KEYBOARD

The keyboard and mouse are the two primary means of user input.

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, but the fundamental principles of typing remain unchanged.

Keyboard Basics

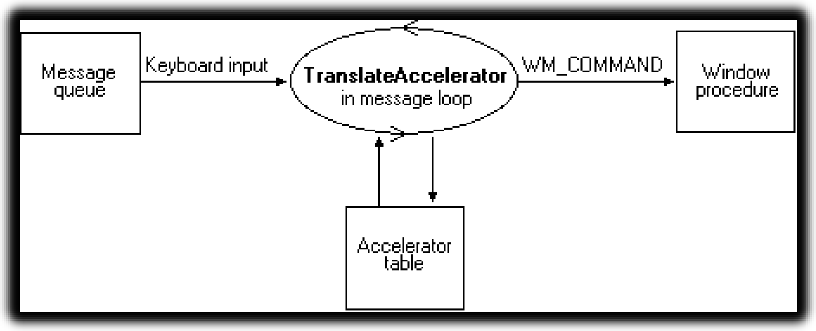
Think of keyboard input in Windows as a constant stream of "mail" arriving at your program's door. Even though Windows sends **eight different types of messages** every time a key is touched, most of them are just "junk mail" for a typical program.

To keep your code clean and fast, the trick is to ignore the noise and only open the specific envelopes you actually need.

- **The Message Flood:** Every press, hold, and release of a key creates a notification.
- **Too Much Info:** These messages contain a lot of technical "fine print" that you usually won't care about.
- **The Goal:** Successful programming isn't about listening to everything; it's about **identifying and picking out** only the relevant messages that matter for your specific task.

Ignoring Keyboard Input

When you're writing a Windows program, you can actually be quite "lazy" with the keyboard. Windows does a massive amount of the heavy lifting for you.



Think of Windows as a personal assistant: it handles the boring stuff (like menu shortcuts and system keys) and just hands you the final result.

Why You Can Ignore the Keyboard 🛡

- **The System Stuff:** Windows handles most keys involving **Alt** by default. These are usually for system operations, and while you *can* watch them, it's easier to just let Windows tell you what happened afterward.
- **The Shortcuts:** "Keyboard Accelerators" (like **Ctrl+S** for Save) are defined in your resource script. Windows automatically turns these keystrokes into simple **Menu Command** messages. You don't have to write code to "detect" the Ctrl key and the S key at the same time.
- **Dialog Boxes:** When a user is typing in a popup box, Windows manages that entire interface. It sends you a message only when the user is done.
- **The Edit Controls:** If you need a text box, you use an "Edit Control". Windows handles the cursor, the typing, and the backspacing. You just ask it for the final string of text when you're ready.
- **Child Windows:** You can use "Child Window Controls" to process input. These controls deal with the raw keyboard/mouse clicks and send "high-level" info to your main window. This keeps your main code clean.

In Windows, the keyboard is like a single microphone shared by everyone in a crowded room. Because everyone is talking (all your open apps), Windows needs a clear rule to decide who gets to "hear" the typing at any given second.

🎯 Who's Got the Focus?

The Shared Resource: The keyboard is shared by every app running on your computer.

The Address Label: Every message sent to your program has an hwnd (Window Handle) field. This acts like a mailing address so DispatchMessage knows exactly which window should get the keystroke.

The Input Focus: Only **one** window at a time has the "Input Focus". If a window has focus, it's the one that receives the keyboard messages.

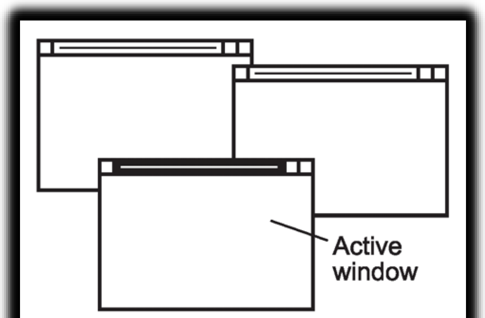


FIGURE 2. Active window

The Active Window: Focus is tied to the **Active Window**. This is always a "Top-Level" window (one with no parent).

Spotting the Winner: You can tell which window is active because Windows highlights its title bar or frame. If it's minimized, it looks like a "sunken" button on your taskbar.

The Family Tree: Inside an active window, the focus can be passed down to "descendants"—like child windows or grandchild windows.

Controls: These descendants are usually things like buttons, checkboxes, or text boxes (edit boxes). Even though they are small parts of a larger window, they can hold the focus and catch your keystrokes.

How Child Windows Handle Focus

Think of **"Focus"** like a spotlight in a theater. Only one person can be under the light at a time. While the main stage (the Active Window) is the boss, it can point that spotlight at any of its performers (the Child Windows).

- **Children are never the "Boss":** A child window (like a button) can never be the "Active Window." It can only hold the focus if its parent is already active.
- **The Visual Clues:** You can tell a child window has the focus because it will show a **blinking cursor (caret)** or a **dotted line** around its label.
- **Silence when Minimized:** If every program is minimized, no window has focus. Windows still sends messages, but they look a little different because there's no "visible" target.
- **The "Hello/Goodbye" Messages:** A window knows when it gets the spotlight because Windows sends it a WM_SETFOCUS message. When the spotlight moves away, it gets a WM_KILLFOCUS message.

The Traffic Controller (System Message Queue) 🚦

Think of the **System Message Queue** as the "Brain" that coordinates your fingers and the screen. Without it, your typing would be a mess of lost letters and confused windows.

Imagine typing is like a fleet of cars entering a busy city. If all the cars rushed to their destination at once, there would be a massive pile-up.

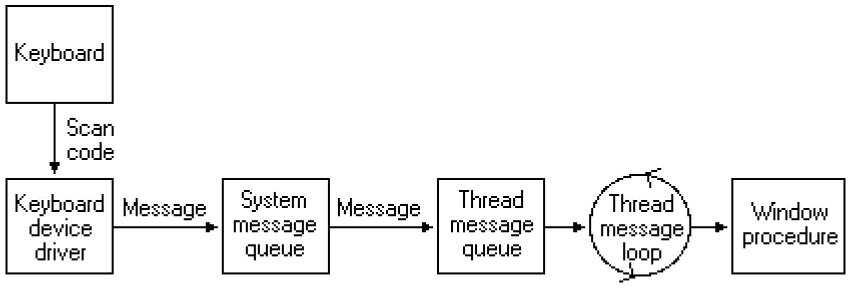
- **The Waiting Room (Buffer):** When you type fast, the keyboard driver sends messages to a "Holding Area" (the System Message Queue). This acts as a buffer so your apps don't get overwhelmed and crash.
- **One-at-a-Time (Sequential):** Windows is very strict. It waits for the current message to be finished before it sends the next one.
- **No Lost Letters:** Because of this "waiting room," your keystrokes aren't lost if an app is slightly slow. They just sit in the queue until the app is ready.
- **The Clean Switch:** When you click from one window to another, the queue makes sure the old window doesn't "steal" the letters meant for the new one. It clears the path so the new window gets the very next keystroke.

How Your Program Knows It's "On"

Your code doesn't have to guess if it has the focus. Windows sends two specific "handshake" messages:

- **WM_SETFOCUS**: This is Windows saying, *"Hey, it's your turn! The user is talking to you now."*
- **WM_KILLFOCUS**: This is Windows saying, *"Time's up. Someone else is taking over. Stop listening."*

In addition to keeping things organized, the **System Message Queue** has one more special job: it acts like an "Emergency Lane" for important commands.



The "Emergency Lane" (Priority Handling) 🚑

Even though most typing is **"first-come, first-served,"** Windows knows that some keys are too important to wait in line.

- **System Hotkeys:** Keys like the **Windows Key** (to open the Start menu) or your **Volume/Media buttons** are treated like emergency vehicles.
- **Cutting the Line:** Windows processes these immediately. If a slow app is currently "jamming" the regular traffic flow, your volume button will still work because it bypasses the standard queue.
- **The Unsung Hero:** Without this traffic controller, a single frozen program could "hijack" your entire keyboard, making it impossible to even open the Start menu to restart.

KEYSTROKES vs CHARACTERS

In Windows, typing is a two-step process. Windows first sees your physical finger movements (**Keystrokes**), and then it figures out what letter you actually meant to type (**Characters**).

The Two Sides of the Keyboard: Think of it like a musician playing a piano versus a writer typing a story.

1. The Keystroke (The Physical Act)

This is the raw signal. Every single button on your keyboard is just a physical switch.

- **Action:** Pressing a key down or letting it up.
- **Result:** Windows says, "The 'A' key was pressed." It doesn't care about uppercase or lowercase yet; it just knows the physical button moved.
- **Messages:** These are sent as WM_KEYDOWN and WM_KEYUP.

2. The Character (The Meaning)

This is the "translation." Windows looks at which physical key you hit AND which other keys were held down at the same time.

- **Action:** Combining keys (like **Shift + A**).
- **Result:** Windows translates the "A" key into a lowercase **'a'**, an uppercase **'A'**, or even a special command like **Ctrl+A**.
- **Messages:** This is sent as a WM_CHAR message.

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.**

When Windows watches you type, it acts like a translator. Sometimes it just tells your program "A button was pushed," and other times it says, "Here is the letter the user wanted."

3. Keys That Don't Talk (Keystrokes Only) 🎹

Not every key produces a letter. If you hit the **Shift** key, nothing appears on the screen. The same goes for the **Arrow keys**, **Delete**, or **Function keys (F1-F12)**.

- For these "silent" keys, Windows **only** sends a **Keystroke Message**.
- There is no "Character Message" because there is no character to display.

4. The "Dead Key" Magic

In some languages, you hit a key (like an accent mark ^) and nothing happens immediately. This is a **Dead Key**. When you hit the next key (like a), Windows combines them to send you a single character: **â**.

The Core Concept: The Piano Analogy

This concept is often confusing because we think of "typing" as one action, but the computer sees it as two very different steps.

Think of your keyboard like a **piano**.

1. **The Keystroke (The Physical Act):** You press a specific plastic key down. It doesn't matter what sound it makes; it only matters *which* mechanical lever you pushed.
2. **The Character (The Result):** The note that actually plays. If you hold a pedal (Shift) while pressing the key, the note changes (Low C vs. High C), even though you pressed the same physical key.

Windows sends you messages for both steps.

1. Keystroke Messages (The Hardware View)

"The user just smashed a piece of plastic" - This message is for when you need to know *what the user is doing with their hands*. It doesn't care about letters; it cares about buttons.

- **The Action (Up/Down):** Did they just push the button down, or did they release it?
- **Virtual Key Code (The Name Tag):** Every key has a permanent ID card. *Example:* The Left Arrow key is always VK_LEFT. The letter 'A' key is VK_A.
- **Scan Code (The Hardware Address):** This is the raw data from the keyboard hardware telling the computer exactly where the key is located on the circuit board (e.g., "Row 2, Button 3"). *Note:* You almost never need this unless you are writing a driver.
- **Repeat Count (The Autofire):** If the user holds the key down, Windows sends this message repeatedly. This number tells you, "This is the 5th time I'm telling you they are holding this key."

Best For: Games (Movement), Shortcut Keys (Ctrl+S), Function Keys (F1, F12).

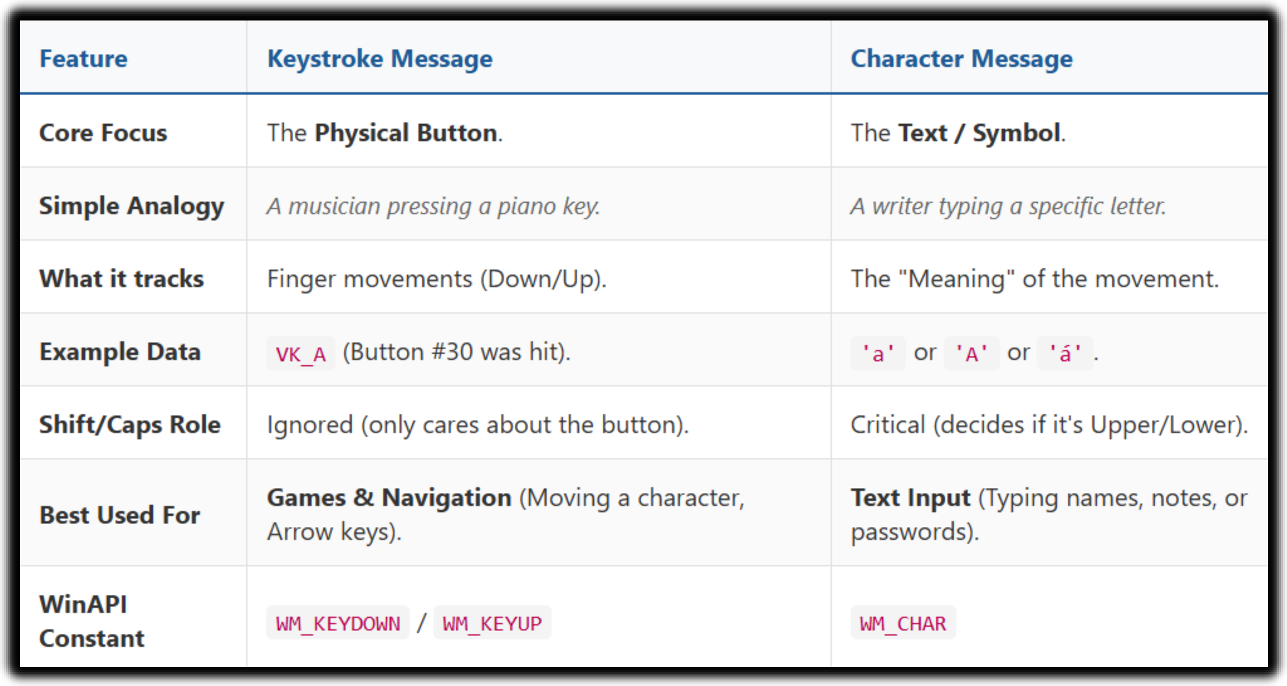
2. Character Messages (The Text View)

"The user wants to type a symbol."

This message is for when you want to know *what to write on the screen*. Windows has already done the hard work of checking the Shift key and Caps Lock for you.

- **Unicode Code (The Letter):** This is the final result.
    - ✓ *Example:* If you press the VK_A key...
        - Windows checks: "Is Shift held?" -> **No.** -> It sends the code for **'a'**.
        - Windows checks: "Is Shift held?" -> **Yes.** -> It sends the code for **'A'**.
- **Modifier State:** Windows implicitly handles this. It combines the Shift + 4 keys to simply give you the $ symbol.

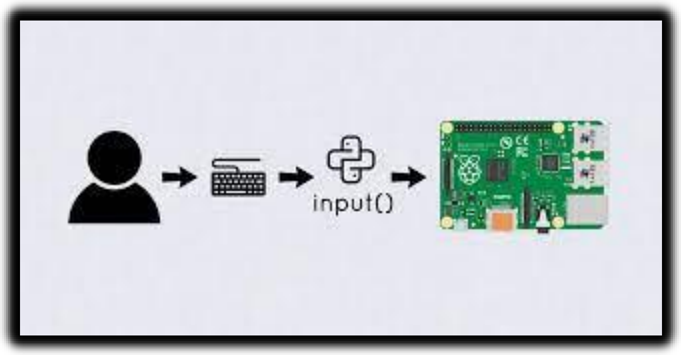
Best For: Word processors, data entry, search bars.



Feature	Keystroke Message	Character Message
Core Focus	The **Physical Button**.	The **Text / Symbol**.
Simple Analogy	*A musician pressing a piano key.*	*A writer typing a specific letter.*
What it tracks	Finger movements (Down/Up).	The "Meaning" of the movement.
Example Data	`VK_A` (Button #30 was hit).	`'a'` or `'A'` or `'á'`.
Shift/Caps Role	Ignored (only cares about the button).	Critical (decides if it's Upper/Lower).
Best Used For	**Games & Navigation** (Moving a character, Arrow keys).	**Text Input** (Typing names, notes, or passwords).
WinAPI Constant	`WM_KEYDOWN` / `WM_KEYUP`	`WM_CHAR`

The Two Types of Keystrokes 🎭

In Windows, every time you touch a key, it's sorted into one of two buckets: **System** or **Nonsystem**. This helps Windows decide if you are trying to talk to the Operating System (like opening a menu) or just typing a letter into your app.



1. System Keystrokes (The "Commanders")

These involve the **Alt** key or the **Windows** key. They are usually meant for Windows itself (like Alt+F4 to close a window).

- **The Rule:** If you don't handle these in your code, you **must** pass them back to Windows so it can do its job.
- **Messages:** WM_SYSKEYDOWN and WM_SYSKEYUP.

2. Nonsystem Keystrokes (The "Typers")

These are your regular keys—letters, numbers, and the **Ctrl** key combinations (interestingly, Ctrl is nonsystem). These are meant for your specific application.

- **Messages:** WM_KEYDOWN and WM_KEYUP.

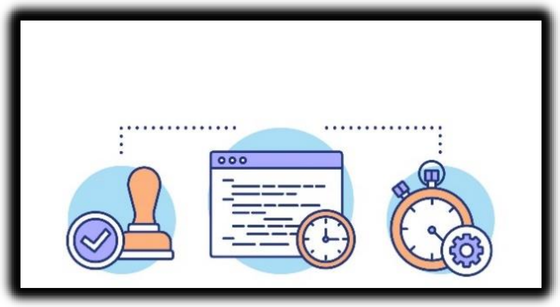
The Autorepeat Pattern (Typematic Action) 🔄

Normally, keys come in pairs (Down then Up). But if you hold a key down, Windows starts "machine-gunning" the Down messages.

- **Standard Press:** Down → Up
- **Holding a Key:** Down → Down → Down → Down ... → Up
- **The Key Point:** You will only ever get **one** KeyUp message at the very end, no matter how many KeyDown messages were sent during the repeat.

Timestamping: The Clock Behind the Keys ⏱️

When you type, Windows doesn't just record **what** happened; it records **when** it happened. This "temporal context" is what makes your keyboard feel responsive and allows for complex features like double-tapping or measuring typing speed.



Every time a key message is put into the system queue, Windows stamps it with a time. This isn't the "wall clock" time (like 3:00 PM), but a **relative time** in milliseconds since the system started.

- **The Function:** You use GetMessageTime() to grab this value.
- **The Precision:** Because it's in milliseconds, you can precisely measure the gap between a WM_KEYDOWN and a WM_KEYUP.
- **The Use Case:** This is how "Filter Keys" works (ignoring accidental brief taps) and how games determine if you performed a "long press" versus a "quick tap."

🏛 Summary: The Keystroke Foundation

Keystroke messages are the "raw atoms" of user input. Everything you see on your screen—from a typed letter to a character jumping in a game—starts here.

- **System vs. Nonsystem:** Decides if the message is for the OS (Alt keys) or your App.
- **Autorepeat:** Handles the "machine-gun" effect of holding a key down.
- **Timestamping:** Adds the dimension of time to every action.

In Windows, every keystroke is like a letter being sorted at a post office. Some letters are addressed to the "City Hall" (Windows OS), and some are addressed to the "Local Business" (Your App).

Who Processes the Message?

1. System Keys (The "City Hall" Mail)

Messages like WM_SYSKEYDOWN (usually involving the **Alt** key) are for the system.

- **The Default Rule:** Your application should be "lazy" here. Let them slide right past your code.
- **The Relay:** You pass these to DefWindowProc. This is a built-in function that says, "I don't know what to do with this Alt key, you handle it, Windows."
- **The Exception:** If you want to build a "Kiosk Mode" or a specialized tool that blocks Alt+F4, you can "trap" the message. But be careful—if you don't pass it back to DefWindowProc eventually, the user might get stuck!

2. Nonsystem Keys (The "Local" Mail)

Messages like WM_KEYDOWN are for **you**.

- **Total Authority:** Your application has 100% control here. You can turn a "W" key into a character move, a "Delete" key into a file wipe, or simply ignore it if it doesn't fit your app's purpose.

Exploring the Power of Window Procedures

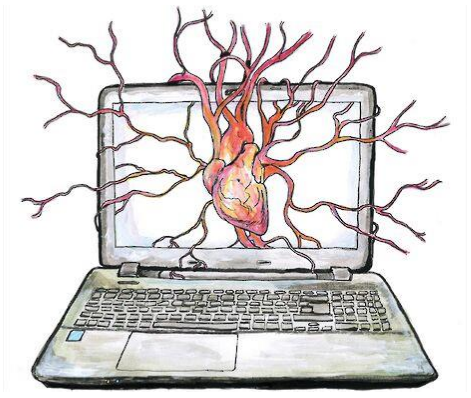
Window Procedures and the specific data inside the **Keystroke Messages**.

1. The Window Procedure: The Gatekeeper

The Concept: The Traffic Controller.

The **Window Procedure** (WndProc) is the function where your code lives. Every time a key is pressed, Windows sends a message to this function saying, "Hey, a key happened!"

- **Your Power:** You can decide to obey the message, ignore it, or change it.
- **Granular Control:** Because you see *every single* press and release, you can create custom behaviors (like "Press F1 to open Help") or block keys entirely (like disabling the Windows key in a game).



WndProc

---

2. The Four Keystroke Messages

Windows separates keyboard activity into four specific messages:

1. **WM_KEYDOWN**: A standard key (A, B, Space, Arrow) was pressed.
2. **WM_KEYUP**: A standard key was released.
3. **WM_SYSKEYDOWN**: A "System" key was pressed (usually **Alt** or **F10**).
4. **WM_SYSKEYUP**: A "System" key was released.

Note: "System" keys are ones that Windows usually handles for menus (Alt+File) or switching windows (Alt+Tab). You *can* intercept them, but usually, you leave them alone.

---

3. The Data: wParam vs lParam

When your Window Procedure receives one of these messages, it gets two data packages:

wParam: The "Virtual Key Code"

The Concept: The Universal ID Card. This tells you **WHICH** key is being interacted with.

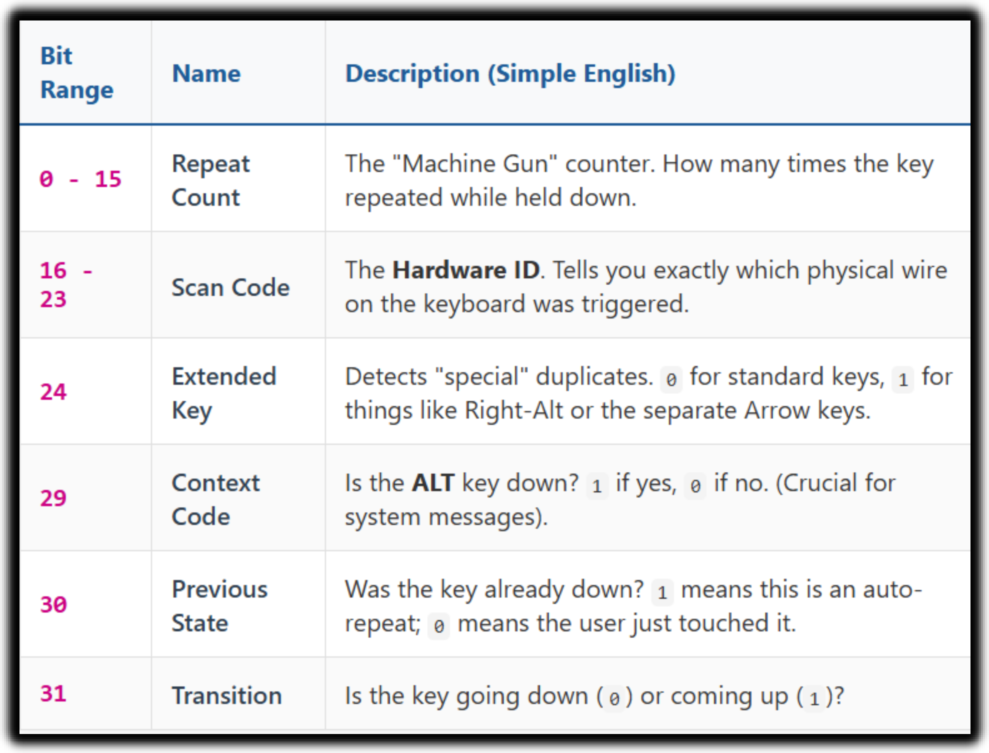
- **Why "Virtual"?** It doesn't matter what keyboard you have (US, UK, Japanese). If you press "Enter," the code is always the same.
- **The Constants:** In your code, you use standard names defined by Windows (starting with VK_).

Common Examples: | Constant | Key | | :--- | :--- | | VK_RETURN | Enter Key | | VK_ESCAPE | Esc Key | | VK_LEFT | Left Arrow | | VK_F1 | F1 Key | | VK_MENU | Alt Key |

lParam: The "Details Package"

The Concept: The Fine Print. This is a 32-bit number where different bits hold different details about *how* the key was pressed. It is "packed" with information.

The Breakdown of the 32 Bits:



Bit Range	Name	Description (Simple English)
`0 - 15`	Repeat Count	The "Machine Gun" counter. How many times the key repeated while held down.
`16 - 23`	Scan Code	The **Hardware ID**. Tells you exactly which physical wire on the keyboard was triggered.
`24`	Extended Key	Detects "special" duplicates. `0` for standard keys, `1` for things like Right-Alt or the separate Arrow keys.
`29`	Context Code	Is the **ALT** key down? `1` if yes, `0` if no. (Crucial for system messages).
`30`	Previous State	Was the key already down? `1` means this is an auto-repeat; `0` means the user just touched it.
`31`	Transition	Is the key going down (`0`) or coming up (`1`)?

Window Procedure: The boss that decides what to do with the input.

wParam: **Who** is calling? (The Identity: VK_SPACE).

lParam: How are they calling? (The Context: "I'm holding it down," "I'm the Right-side Ctrl key," etc.).

VIRTUAL KEY CODES

1. The Problem: Hardware Chaos

The Concept: Every keyboard is different.

Imagine you are writing a game. You want the player to press the key in the top-left corner to jump.

- On an **American** keyboard, that physical key is **Q**.
- On a **French** keyboard, that same physical key is **A**.
- On a **German** keyboard, it's something else.

If Windows only listened to the "Scan Code" (the physical wire connected to that button), your game would be broken for half the world. Pressing the top-left button would send different signals depending on the hardware.

2. The Solution: The "Virtual" Translator

The Concept: The Logical ID.

Windows introduces a translation layer called the **Virtual Key Code**.

- It doesn't care *where* the key is physically located.
- It cares *what* the key is supposed to do.

The Workflow:

1. **Hardware:** You press the physical key in the top-left.
2. **Driver:** The keyboard driver says, "Physical Switch #16 was pressed."
3. **Windows (The Translation):** Windows checks your language settings.
    - ✓ If set to **English**, it translates Switch #16 to VK_Q.
    - ✓ If set to **French**, it translates Switch #16 to VK_A.
4. **Your App:** Your program receives VK_A. You don't need to know *how* it happened, just that the user wanted an 'A'.

3. Why "Virtual"?

The text notes that "Virtual" sounds fake or imaginary. In this context, it means **Abstract**.

- **Scan Code = Physical Reality:** "The switch at Row 3, Column 2."
- **Virtual Code = Logical Reality:** "The 'Enter' Command."

Because the code represents the *function* of the key rather than the plastic switch itself, it is "Virtual."

4. The Three Big Advantages

A. Device Independence (Write Once, Run Anywhere)

You write your code: if (wParam == VK_RETURN). This works perfectly on:

- A standard desktop keyboard.
- A laptop with a mini-keyboard.
- A Bluetooth projection keyboard.
- An on-screen touch keyboard.

B. Future-Proofing

If a company invents a new "3D Keyboard" next year, your old software will still work. The new keyboard's driver will handle the translation to Virtual Codes; your application doesn't need to change.

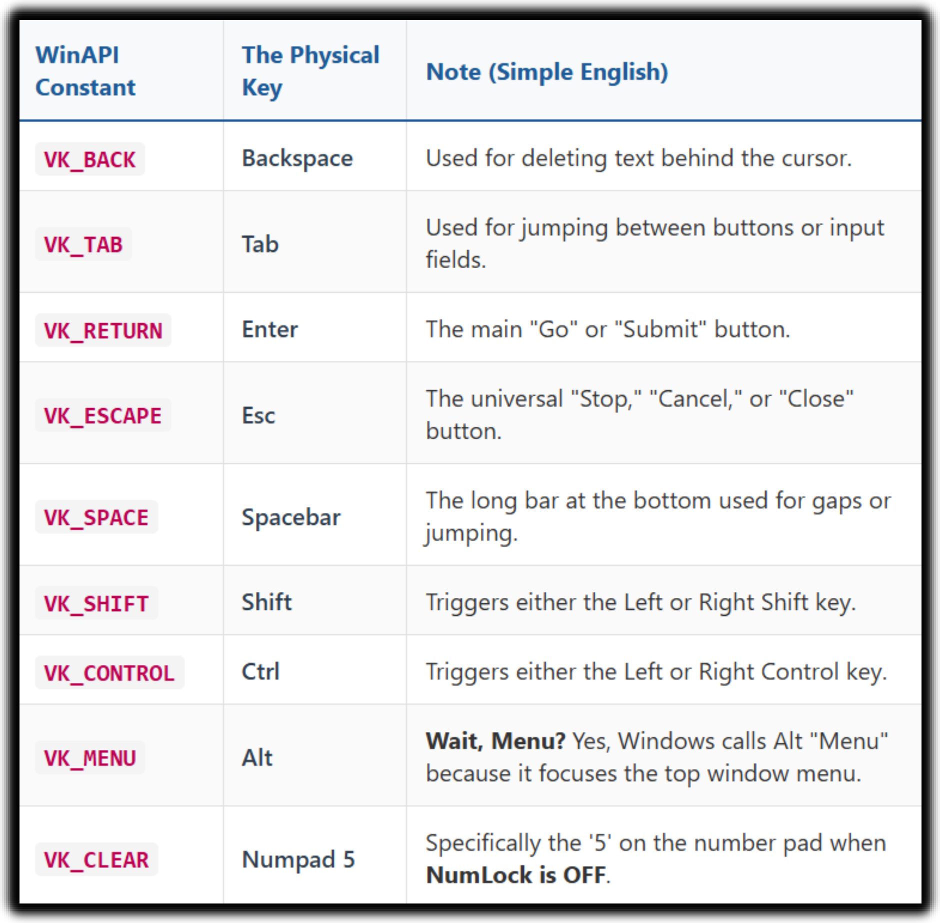
C. Accessibility

This is vital for users with disabilities.

- Some users control computers with **Eye Trackers** or **Voice Commands**.
- These tools don't have physical switches. Instead, they simulate input by sending **Virtual Key Codes** directly to Windows.
- Your program sees VK_SPACE and reacts, never knowing that the user didn't actually touch a physical keyboard.

D. The Cheat Sheet (Common Codes)

Windows uses names starting with VK_ (Virtual Key) to identify buttons. Most are obvious, but a few are tricky.



WinAPI Constant	The Physical Key	Note (Simple English)
`VK_BACK`	Backspace	Used for deleting text behind the cursor.
`VK_TAB`	Tab	Used for jumping between buttons or input fields.
`VK_RETURN`	Enter	The main "Go" or "Submit" button.
`VK_ESCAPE`	Esc	The universal "Stop," "Cancel," or "Close" button.
`VK_SPACE`	Spacebar	The long bar at the bottom used for gaps or jumping.
`VK_SHIFT`	Shift	Triggers either the Left or Right Shift key.
`VK_CONTROL`	Ctrl	Triggers either the Left or Right Control key.
`VK_MENU`	Alt	**Wait, Menu?** Yes, Windows calls Alt "Menu" because it focuses the top window menu.
`VK_CLEAR`	Numpad 5	Specifically the '5' on the number pad when **NumLock is OFF**.

E. When to use VK Codes vs. Characters

The text highlights a subtle but important rule: **Know your goal.**

A. If you are a Word Processor (Typing Text): You typically **Ignore** VK_ codes for keys like A, B, or C.

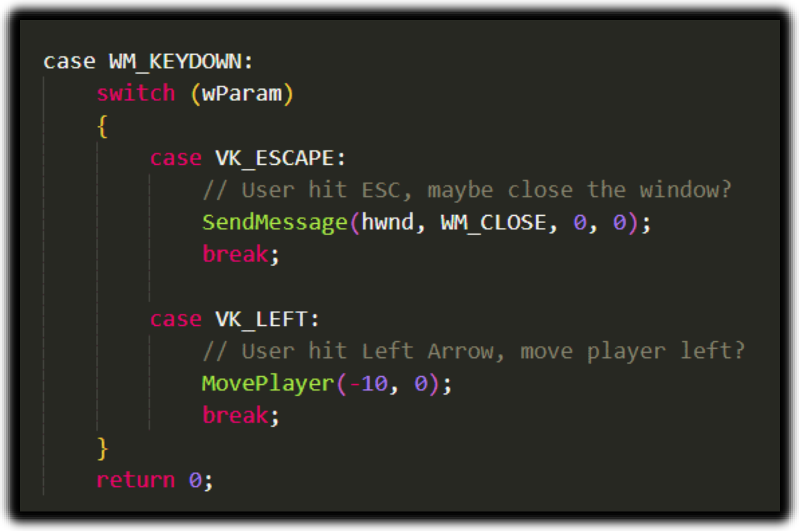
- *Why?* Because VK_A doesn't tell you if it's "a" or "A". You wait for the WM_CHAR message instead, which handles the capitalization for you.

B. If you are a Game or Utility (Controlling Things): You **Must Use** VK_ codes.

- *Why?* If the user presses the **Left Arrow**, there is no "text character" for that. It's a command. You must check for VK_LEFT.
- *Why?* If the user holds **Shift**, that is a "state," not a letter. You check VK_SHIFT.

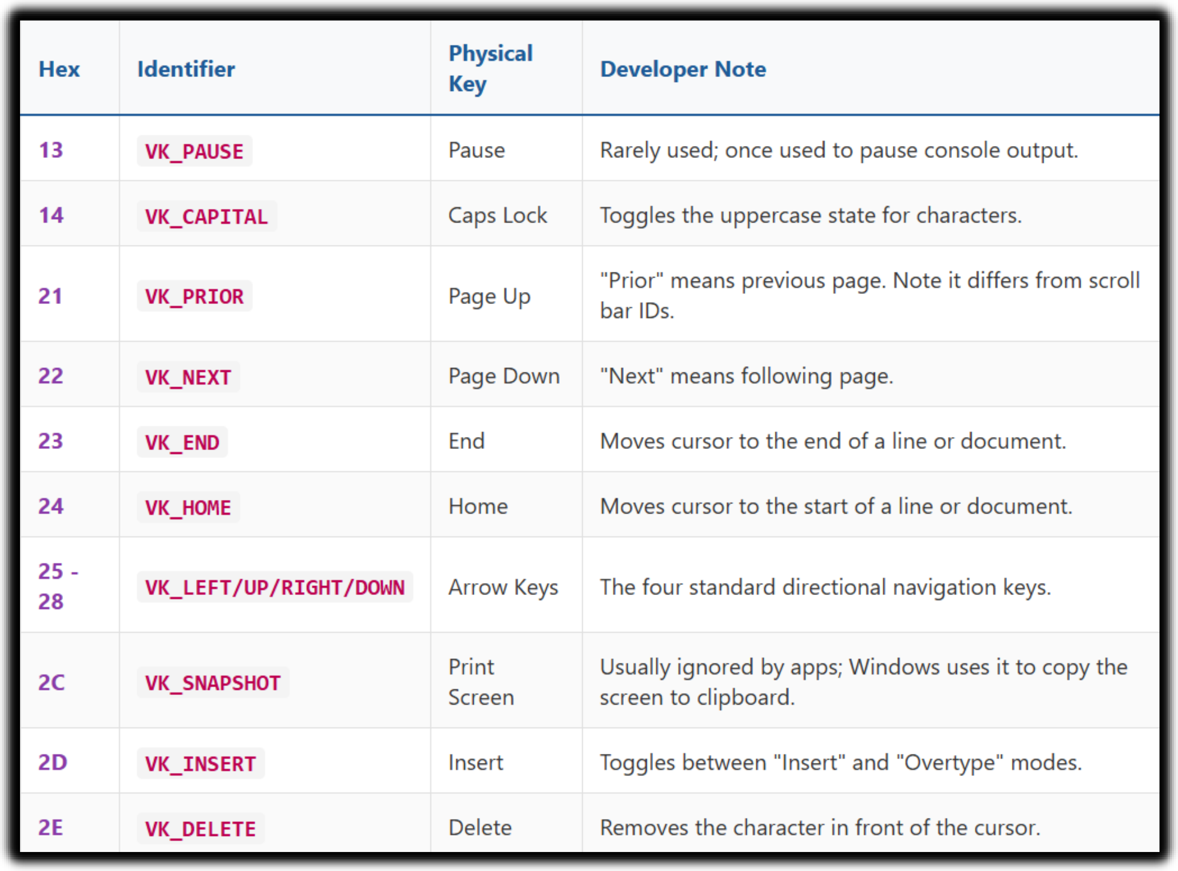
F. How to Catch Them

As learned in the previous section, these ID cards arrive in the **wParam** bucket of the message.



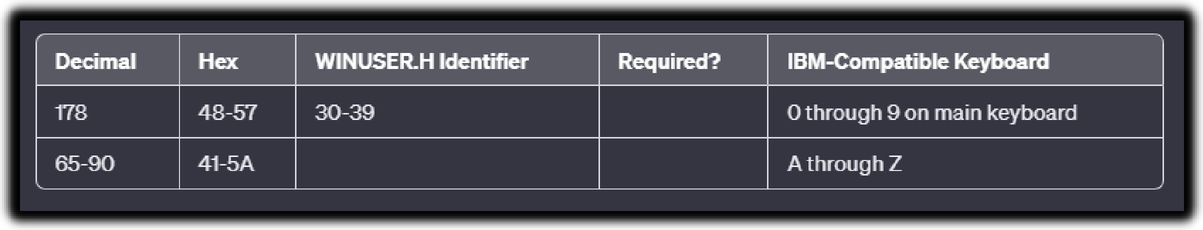
```
case WM_KEYDOWN:
    switch (wParam)
    {
        case VK_ESCAPE:
            // User hit ESC, maybe close the window?
            SendMessage(hwnd, WM_CLOSE, 0, 0);
            break;

        case VK_LEFT:
            // User hit Left Arrow, move player left?
            MovePlayer(-10, 0);
            break;
    }
    return 0;
```

Virtual Key Code Tables



Hex	Identifier	Physical Key	Developer Note
13	VK_PAUSE	Pause	Rarely used; once used to pause console output.
14	VK_CAPITAL	Caps Lock	Toggles the uppercase state for characters.
21	VK_PRIOR	Page Up	"Prior" means previous page. Note it differs from scroll bar IDs.
22	VK_NEXT	Page Down	"Next" means following page.
23	VK_END	End	Moves cursor to the end of a line or document.
24	VK_HOME	Home	Moves cursor to the start of a line or document.
25 - 28	VK_LEFT/UP/RIGHT/DOWN	Arrow Keys	The four standard directional navigation keys.
2C	VK_SNAPSHOT	Print Screen	Usually ignored by apps; Windows uses it to copy the screen to clipboard.
2D	VK_INSERT	Insert	Toggles between "Insert" and "Overtype" modes.
2E	VK_DELETE	Delete	Removes the character in front of the cursor.

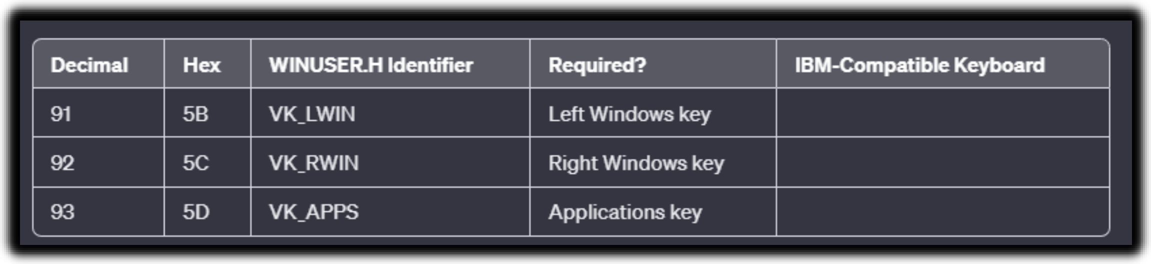
Virtual Key Codes for Main Keyboard Keys:



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.

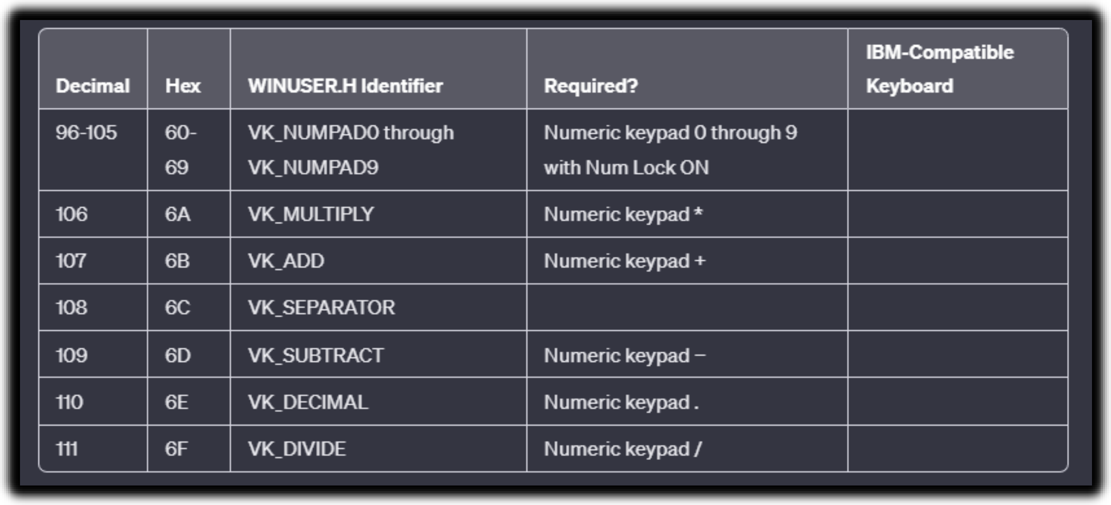
Virtual Key Codes for Microsoft Natural Keyboard:



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.

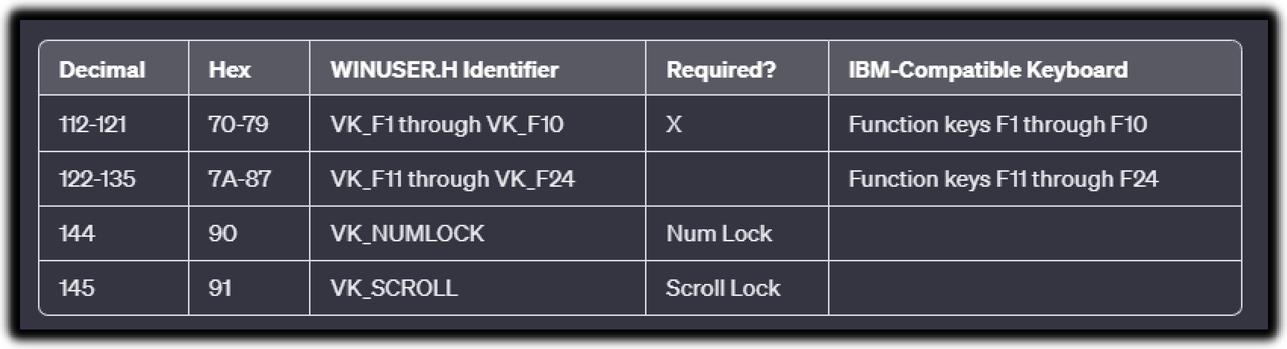
Virtual Key Codes for Numeric Keypad:



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	

Windows doesn't just tell you *which* key was hit; it gives you a high-tech "flight data recorder" for every single press. While the **wParam** acts as the ID tag (the Virtual Key Code), the **lParam** is a 32-bit package stuffed with metadata about the hardware and the timing of the event.

The Hidden Details of a Keystroke

- **Function Key Flexibility**: Windows only strictly requires 10 function keys (F1 through F10), but it actually has IDs ready for up to 24 keys. You'll mostly see these used as "Shortcuts" (Accelerators) rather than raw input.
- **Specialty Hardware**: If you ever find yourself reversing software for old mainframe terminals or weird non-standard keyboards, Windows has extra Virtual Key codes reserved just for them.
- **The Power of lParam**: Think of lParam as the "Fine Print" of a message. It tells you the "how" and "when" behind the "what" found in wParam.

Unpacking the lParam Suitcase

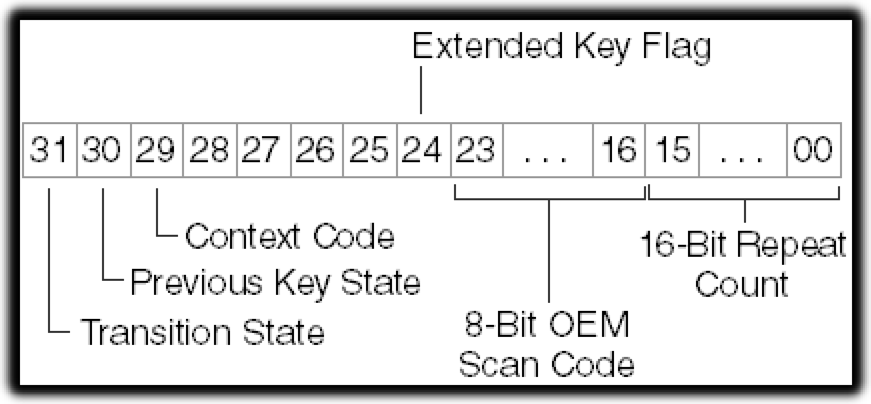
When you see an lParam in your debugger, it looks like one big number, but it's actually several small pieces of data "glued" together into 32 bits.

In your malware analysis reverse engineering journey, you'll notice that most programs use the **Virtual Key** (wParam). However, some clever anti-cheat systems or low-level drivers look at the **Scan Code** in lParam.

Why? Because Scan Codes are tied to the physical wires. If a program sees a Virtual Key for "A" but the Scan Code doesn't match a real keyboard layout, it might realize the input is being "faked" by a script!

The Structure of lParam

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



Think of lParam not as a single number, but as a "suitcase" that is 32 bits long. Windows packs **six specific pieces of information** into this one suitcase to tell you exactly how a key was pressed.

Here is what is inside that suitcase, reading the bits from the image:

1. Repeat Count (Bits 0 - 15)

- **What it is:** The "Autofire" counter.
- **Meaning:** If you hold down the letter 'A', Windows sends multiple messages. This number tells you how many times the keystroke is repeated as a result of the user holding down the key.
- **Why use it?** Essential for typing. If this number is greater than 1, you know the user is holding the key, not tapping it repeatedly.

2. OEM Scan Code (Bits 16 - 23)

- **What it is:** The "Hardware Address".
- **Meaning:** This is the value generated by the keyboard hardware itself. It depends on the manufacturer (OEM = Original Equipment Manufacturer).
- **Why use it?** You rarely need this. It tells you physically *where* on the circuit board the key is located, regardless of what is printed on the keycap.

3. Extended Key Flag (Bit 24)

- **What it is:** The "Duplicate Key" detector.
- **Meaning:**
  - ✓ **0:** It is a standard key (like the main Enter key).
  - ✓ **1:** It is an "extended" key (like the Enter key on the numeric keypad, or the Right Alt/Ctrl keys).
- **Why use it?** This is how games distinguish between the **Left Ctrl** (shoot) and **Right Ctrl** (jump).

4. Context Code (Bit 29)

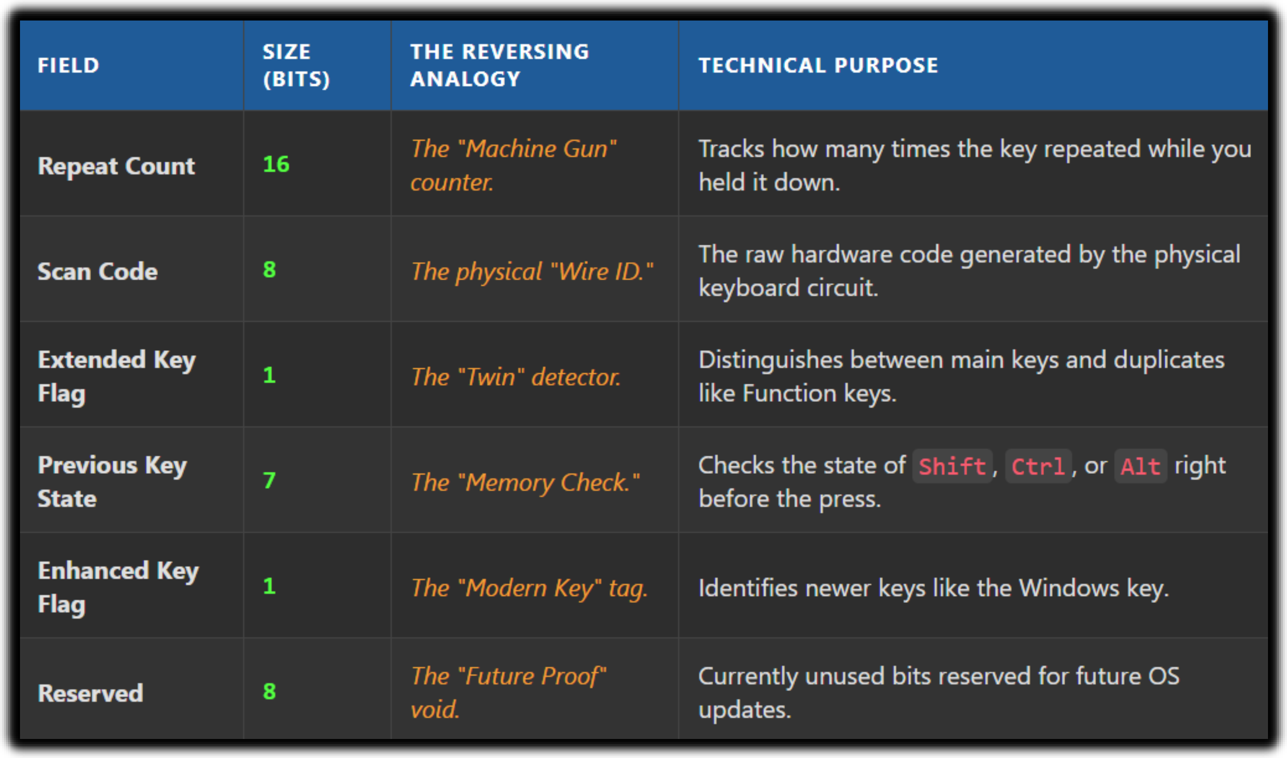
- **What it is:** The "Alt" Flag.
- **Meaning:**
  - ✓ **1:** The **Alt** key is currently being held down.
  - ✓ **0:** The **Alt** key is not being held.
- **Why use it?** This helps distinguish commands. Pressing 'F' is just typing; pressing 'Alt+F' usually opens a File menu.

5. Previous Key State (Bit 30)

- **What it is:** The "Was it already down?" check.
- **Meaning:**
  - ✓ **0:** The key was previously **UP**. (This is the very first instant the user touched the key).
  - ✓ **1:** The key was previously **DOWN**. (The user is still holding it).
- **Why use it?** This is the best way to ignore "autorepeat" if you only want an action to happen *once* (like firing a single bullet) rather than machine-gunning while the key is held.

6. Transition State (Bit 31)

- **What it is:** The "Action" Flag.
- **Meaning:**
  - ✓ **0:** The key is being **Pressed** (WM_KEYDOWN).
  - ✓ **1:** The key is being **Released** (WM_KEYUP).
- **Why use it?** It strictly tells you if the finger is going down or coming up.



FIELD	SIZE (BITS)	THE REVERSING ANALOGY	TECHNICAL PURPOSE
Repeat Count	16	*The "Machine Gun" counter.*	Tracks how many times the key repeated while you held it down.
Scan Code	8	*The physical "Wire ID."*	The raw hardware code generated by the physical keyboard circuit.
Extended Key Flag	1	*The "Twin" detector.*	Distinguishes between main keys and duplicates like Function keys.
Previous Key State	7	*The "Memory Check."*	Checks the state of `Shift`, `Ctrl`, or `Alt` right before the press.
Enhanced Key Flag	1	*The "Modern Key" tag.*	Identifies newer keys like the Windows key.
Reserved	8	*The "Future Proof" void.*	Currently unused bits reserved for future OS updates.

Repeat Count: The "Turbo Mode" Handler

The Concept: Catching up with the typist.

Usually, when you press a key, the Repeat Count is **1**. However, if you hold a key down (like "aaaaaaaa"), your keyboard sends signals very fast (the "typematic rate").

- **The Problem:** Sometimes your program is busy doing something else (like saving a file) and can't process these messages fast enough.
- **The Solution:** Instead of flooding your queue with 50 separate messages for "a", Windows combines them. It sends **one** message but sets the **Repeat Count** to 50.
- **Key Rule:** This only happens for Key **Down** messages. For Key **Up** messages (WM_KEYUP), the count is always **1** because you can only release a key once.

OEM Scan Code: The Hardware Relic

The Concept: The physical wire address.

- **What it is:** The raw code the keyboard hardware sends to the computer.
- **History:** In the old days of DOS and Assembly, programmers used this to talk directly to the BIOS.
- **Modern Usage:** You can almost always **ignore this**. Windows handles the hardware translation for you. You only look at this if your app needs to know the exact physical layout of the user's specific keyboard model.

Extended Key Flag: The "Duplicate Key" Finder

The Concept: Distinguishing between twins.

Modern keyboards (IBM Enhanced) have duplicate keys. For example, there is an **Enter** key in the main area and another **Enter** key on the number pad.

- **Flag = 1:** The key is from the "Extended" section (the extra keys).
- **Flag = 0:** The key is from the standard main section.

Which keys are "Extended"?

- The **Right-side** Alt and Ctrl keys.
- The arrow keys/navigation keys (Insert, Delete, Home, End) that are **separate** from the number pad.
- The **Num Lock** key.
- The **Enter** and **/** keys located on the numeric keypad.

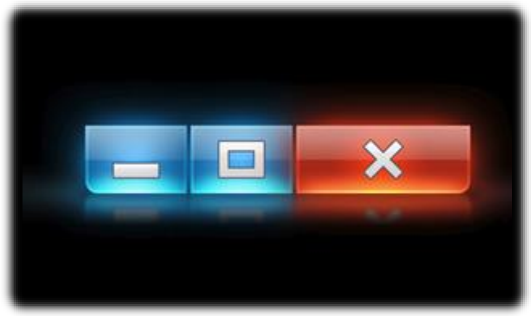
Context Code: The "Alt" Detector

The Concept: Are we in a menu or system command?

This bit tells you if the **Alt** key was held down while the key was pressed.

- **Value 1:** Alt was down (Triggering WM_SYSKEYDOWN / WM_SYSKEYUP).
- **Value 0:** Alt was NOT down (Triggering standard WM_KEYDOWN / WM_KEYUP).

The Minimized Exception: If your window is minimized, it cannot have "Input Focus" (you can't type into a minimized window). However, if you type while it is minimized, Windows sends **System Messages** (WM_SYS...) with the Context Code set to 1, effectively treating them like system commands rather than standard text input.



Usually, **Context Code 1** means "The Alt key is held down." However, there are two tricky exceptions where this logic changes slightly.

- **The Minimized Window Shield:** If your window is minimized, you can't type text into it. To protect your app, Windows converts *all* your keystrokes into "System Messages" (WM_SYS...). In this specific case, the Context Code might be **0** (because you aren't holding Alt), but it is still treated as a System message effectively "disabling" standard typing input.
- **The Foreign Keyboard Exception:** On some international keyboards, you must hold **Alt** or **Ctrl** to type specific letters (like é or ñ). In these cases, the Context Code will be **1** (because Alt is down), but Windows is smart enough *not* to treat it as a System Command. It lets the keystroke pass through as a normal character so the user can type their language.

Previous Key State: The "Am I being held?" Flag

The Concept: The Auto-Repeat Filter.

This bit tells you what the key was doing *milliseconds before* this current message arrived.

- **0 (Was Up):** This is a **Fresh Press**. The user just touched the key.
- **1 (Was Down):** The key is being **Held Down**. This message is just an auto-repeat echo.

Why is this useful? Imagine a "Pause" button in a game.

- If you just check for "Key Down," holding the 'P' key will rapidly pause and unpause the game 30 times a second.
- By checking if (PreviousState == 0), you ensure the toggle happens only **once**, no matter how long they hold the button.

Transition State: The Simple Switch

The Concept: Direction of movement.

This is the simplest bit in the entire suitcase. It strictly tells you the direction the finger is moving.

- **0:** The key is being pushed **Down** (Pressing).
- **1:** The key is popping **Up** (Releasing).

Why is it repetitive? You are right; it seems redundant because the message name (WM_KEYDOWN vs WM_KEYUP) usually tells you this already.

However, Windows includes this bit so that advanced programmers can write a single function to handle *both* ups and downs by just checking this one number, rather than writing two separate case statements.

SHIFT STATES: IDENTIFYING MODIFIER KEYS

Here is the guide to **detecting Shift States and Modifier Keys**. This section answers the question: *"I know they pressed 'A', but were they holding Shift?"*

1. The Problem: Context Matters

Knowing a key was pressed isn't always enough. You often need the **context**.

- **Action:** User presses S.
- **Context 1:** Just S -> Type the letter "s".
- **Context 2:** Ctrl + S -> Save the file.
- **Context 3:** Shift + S -> Type capital "S".

To get this context, we use the function GetKeyState.

2. The Tool: GetKeyState

This function lets you peek at the keyboard's current status *right now*.

The Syntax: SHORT state = GetKeyState(VK_CODE);

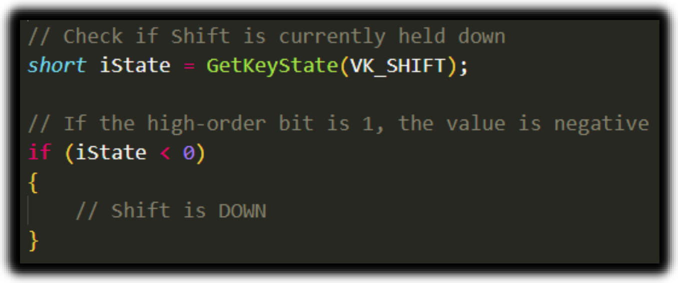
It returns a number (a 16-bit integer) that contains two distinct pieces of information, hidden in the bits.

3. How to Read the Return Value

The return value uses specific bits to tell you different things.

A. Checking for "Pressed Down" (Shift, Ctrl, Alt)

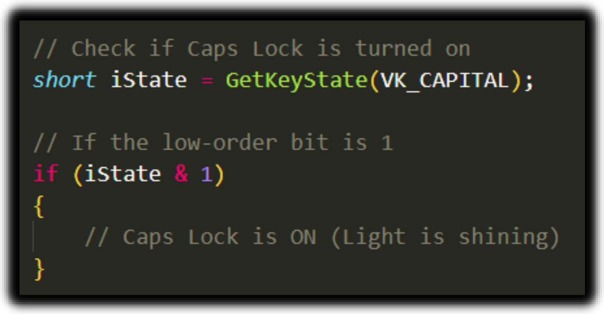
To see if a key is currently being held down, you check the **High-Order Bit** (the very first bit of the number). If the number is **negative**, the key is down.



```
// Check if Shift is currently held down
short iState = GetKeyState(VK_SHIFT);

// If the high-order bit is 1, the value is negative
if (iState < 0)
{
    // Shift is DOWN
}
```

B. Checking for "Toggled On" (Caps Lock, Num Lock)

To see if a toggle key is active (i.e., is the little light on the keyboard ON?), you check the **Low-Order Bit** (the very last bit).



```
// Check if Caps Lock is turned on
short iState = GetKeyState(VK_CAPITAL);

// If the low-order bit is 1
if (iState & 1)
{
    // Caps Lock is ON (Light is shining)
}
```

4. Left vs. Right Keys

Sometimes you need to know *exactly* which key was pressed (e.g., in a flight simulator, Left Shift might be "Throttle Up" while Right Shift is "Flaps").

GetKeyState supports these specific constants:

- VK_LSHIFT / VK_RSHIFT
- VK_LCONTROL / VK_RCONTROL
- VK_LMENU (Left Alt) / VK_RMENU (Right Alt)

5. Mouse Buttons (The Exception)

You *can* use GetKeyState to check mouse buttons using VK_LBUTTON or VK_RBUTTON, but the text advises against it.

- **Why?** Because when Windows sends you a mouse message (like "User clicked here"), it typically includes the state of the Shift and Ctrl keys inside that message automatically. It is cleaner to use the data Windows gives you rather than asking GetKeyState manually.

Summary:

- **GetKeyState < 0**: The key is being **HELD**.
- **GetKeyState & 1**: The key is **TOGGLED** (Light is on).

GetKeyState vs. GetAsyncKeyState

This is a common interview question and a frequent bug source. The difference lies in **time**.

A. GetKeyState: The "Historical" Check

"What was the status *when this message was created*?"

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

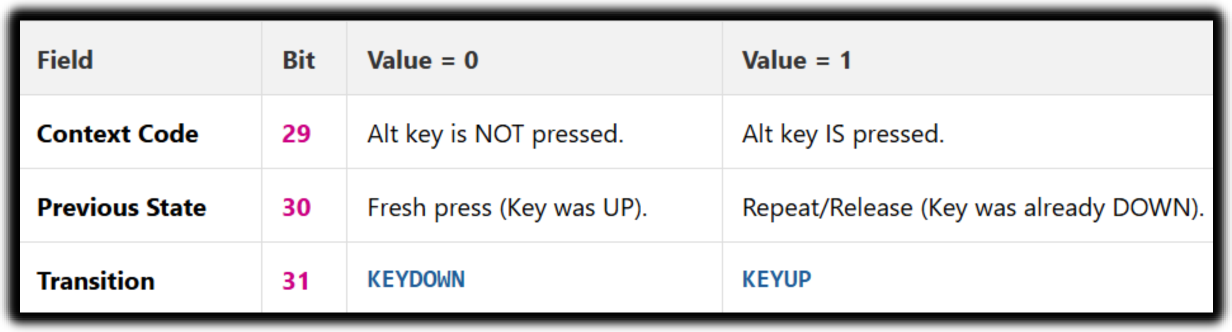
- **How it works:** It does **not** check the hardware. It looks at the message queue.
- **The Scenario:** Imagine your computer is lagging. You press Shift, then Tab, then release both. The computer freezes for 5 seconds.
- **The Result:** When your app finally wakes up and processes the Tab message, GetKeyState(VK_SHIFT) will still report that Shift is **DOWN**, even though your finger let go 5 seconds ago.
- **Why is this good?** It ensures your input logic remains consistent. If the user typed Shift+Tab, you want to process it as Shift+Tab, even if the computer was slow.

B. GetAsyncKeyState: The "Real-Time" Check

"Is the finger on the button *right now*?"

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

- **How it works:** It queries the hardware driver immediately.
- **The Scenario:** You are writing a game loop or a long calculation. You want to break out of a loop instantly if the user panics and hits F1.



Field	Bit	Value = 0	Value = 1
Context Code	29	Alt key is NOT pressed.	Alt key IS pressed.
Previous State	30	Fresh press (Key was UP).	Repeat/Release (Key was already DOWN).
Transition	31	KEYDOWN	KEYUP

System Keystrokes (WM_SYS...)

The Rule: Ignore them.

- **WM_SYSKEYDOWN / WM_SYSKEYUP:** These are triggered by "System" keys, usually **Alt** or **F10**.
- **Windows' Job:** Windows uses these to open menus (File, Edit) or switch windows (Alt-Tab). If you intercept them and don't pass them to DefWindowProc, you might break standard Windows shortcuts (like Alt-F4 to close).

Standard Keystrokes (WM_KEYDOWN)

The Rule: Use for *control*, not *text*.

You should use WM_KEYDOWN to detect:

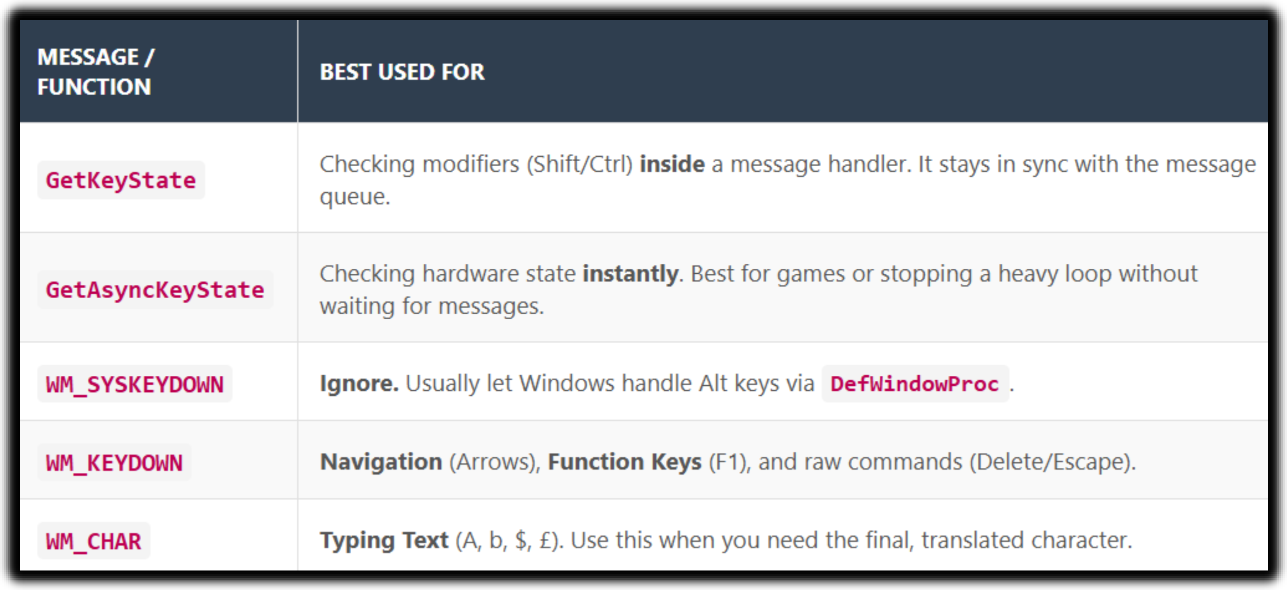
- **Cursor Keys:** Left, Right, Up, Down.
- **Function Keys:** F1, F2, etc.
- **Editing Keys:** Insert, Delete, Home, End.

The Trap (Do NOT do this): Do **not** try to translate keys into text yourself.

- **Bad Logic:** "User pressed 3 + Shift, so I will draw a #."
- **The Problem:** Keyboards are different!
  - ✓ **US Keyboard:** Shift + 3 = #
  - ✓ **UK Keyboard:** Shift + 3 = £
  - ✓ **German Keyboard:** Shift + 3 = §

If you try to hard-code this math, your program will be broken for international users. Wait for the WM_CHAR message (covered next), where Windows has already done this translation for you.

Summary Table



MESSAGE / FUNCTION	BEST USED FOR
`GetKeyState`	Checking modifiers (Shift/Ctrl) **inside** a message handler. It stays in sync with the message queue.
`GetAsyncKeyState`	Checking hardware state **instantly**. Best for games or stopping a heavy loop without waiting for messages.
`WM_SYSKEYDOWN`	**Ignore.** Usually let Windows handle Alt keys via `DefWindowProc` .
`WM_KEYDOWN`	**Navigation** (Arrows), **Function Keys** (F1), and raw commands (Delete/Escape).
`WM_CHAR`	**Typing Text** (A, b, $, £). Use this when you need the final, translated character.

Function Keys: The "Double-Edged Sword"

The Concept: Just because you *can* use F1-F12 doesn't mean you *should*.

- **The Trap:** In old MS-DOS apps, users had to memorize "Ctrl+Shift+F3" to print. This was powerful but hard to learn.
- **The Windows Philosophy:** Do not force users to memorize "secret handshakes."
    - ✓ **Rule of Thumb:** Function keys should only be **shortcuts** for things that are already visible in a menu.
    - ✓ **Goal:** A user should be able to run your program using only the mouse and menus without knowing a single hotkey.

Recommended Keystroke Handling

The Concept: Don't reinvent the wheel. Copy the masters.

When handling WM_KEYDOWN, stick to these standard behaviors found in popular apps like Microsoft Word or Notepad:

- **Primary Keys to Handle:**
  - ✓ Cursor Keys (Arrows)
  - ✓ Insert / Delete.
- **Using Modifiers (Shift/Ctrl):**
  - ✓ **Shift + Arrow:** Standard for **Selecting** items (highlighting text).
  - ✓ **Ctrl + Arrow:** Standard for **Amplified Movement** (e.g., Ctrl+Right usually jumps a whole word instead of one character).

Strategy: If you don't know what a key should do, open Microsoft Word and see what it does there. Deviating from these standards confuses users.

The "SYSMETS" Upgrade Strategy

The Concept: Adding keyboard scrolling without writing duplicate code.

We want to upgrade our "System Metrics" viewer so users can scroll using keys (Home, End, PgUp, Arrows) instead of just the mouse.

The "Bad" Way: Redundancy

You *could* write new logic inside WM_KEYDOWN to calculate lines and repaint the screen.

Why it's bad: You already wrote that logic inside WM_VSCROLL (for the mouse). Copy-pasting it means you now have two separate scrolling engines to maintain. If you fix a bug in one, you might forget the other.

The "Elegant" Way: SendMessage

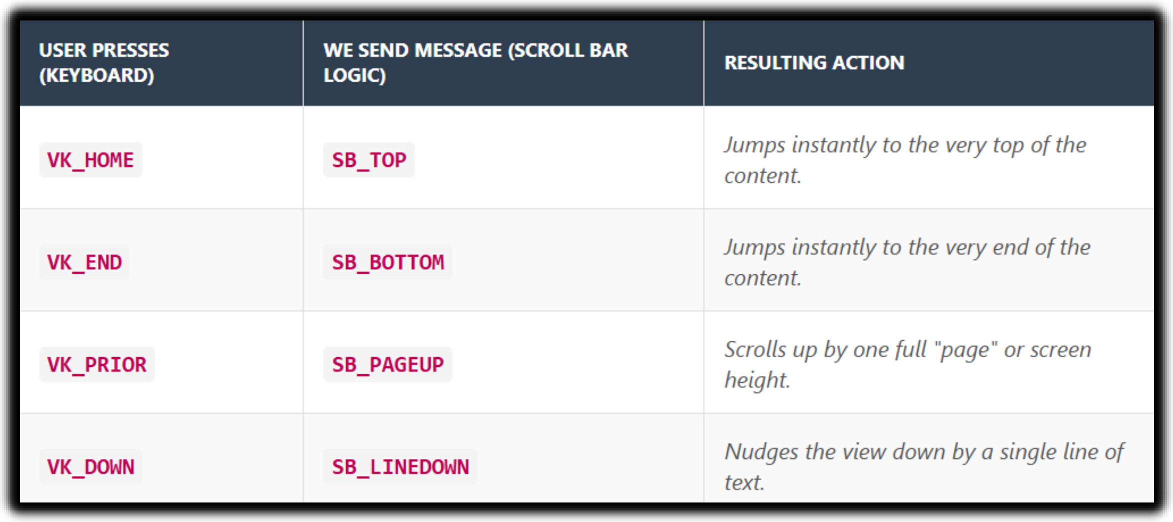
Instead of rewriting the logic, we will **trick** the window into thinking the mouse was used.

We use the SendMessage function to translate a keystroke into a scroll message.

The Workflow:

1. **User Action:** Presses Down Arrow.
2. **Your Code:** Catches WM_KEYDOWN (VK_DOWN).
3. **The Trick:** You call SendMessage(hwnd, WM_VSCROLL, SB_LINEDOWN, ...).
4. **The Result:** Windows receives a WM_VSCROLL message. It doesn't know (or care) that it came from the keyboard. It runs your existing mouse-scrolling code perfectly.

Visualizing the Translation:



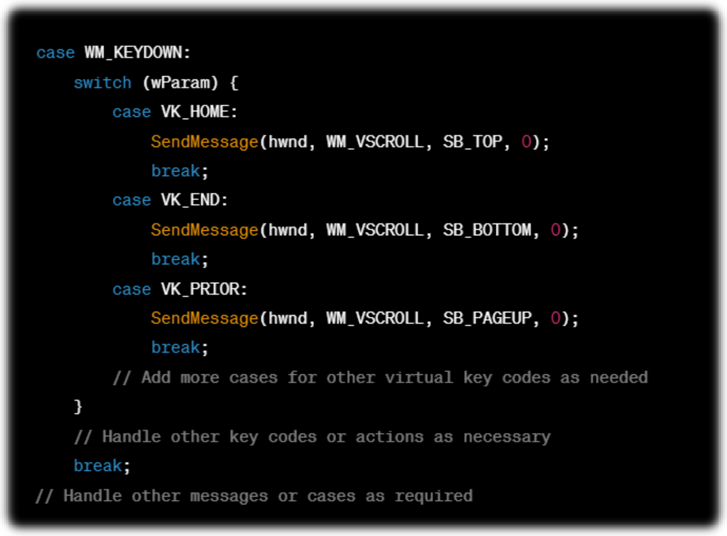
USER PRESSES (KEYBOARD)	WE SEND MESSAGE (SCROLL BAR LOGIC)	RESULTING ACTION
`VK_HOME`	`SB_TOP`	*Jumps instantly to the very top of the content.*
`VK_END`	`SB_BOTTOM`	*Jumps instantly to the very end of the content.*
`VK_PRIOR`	`SB_PAGEUP`	*Scrolls up by one full "page" or screen height.*
`VK_DOWN`	`SB_LINEDOWN`	*Nudges the view down by a single line of text.*

Implementing Keyboard-Triggered Scrolling

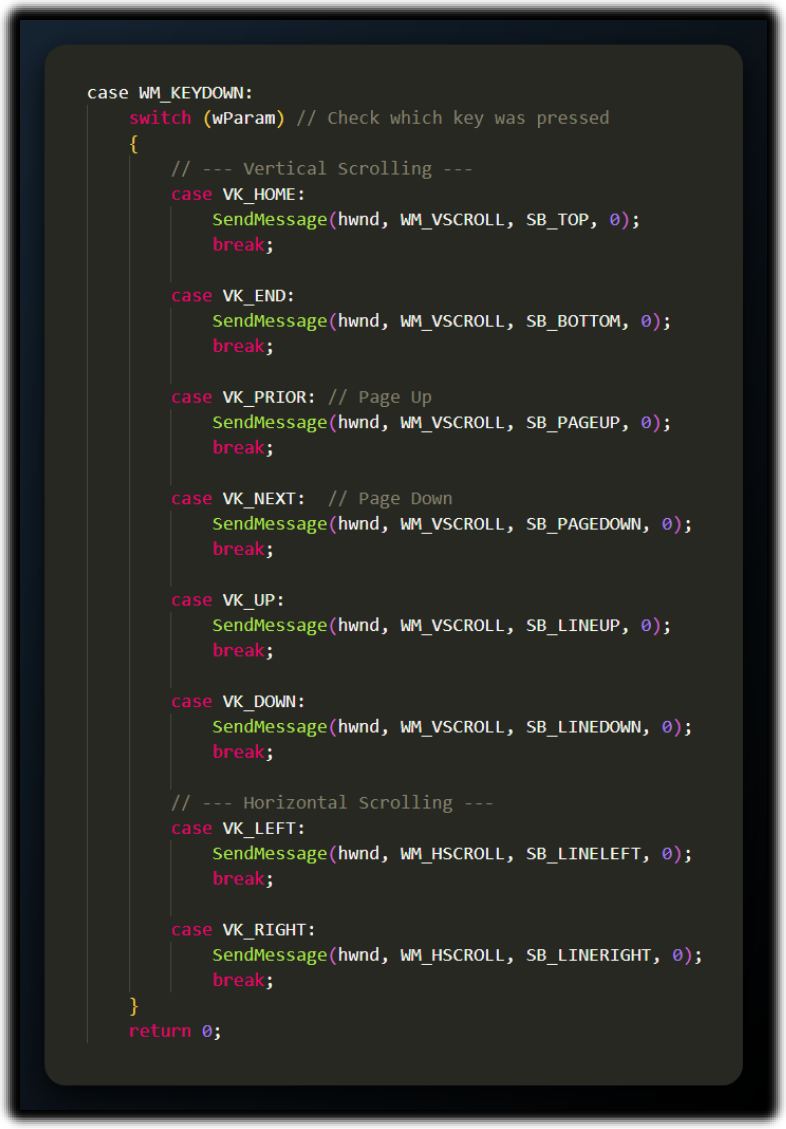
This section demonstrates the "elegant solution" mentioned previously: instead of writing new code to scroll the screen, we simply translate keyboard clicks into scroll bar messages.

1. The Strategy: Translation

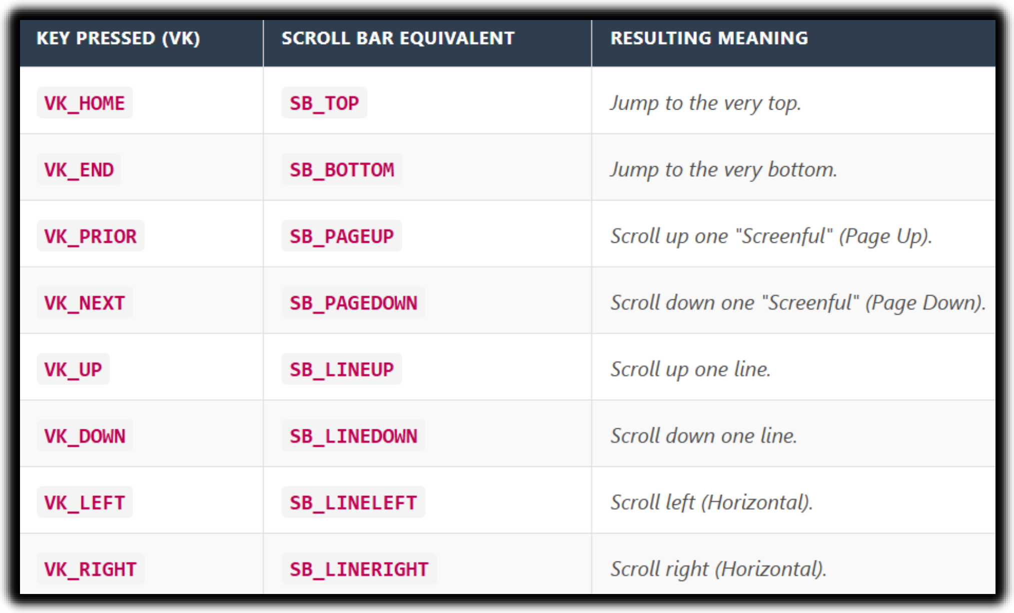
We are going to map **Virtual Keys** (Inputs) to **Scroll Commands** (Actions).



```
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
```



```
case WM_KEYDOWN:
    switch (wParam) // Check which key was pressed
    {
        // --- Vertical Scrolling ---
        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: // Page Up
            SendMessage(hwnd, WM_VSCROLL, SB_PAGEUP, 0);
            break;

        case VK_NEXT:  // Page Down
            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;

        // --- Horizontal Scrolling ---
        case VK_LEFT:
            SendMessage(hwnd, WM_HSCROLL, SB_LINELEFT, 0);
            break;

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



KEY PRESSED (VK)	SCROLL BAR EQUIVALENT	RESULTING MEANING
`VK_HOME`	`SB_TOP`	*Jump to the very top.*
`VK_END`	`SB_BOTTOM`	*Jump to the very bottom.*
`VK_PRIOR`	`SB_PAGEUP`	*Scroll up one "Screenful" (Page Up).*
`VK_NEXT`	`SB_PAGEDOWN`	*Scroll down one "Screenful" (Page Down).*
`VK_UP`	`SB_LINEUP`	*Scroll up one line.*
`VK_DOWN`	`SB_LINEDOWN`	*Scroll down one line.*
`VK_LEFT`	`SB_LINELEFT`	*Scroll left (Horizontal).*
`VK_RIGHT`	`SB_LINERIGHT`	*Scroll right (Horizontal).*

SYSMETS4, which represents the "Complete Package" in this chapter of your learning. It combines everything from previous versions (System Metrics + Scroll Bars) and adds the final piece: **Keyboard Control**.

The Life Cycle of SYSMETS4

The text outlines the standard heartbeat of a Windows application. Here is how the messages flow to create the user experience:

- **Birth (WM_CREATE)**: The window opens. The app gathers the system metrics (width of screen, height of menus, etc.) and prepares the data structures.
- **Adaptation (WM_SIZE)**: The user stretches the window. The app recalculates the "Page Size" (how many lines fit on screen) to adjust the scroll bars.
- **The Action Loop (WM_PAINT)**: The heavy lifter. Whenever the user scrolls or resizes, this message redraws the text in the correct new position.
- **Death (WM_DESTROY)**: The user closes the window. The app cleans up and posts the Quit message.

1. The Interaction (Mouse vs. Keyboard)

This is where the program shines by handling two different inputs for the same result.

- **Mouse Input (WM_VSCROLL / WM_HSCROLL)**: The user drags the scroll bar thumb. The app updates the view directly.
- **Keyboard Input (WM_KEYDOWN)**: The user presses "Down Arrow." The app catches this and **pretends** it was a scroll bar click, triggering the scrolling logic.

2. Comparison: SYSMETS3 vs. SYSMETS4

You asked specifically about the differences. Based on the progression of the code:

SYSMETS3 (The previous version) had:

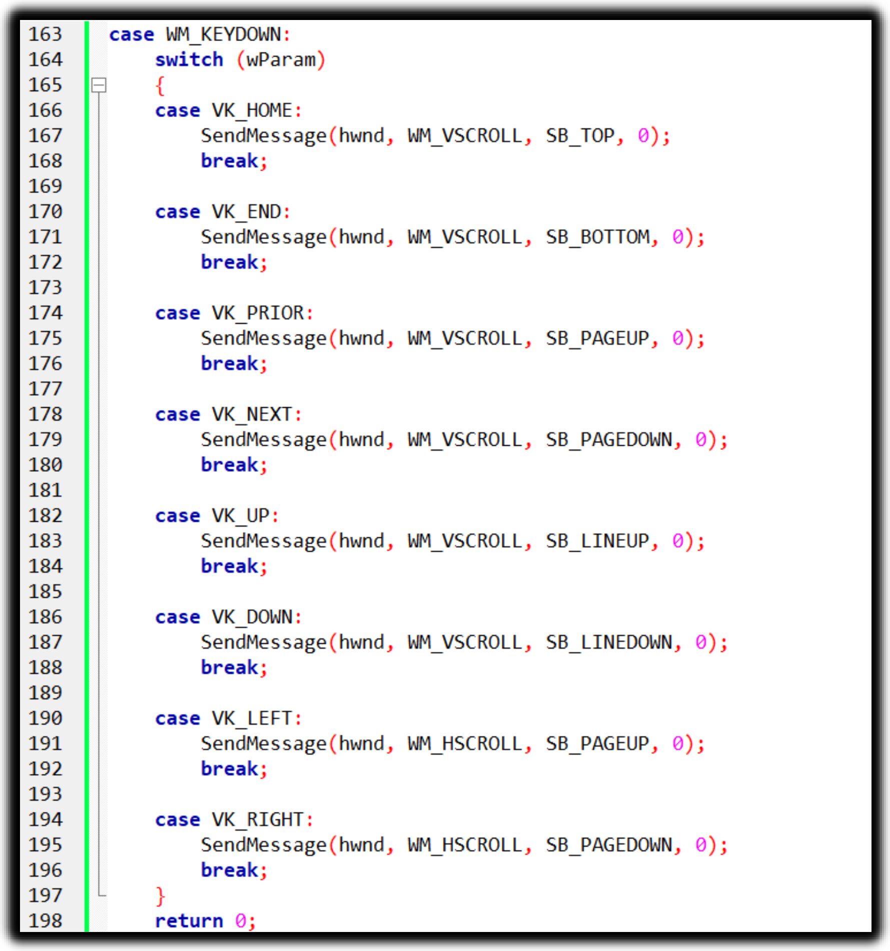
- [x] System Metrics display.
- [x] Mouse support (Scroll bars worked by clicking).
- [ ] **NO Keyboard support** (Pressing "Page Down" did nothing).

SYSMETS4 adds exactly one major feature: **The WM_KEYDOWN Logic.**

It includes the specific code block we just reviewed that "translates" keys into scroll messages:

1. **Cursor Keys:** VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT.
2. **Navigation Keys:** VK_HOME, VK_END.
3. **Page Keys:** VK_PRIOR (Page Up), VK_NEXT (Page Down).

In short: SYSMETS4 makes the application **accessible** to users who prefer (or need) to use the keyboard instead of the mouse.



```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;
```

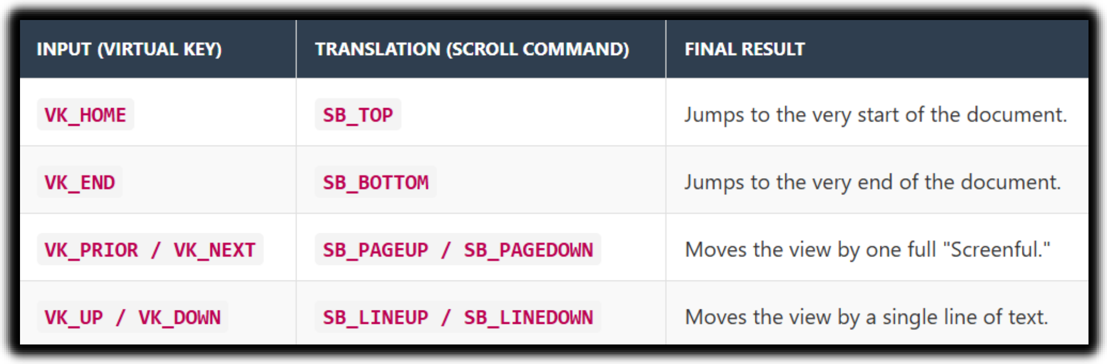
Summary: The Keyboard-Scroll Connection

The **SYSMETS4** program adds keyboard navigation by treating the keyboard as a remote control for the scroll bars. Here is the final logic flow:

1. The Trigger (WM_KEYDOWN) - When the user presses a non-system key (like an Arrow or Page key), Windows sends this message to your window procedure.

2. The Identifier (wParam) - The message carries the Virtual Key Code in wParam, telling the program exactly which button was pushed (e.g., VK_HOME or VK_DOWN).

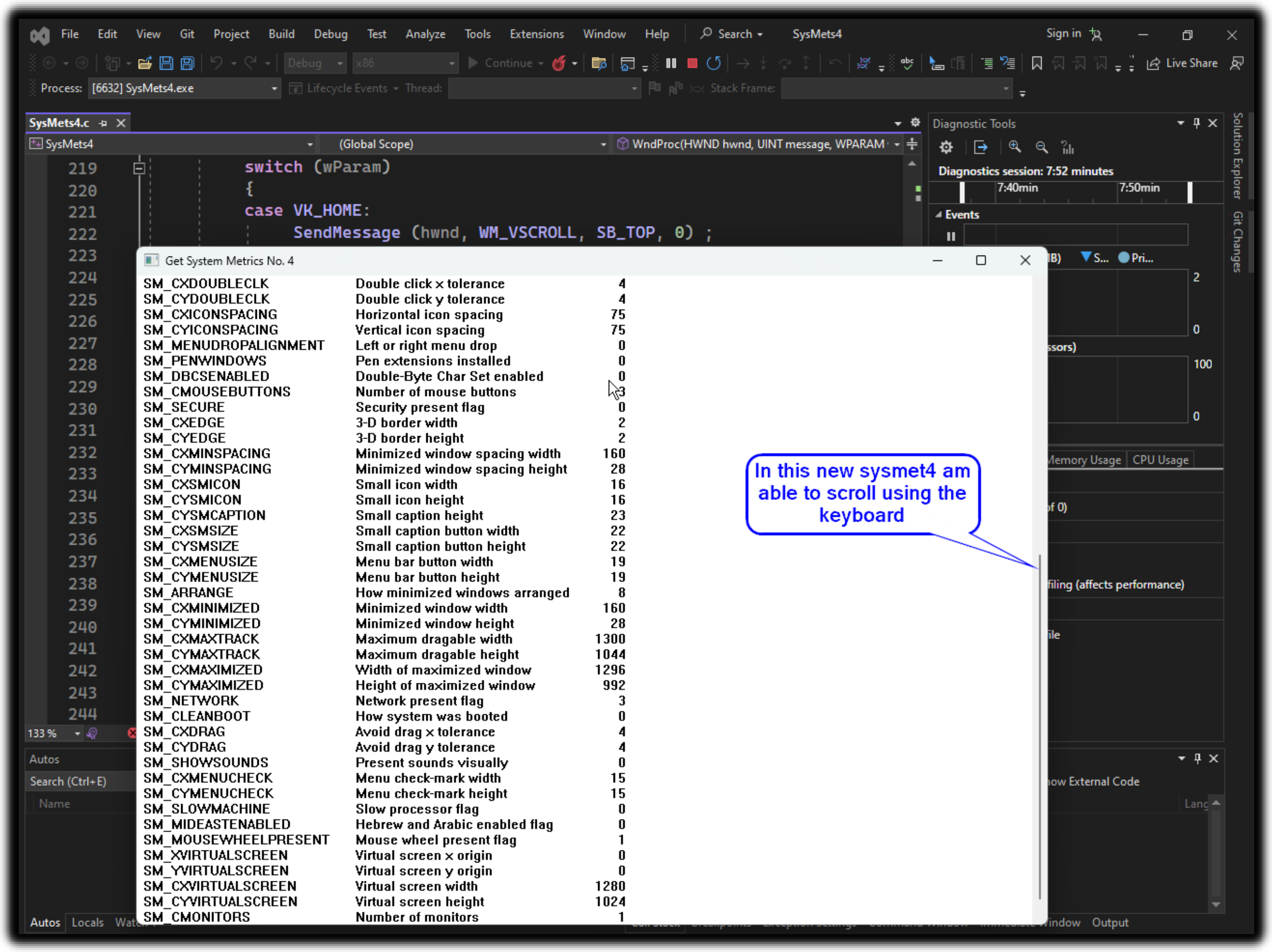
3. The Translation (The Switch Statement) - The program uses a switch statement to check the key code and translate it into a scrolling command:



INPUT (VIRTUAL KEY)	TRANSLATION (SCROLL COMMAND)	FINAL RESULT
`VK_HOME`	`SB_TOP`	Jumps to the very start of the document.
`VK_END`	`SB_BOTTOM`	Jumps to the very end of the document.
`VK_PRIOR / VK_NEXT`	`SB_PAGEUP / SB_PAGEDOWN`	Moves the view by one full "Screenful."
`VK_UP / VK_DOWN`	`SB_LINEUP / SB_LINEDOWN`	Moves the view by a single line of text.

4. The Execution (SendMessage) - Instead of rewriting the scrolling code, the program uses SendMessage to fire a WM_VSCROLL or WM_HSCROLL message at itself. This triggers the existing scroll bar logic, moving the content seamlessly.

5. Conclusion - This implementation completes the basic window interface. Your application now fully supports both Mouse (clicking scroll bars) and Keyboard (navigation keys) interactions using a single, unified logic engine.



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

```
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
```

In this new sysmet4 am able to scroll using the keyboard