CHARACTER MESSAGES

Character messages are messages that represent characters typed by the user. There are four types of character messages:

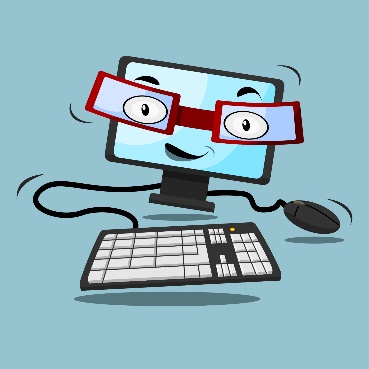
WM\_CHAR: This message is sent to the window procedure of the active window when a character is typed. The lParam parameter of this message is the same as the lParam parameter of the WM\_KEYDOWN message that generated the character code. The wParam parameter is an ANSI or Unicode character code.



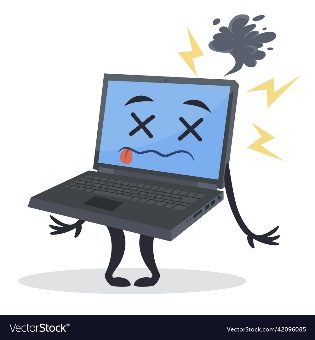
WM\_DEADCHAR: This message is sent to the window procedure of the active window before a character is displayed. The lParam parameter of this message is the same as the lParam parameter of the WM\_KEYDOWN message that generated the character code. The wParam parameter is an ANSI or Unicode character code.



WM\_SYSCHAR: This message is sent to the window procedure of the active window when a system character is typed. A system character is a character that is not displayed, but that can be used to control the window, such as the Alt key or the Escape key. The lParam parameter of this message is the same as the lParam parameter of the WM\_SYSKEYDOWN message that generated the system character code. The wParam parameter is an ANSI or Unicode character code.



WM\_SYSDEADCHAR: This message is sent to the window procedure of the active window before a system character is displayed. The lParam parameter of this message is the same as the lParam parameter of the WM\_SYSKEYDOWN message that generated the system character code. The wParam parameter is an ANSI or Unicode character code.



Dead Characters

A dead character is a character that requires additional input before it can be displayed. For example, the character é is a dead character because it requires the user to type the accent mark (`) after the e. Dead characters are sent to the window procedure as WM\_DEADCHAR messages. The window procedure can then decide whether to display the dead character or wait for additional input.



Nonsystem Characters vs. System Characters

Nonsystem characters are characters that are not used to control the window, such as letters, numbers, and punctuation marks. System characters are characters that are used to control the window, such as the Alt key and the Escape key.

ANSI vs. Unicode

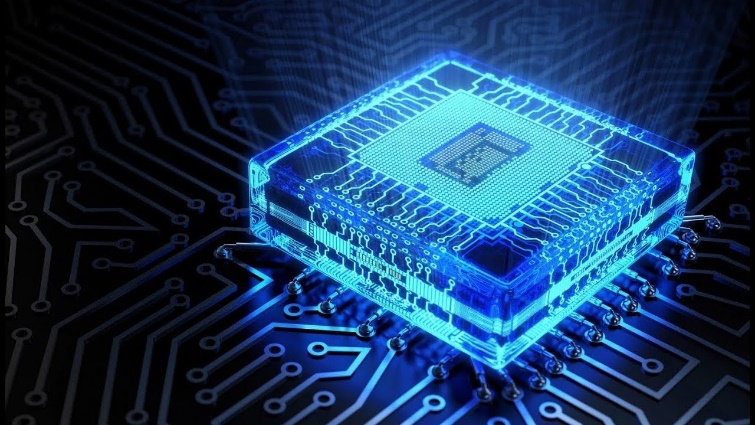
ANSI is an 8-bit character encoding that can represent 256 characters.

Unicode is a 16-bit character encoding that can represent over 1 million characters. Windows programs can use either ANSI or Unicode character codes.



How to Process Character Messages

In most cases, Windows programs can process the WM\_CHAR message while ignoring the other three character messages. The lParam parameter of the four character messages is the same as the lParam parameter for the keystroke message that generated the character code message. However, the wParam parameter is not a virtual key code. Instead, it is an ANSI or Unicode character code.



How the Window Procedure Knows Whether Character Data is ANSI or Unicode

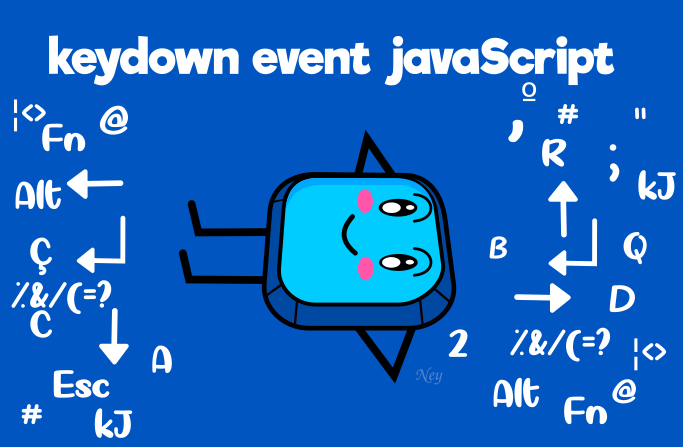
The window procedure knows whether character data is ANSI or Unicode by looking at the window class that the window procedure is associated with. If the window class was registered with RegisterClassA, then the character data is ANSI. If the window class was registered with RegisterClassW, then the character data is Unicode.



Character Messages and Keystroke Messages

Character messages are generated from keystroke messages by the TranslateMessage function. This means that character messages are always delivered to the window procedure sandwiched between keystroke messages. The order of messages is as follows:

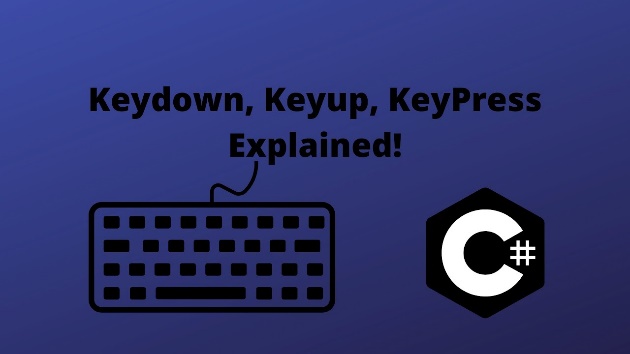
WM\_KEYDOWN: This message is sent when a key is pressed down. The wParam parameter contains the virtual key code of the key that was pressed. The lParam parameter contains additional information about the keystroke, such as the state of the shift and control keys.



WM\_CHAR (or WM\_DEADCHAR): If the keystroke produces a character, a character message is sent after the WM\_KEYDOWN message. The wParam parameter of the character message contains the ANSI or Unicode character code of the character. The lParam parameter is the same as the lParam parameter of the WM\_KEYDOWN message.



WM\_KEYUP: This message is sent when a key is released. The wParam parameter contains the virtual key code of the key that was released. The lParam parameter is the same as the lParam parameter of the WM\_KEYDOWN message.



Example: Typing the Letter "A"

If you type the letter "a" without pressing the Shift key, the following messages are sent to the window procedure:

* WM\_KEYDOWN: wParam = 0x41, lParam = 0
* WM\_CHAR: wParam = 0x61, lParam = 0
* WM\_KEYUP: wParam = 0x41, lParam = 0

Example: Typing the Letter "A" with Shift Key

If you type the letter "A" with the Shift key pressed, the following messages are sent to the window procedure:

* WM\_KEYDOWN: wParam = 0x10, lParam = 0x00000001
* WM\_KEYDOWN: wParam = 0x41, lParam = 0x00000001
* WM\_CHAR: wParam = 0x41, lParam = 0x0000000
* WM\_KEYUP: wParam = 0x41, lParam = 0x00000001
* WM\_KEYUP: wParam = 0x10, lParam = 0x00000001

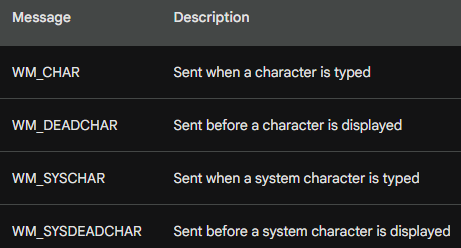
Handling Repeat Count

If you hold down a key so that the typematic action generates keystrokes, you will get a character message for each WM\_KEYDOWN message. The character message will have the same Repeat Count as the WM\_KEYDOWN message.

Determining ANSI or Unicode Character Codes

The window procedure can determine whether a character message is ANSI or Unicode by calling the IsWindowUnicode function. This function takes an HWND parameter and returns TRUE if the window procedure for that window receives Unicode messages.

*This table summarizes the four character messages:*

**

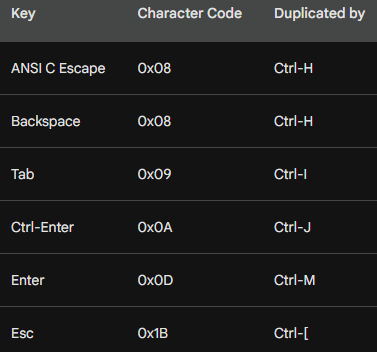
Ctrl Key Combinations and ASCII Control Characters

The Ctrl key, when combined with a letter key, generates ASCII control characters from 0x01 (Ctrl-A) through 0x1A (Ctrl-Z). These control characters are used to control various functions within a program or operating system.



Duplicate Control Characters

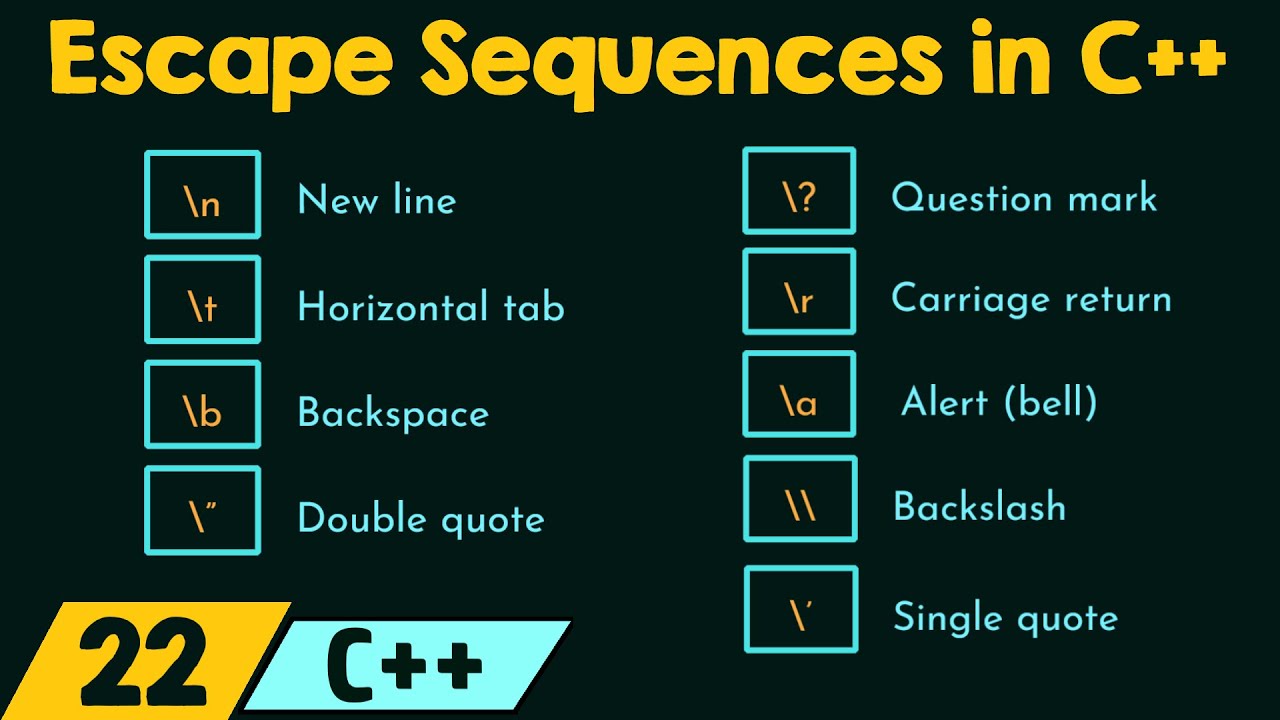
Several of these control characters are also generated by individual keys on the keyboard, as shown in the table below:



ANSI C Escape Codes

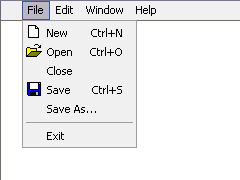
The rightmost column in the table shows the escape code defined in ANSI C to represent the character codes for these keys.

Escape codes are sequences of characters that start with the backslash () character, followed by one or more additional characters. They are used to represent non-printing characters, such as control characters.



Menu Accelerators

In Windows programs, the Ctrl key combination with letter keys is often used for menu accelerators. Menu accelerators are shortcuts that allow users to quickly access menu options using the keyboard.



For example, the Ctrl-O key combination might be used to open the Open File dialog box. In this case, the letter keys are not translated into character messages. Instead, they are interpreted as menu accelerator commands.

Processing Tab, Enter, Backspace, and Escape Keys

The Tab, Enter, Backspace, and Escape keys have a dual nature: they can generate both ASCII control characters and virtual key codes.

This raises the question of whether to process these keys during WM\_CHAR processing or WM\_KEYDOWN processing.

Traditional Approach

Traditionally, these keys have been processed during WM\_KEYDOWN processing. This is because they were originally intended to generate ASCII control characters, which are used to control various functions within a program or operating system.

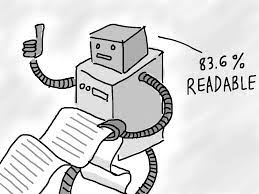
Modern Approach

However, there are several reasons why it is often more convenient to process these keys during WM\_CHAR processing:

Consistency: Processing these keys during WM\_CHAR processing provides a more consistent approach to handling keyboard input. This is because all other character keys are processed during WM\_CHAR processing.



Readability: Processing these keys during WM\_CHAR processing can make the code more readable, as it avoids the need to switch between WM\_CHAR and WM\_KEYDOWN processing.



Efficiency: Processing these keys during WM\_CHAR processing can be more efficient, as it avoids the need to extract the ASCII control character from the virtual key code.

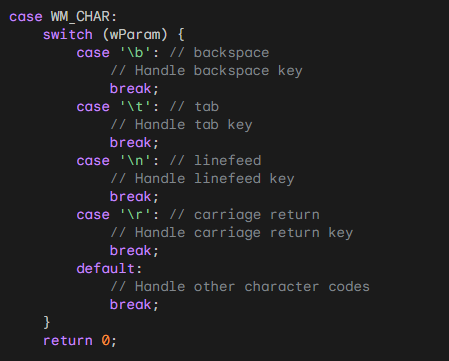


Recommended Approach

Based on these considerations, it is generally recommended to process the Tab, Enter, Backspace, and Escape keys during WM\_CHAR processing. This approach provides a more consistent, readable, and efficient way to handle keyboard input.

Example Code

Here is an example of how to process the Tab, Enter, Backspace, and Escape keys during WM\_CHAR processing:



This code will handle the Tab, Enter, Backspace, and Escape keys as control characters. All other character codes will be handled by the default case.

Dead Characters

Dead characters are characters that require additional input before they can be displayed. For example, the character é is a dead character because it requires the user to type the accent mark (`) after the e.

Dead characters are sent to the window procedure as WM\_DEADCHAR messages. The window procedure can then decide whether to display the dead character or wait for additional input.

Dead Character Processing

Windows programs can usually ignore WM\_DEADCHAR and WM\_SYSDEADCHAR messages, but it is important to understand how they work. This is because dead characters are used on some non-U.S. English keyboards to add diacritic marks to letters.

Example: German Keyboard

On a German keyboard, the key that is in the same position as the +/= key on a U.S. keyboard is a dead key for the grave accent (`) when shifted and the acute accent (´) when unshifted.

When a user presses this dead key, the window procedure receives a WM\_DEADCHAR message with wParam equal to the ASCII or Unicode code for the diacritic by itself.

When the user then presses a letter key that can be written with this diacritic (such as the A key), the window procedure receives a WM\_CHAR message where wParam is the ANSI code for the letter `a' with the diacritic.

Error Handling

The Windows logic even has built-in error handling: If the dead key is followed by a letter that can't take a diacritic (such as `s'), the window procedure receives two WM\_CHAR messages in a row:

* The first message has wParam equal to the ASCII code for the diacritic by itself (the same wParam value delivered with the WM\_DEADCHAR message).
* The second message has wParam equal to the ASCII code for the letter `s'.

Testing Dead Characters

The best way to get a feel for dead characters is to see them in action. To do this, you need to load a foreign keyboard that uses dead keys, such as the German keyboard that I described earlier. You can do this in the Control Panel by selecting Keyboard and then the Language tab.

Once you have loaded a foreign keyboard, you can use an application like KEYVIEW1 to see the details of every keyboard message that a program receives. This will help you to understand how dead characters are processed by Windows.

*KeyView1 program In chapter 6 KeyView1 to see this in action…*

Program Structure:

The program is implemented in C and utilizes the Win32 API for creating a graphical user interface. Let's break down the key aspects of the program:

1. Window Procedure (WndProc):

The heart of the program is the window procedure, WndProc. This function is responsible for handling various messages sent to the window. In this case, it processes messages such as WM\_CREATE, WM\_SIZE, WM\_KEYDOWN, WM\_CHAR, WM\_SYSKEYUP, and others.

2. Initialization and Window Creation:

The WinMain function initializes the program by registering a window class, creating a window, and setting it up for display. The window class specifies the appearance and behavior of the window.

3. Displaying Keyboard Messages:

The core functionality of KEYVIEW1 involves capturing and displaying keyboard messages. The program maintains an array of MSG structures to store information about each message. When a keyboard-related message is received (e.g., WM\_KEYDOWN or WM\_CHAR), the program updates the array with relevant details.

4. Dynamic Memory Allocation:

To adapt to changes in the window size, the program dynamically allocates memory for the array of MSG structures. This ensures that the array can accommodate the information for the maximum number of lines that can be displayed in the window.

5. Scrolling and Display Update:

The program handles scrolling to show the most recent keyboard messages. When a new message is received, the array is rearranged, and the display is scrolled up to make room for the new information. The ScrollWindow function is employed for this purpose.

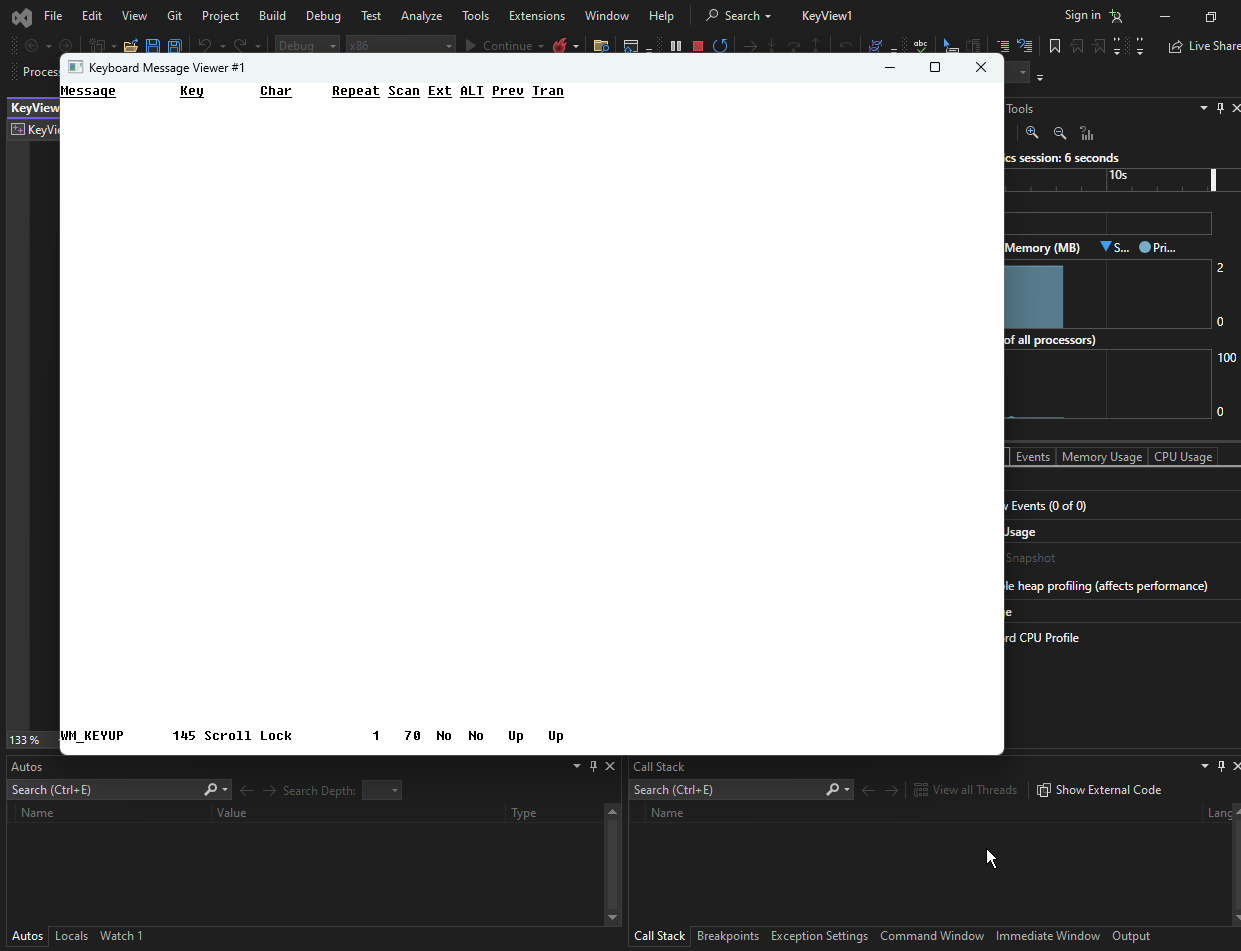
6. Painting the Display:

The WM\_PAINT message triggers the painting of the client area. The program uses the TextOut function to display information about each keyboard message in a tabular format. The displayed columns include the message type, virtual key code, character code, and various flags extracted from the lParam parameter.

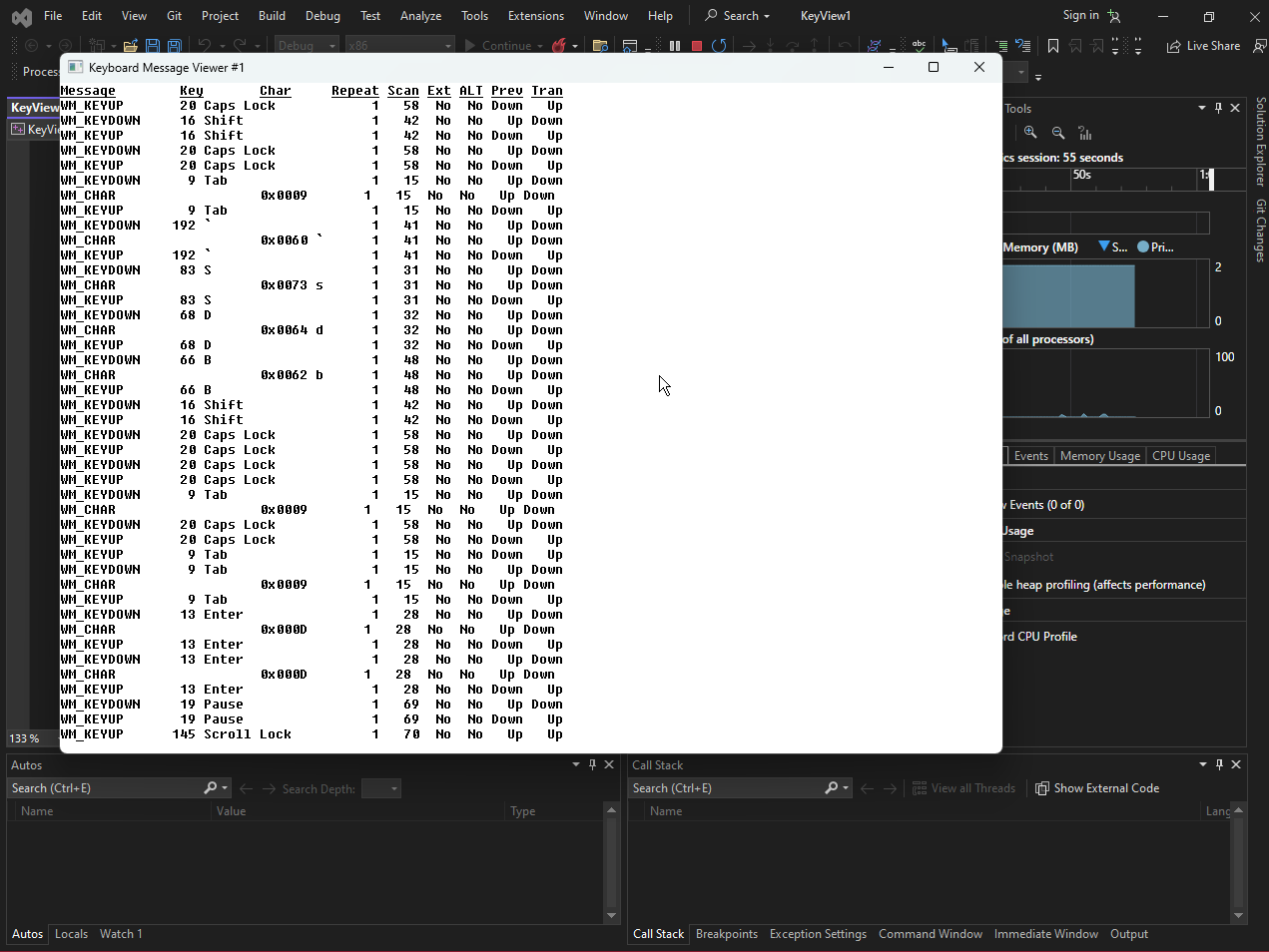
7. Clean-Up on Window Closure:

When the window is closed (WM\_DESTROY), the program releases any allocated resources and posts a quit message to terminate the application.

The apps is empty at first:



Clicking any key, it registers:



Fixed-Pitch Font Usage:

KEYVIEW1 utilizes a fixed-pitch font for its columnar display, ensuring uniform character width.

The relevant code for this is: SelectObject(hdc, GetStockObject(SYSTEM\_FIXED\_FONT));

Header Display:

KEYVIEW1 includes a header at the top of the client area, identifying nine columns for keyboard and character information.

Underlining Text Approach:

Instead of creating an underlined font, the program defines two character string variables, szTop (text) and szUnd (underlining), displaying both during the WM\_PAINT message.

Opaque Mode Issue:

In the "opaque" mode of text display, Windows erases the character background area while showing a character. This could cause the second character string (szUnd) to erase the first (szTop).

Transparent Mode Solution:

To prevent the erasing issue, the program switches the device context into "transparent" mode using SetBkMode(hdc, TRANSPARENT);.

Underlining Compatibility:

The chosen method of underlining is effective when using a fixed-pitch font, ensuring that the underline character aligns correctly with the characters it underlines.

*Conclusion:*

In summary, KEYVIEW1 serves as a diagnostic tool for understanding keyboard and character messages in a Windows environment.

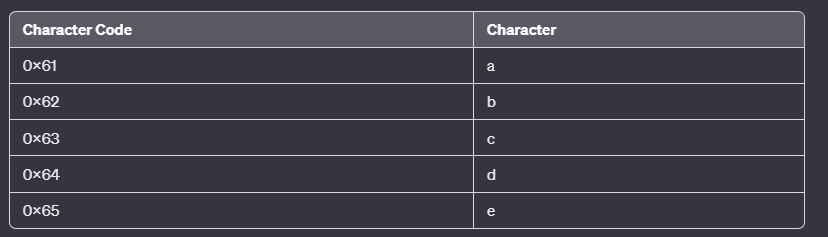
It provides a real-time display of keyboard interactions, offering developers insights into the messages generated by user input. The program's dynamic allocation and scrolling mechanisms ensure that it can adapt to different window sizes and accommodate a history of keyboard events.

Foreign-Language Keyboard Problem

When using a foreign-language keyboard layout, Windows programs may display incorrect characters. This is because the programs are not aware of the new keyboard layout and are still interpreting the character codes according to the default English keyboard layout.

Example: German Keyboard

For example, if you switch to the German keyboard layout and type the letters "abcde," you will get the following WM\_CHAR messages:

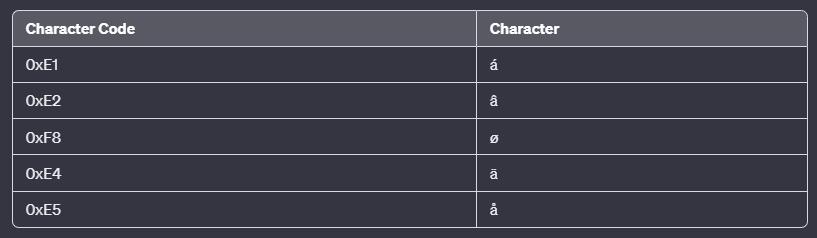


These are the same character codes that you would get if you typed the same letters on an English keyboard.

However, the displayed characters will be different. This is because the German keyboard layout maps the a, b, c, d, and e keys to different character codes than the English keyboard layout.

Example: Greek Keyboard

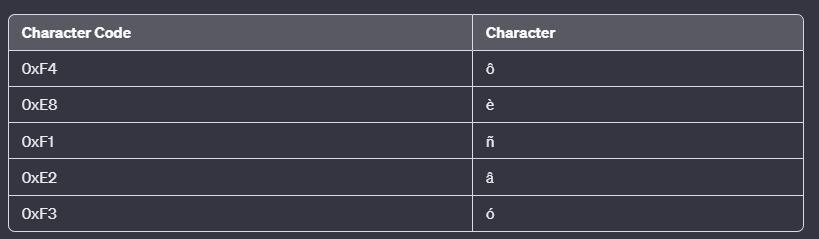
If you switch to the Greek keyboard layout and type "abcde," you will get the following WM\_CHAR messages:



These are not the characters that you would expect to see. This is because the Greek keyboard layout maps the a, b, c, d, and e keys to different character codes than the English keyboard layout, and the Greek version of Windows is able to interpret these character codes correctly.

Example: Russian Keyboard

If you switch to the Russian keyboard layout and type "abcde," you will get the following WM\_CHAR messages:



These are not the characters that you would expect to see. This is because the Russian keyboard layout maps the a, b, c, d, and e keys to different character codes than the English keyboard layout, and the Russian version of Windows is able to interpret these character codes correctly.

Solution

The solution to the foreign-language keyboard problem is to inform GDI of the new keyboard layout so that GDI can interpret the character codes correctly.

This can be done by calling the SetWindowsHookEx function with the WH\_KEYBOARD\_LL hook type. The hook procedure can then intercept the WM\_CHAR messages and translate the character codes according to the new keyboard layout.