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.

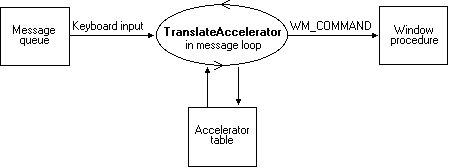
Ignoring Keyboard Input

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.



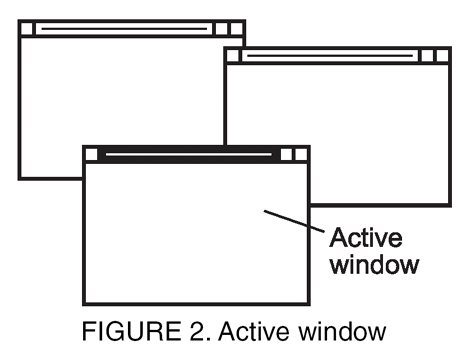
When a key is pressed, only one window procedure receives the corresponding keyboard message, which includes a handle to the receiving window.

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.



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.



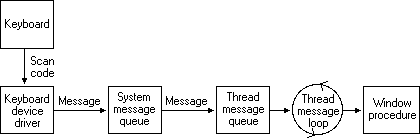
Windows, acting as the city's traffic management system, doesn't simply let all these vehicles flood the streets at once. Instead, it employs a designated holding area, analogous to the system message queue, to temporarily park the incoming vehicles until the traffic flow is smooth and organized.



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.



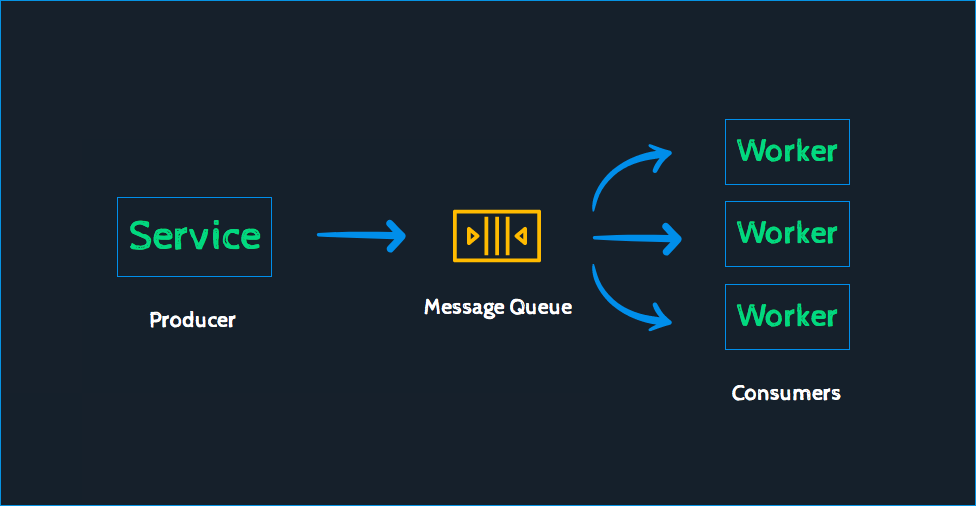
Windows meticulously manages this queue, ensuring that each message is processed sequentially and routed to the appropriate application only when the previous message has been handled.



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

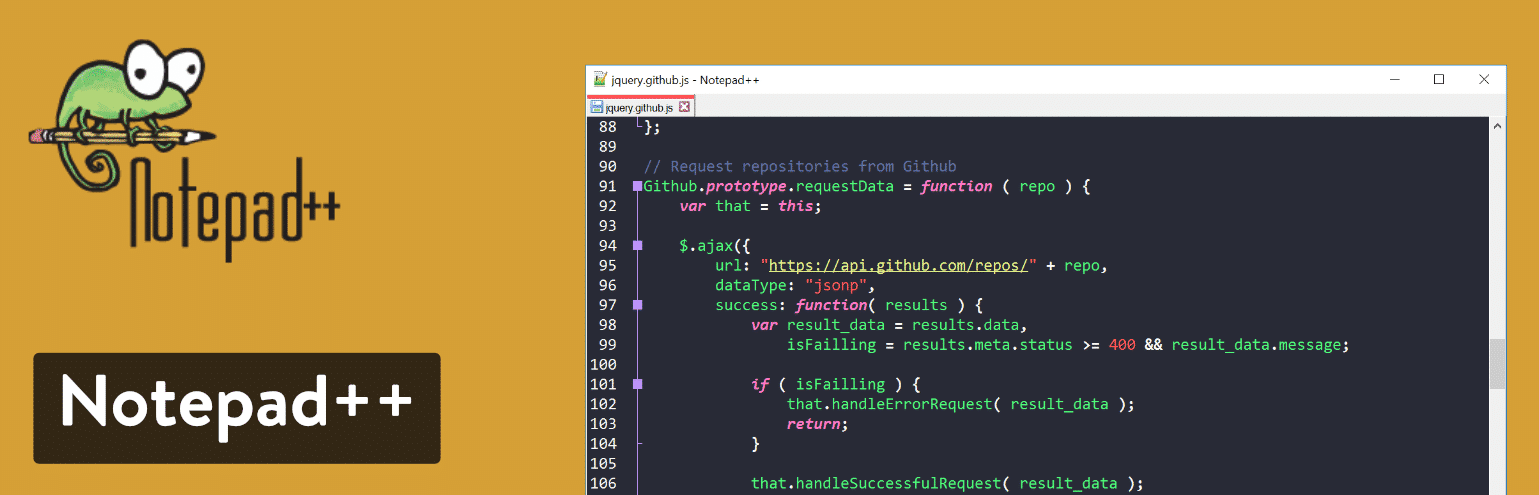
Character messages convey information about the character that was generated by the keystroke. They include the following details:

* *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.



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.