CHARACTER MESSAGES

We are now crossing the bridge from "Hardware" to "Text."

- **Previous Section:** "The user pressed Button #30." (WM_KEYDOWN)

- **This Section:** "The user typed the letter 'a'." (WM_CHAR)

1. The Big Concept: Translation

Windows takes the raw keystroke messages you learned earlier, processes them (checking Shift, Caps Lock, and Language settings), and translates them into **Character Messages**.

You usually ignore WM_KEYDOWN for text entry and listen for **WM_CHAR** instead.

---

2. The Four Character Messages

A. WM_CHAR (The Main Event)

This is the most important message for text input.

- **When it happens:** When a user presses a key that corresponds to a printable character (A-Z, 0-9, symbols).

- **wParam:** Contains the **Character Code** (ASCII or Unicode value).

  - *Example:* If the user types "A", wParam is 65.

- **lParam:** Contains the physical details (Repeat count, Scan code) copied directly from the WM_KEYDOWN message that caused this.

B. WM_DEADCHAR (The Accent Helper)

This is for international characters (diacritics).

- **The Concept:** A "Dead Key" is a key that doesn't make a character appear immediately. It waits for the *next* keystroke to combine them.

- **Example:** On many keyboards, to type **é**, you first press the ´ (Acute Accent) key.

  1. User presses ´: Nothing appears on screen. The cursor does not move (it is "dead"). Windows sends WM_DEADCHAR.

  2. User presses **e**: Windows combines them and sends WM_CHAR for **é**.

C. WM_SYSCHAR (System Characters)

- **The Concept:** These are characters generated while holding the **Alt** key.

- **Usage:** Windows uses these for menu shortcuts (e.g., Alt+F to open the "File" menu).

- **Your Job:** Usually, you ignore these. You pass them to DefWindowProc so Windows can handle the menu navigation for you.

D. WM_SYSDEADCHAR (System Dead Characters)

- **The Concept:** A "Dead Key" pressed while holding **Alt**.

- **Usage:** Extremely rare. Unless you are writing very specific system tools for international keyboards, you will never touch this.

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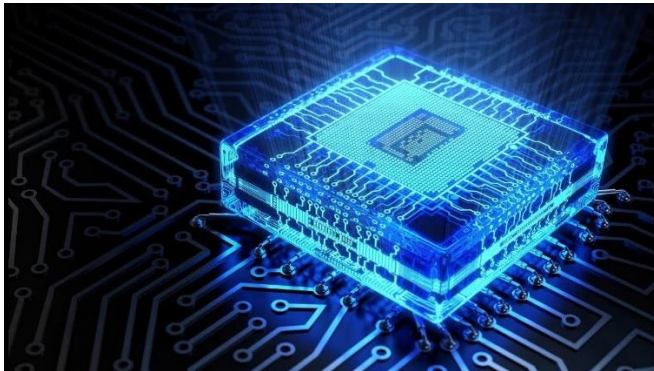
How the Window Procedure Knows Whether Character Data is ANSI or Unicode

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.
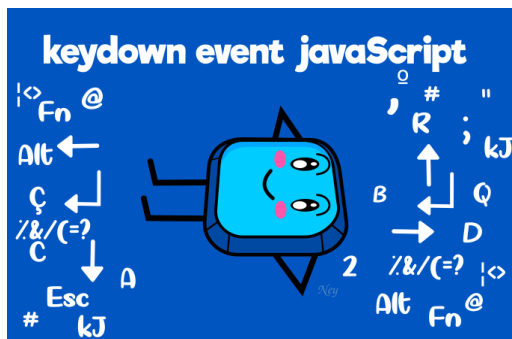
WM_CHAR (or WM_DEADCHAR): If the keystroke produces a character, a character message is sent after the WM_KEYDOWN message. The wParam parameter of the character message contains the ANSI or Unicode character code of the character. The lParam parameter is the same as the lParam parameter of the WM_KEYDOWN 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:

Message	Description
WM_CHAR	Sent when a character is typed
WM_DEADCHAR	Sent before a character is displayed
WM_SYSCHAR	Sent when a system character is typed
WM_SYSDEADCHAR	Sent before a system character is displayed

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:

| Key	Character Code	Duplicated by
ANSI C Escape	0x08	Ctrl-H
Backspace	0x08	Ctrl-H
Tab	0x09	Ctrl-I
Ctrl-Enter	0x0A	Ctrl-J
Enter	0x0D	Ctrl-M
Esc	0x1B	Ctrl-[

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.
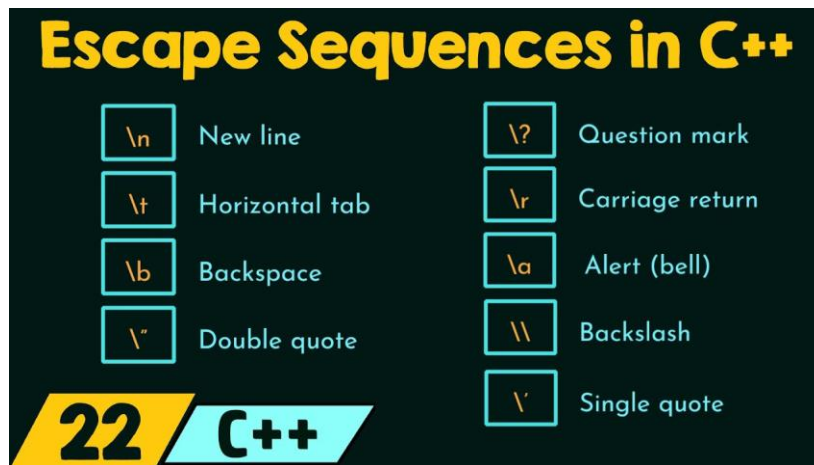
Menu Accelerators

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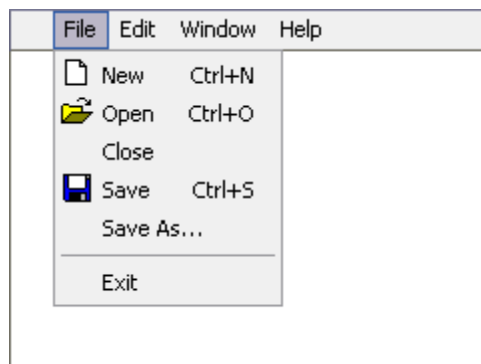
For example, the Ctrl-O key combination might be used to open the Open File dialog box. In this case, the letter keys are not translated into character messages. Instead, they are interpreted as menu accelerator commands.

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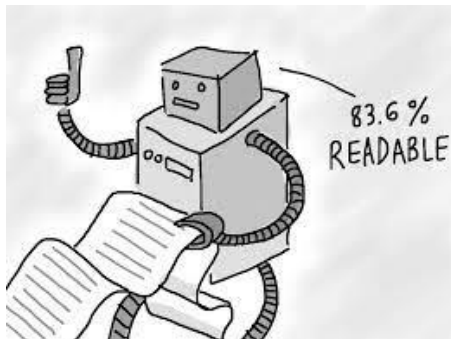
Efficiency: Processing these keys during WM_CHAR processing can be more efficient, as it avoids the need to extract the ASCII control character from the virtual key code.

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:

```
case WM_CHAR:
    switch (wParam) {
        case '\b': // backspace
            // Handle backspace key
            break;
        case '\t': // tab
            // Handle tab key
            break;
        case '\n': // linefeed
            // Handle linefeed key
            break;
        case '\r': // carriage return
            // Handle carriage return key
            break;
        default:
            // Handle other character codes
            break;
    }
    return 0;
```

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.
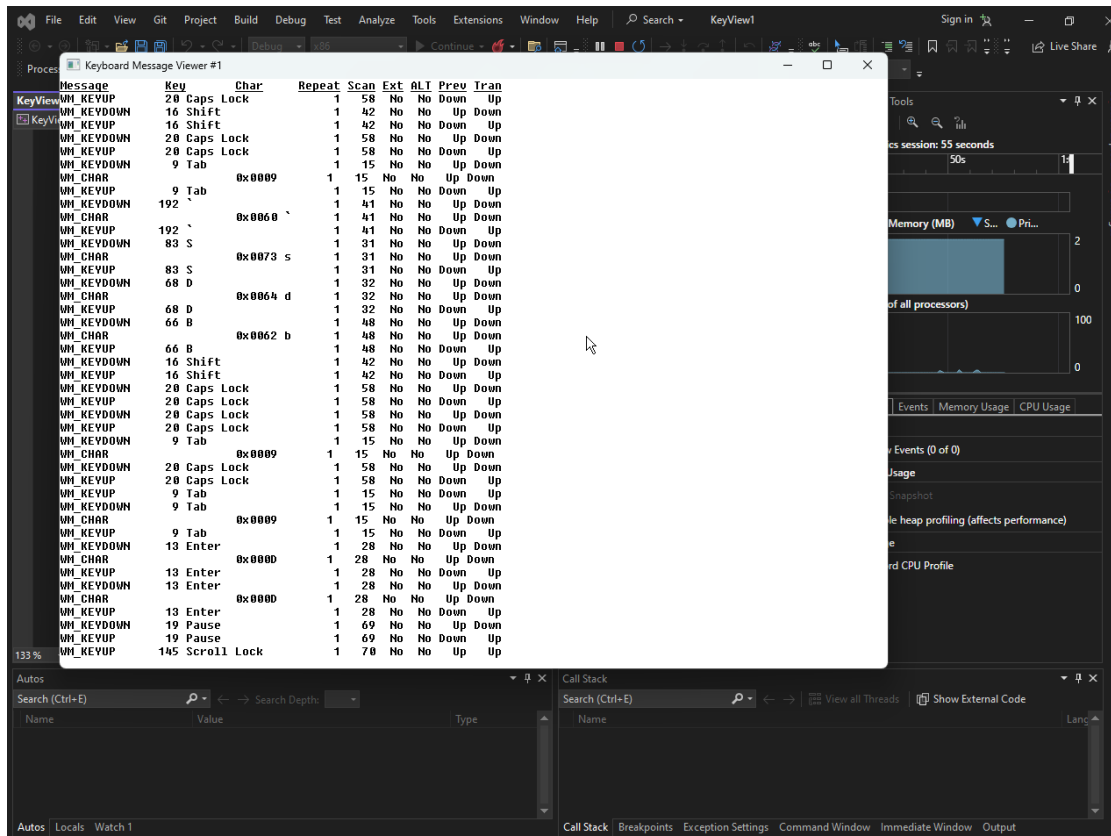
7. Clean-Up on Window Closure:

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:
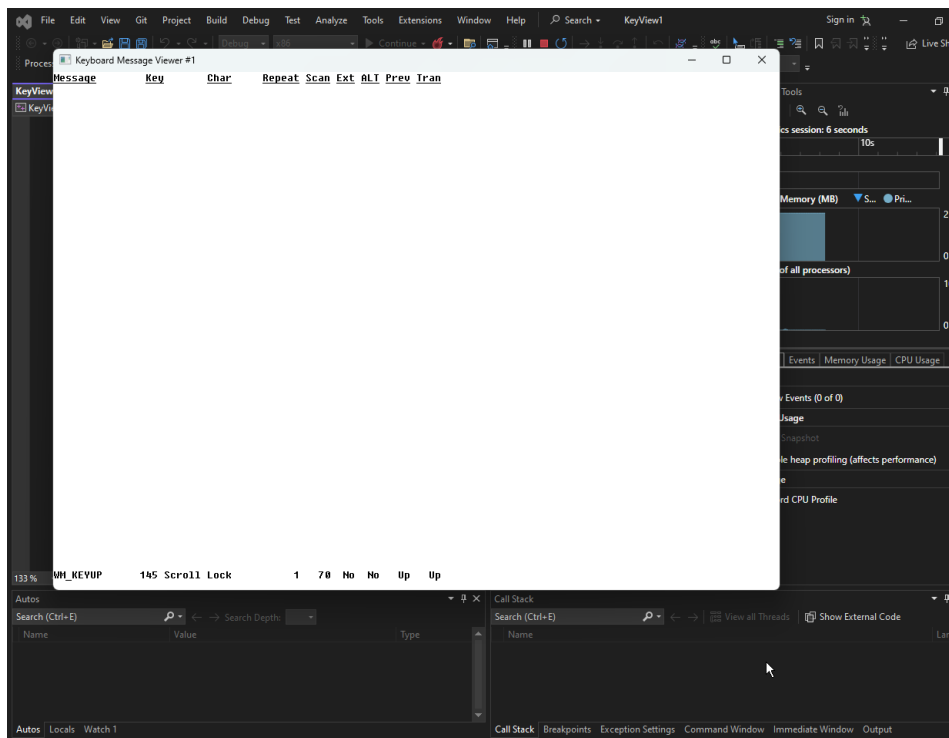
Clicking any key, it registers:

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:

Character Code	Character
0×61	a
0×62	b
0×63	c
0×64	d
0×65	e

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:

Character Code	Character
0xE1	á
0xE2	â
0xF8	ø
0xE4	ä
0xE5	å

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:

Character Code	Character
0xF4	ô
0xE8	è
0xF1	ñ
0xE2	â
0xF3	ó

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.

CHARACTER SETS AND FONTS

A character set is a collection of characters, such as the letters of the alphabet, numbers, and punctuation marks.

A font is a collection of glyphs, which are visual representations of characters. Glyphs can be defined in various ways, such as using bitmaps or vector strokes.

Types of Fonts

Windows supports three types of fonts:

Bitmap fonts: Bitmap fonts are defined using an array of bits, which corresponds to the pixels on the screen. Bitmap fonts can be scaled to any size, but they can appear jagged if scaled too large.

Vector fonts: Vector fonts are defined using lines and curves, which can be scaled to any size without losing quality. However, vector fonts are not as widely used as bitmap fonts.

TrueType fonts: TrueType fonts are a type of outline font that is defined using both lines and curves, as well as hints to improve the appearance of the font at different sizes. TrueType fonts are the most common type of font used on Windows.

Bitmap   TrueType

Character Set and Font Compatibility

The character set used by a font must be compatible with the character set that the application is using. If the character set is not compatible, the application will not be able to display the characters correctly.

For example, if an application is using the Latin-1 character set, but the font is only designed for the ASCII character set, then the application will not be able to display the accented characters in the Latin-1 character set.

The KEYVIEW1 Problem

The KEYVIEW1 program is displaying incorrect characters because it is using the system font, which is a bitmap font that is not designed for the character sets of the foreign languages that the program is displaying.

To solve the KEYVIEW1 problem, the program should use a TrueType font that is designed for the character sets of the foreign languages that the program is displaying. The program can use the GetStockObject function to get a handle to a TrueType font, or it can load a TrueType font from a file.

Stock Fonts in Windows

The stock fonts in Windows are a collection of fonts that are provided by the operating system. These fonts can be used by applications to display text on the screen. There are several different stock fonts available, each with its own unique characteristics.

The System Font

The system font is the default font that is used by Windows for displaying text. It is a bitmap font that is designed to be easily readable on the screen. The system font is stored in the VGAFIX.FON file for video displays with a resolution of 96 dpi and the 8514FIX.FON file for video displays with a resolution of 120 dpi.
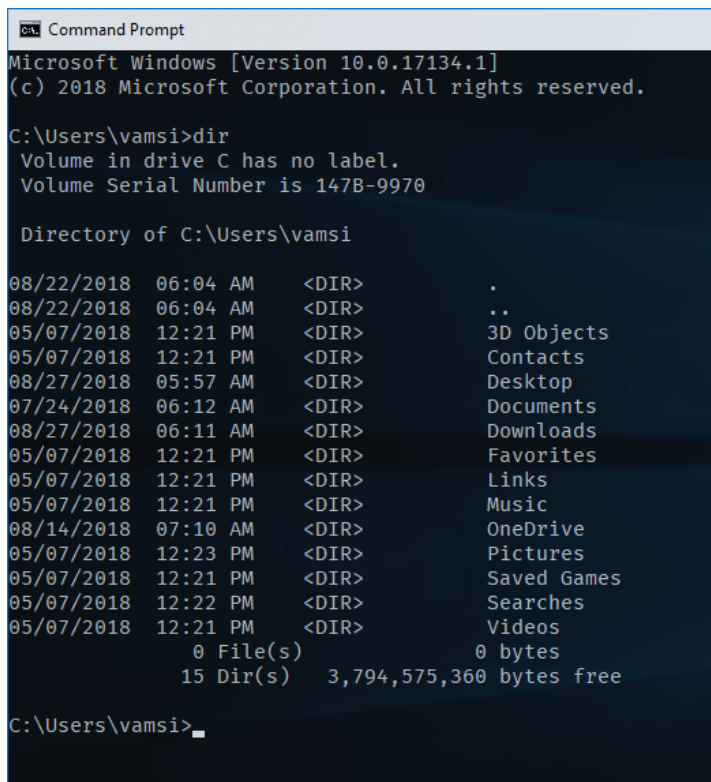

The FixedSys Font

The FixedSys font is a fixed-pitch font that is used by Windows for displaying text in menus, dialog boxes, and other controls. It is a bitmap font that is stored in the VGAFIX.FON file for video displays with a resolution of 96 dpi and the 8514FIX.FON file for video displays with a resolution of 120 dpi.
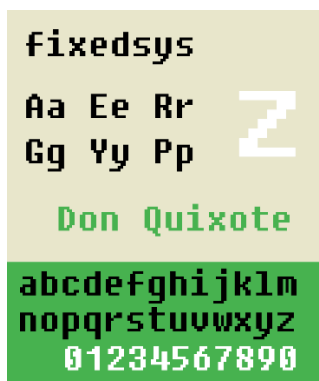
The Terminal Font

The Terminal font is a monospaced font that is used by Windows for displaying text in the Command Prompt window. It is a bitmap font that is stored in the VGAOEM.FON file for video displays with a resolution of 96 dpi and the 8514OEM.FON file for video displays with a resolution of 120 dpi.

The MS Sans Serif Font

The MS Sans Serif font is a bitmap font that is used by Windows for displaying text in standard controls and user interface items. It is stored in the SSERIFE.FON file and is available in point sizes of 8, 10, 12, 14, 18, and 24.

The DEFAULT_GUI_FONT Identifier

The DEFAULT_GUI_FONT identifier can be used with the GetStockObject function to obtain a handle to the MS Sans Serif font. The point size of the font that is returned will be based on the display resolution that the user has selected in the Display applet of the Control Panel.

Other Stock Fonts

There are three other stock fonts that are available in Windows:

ANSI_FIXED_FONT: This font is a fixed-pitch font that is used by Windows for displaying text in code editor windows.
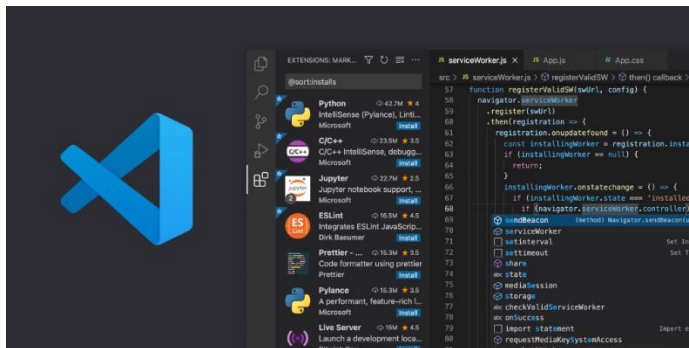
ANSI_VAR_FONT: This font is a proportional font that is used by Windows for displaying text in most applications.

DEVICE_DEFAULT_FONT: This font is the default font that is used by the current device context.
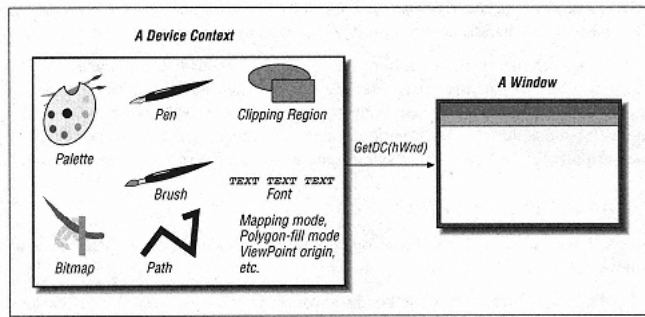

Program code in stonkfont folder chapter 6…

- The STOKFONT program is a simple Windows application that displays the 256 characters of a selected stock font in a grid. The user can select a font using the scroll bar or the cursor movement keys. The program also displays the identifier of the selected font, the typeface name of the font, and the character set identifier of the font.
- The main function of the STOKFONT program is WinMain. This function creates a window and initializes the program's variables. The main loop of the program is located in the while loop that starts after the call to GetMessage. This loop gets messages from the message queue and dispatches them to the appropriate window procedure.
- The window procedure for the STOKFONT program is WndProc. This procedure handles all of the messages that are sent to the window. The majority of the code in this procedure is responsible for drawing the font grid and updating the font information when the user selects a new font.
- The code that draws the font grid is located in the WM_PAINT message handler. This code first selects the stock font that the user has selected. Then, it iterates over all of the characters in the font and draws them in the grid. The code also draws the headings for the grid, which show the hexadecimal values of the character codes.

➢ The code that updates the font information is located in the WM_DISPLAYCHANGE, WM_VSCROLL, and WM_KEYDOWN message handlers. These message handlers update the identifier of the selected font, the typeface name of the font, and the character set identifier of the font.

➢ The character set identifier is a crucial piece of information in understanding how Windows deals with foreign-language versions of Windows. The character set identifier tells Windows which character set the font is using. This information is used by Windows to map character codes to glyphs.

Output:
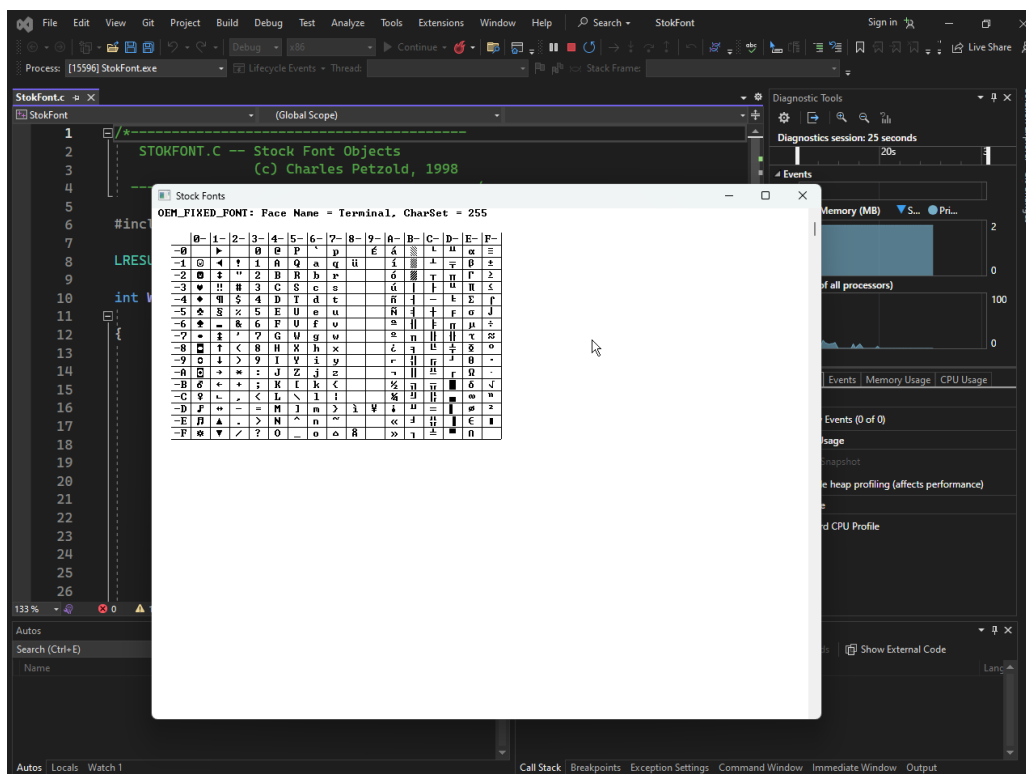
The OEM Character Set

The OEM character set is an extended ASCII character set that was developed by IBM for use with the original IBM PC. It includes line-drawing and block-drawing characters, accented letters, Greek letters, math symbols, and some miscellany. The OEM character set was used by many character-mode MS-DOS programs and is still supported by Windows for compatibility reasons.

OEM Extended ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	▒	▓	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	β	Γ	π	Σ	σ	µ	τ	Φ	θ	Ω	δ	∞	φ	ε	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

cplusplus.com

The ANSI Character Set

The ANSI character set is an extended ASCII character set that is based on the ISO 8859-1 standard. It is also known as Latin 1, Western European, or code page 1252. The ANSI character set includes the ASCII characters, plus 64 additional characters that are commonly found in Western European languages. The ANSI character set is the default character set for Windows applications.

```
 · · · · · · · · · · · · · · · ·
 · · · · · · · · · · · · · · · ·
  !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~.
€.‚ƒ„…†‡ˆ‰Š‹Œ.Ž.
.''""•–—˜™š›œ.žŸ
 ¡¢£¤¥¦§¨©ª«¬®¯
°±²³´µ¶·¸¹º»¼½¾¿
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ
ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
àáâãäåæçèéêëìíîï
ðñòóôõö÷øùúûüýþÿ
```

The System Font

The system font is the default font that is used by Windows for displaying text. It is a fixed-pitch font that is based on the OEM character set. The system font is used for displaying text in menus, dialog boxes, and other controls.

The System Fixed Font

The system fixed font is a fixed-pitch font that is based on the ANSI character set. It is used for displaying text in code editor windows.

The Character Set in Foreign-Language Versions of Windows

The default character set for Windows applications is the ANSI character set. However, the default character set for the system font and system fixed font can be different in foreign-language versions of Windows. For example, the default character set for the system font in the Greek version of Windows is the Greek alphabet, and the default character set for the system font in the Russian version of Windows is the Cyrillic alphabet.

Conclusion

The character set used by a Windows application depends on the application and the language version of Windows that is being used. Applications can use the GetStockObject function to obtain a handle to a stock font, such as the system font or system fixed font. The GetTextFace function can be used to obtain the typeface name of a font, and the GetTextMetrics function can be used to obtain information about the font, such as its character set.

UNICODE IN WINDOWS NT

Unicode is a character encoding standard that can represent all of the characters in the world's major writing systems. Windows NT supports Unicode, which means that Windows NT applications can use Unicode to manipulate and display text in any language.

KEYVIEW1 Unicode Version

The Unicode version of KEYVIEW1 is a modified version of the original KEYVIEW1 program that uses Unicode.

The main difference between the two programs is that the Unicode version of KEYVIEW1 uses the RegisterClassW function to register the "KeyView1" window class, while the original KEYVIEW1 program uses the RegisterClassA function.

This means that the Unicode version of KEYVIEW1 will receive 16-bit character codes rather than 8-bit character codes.

Running KEYVIEW1 Unicode Version under American English Windows NT

When you run the Unicode version of KEYVIEW1 under the American English version of Windows NT, it will appear to work the same as the non-Unicode version.

This is because the first 256 characters of Unicode are the same as the ANSI character set used in Windows.

However, when you switch to the Greek or Russian keyboard layout, you will see that the Unicode version of KEYVIEW1 displays solid blocks instead of the correct characters. This is because the SYSTEM_FIXED_FONT only has 256 characters.

Differences between Unicode and Non-Unicode KEYVIEW1

The main differences between the Unicode and non-Unicode versions of KEYVIEW1 are as follows:

Character codes: The Unicode version of KEYVIEW1 uses 16-bit character codes, while the non-Unicode version uses 8-bit character codes. This means that the Unicode version of KEYVIEW1 can represent a wider range of characters.

Character display: The Unicode version of KEYVIEW1 uses the TextOutW function to display characters, while the non-Unicode version uses the TextOutA function. The TextOutW function can display Unicode characters, while the TextOutA function can only display 8-bit characters.

Benefits of Unicode Version

The Unicode version of KEYVIEW1 has several benefits over the non-Unicode version:

Correct character display: The Unicode version of KEYVIEW1 will always display the correct characters, regardless of the keyboard layout that is active.

Unambiguous character codes: The 16-bit character codes in the Unicode version of KEYVIEW1 are totally unambiguous, which means that there is no ambiguity about which character is being represented.

Improved GDI support: The Unicode version of KEYVIEW1 takes advantage of GDI's support for Unicode, which means that KEYVIEW1 can take advantage of GDI features that are only available for Unicode characters.

TRUETYPE FONTS AND BIGFONTS

TrueType fonts are a type of font that can contain more than 256 characters. This is in contrast to bitmap fonts, which are limited to 256 characters.

TrueType fonts are able to contain more characters because they use a vector graphics format to store the font glyphs. This means that the font glyphs can be scaled to any size without losing quality.

Big fonts are TrueType fonts that support multiple character sets. This means that they can display characters in multiple languages. The TrueType fonts that are shipped with Windows 98 and Windows NT support the following character sets:

- ➢ Western European (ANSI) character set: This character set is used for virtually all languages except those in the Far East (Chinese, Japanese, and Korean).
- ➢ Baltic character set: This character set is used for Estonian, Latvian, and Lithuanian.
- ➢ Central European character set: This character set is used for Albanian, Czech, Croatian, Hungarian, Polish, Romanian, Slovak, and Slovenian.
- ➢ Cyrillic character set: This character set is used for Bulgarian, Belarusian, Russian, Serbian, and Ukrainian.
- ➢ Greek character set: This character set is used for Greek.
- ➢ Turkish character set: This character set is used for Turkish.

Using Big Fonts in Non-Unicode Programs

You can use big fonts in non-Unicode programs by using the CreateFont function to create a logical font. The CreateFont function has 14 arguments, but you only need to set the following arguments:

- ➢ **lfFaceName:** This argument specifies the name of the font. For example, to use the Arial font, you would specify "Arial" as the value of this argument.
- ➢ **lfCharSet:** This argument specifies the character set of the font. For example, to use the Baltic character set, you would specify BALTIC_CHARSET as the value of this argument.
- ➢ **lfPitch:** This argument specifies whether the font is a fixed-pitch font or a proportional font. For example, to use a fixed-pitch font, you would specify FIXED_PITCH as the value of this argument.

Once you have created a logical font, you can select it into a device context by using the SelectObject function. The SelectObject function takes two arguments: the device context handle and the logical object handle. The logical object handle is the handle of the logical font that you created.

Character Set ID vs. Code Page ID

Windows uses two different numbers to refer to the same character sets: a character set ID and a code page ID. This is one of the confusing quirks in Windows.

Character Set ID

The character set ID is a number that identifies a character set. It is stored in the LOGFONT structure, which is used to describe a logical font. The character set ID is only one byte in size.

Code Page ID

The code page ID is a number that identifies a code page. A code page is a specific mapping of characters to code points. Code pages are typically two bytes in size.

Why Two IDs?

Windows uses two different IDs to refer to the same character sets for several reasons:

Backward compatibility: The character set ID was originally used in Windows 1.0, when memory and storage space were limited. The code page ID was added later to support multiple code pages for the same character set.

International support: The character set ID is used to identify the character set of the OEM character set, which is the character set that is used by the MS-DOS operating system. The code page ID is used to identify the character set of other code pages that are supported by Windows.

How to Use the IDs

The character set ID is used to create a logical font. A logical font is a description of a font that can be selected into a device context. To create a logical font, you use the CreateFont function. The CreateFont function takes several arguments, including the character set ID.

The code page ID is used to convert between characters and code points. To convert a character to a code point, you use the MultiByteToWideChar function. The MultiByteToWideChar function takes several arguments, including the code page ID.

WM_INPUTLANGCHANGE Message

Whenever you change the keyboard layout using the popup menu in the desktop tray, Windows sends your window procedure the WM_INPUTLANGCHANGE message. The wParam message parameter is the character set ID of the new keyboard layout.

KEYVIEW2 Program

The KEYVIEW2 program implements logic to change the font whenever the keyboard layout changes. The program does this by handling the WM_INPUTLANGCHANGE message.

When the program receives the WM_INPUTLANGCHANGE message, it calls the CreateFont function to create a new logical font with the new character set ID. It then selects the new logical font into the device context.

KeyView2 program folder in Chapter 6…
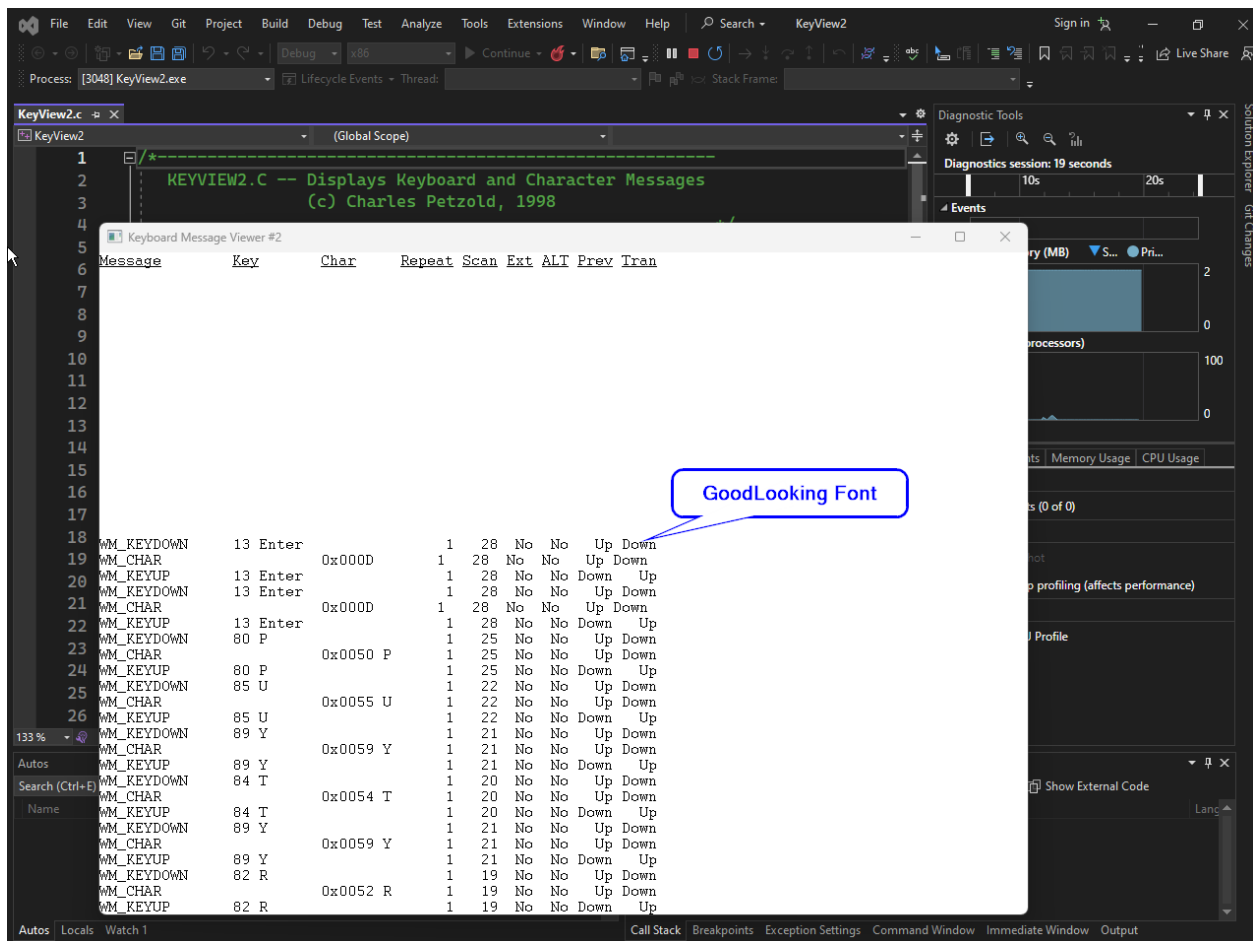
What is the Caret?
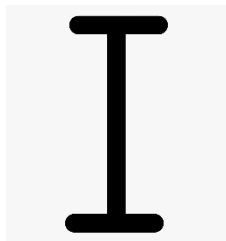
The caret is a small indicator that shows you where the next character you type will appear on the screen. It is commonly represented as a horizontal line or box that is the size of a character, or a vertical line that is the height of a character.

The vertical line caret is recommended when using a proportional font, such as the Windows default system font. It provides a visual cue to the user of where their text will be inserted. This can help to improve the user's typing accuracy and efficiency.

There are five essential caret functions:

- ➢ **CreateCaret:** Creates a caret associated with a window.
- ➢ **SetCaretPos:** Sets the position of the caret within the window.
- ➢ **ShowCaret:** Shows the caret.
- ➢ **HideCaret:** Hides the caret.
- ➢ **DestroyCaret:** Destroys the caret.

There are also functions to get the current caret position (GetCaretPos) and to get and set the caret blink time (GetCaretBlinkTime and SetCaretBlinkTime).

When to Create and Destroy the Caret

A program should not simply create the caret during the WM_CREATE message of its window procedure and destroy it during the WM_DESTROY message. This is because a message queue can support only one caret. Thus, if your program has more than one window, the windows must effectively share the same caret.

Instead, a program should create the caret when it receives the WM_SETFOCUS message and destroy it when it receives the WM_KILLFOCUS message. This ensures that only one window has a caret blinking at any time.

How to Use the Caret

The main rule for using the caret is simple:

- ➢ A window procedure calls CreateCaret during the WM_SETFOCUS message.
- ➢ The caret is created hidden. After calling CreateCaret, the window procedure must call ShowCaret for the caret to be visible.
- ➢ The window procedure must hide the caret by calling HideCaret whenever it draws something on its window during a message other than WM_PAINT.
- ➢ After it finishes drawing on the window, the program calls ShowCaret to display the caret again.

Additional Notes

- ➢ *The effect of HideCaret is additive. If you call HideCaret several times without calling ShowCaret, you must call ShowCaret the same number of times before the caret becomes visible again.*
- ➢ *You can use the GetCaretPos function to get the current position of the caret.*
- ➢ *You can use the GetCaretBlinkTime and SetCaretBlinkTime functions to get and set the caret blink time.*

The typer program in Chapter 6 Folder Typer folder...
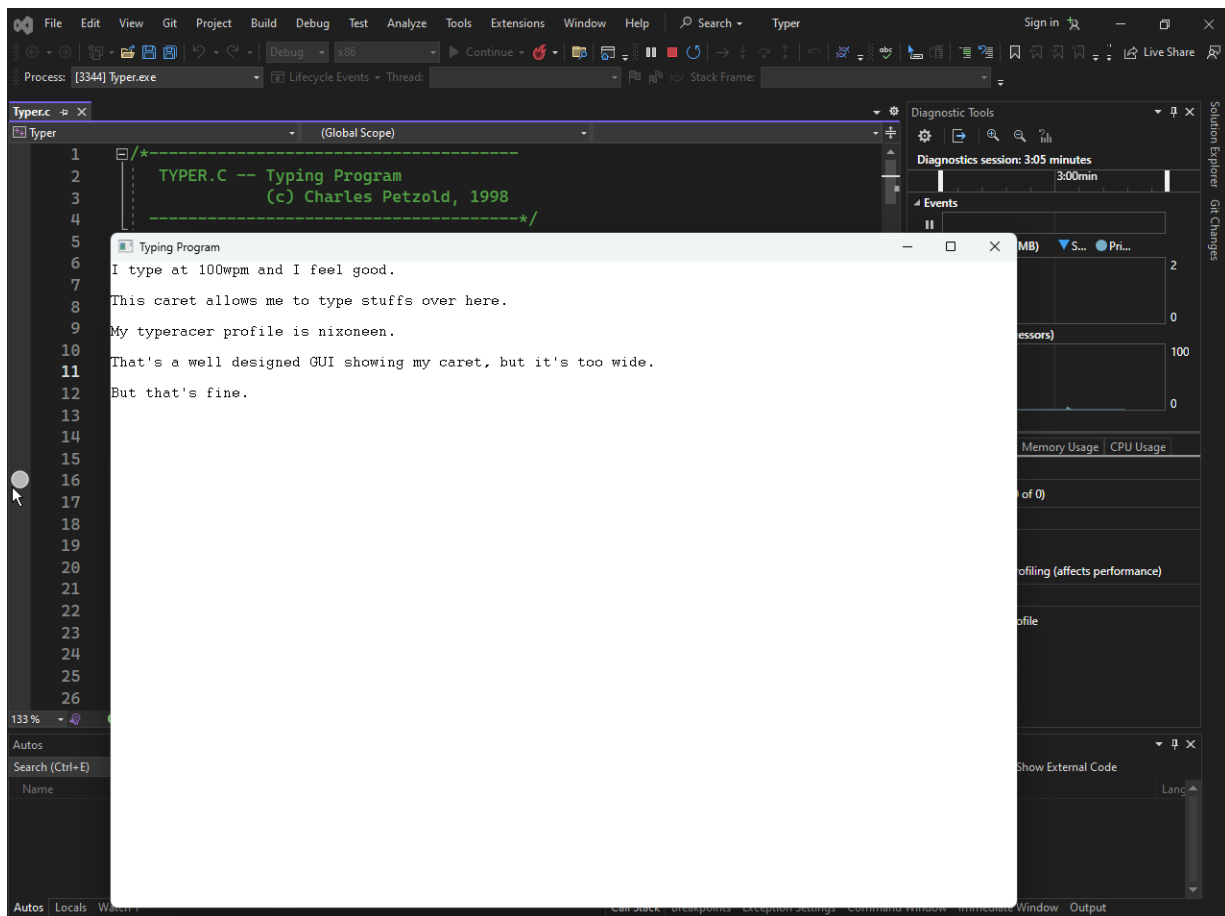
The TYPER program manages the caret by creating it when the window receives the WM_SETFOCUS message and destroying it when the window receives the WM_KILLFOCUS message.

The program also hides the caret whenever it draws something on the window during a message other than WM_PAINT, and shows the caret again after it finishes drawing.

Additionally, the program updates the position of the caret whenever the user types a character, presses the backspace key, or moves the cursor using the arrow keys. This ensures that the caret always indicates where the next character will be typed.

The program:

➢ TYPER uses a fixed-pitch font to simplify the text editor.
➢ TYPER obtains a device context during various messages and selects a fixed-pitch font with the current character set.
➢ TYPER calculates the character width and height of the window and allocates a buffer to hold all the characters.
➢ TYPER stores the character position of the caret in xCaret and yCaret variables.
➢ TYPER creates and shows the caret during the WM_SETFOCUS message and hides and destroys the caret during the WM_KILLFOCUS message.
➢ TYPER handles cursor movement keys and the Delete key during the WM_KEYDOWN message.
➢ TYPER handles Backspace, Tab, Linefeed, Enter, Escape, and character keys during the WM_CHAR message.
➢ TYPER hides the caret when drawing on the window during messages other than WM_PAINT.
➢ TYPER switches between character sets as the user switches keyboard layouts, but it does not handle double-width characters correctly.