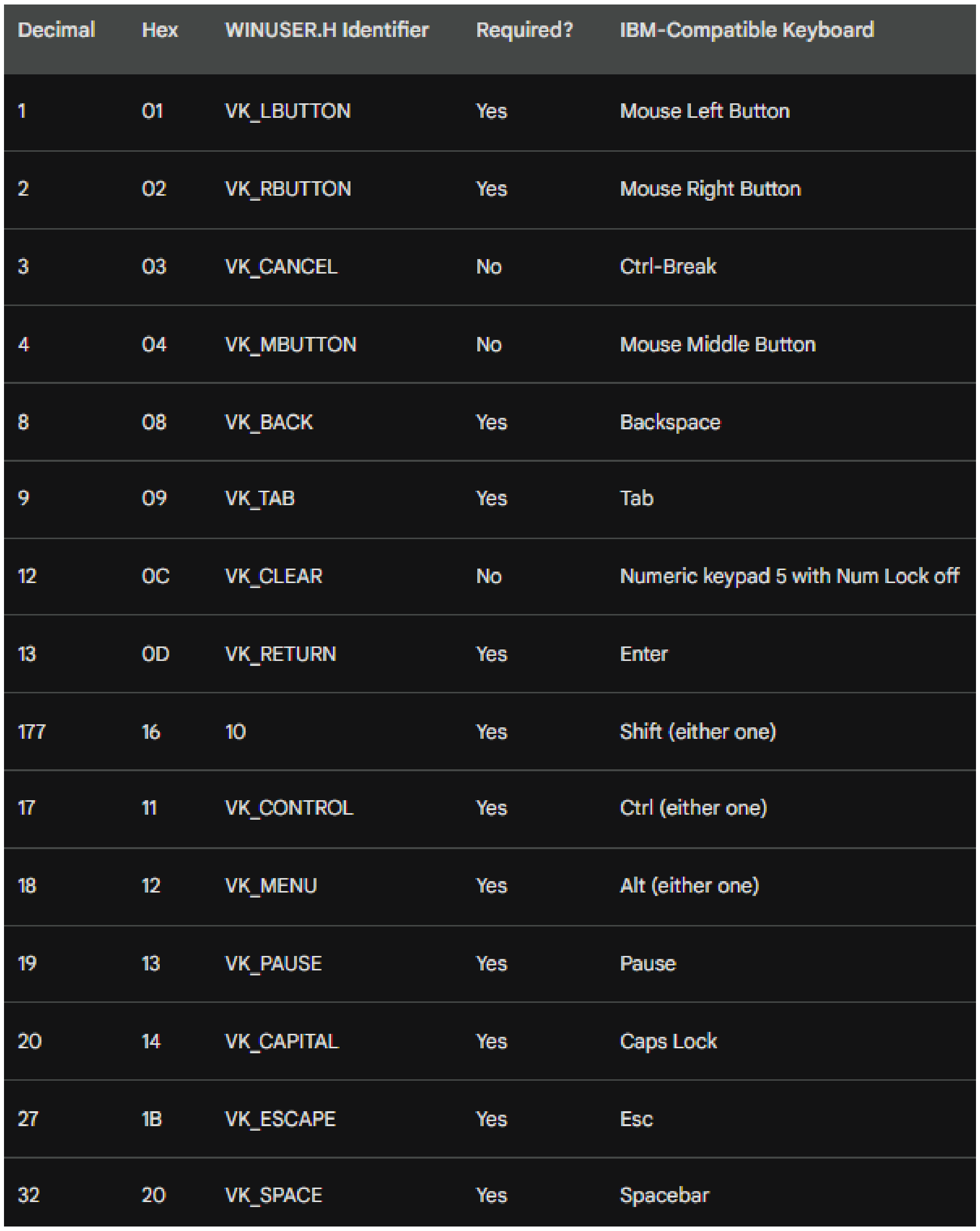
While Windows applications often rely on character messages for processing common keys like Backspace, Tab, Enter, and Spacebar, virtual key codes remain essential for handling less frequent keys and for monitoring the state of modifier keys (Shift, Ctrl, Alt).

Virtual Key Codes in Action

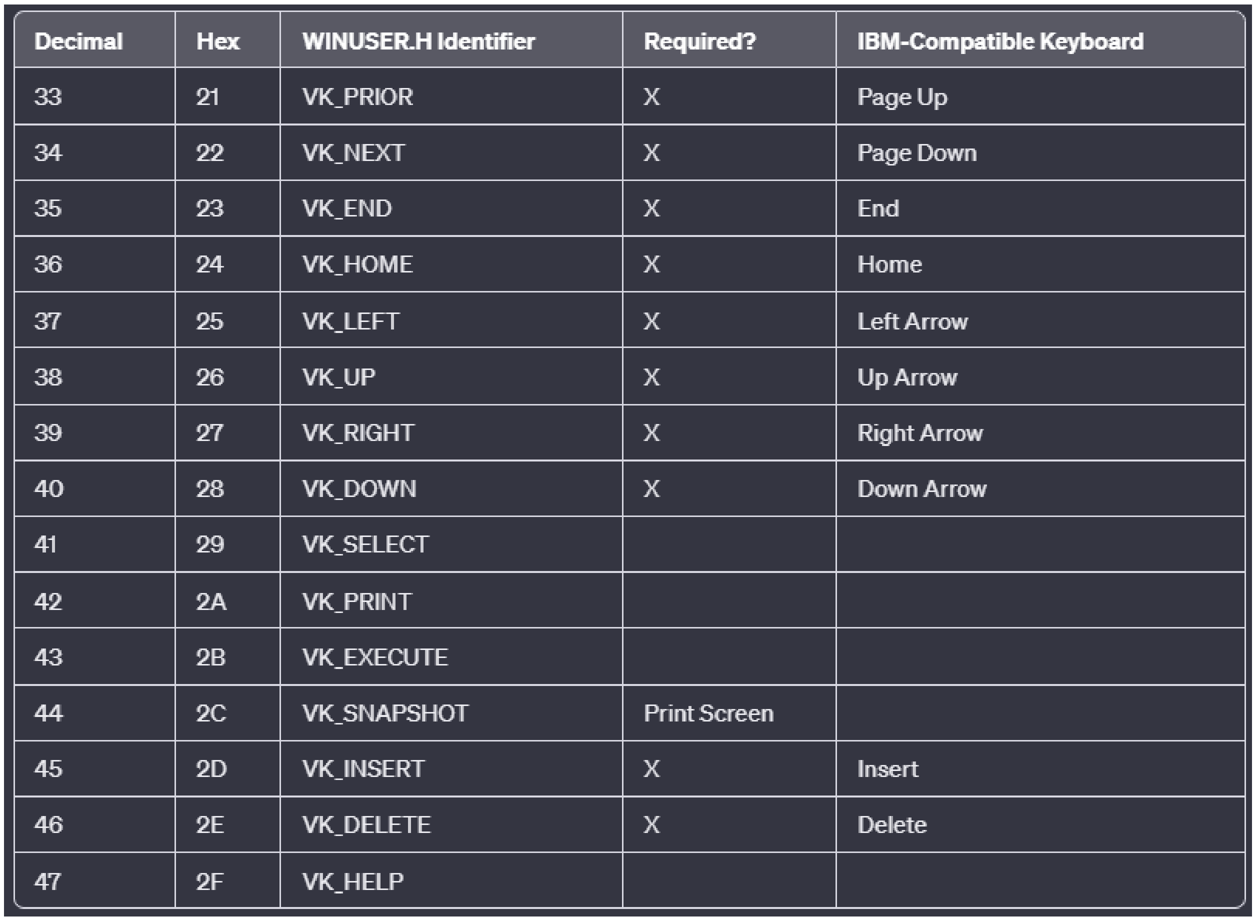
Virtual key codes are embedded within keystroke messages (WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, and WM_SYSKEYUP) and passed to window procedures for handling. Applications can process these messages to respond to user input and perform desired actions.

Virtual Key Code Tables

For reference, here are tables summarizing the commonly used virtual key codes:



Decimal	Hex	WINUSER.H Identifier	Required?	IBM-Compatible Keyboard
1	01	VK_LBUTTON	Yes	Mouse Left Button
2	02	VK_RBUTTON	Yes	Mouse Right Button
3	03	VK_CANCEL	No	Ctrl-Break
4	04	VK_MBUTTON	No	Mouse Middle Button
8	08	VK_BACK	Yes	Backspace
9	09	VK_TAB	Yes	Tab
12	0C	VK_CLEAR	No	Numeric keypad 5 with Num Lock off
13	0D	VK_RETURN	Yes	Enter
177	16	10	Yes	Shift (either one)
17	11	VK_CONTROL	Yes	Ctrl (either one)
18	12	VK_MENU	Yes	Alt (either one)
19	13	VK_PAUSE	Yes	Pause
20	14	VK_CAPITAL	Yes	Caps Lock
27	1B	VK_ESCAPE	Yes	Esc
32	20	VK_SPACE	Yes	Spacebar

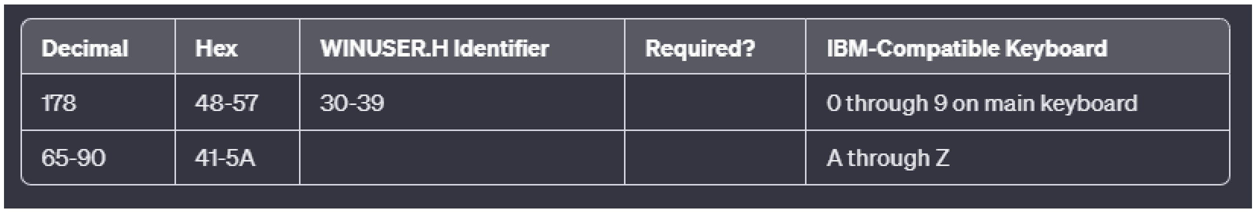


Decimal	Hex	WINUSER.H Identifier	Required?	IBM-Compatible Keyboard
33	21	VK_PRIOR	X	Page Up
34	22	VK_NEXT	X	Page Down
35	23	VK_END	X	End
36	24	VK_HOME	X	Home
37	25	VK_LEFT	X	Left Arrow
38	26	VK_UP	X	Up Arrow
39	27	VK_RIGHT	X	Right Arrow
40	28	VK_DOWN	X	Down Arrow
41	29	VK_SELECT		
42	2A	VK_PRINT		
43	2B	VK_EXECUTE		
44	2C	VK_SNAPSHOT	Print Screen	
45	2D	VK_INSERT	X	Insert
46	2E	VK_DELETE	X	Delete
47	2F	VK_HELP		

Note: Some key names (e.g., VK_PRIOR and VK_NEXT) may differ from key labels and lack consistency with scroll bar identifiers.

The Print Screen key is often ignored by Windows applications, used mainly by Windows itself for storing a bitmap copy of the display into the clipboard. Keys like VK_SELECT, VK_PRINT, VK_EXECUTE, and VK_HELP might be found on less common keyboards.

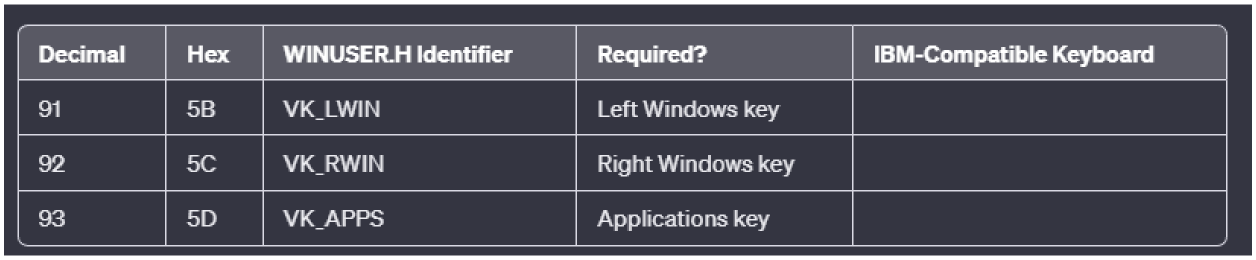
Virtual Key Codes for Main Keyboard Keys:



Decimal	Hex	WINUSER.H Identifier	Required?	IBM-Compatible Keyboard
178	48-57	30-39		0 through 9 on main keyboard
65-90	41-5A			A through Z

Note: These virtual key codes align with ASCII codes for numbers and letters. Windows programs often use character messages for ASCII characters rather than these virtual key codes.

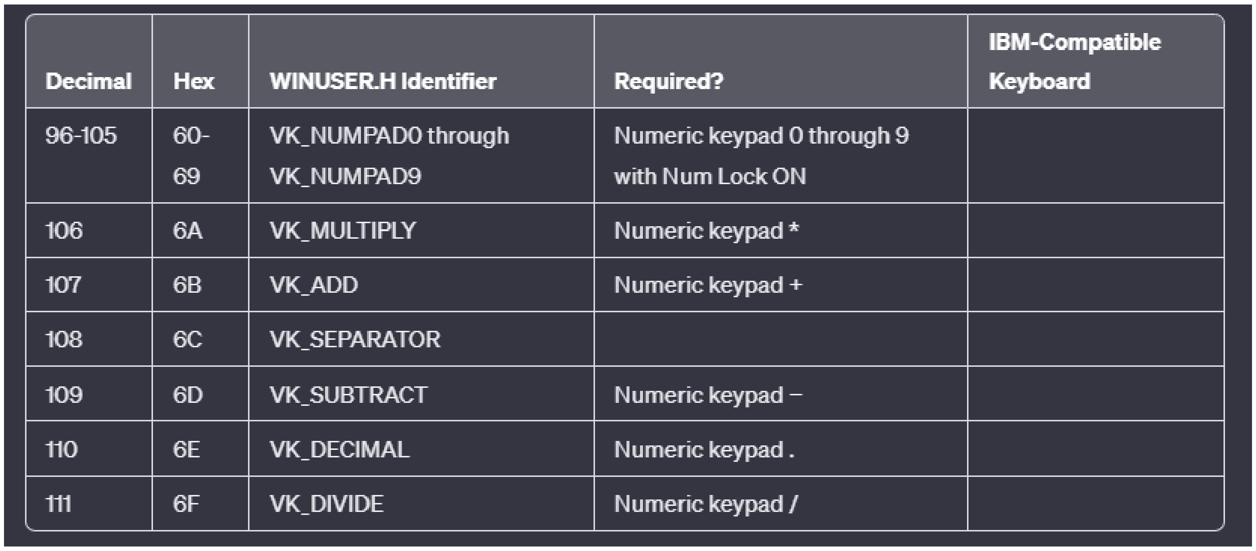
Virtual Key Codes for Microsoft Natural Keyboard:



Decimal	Hex	WINUSER.H Identifier	Required?	IBM-Compatible Keyboard
91	5B	VK_LWIN	Left Windows key	
92	5C	VK_RWIN	Right Windows key	
93	5D	VK_APPS	Applications key	

Note: VK_LWIN and VK_RWIN keys are handled by Windows for functions like opening the Start menu or launching the Task Manager. VK_APPS can be processed by applications for displaying help information or shortcuts.

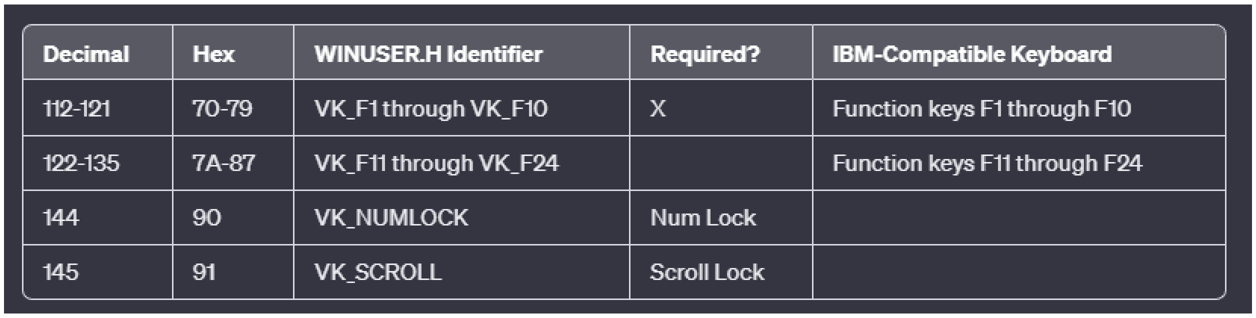
Virtual Key Codes for Numeric Keypad:



Decimal	Hex	WINUSER.H Identifier	Required?	IBM-Compatible Keyboard
96-105	60-69	VK_NUMPAD0 through VK_NUMPAD9	Numeric keypad 0 through 9 with Num Lock ON	
106	6A	VK_MULTIPLY	Numeric keypad *	
107	6B	VK_ADD	Numeric keypad +	
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT	Numeric keypad –	
110	6E	VK_DECIMAL	Numeric keypad .	
111	6F	VK_DIVIDE	Numeric keypad /	

Note: These codes correspond to keys on the numeric keypad, and their availability depends on the state of the Num Lock.

Function Key Virtual Codes:



Decimal	Hex	WINUSER.H Identifier	Required?	IBM-Compatible Keyboard
112-121	70-79	VK_F1 through VK_F10	X	Function keys F1 through F10
122-135	7A-87	VK_F11 through VK_F24		Function keys F11 through F24
144	90	VK_NUMLOCK	Num Lock	
145	91	VK_SCROLL	Scroll Lock	

Note: Windows requires only 10 function keys but has numeric identifiers for up to 24. Function keys are often used as keyboard accelerators, and programs typically don't process all keystrokes in this table.

There are additional virtual key codes defined for specific keys on nonstandard keyboards or those commonly found on mainframe terminals.

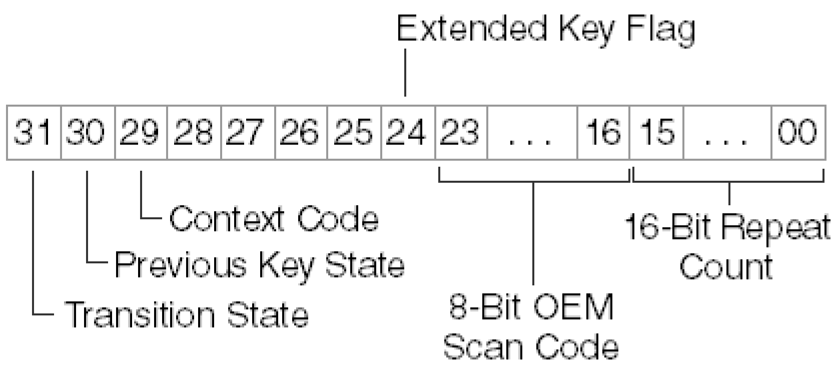
Refer to /Platform SDK/User Interface Services/User Input/Virtual−Key Codes for a comprehensive list.

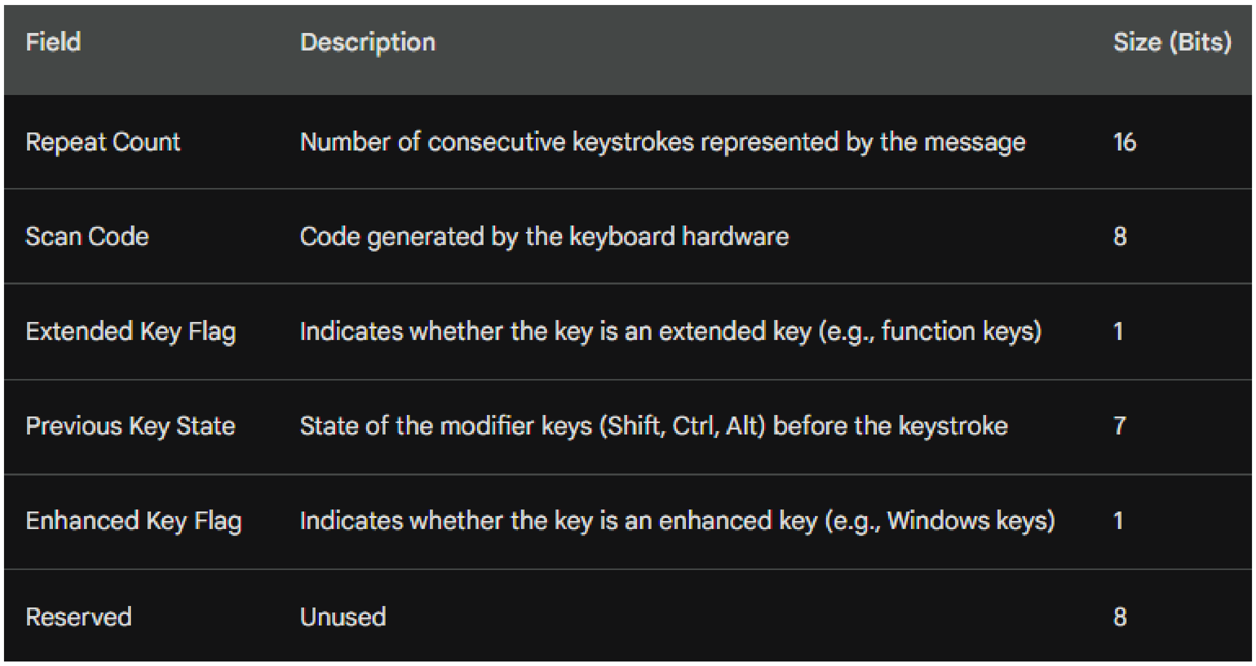
lParam: Unpacking the Additional Information

In addition to the virtual key code, which is stored in the wParam parameter of keystroke messages, the lParam parameter provides further details about the keystroke. This information is crucial for understanding the context and intent behind the key press or release.

The Structure of lParam

The 32 bits of lParam are divided into six fields, as depicted in Figure 6-1:





Field	Description	Size (Bits)
Repeat Count	Number of consecutive keystrokes represented by the message	16
Scan Code	Code generated by the keyboard hardware	8
Extended Key Flag	Indicates whether the key is an extended key (e.g., function keys)	1
Previous Key State	State of the modifier keys (Shift, Ctrl, Alt) before the keystroke	7
Enhanced Key Flag	Indicates whether the key is an enhanced key (e.g., Windows keys)	1
Reserved	Unused	8

Repeat Count: Tracking Rapid Keystrokes

The Repeat Count field indicates the number of consecutive keystrokes represented by the message. This value is typically set to 1 for individual key presses.

However, when a key is held down and the window procedure cannot process key-down messages at the typematic rate, Windows combines multiple WM_KEYDOWN or WM_SYSKEYDOWN messages into a single message and increments the Repeat Count accordingly. For WM_KEYUP or WM_SYSKEYUP messages, the Repeat Count is always 1.

OEM Scan Code: A Relic of the Past

The OEM Scan Code field contains the code generated by the keyboard hardware. This was a valuable piece of information for assembly language programmers who interacted with the keyboard through BIOS services.

However, in today's Windows environment, applications can generally disregard the OEM Scan Code, except in rare cases where they need to consider the physical layout of the keyboard.

Navigating the Keystroke Landscape with lParam

The lParam parameter provides valuable insights into the context of keystrokes. By understanding the Repeat Count, OEM Scan Code, and other fields within lParam, developers can effectively handle keystroke events, respond to user input, and create intuitive applications.

Extended Key Flag: Identifying Enhanced Keyboard Keys

The Extended Key Flag indicates whether the keystroke originates from one of the additional keys on the IBM enhanced keyboard.

This keyboard typically has 101 or 102 keys, including function keys across the top, separate cursor movement keys, and a numeric keypad that duplicates the cursor movement keys.

The Extended Key Flag is set to 1 for specific keys, including the Alt and Ctrl keys at the right of the keyboard, the cursor movement keys (including Insert and Delete) that are not part of the numeric keypad, the slash (/) and Enter keys on the numeric keypad, and the Num Lock key. Windows applications generally ignore the Extended Key Flag.

Context Code: Unraveling the Role of Alt Key

The Context Code signifies whether the Alt key was pressed during the keystroke. This bit is always set to 1 for WM_SYSKEYUP and WM_SYSKEYDOWN messages and 0 for WM_KEYUP and WM_KEYDOWN messages. However, there are two exceptions:

Minimized Window Handling: When the active window is minimized, it does not have the input focus. Consequently, all keystrokes generate WM_SYSKEYUP and WM_SYSKEYDOWN messages.

In these cases, if the Alt key is not pressed, the Context Code field is set to 0. Windows uses WM_SYSKEYUP and WM_SYSKEYDOWN messages to prevent the minimized active window from processing these keystrokes.

Foreign Language Characters: On some foreign-language keyboards, specific characters are generated by combining Shift, Ctrl, or Alt with another key. In these instances, the Context Code is set to 1, but the messages are not system keystroke messages.

Previous Key State: Tracking Key Toggle State

The Previous Key State indicates whether the key was previously up (0) or down (1). This field is always set to 1 for a WM_KEYUP or WM_SYSKEYUP message, but it can be 0 or 1 for a WM_KEYDOWN or WM_SYSKEYDOWN message. A 1 indicates second and subsequent messages that result from typematic repeats.

Transition State: Identifying Press and Release Events

The Transition State distinguishes between key press and release events. This field is set to 0 for a WM_KEYDOWN or WM_SYSKEYDOWN message and to 1 for a WM_KEYUP or WM_SYSKEYUP message.

SHIFT STATES: IDENTIFYING MODIFIER KEYS

When handling keystroke events, it's often crucial to determine whether any of the modifier keys (Shift, Ctrl, and Alt) or toggle keys (Caps Lock, Num Lock, and Scroll Lock) are pressed. This information plays a vital role in interpreting user actions and enabling context-aware functionality.

GetKeyState: Unveiling Key State Information

The GetKeyState function provides a convenient way to retrieve the state of a specified virtual key code. To determine if the Shift key is currently pressed, you can use the following code:

```
iState = GetKeyState(VK_SHIFT);
```

If the Shift key is down, the iState variable will be negative, indicating that the high bit is set. Similarly, to check if the Caps Lock key is toggled on, you can use:

```
iState = GetKeyState(VK_CAPITAL);
```

The returned value from iState will have the low bit set if the Caps Lock key is on. This bit corresponds to the little light on the keyboard.

Common Use Cases for GetKeyState

GetKeyState is primarily used with the virtual key codes VK_SHIFT, VK_CONTROL, and VK_MENU (which represents the Alt key).

Additionally, it can be used with the following identifiers to determine if the left or right Shift, Ctrl, or Alt keys are pressed:

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

These identifiers are specifically designed for use with GetKeyState and GetAsyncKeyState.

Mouse Button States

GetKeyState can also be used to obtain the state of the mouse buttons using the virtual key codes VK_LBUTTON, VK_RBUTTON, and VK_MBUTTON.

However, most Windows programs that monitor mouse buttons and keystrokes simultaneously typically do so by checking keystrokes when receiving a mouse message.

This approach is often preferred due to the inclusion of shift-state information within mouse messages.

GetKeyState vs. Real-Time Keyboard Status

It's essential to note that GetKeyState does not provide a real-time keyboard status check. Instead, it reflects the keyboard status up to and including the current message being processed. This distinction is important for accurately interpreting user actions.

Example: Handling Shift-Tab

To determine if the user pressed Shift-Tab, you can call GetKeyState with the VK_SHIFT parameter while processing the WM_KEYDOWN message for the Tab key:

```
if (GetKeyState(VK_SHIFT) < 0) {
  // Shift key was pressed before Tab key
}
```

This code snippet effectively captures the state of the Shift key when the Tab key is pressed, regardless of whether the Shift key has been released by the time the Tab key event is processed.

GetAsyncKeyState: Real-Time Keyboard Status

GetKeyState is limited to retrieving keyboard information within the context of normal keyboard messages.

For scenarios where you need to determine the current real-time state of a key, such as waiting for the F1 key to be pressed, you should use GetAsyncKeyState:

```
while (GetAsyncKeyState(VK_F1) >= 0) {
  // Wait until F1 key is pressed
}
```

GetAsyncKeyState provides a real-time representation of the key state, allowing you to respond to user actions without relying on the message queue.

Harnessing Keystroke Messages Effectively

Windows applications receive detailed information about each keystroke that occurs while they are running. While this information can be quite valuable, most Windows programs only process a limited subset of keystroke messages.

WM_SYSKEYDOWN and WM_SYSKEYUP Messages

The WM_SYSKEYDOWN and WM_SYSKEYUP messages are primarily used by Windows system functions and can generally be ignored by your application.

WM_KEYDOWN Messages: Processing Keystrokes

Windows programs typically use WM_KEYDOWN messages to handle keystrokes that do not generate characters.

While it might seem tempting to combine keystroke messages with shift-state information to translate them into characters, this approach is not recommended due to complexities with non-English keyboards.

For instance, receiving a WM_KEYDOWN message with wParam equal to 0x33 indicates that the user pressed the 3 key.

However, assuming the user is typing a pound sign (#) based on the Shift key state might be incorrect. A British user might be typing a different pound sign, the one that looks like this: £.

WM_KEYDOWN Messages for Cursor Movement and Special Keys

WM_KEYDOWN messages are most useful for handling cursor movement keys, function keys, Insert, and Delete.

However, Insert, Delete, and function keys often appear as menu accelerators, which are translated by Windows into menu command messages, eliminating the need to process the keystrokes themselves.

Function Keys: A Double-Edged Sword

Pre-Windows MS-DOS applications often relied heavily on function keys in combination with the Shift, Ctrl, and Alt keys.

While you can implement a similar approach in your Windows programs (as demonstrated by Microsoft Word's extensive use of function keys as command shortcuts), this practice is generally discouraged.

Ideally, function keys should replicate menu commands, aligning with the Windows philosophy of providing a user interface that minimizes memorization and complex command charts.

Recommended Keystroke Handling

As a general rule, you should only process WM_KEYDOWN messages for cursor movement keys and, occasionally, Insert and Delete.

When interacting with these keys, you can check the Shift-key and Ctrl-key states using GetKeyState.

Windows programs frequently utilize the Shift key in conjunction with cursor keys to extend selections in documents.

Similarly, the Ctrl key often modifies the meaning of the cursor key. For example, Ctrl+Right Arrow might move the cursor one word to the right.

Learning from Popular Windows Programs

One of the best strategies for determining how to use the keyboard in your application is to observe how popular Windows programs handle keyboard interactions. While you have the freedom to deviate from these norms, doing so might hinder a user's ability to learn your program efficiently.

Enriching SYSMETS with Keyboard Support

The initial versions of the SYSMETS program lacked keyboard functionality, relying solely on mouse interactions with scroll bars for navigation. Now, armed with knowledge of keystroke messages, we can enhance the program to incorporate keyboard controls.

Leveraging Cursor Movement Keys for Scrolling

We'll employ cursor movement keys for vertical scrolling, utilizing Home, End, Page Up, Page Down, Up Arrow, and Down Arrow. Additionally, the Left Arrow and Right Arrow keys will handle horizontal scrolling.

Avoiding Redundancy in WM_KEYDOWN Logic

A straightforward approach would be to add WM_KEYDOWN logic to the window procedure, replicating the functionality of WM_VSCROLL and WM_HSCROLL messages. However, this approach introduces redundancy, making it difficult to maintain consistency in scroll bar logic across different message handlers.

Utilizing SendMessage for Efficient Keyboard Integration

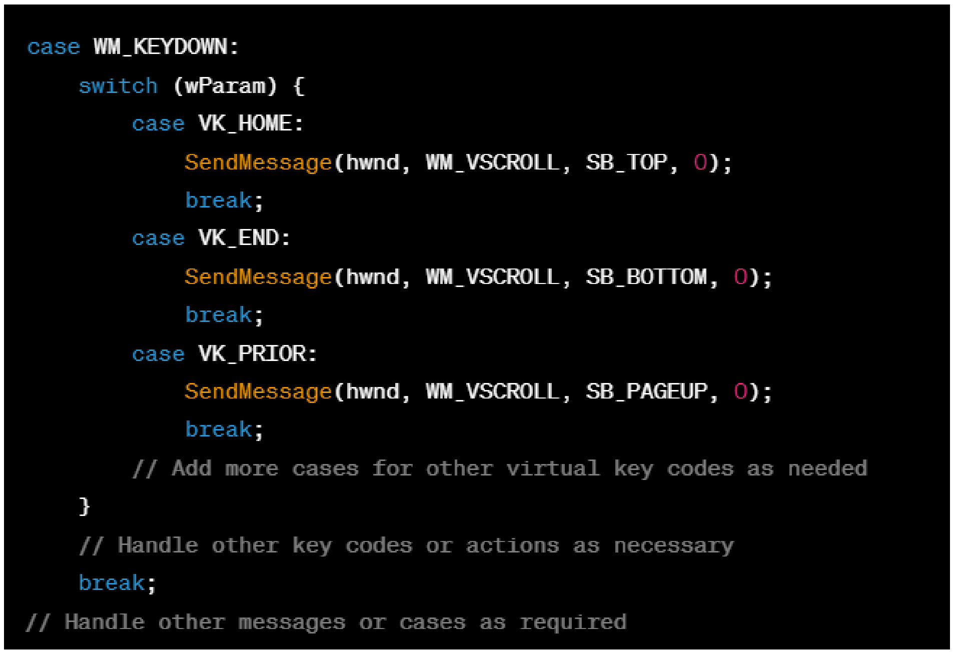
A more elegant solution is to translate each WM_KEYDOWN message into an equivalent WM_VSCROLL or WM_HSCROLL message. This approach leverages the existing scroll bar logic, ensuring consistency and simplifying maintenance.

The SendMessage Function: Translating Keystrokes into Scroll Bar Messages

The SendMessage function serves as the bridge between keystroke messages and scroll bar messages. It takes the same parameters as those passed to the window procedure, effectively sending a simulated message to the window procedure.

Implementing Keyboard-Triggered Scrolling

Within the WM_KEYDOWN case, a switch statement can be used to handle specific keystrokes and translate them into corresponding scroll bar messages:



```
case WM_KEYDOWN:
    switch (wParam) {
        case VK_HOME:
            SendMessage(hwnd, WM_VSCROLL, SB_TOP, 0);
            break;
        case VK_END:
            SendMessage(hwnd, WM_VSCROLL, SB_BOTTOM, 0);
            break;
        case VK_PRIOR:
            SendMessage(hwnd, WM_VSCROLL, SB_PAGEUP, 0);
            break;
        // Add more cases for other virtual key codes as needed
    }
    // Handle other key codes or actions as necessary
    break;
// Handle other messages or cases as required
```

This approach seamlessly integrates keyboard controls with the existing scroll bar logic, providing users with an intuitive and versatile way to navigate the text.

SYSMETS4: The Keyboard-Enhanced Version

The *SYSMETS4 program in chapter 6 folder*, incorporates these enhancements, enabling keyboard-driven scrolling alongside mouse interactions. This comprehensive approach provides users with a more versatile and convenient way to interact with the program.

Purpose and Functionality

The SYSMETS4 program is a graphical user interface (GUI) application designed to display system metrics information in a window. It provides a user-friendly interface for exploring and understanding the various system metrics that characterize the operating system environment.

Window Creation and Initialization (WM_CREATE)

When the SYSMETS4 program is executed, the WM_CREATE message is sent to the window procedure, signaling the creation of the application's window. This message triggers the initialization process within the window procedure, which involves setting up the window's initial appearance and preparing the necessary data structures.

Window Resizing and Scroll Bar Adjustments (WM_SIZE)

As the user resizes the application window, the WM_SIZE message is sent to the window procedure. This message informs the window procedure of the new window dimensions, enabling it to adapt the layout and adjust the scroll bars accordingly.

Handling Vertical Scroll Bar Interactions (WM_VSCROLL)

When the user interacts with the vertical scroll bar, whether by clicking the scroll arrows or dragging the thumb, the WM_VSCROLL message is sent to the window procedure. This message conveys information about the scroll bar's current position and the user's actions. The window procedure utilizes this information to update the vertical scroll bar position and scroll the window contents vertically, ensuring that the desired portion of the system metrics list is visible.

Managing Horizontal Scroll Bar Operations (WM_HSCROLL)

Similar to the vertical scroll bar, user interactions with the horizontal scroll bar trigger the WM_HSCROLL message, which carries information about the scroll bar's position and user actions. The window procedure processes this message to update the horizontal scroll bar position and scroll the window contents horizontally, providing users with a means to navigate through the system metrics list horizontally.

Responding to Keyboard-Based Scrolling (WM_KEYDOWN)

When the user presses certain keyboard keys, such as the arrow keys, the WM_KEYDOWN message is sent to the window procedure. This message indicates the specific key that was pressed. The window procedure interprets the pressed key and initiates the corresponding scrolling action, enabling users to control the window contents using keyboard shortcuts.

Refreshing the Display (WM_PAINT)

Whenever the window's appearance needs to be updated, such as after a scroll operation or a change in window size, the WM_PAINT message is sent to the window procedure. This message prompts the window procedure to redraw the system metrics information within the window. The window procedure iterates through the system metrics list, formatting and displaying each metric's label, description, and current value in the appropriate positions.

Program Termination (WM_DESTROY)

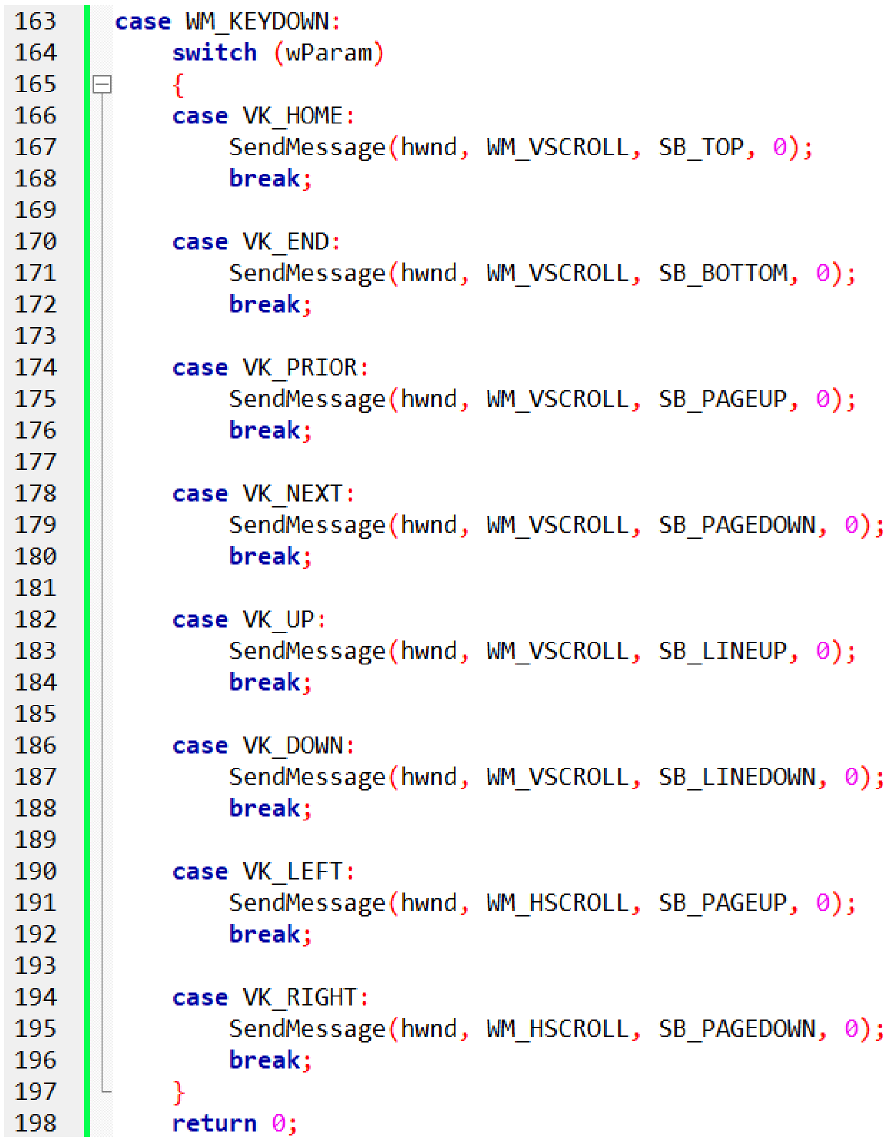
When the user initiates the program's termination, typically by closing the window, the WM_DESTROY message is sent to the window procedure. This message signals the impending closure of the window and the termination of the program. The window procedure performs any necessary cleanup tasks, such as releasing allocated resources, before returning control to the message loop.

Comprehensive Window Management

The SYSMETS4 program effectively handles various window events and user interactions, ensuring smooth operation and a responsive user experience. It demonstrates the use of scroll bars and keyboard shortcuts for navigating through content, providing users with multiple options for controlling the display of system metrics information.

Compared to systmets3.c

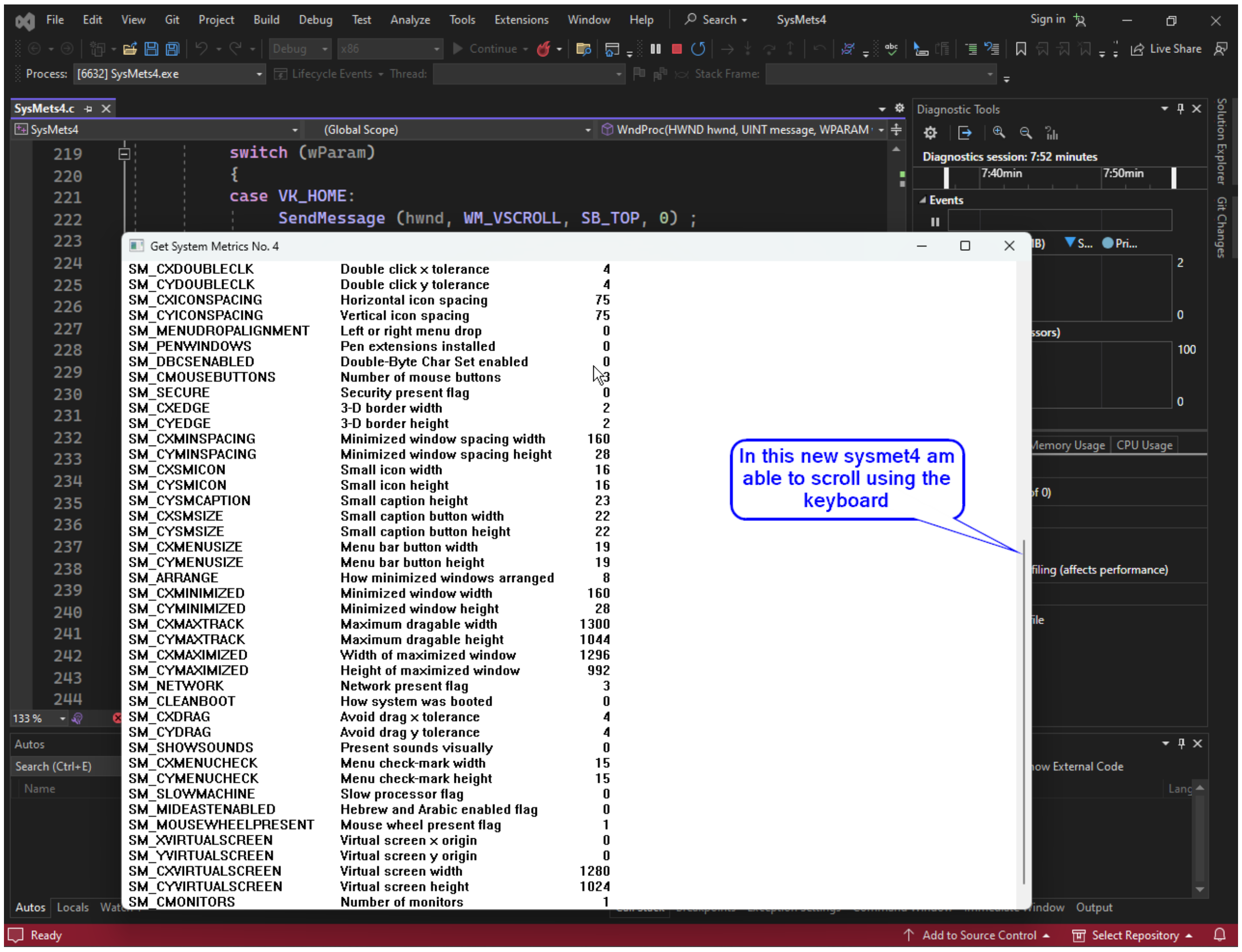
Sysmets4 includes additions like these:



```c
case WM_KEYDOWN:
    switch (wParam)
    {
    case VK_HOME:
        SendMessage(hwnd, WM_VSCROLL, SB_TOP, 0);
        break;

    case VK_END:
        SendMessage(hwnd, WM_VSCROLL, SB_BOTTOM, 0);
        break;

    case VK_PRIOR:
        SendMessage(hwnd, WM_VSCROLL, SB_PAGEUP, 0);
        break;

    case VK_NEXT:
        SendMessage(hwnd, WM_VSCROLL, SB_PAGEDOWN, 0);
        break;

    case VK_UP:
        SendMessage(hwnd, WM_VSCROLL, SB_LINEUP, 0);
        break;

    case VK_DOWN:
        SendMessage(hwnd, WM_VSCROLL, SB_LINEDOWN, 0);
        break;

    case VK_LEFT:
        SendMessage(hwnd, WM_HSCROLL, SB_PAGEUP, 0);
        break;

    case VK_RIGHT:
        SendMessage(hwnd, WM_HSCROLL, SB_PAGEDOWN, 0);
        break;
    }
    return 0;
```

Explanation:

- WM_KEYDOWN: This message is sent to the window when a nonsystem key is pressed.
- wParam: It contains the virtual key code of the key that was pressed.
- Switch on wParam: Depending on the virtual key code, different actions are taken.
- For example, when VK_HOME is pressed, it sends a vertical scroll message to the window (WM_VSCROLL) to scroll to the top (SB_TOP). Similar actions are taken for other keys, like VK_END, VK_PRIOR, etc.
- For arrow keys (VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT), it sends scroll messages accordingly.
- This implementation allows the user to navigate through the content of the window using keyboard input. Arrow keys, Home, End, Page Up, and Page Down keys are mapped to corresponding scroll actions.



```
switch (wParam)
{
case VK_HOME:
     SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
```

Get System Metrics No. 4

```
SM_CXDOUBLECLK           Double click x tolerance             4
SM_CYDOUBLECLK           Double click y tolerance             4
SM_CXICONSPACING         Horizontal icon spacing             75
SM_CYICONSPACING         Vertical icon spacing               75
SM_MENUDROPALIGNMENT     Left or right menu drop              0
SM_PENWINDOWS            Pen extensions installed             0
SM_DBCSENABLED           Double-Byte Char Set enabled         0
SM_CMOUSEBUTTONS         Number of mouse buttons              3
SM_SECURE                Security present flag                0
SM_CXEDGE                3-D border width                     2
SM_CYEDGE                3-D border height                    2
SM_CXMINSPACING          Minimized window spacing width     160
SM_CYMINSPACING          Minimized window spacing height     28
SM_CXSMICON              Small icon width                    16
SM_CYSMICON              Small icon height                   16
SM_CYSMCAPTION           Small caption height                23
SM_CXSMSIZE              Small caption button width          22
SM_CYSMSIZE              Small caption button height         22
SM_CXMENUSIZE            Menu bar button width               19
SM_CYMENUSIZE            Menu bar button height              19
SM_ARRANGE               How minimized windows arranged       8
SM_CXMINIMIZED           Minimized window width             160
SM_CYMINIMIZED           Minimized window height             28
SM_CXMAXTRACK            Maximum dragable width            1300
SM_CYMAXTRACK            Maximum dragable height           1044
SM_CXMAXIMIZED           Width of maximized window         1296
SM_CYMAXIMIZED           Height of maximized window         992
SM_NETWORK               Network present flag                 3
SM_CLEANBOOT             How system was booted                0
SM_CXDRAG                Avoid drag x tolerance               4
SM_CYDRAG                Avoid drag y tolerance               4
SM_SHOWSOUNDS            Present sounds visually              0
SM_CXMENUCHECK           Menu check-mark width               15
SM_CYMENUCHECK           Menu check-mark height              15
SM_SLOWMACHINE           Slow processor flag                  0
SM_MIDEASTENABLED        Hebrew and Arabic enabled flag       0
SM_MOUSEWHEELPRESENT     Mouse wheel present flag             1
SM_XVIRTUALSCREEN        Virtual screen x origin              0
SM_YVIRTUALSCREEN        Virtual screen y origin              0
SM_CXVIRTUALSCREEN       Virtual screen width              1280
SM_CYVIRTUALSCREEN       Virtual screen height             1024
SM_CMONITORS             Number of monitors                   1
```