

THE MOUSE: DETECTION AND LOGIC

1. Determining Mouse Presence

To check if a mouse is connected, you use the GetSystemMetrics function.

- **The Constant:** SM_MOUSEPRESENT.
- **The Legacy Quirk:** In older versions of Windows (like Windows 98), this function often lied. It would return TRUE even if no mouse was plugged in. In modern Windows (NT-based), it is accurate.



2. Counting Buttons

Windows supports mice with one, two, or three buttons.

- **The Constant:** SM_CMOUSEBUTTONS.
- **The Logic:** This returns the number of physical buttons. If no mouse is installed, it should return 0 (though legacy Windows 98 often defaulted to 2).

3. Left-Handed vs. Right-Handed (Logical Buttons)

Users can swap their mouse buttons in the Control Panel.

- **The Rule:** As a programmer, you don't care which physical button is clicked. You care about the **Logical Button**.
- **Logical Left:** This is the "Primary" button used for selecting and clicking.
- **Logical Right:** This is the "Secondary" button used for context menus.
- **The Check:** GetSystemMetrics(SM_SWAPBUTTON) tells you if the user has swapped them, but Windows handles the messages (WM_LBUTTONDOWN, etc.) automatically so your code stays the same.

4. Adjusting Settings

The function `SystemParametersInfo` is the "Master Switch" for mouse behavior. It can get or set:

- **Double-click speed:** How fast two clicks must happen to count as one double-click.
 - **Mouse sensitivity:** How far the pointer moves when you move the physical mouse.
-

Reverse Engineering Perspective: Malware & The Mouse

In malware analysis and ethical hacking, the mouse is a major "signal" used to detect human presence.

- **Sandbox Evasion:** Many automated malware analysis "sandboxes" (like Cuckoo or Any.Run) do not simulate realistic mouse movement. Malware will call `GetCursorPos`, wait 10 seconds, and call it again. If the mouse hasn't moved, the malware assumes it is in a virtual environment and will not execute the payload.
 - **"Prank" Malware:** Simple malicious scripts often use `SystemParametersInfo` to swap mouse buttons or change the double-click speed to an impossible value to frustrate the user.
 - **Clickjacking:** Malicious apps can create invisible windows over legitimate ones. When the user thinks they are clicking a button on a trusted app, they are actually clicking a hidden "Allow" button on the malware.
 - **Global Hooks:** Malware often uses `SetWindowsHookEx` with `WH_MOUSE_LL` (Low-Level Mouse Hook). This allows the malware to record every click a user makes, which can be used to steal coordinates or track activity.
-

20 Random Non-Mouse related Questions

These questions cover Fonts, Carets, and the Mouse sections we have revamped.

1. How does a **Vector Font** (like TrueType) differ from a **Bitmap Font** in terms of storage?
2. Why is the **Terminal** font often used in command-line environments?
3. What happens to the **Caret** when a window receives the `WM_KILLFOCUS` message?
4. Why is `HideCaret` considered an "additive" function?
5. What is the purpose of **Hinting** in TrueType fonts?
6. Which function do you call to determine the number of buttons on a mouse?

7. If a user swaps their mouse buttons in the Control Panel, how does a programmer distinguish between the "Action" button and the "Menu" button?
 8. Why does malware check for mouse movement before activating its main code?
 9. What is the difference between a **Character Set** and a **Font**?
 10. Which WinAPI function is used to create a custom "Logical Font"?
 11. What is the **OEM Character Set** and why is it still supported?
 12. Why should CreateCaret be called during WM_SETFOCUS instead of WM_CREATE?
 13. What is a **Fixed-Pitch** font, and why does the TYPER program use one?
 14. If GetSystemMetrics(SM_MOUSEPRESENT) returns TRUE on Windows 98, can you be 100% sure a mouse is attached?
 15. What does the SystemParametersInfo function allow a developer to change regarding the mouse?
 16. In the TYPER program, what is the purpose of the xCaret and yCaret variables?
 17. What is the "Tofu" block (□) in typography, and when does it appear?
 18. How can a reverse engineer use GetCursorPos to detect if a program is anti-debugging?
 19. Which WinAPI message is sent when the user presses the "Primary" mouse button?
 20. Why must you call ShowCaret after calling CreateCaret?
-

Fun facts

The mouse cursor is a small bitmapped picture that moves on the display as the user moves the mouse.

The hot spot is the single-pixel point on the cursor that indicates the precise location on the display.

Windows supports several predefined mouse cursors, such as IDC_ARROW, IDC_CROSS, and IDC_WAIT.

Programmers can also design their own custom cursors.

The default cursor for a particular window is specified when defining the window class structure.

Common mouse actions include clicking, double-clicking, and dragging.

On a three-button mouse, the buttons are called the left button, the middle button, and the right button.

On a two-button mouse, there is only a left button and a right button.

The single button on a one-button mouse is a left button.

The plural of "mouse" is a matter of debate, with both "mice" and "mouses" being considered acceptable.

The Microsoft Manual of Style for Technical Publications recommends avoiding the plural "mice" and using "mouse devices" instead.

Mouse vs. Keyboard: The "Focus" Rule

The biggest difference between the keyboard and the mouse is **who** gets the message.

- **Keyboard:** Messages *only* go to the window that has **Input Focus** (the active window). If you type while hovering over a background window, the background window ignores it.
 - **Mouse:** Messages go to **whatever window is under the cursor**.
 - **Result:** You can click a button on a background window without activating it first. Windows calculates which window is physically under the pixel you clicked and sends the message there.
-

The 10 Client-Area Messages

There are 21 total mouse messages, but we care about the **10 Client-Area messages**.

These happen when the mouse is inside the "white space" of your application (not the title bar or borders).

ACTION	LEFT BUTTON	MIDDLE BUTTON	RIGHT BUTTON
Pressed	WM_LBUTTONDOWN	WM_MBUTTONDOWN	WM_RBUTTONDOWN
Released	WM_LBUTTONUP	WM_MBUTTONUP	WM_RBUTTONUP
Double Click	WM_LBUTTONDOWNDBLCLK	WM_MBUTTONDOWNDBLCLK	WM_RBUTTONDOWNDBLCLK

Plus one for movement:

WM_MOUSEMOVE: Sent constantly as the cursor slides across the window.

Anatomy of a Mouse Message

When you receive a mouse message (like WM_LBUTTONDOWN), the parameters contain all the data you need.

A. Where is the mouse? (lParam)

The lParam holds the X and Y coordinates of the cursor **relative to the top-left corner** of your client area.

Technique: They are packed into a single 32-bit number. You must split them.

```
x = LOWORD(lParam); // The horizontal position  
y = HIWORD(lParam); // The vertical position
```

B. What keys are held? (wParam)

The wParam tells you if keys (Shift, Ctrl) or *other* mouse buttons are currently held down. This lets you handle "Shift+Click" or "Ctrl+Click" logic.

The Flags (Bitmasks):

- ✓ MK_LBUTTON: Left button is down.
- ✓ MK_RBUTTON: Right button is down.
- ✓ MK_SHIFT: Shift key is down.
- ✓ MK_CONTROL: Ctrl key is down.

```
if (wParam & MK_CONTROL) {  
    // This was a Ctrl + Click  
}
```

4. Important Behaviors

A. The "Wake Up" Click

If your window is inactive (in the background) and you click inside it:

1. Windows automatically switches focus to your window (Active).
2. Windows *then* sends the WM_LBUTTONDOWN message. **Benefit:** You don't need special code to "wake up" your app; the click is processed immediately.

B. The "Unpaired" Problem

Mouse messages are strictly about **location**.

Scenario:

- User presses Mouse Down inside your window (WM_LBUTTONDOWN).
- User drags the mouse *outside* your window.
- User releases the button.

Result: You will **NOT** receive the WM_LBUTTONUP message.

- Why? Because the release happened over someone else's window.
- **Implication:** If you are dragging an object, your code needs to account for the mouse leaving the window (often solved by "Capturing the Mouse," which is likely the next topic).

5. Mouse Capture: The "Invisible Leash"

The Problem: As mentioned earlier, if you click inside your window and drag the mouse outside the border, your window stops receiving messages.

Scenario: You are drawing a line. You drag the mouse off the edge of the window. The line stops following you because the window "lost" the mouse.

The Solution: You can "Capture the Mouse" (SetCapture).

- This creates a "lock" where Windows sends **ALL** mouse messages to your window, even if the cursor is floating over the Desktop, Chrome, or the Start Menu.
- **Usage:** You usually turn this on during WM_LBUTTONDOWN (start dragging) and turn it off during WM_LBUTTONUP (stop dragging) using ReleaseCapture.

6. System Modal: The "Blockade"

The Concept:

A "System Modal" dialog is a message box that demands absolute attention.

- **Behavior:** When active, it freezes the mouse input for **every other application** on the system. You cannot click Start, you cannot click other windows. You must deal with the dialog box first.
- **Usage:** Used for critical system errors (though modern Windows tries to avoid this to prevent freezing the user).

7. The Code: Handling the Click

Here is the corrected and commented version of the code snippet you provided.

Fixed 1Param → lParam.

Fixed LBUTTONDOWN → WM_LBUTTONDOWN.

Connect folder in Chapter 7 has the code. Here's the video for it's working...

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_LBUTTONDOWN: // User clicked Left Button
        {
            // Extract Coordinates using bit-masks
            int x = LOWORD(lParam);
            int y = HIWORD(lParam);

            // --- Your Logic Here ---
            // For the "Connect" app, this is where you would:
            // 1. Save the point (x,y) into memory.
            // 2. Repaint the window to show a dot or line.
            // 3. SetCapture(hwnd) if you plan to drag.

            return 0;
        }

        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

8. The "Connect" Program (Preview)

You mentioned the **Connect** folder. This is the classic "Connect-the-Dots" example for this chapter.

How it works:

1. **Input:** The user clicks the mouse at random spots in the window.
2. **Storage:** The program saves up to 100 points in an array (similar to KEYVIEW1 storing keys).
3. **Drawing (WM_PAINT):**
 - ✓ Windows connects Point 1 to Point 2.
 - ✓ Point 2 to Point 3.
 - ✓ And so on...
4. **The Result:** A polyline drawing created entirely by mouse clicks.



Connect Mouse
chapter 7 Part 1.mp4

This program demonstrates "Tracking," which is the art of following the mouse as it moves across the screen to create a drawing.

The Workflow: How it Works

The program behaves like a specialized drawing tool. It has three distinct phases controlled by the mouse buttons:

1. **The Reset (WM_LBUTTONDOWN):**
 - ✓ **Action:** User clicks the left button.
 - ✓ **Result:** The screen is wiped clean. The program is ready for a new drawing.
2. **The Tracking (WM_MOUSEMOVE):**
 - ✓ **Action:** User drags the mouse while holding the button.
 - ✓ **Result:** The program acts like a pen. It draws black dots (SetPixel) wherever the cursor goes and secretly records those X,Y coordinates in memory.

3. The Calculation (WM_LBUTTONDOWN):

- ✓ **Action:** User releases the button.
- ✓ **Result:** The "Spiderweb" effect. The program connects **every single dot to every other dot**.

The Logic: "Connect Everything"

The unique feature of this program is what happens when you let go. It doesn't just draw a line from start to finish. It creates a dense mesh.

- **Complexity:** If you draw 100 dots, the computer has to calculate and draw lines between point 1 and 2, 1 and 3, 1 and 4... all the way to 100. Then 2 and 3, 2 and 4, etc.
- **The Cursor:** Because this math can take a few seconds on older hardware (or with many dots), the program switches the cursor to an **Hourglass** (IDC_WAIT) to tell the user "I'm thinking."

The GDI Toolset

The program uses three specific graphics functions to achieve this:

FUNCTION	PURPOSE	USAGE IN CONNECT PROGRAM
SetPixel	Draw a single dot.	<i>Used during WM_MOUSEMOVE to show the user where they are currently drawing.</i>
MoveToEx	Lift the pen and move it.	<i>Used in WM_LBUTTONDOWN to set the "starting point" for a new line.</i>
LineTo	Lower the pen and draw.	<i>Used in WM_LBUTTONUP to draw the actual connecting lines between points.</i>

The Limitations (The "Capture" Bug)

The book highlight a very specific quirk: "**If you move the mouse cursor out of the client area... CONNECT does not connect the dots.**"

- **Why?** The program is **not** using SetCapture.
- **The Consequence:** As soon as the mouse leaves the window, the window stops receiving WM_MOUSEMOVE messages. More importantly, if you release the button *outside* the window, the window never receives the WM_LBUTTONUP message. It never knows you finished, so it never triggers the "Connect" logic.

Program Behavior Summary

- **Storage:** It has a hard limit of **1000 points**. If you draw a line that is too long, it stops recording.
- **Multitasking:** Because this is Windows (preemptive multitasking), even if the CONNECT program is frozen for 5 seconds drawing lines, you can Alt-Tab away and use other programs. The OS does not freeze.

Processing Shift Keys (wParam) and Button Emulation.

This section teaches you how to detect "Modifier Keys" (Shift and Ctrl) during mouse actions. This is essential for creating advanced tools, like "Shift-Click to Select" or "Ctrl-Click to Copy."

The Role of wParam

When you receive a mouse message (like WM_MOUSEMOVE), Windows packs extra information into the wParam variable. It acts like a "Status Report" of the keyboard *at that exact moment*.

How to read it: The variable wParam is a collection of bits (flags). To check one specific flag (like "Is Shift down?"), you use the **Bitwise AND (&)** operator with a "Mask" constant (like MK_SHIFT).

Logic: if (StatusReport AND ShiftMask) == TRUE

Here's an example of how to check if the Shift key is pressed:

```
if (wParam & MK_SHIFT)
{
    // The Shift key is physically held down right now.
    // Example: Draw a straight line instead of a freehand curve.
```

Advanced Combinations (Shift + Ctrl)

You often need to handle complex combos. For example, in Photoshop:

- **Click:** Paint.
- **Shift + Click:** Draw straight line.
- **Ctrl + Click:** Move layer.
- **Shift + Ctrl + Click:** Copy layer.

The logic requires checking multiple flags simultaneously. The nested structure in your notes works, but here is the logic visualized:

Here's an example of how to check for Shift and Ctrl key combinations:

```
// Inside your Mouse Message Handler
if (wParam & MK_SHIFT)
{
    if (wParam & MK_CONTROL) {
        // CASE: Shift + Ctrl are BOTH down
    }
    else {
        // CASE: Only Shift is down
    }
}
else
{
    if (wParam & MK_CONTROL) {
        // CASE: Only Ctrl is down
    }
    else {
        // CASE: No modifiers (Just a normal click)
    }
}
```

Emulating the Right Button (The "One-Button Mouse" Fix)

The Context: Some older systems (and Macs) only had one mouse button. To let those users access "Right Click" features (like Context Menus), developers used a shortcut: **Shift + Left Click = Right Click.**

The Implementation: You intercept the *Left Click* message, check for Shift, and if it's there, you run your *Right Click* code instead.

Here's an example of how to emulate the right button click:

```
case WM_LBUTTONDOWN:
    // Check if shift is NOT held down
    if (!(wParam & MK_SHIFT))
    {
        // STANDARD BEHAVIOR:
        // The user just clicked Left Button (No Shift).
        // Perform normal left-click action (e.g., select item).
    }
    else
    {
        // EMULATION BEHAVIOR:
        // The user clicked Left Button + Shift.
        // Pretend this was actually a Right Button click.

        // Call your Right-Click function here:
        OnRightclick();
    }
return 0;
```

Using GetKeyState for Mouse and Key States

1. Using GetKeyState (The "Live" Check)

Sometimes wParam isn't enough. wParam tells you the state of keys **at the exact moment the message was created**. GetKeyState allows you to ask Windows for the status of a key during your processing.

- **How it works:** It returns a SHORT (16-bit integer).
- **The Check:** If the **High-Order bit** is 1 (which makes the number negative), the key is **DOWN**.

```
// Check if Left Mouse Button is currently held down
if (GetKeyState(VK_LBUTTON) < 0)
{
    // Button is DOWN
}
```

METHOD	SOURCE	WHEN TO USE
wParam	Passed as a variable in the message.	Preferred. Fast and synced exactly to the event. It tells you the state of the key at the moment the message was generated.
GetKeyState	Function call to the system.	Use when you need to check a key that isn't included in wParam (like checking if 'A' is held during a mouse click).

Double-Clicks: The "Two-in-One" Logic

Handling double-clicks is tricky because a double-click is actually **two** single clicks that happen very fast.

a. The Requirement: CS_DBLCLKS

By default, Windows does **not** detect double-clicks. It just sees two separate clicks. To enable detection, you must add a specific flag when you first design your window class (back in WinMain):

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
```

If you forget this flag:

Your window will receive

WM_LBUTTONDOWN → WM_LBUTTONUP → WM_LBUTTONDOWN → WM_LBUTTONUP

You will never get the "**Double Click**" message.

b. The Sequence

If CS_DBLCLKS is set, Windows watches the clock. If the second click happens fast enough (within the system's **Double Click Speed**), it replaces the second "**Down**" message with a special "**Double Click**" message.

Action	Message Sent	What Your Code Sees
User presses button	WM_LBUTTONDOWN	"First Click started."
User releases button	WM_LBUTTONUP	"First Click finished."
User presses again (Fast)	WM_LBUTTONDOWNDBLCLK	"This is a Double Click!"
User releases again	WM_LBUTTONUP	"Double Click finished."

c. The Coding Challenge

Because the first click *always* happens before the double-click, your code must be careful.

- **Rule:** The "Single Click" action must be safe to run even if a "Double Click" is about to happen.
- **Example (Windows Explorer):**
 - ✓ **Single Click:** Selects the file (Highlight blue).
 - ✓ **Double Click:** Opens the file.
 - ✓ *Sequence:* The file gets highlighted (Click 1), *then* it opens (Double Click). This feels natural.
 - ✓ *Bad Design:* If Single Click meant "Delete File," you could never Double Click to open it because the file would be deleted after the first click!

The Strategy of Double-Clicks

Designing a double-click feature is more about logic than code.

a) The Simple (Standard) Approach

This is how Windows Explorer works, and it is the easiest to code.

Logic: The actions are **additive**.

- **Click 1:** "Select this item."
- **Click 2:** "Open this item."

Why it works: When the user double-clicks, your code actually runs *both* actions.

- WM_LBUTTONDOWN: You select the file.
- WM_LBUTTONUP: (Nothing usually).
- WM_LBUTTONDOWNDBLCLK: You open the file.

Result: The user sees the file quickly highlight before it opens. This feels natural.

b) The Complex Approach (Non-Additive)

This is rare and difficult.

- **Logic:** Single Click = Action A. Double Click = Action B (Action A should **NOT** happen).
- **The Problem:** Since the single click *always* happens first, Action A triggers before you know if the user is going to click again.
- **The Fix:** You have to intentionally **delay** Action A. You wait a few milliseconds (using GetMessageTime) to see if a second click arrives. If it does, you run Action B. If the timer runs out, you run Action A.
- **Recommendation:** Avoid this design if possible. It makes the UI feel "laggy."

Non-Client Area Messages (The "Frame" Messages)

Up until now, we have discussed the **Client Area** (the white canvas where your app lives). But the mouse also interacts with the **Non-Client Area** (the Title Bar, Borders, Scrollbars, and Buttons provided by Windows).

a) The Message Format

These messages look just like normal mouse messages, but they have the prefix **NC**.

ACTION	CLIENT AREA (YOUR CANVAS)	NON-CLIENT AREA (TITLE BAR/FRAME)
Move	WM_MOUSEMOVE	WM_NCMOUSEMOVE
Left Button Down	WM_LBUTTONDOWN	WM_NCLBUTTONDOWN
Left Button Up	WM_LBUTTONUP	WM_NCLBUTTONUP

b) What do you do with them?

Usually, nothing. You almost always pass these to DefWindowProc.

- If you ignore WM_NCLBUTTONDOWN, the user won't be able to drag your window or click the "X" button.
- DefWindowProc does the heavy lifting: it detects if the click was on the Title Bar and handles the dragging logic for you.

3. The Strange Parameters of NC Messages

If you do decide to process these messages, be careful. The parameters (wParam/lParam) work completely differently than standard mouse messages.

a) wParam: The "Hit-Test" Code

In standard messages, wParam tells you about Shift/Ctrl keys. In NC messages, wParam tells you **what part of the window structure** the mouse is over. These are constants starting with HT (Hit Test).

- **HTCAPTION:** The Title Bar.
- **HTCLOSE:** The 'X' Button.
- **HTCLIENT:** The Client Area (inside the frame).
- **HTBORDER:** The sizing border.

b) lParam: Screen Coordinates

This is the most dangerous difference.

- **Standard (WM_MOUSEMOVE):** x,y are relative to the **Window's top-left corner**. (0,0 is the top-left of your canvas).
- **Non-Client (WM_NCMOUSEMOVE):** x,y are relative to the **Screen's top-left corner**. (0,0 is the top-left of your Monitor).

Warning: If you mix these up, your mouse logic will be off by hundreds of pixels. You often need functions like ScreenToClient() or ClientToScreen() to convert between them.

Converting Screen Coordinates to Client-Area Coordinates

1. The "GPS" Problem (Screen vs. Client)

Imagine you are trying to find a specific seat in a movie theater.

- **Screen Coordinates (Global GPS):** This is like giving your latitude and longitude on the planet Earth. (0,0) is the absolute top-left corner of your monitor.
- **Client Coordinates (Local Seat Number):** This is like saying "Row 3, Seat 5" *inside* the theater. (0,0) is the top-left corner of your window's white drawing area.

The Issue: Non-Client messages (like when you click the Title Bar) give you **Global GPS** coordinates. But your drawing functions (like SetPixel or drawing lines) only understand **Local Seat Numbers**.

If you try to use the Global numbers to draw inside your window, you will be drawing hundreds of pixels off-target (or off the screen entirely). You need a translator.

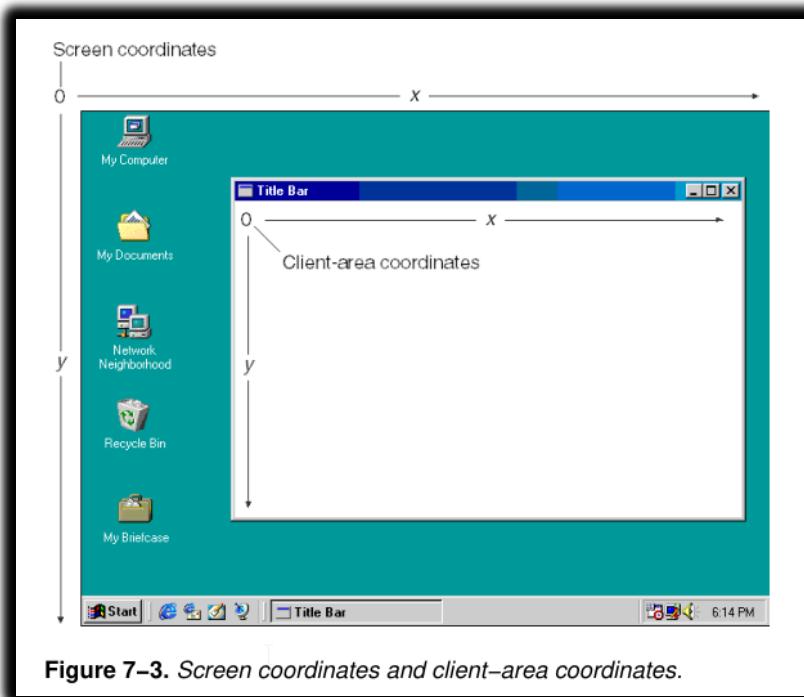


Figure 7–3. Screen coordinates and client-area coordinates.

2. The Translators: ScreenToClient and ClientToScreen

Windows gives you two functions to fix this math instantly. They take a POINT (a simple bundle of X and Y) and flip the numbers for you.

ScreenToClient(hwnd, &point):

- *Input:* "The mouse is at pixel 1024, 768 on the monitor."
- *Output:* "Okay, relative to *your* window, that is pixel 50, 50."
- *Usage:* Use this when you get a weird message (like a Non-Client message) but need to react to it inside your window.

ClientToScreen(hwnd, &point):

- *Input:* "I have a button at 10, 10 inside my app."
- *Output:* "On the whole monitor, that button is physically located at 500, 300."
- *Usage:* Use this if you want to spawn a popup menu exactly where the mouse is.

3. Summary: The "Non-Client" Stuff

Let's simplify that final block of notes.

The **Non-Client Area** is just the "Frame" that Windows draws for you—the Blue Title bar, the Minimize button, and the Resize border.

- **Why do these messages exist?** So Windows knows when you are trying to drag the window or close it.
- **What do you do with them? Ignore them.** Pass them to DefWindowProc. It acts as the "autopilot" that handles all the dragging and resizing logic for you. You only mess with these messages if you are trying to do something crazy, like making a window shaped like a donut.

WM_NCHITTEST (The Non-Client Hit Test)

This is arguably the most important message you never see, because it happens invisibly in the background. It is the "Root Cause" of all the other messages we have discussed.

1. The "Radar" Signal

Think of WM_NCHITTEST as a sonar ping.

Before Windows decides to send you a WM_MOUSEMOVE or a WM_LBUTTONDOWN, it has to answer a fundamental question: **"Where exactly is the mouse?"**

Is it on the Title Bar? The Minimize button? The Border? The Client Area?

The Sequence of Events:

1. **Physical Action:** You move the mouse.
2. **The Question:** Windows sends WM_NCHITTEST to your window. ("Hey, I see the mouse at Screen Coordinate (500, 300). What part of your window is that?")
3. **The Calculation:** DefWindowProc does the math. It checks your window size and borders.
4. **The Answer:** It returns a code (e.g., "That's the Title Bar!").
5. **The Result:** *Based on that answer*, Windows **then** generates the specific message (e.g., WM_NCMOUSEMOVE) and sends it to you.

2. The Input: Screen Coordinates

As with other non-client messages, WM_NCHITTEST deals with **Global GPS** (Screen Coordinates).

- **lParam:** Contains the X (Low Word) and Y (High Word) relative to the screen.
- **wParam:** Ignored (as per your notes).

3. The Output: The HT Codes

The return value of this message is not 0. It is a specific "Hit Test Code" that tells Windows how the cursor should behave (e.g., should it turn into a resize arrow?).

Common Return Values:

CODE	MEANING	RESULTING BEHAVIOR
HTCLIENT	The White Canvas	Windows sends standard WM_MOUSEMOVE. Cursor stays an arrow.
HTCAPTION	The Title Bar	Windows allows you to drag the window.
HTCLOSE	The 'X' Button	Windows highlights the Close button.
HTTOP / HTBOTTOM	The Borders	Windows changes cursor to a Resize Arrow (↔).
HTTRANSPARENT	"I'm a Ghost"	Windows ignores your window and sends the click to the window behind you.
HTERROR	Error	Windows beeps and rejects the input.

4. Why would you ever mess with this?

99% of the time, you let DefWindowProc handle this. However, hacking WM_NCHITTEST allows for cool tricks.

The "Fake Title Bar" Trick: If you want to create a window that has no title bar but can still be dragged by clicking *anywhere* (like Winamp or a modern skin), you simply do this:

```
case WM_NCHITTEST:  
    // Lie to Windows!  
    // Tell it: "No matter where the user clicks, pretend it's the Title Bar."  
    return HTCAPTION;
```

Now, when the user clicks the client area, Windows *thinks* they clicked the Title Bar and initiates the "Move Window" logic automatically.

Summary

- **WM_NCHITTEST** is the "Scout" message sent first.
 - It asks "Where is the mouse?"
 - DefWindowProc usually answers it for you.
 - The answer determines if you get a Client message (WM_LBUTTONDOWN) or a Non-Client message (WM_NCLBUTTONDOWN).
-

Disabling Mouse Interaction with HTNOWHERE

"Ghost Window" technique using WM_NCHITTEST

This technique acts as the ultimate "Ignore Button" for your application.

1. The Concept: The "Invisibility Cloak"

As we established, WM_NCHITTEST is the gatekeeper. Before Windows sends a click, it asks your window: "Are you here?"

If you reply with **HTNOWHERE**, you are effectively telling Windows: "There is no window here. The mouse is floating in empty space."

The Result:

- Windows assumes the mouse is **not** touching your window.
- It **cancels** the generation of all subsequent messages (WM_MOUSEMOVE, WM_LBUTTONDOWN, WM_SETCURSOR).
- Your window becomes **unclickable**. The mouse events might even fall through to the window sitting behind yours (like the Desktop).

2. The Code

```
HRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_NCHITTEST:
            // Intercept the query and lie to Windows
            return HTNOWHERE;

        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

3. Why is this powerful?

This is different from just ignoring WM_LBUTTONDOWN.

Ignoring WM_LBUTTONDOWN:

The user clicks, nothing happens, but the mouse cursor still sits on top of your window.

Returning HTNOWHERE:

The mouse cursor behaves as if your window doesn't exist physically.

It allows for *click-through* overlays (like a transparent HUD in a game or a screen-tinting app), where you want the user to see your window but click on the icons behind it.

4. Selective Interaction (The "Hybrid" Approach)

The notes mention that you can modify this to allow interaction in some parts.

Scenario: A window shaped like a donut. You want clicks in the *hole* to pass through, but clicks on the *dough* to work. **Logic:**

- Get the mouse X, Y from lParam
- Check if X, Y is inside the *hole*
- If yes → return HTNOWHERE
- If no → return DefWindowProc(...) (let Windows decide normally)

Message Chaining: A Series of Events

This section illustrates the "Conversational" nature of Windows. The operating system doesn't just do things; it talks to itself (and you) at every step.

1. The Domino Effect (The Chain)

The notes use the **System Menu Icon** (the little icon in the top-left corner of the window) to demonstrate this.

The User's Action: A simple Double-Click on that icon. **The Goal:** Close the program.

Here is the invisible conversation that happens to make that work:

STEP	MESSAGE	MEANING	WHO HANDLES IT?
1	WM_NCHITTEST	"Where is the mouse?" → Answer: HTSYSMENU (On the Icon).	DefWindowProc
2	WM_NCLBUTTONDOWNDBLCLK	"Okay, the user double-clicked the System Menu."	DefWindowProc
3	WM_SYSCOMMAND	"That click translates to a command: SC_CLOSE."	DefWindowProc
4	WM_CLOSE	"The System is requesting to close. Do you object?"	YOU (The Programmer)
5	WM_DESTROY	"No objection? Okay, I am tearing down the window now."	DefWindowProc
6	WM_QUIT	"The window is gone. Time to stop the loop and exit RAM."	WinMain

2. The "Intervention" Point: WM_CLOSE

This chain is important because it reveals the perfect place to interrupt the process.

- If you intercept WM_SYSCOMMAND, you are fighting the system too early.
- If you wait for WM_DESTROY, it is too late (the window is already being dismantled).
- **WM_CLOSE** is the "Polite Request." It is the moment Windows asks: *"Are you sure?"*

3. The Code: Trapping WM_CLOSE

Your notes provided a code snippet with several typos (e.g., MB_ICONWESTION). Here is the corrected, working C code to add a "Are you sure?" popup.

How it works:

- If the user clicks "Yes": We call DestroyWindow, enabling the chain to continue to Step 5.
- If the user clicks "No": We **return 0**. We do *not* call DefWindowProc. The chain is broken, and the window stays open.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_CLOSE:
            // 1. Ask the user for confirmation
            int result = MessageBox(hwnd,
                L"Are you sure you want to close?",
                L"Confirmation",
                MB_YESNO | MB_ICONQUESTION);

            // 2. Check their answer
            if (result == IDYES)
            {
                // User said Yes: Continue the chain manually
                DestroyWindow(hwnd);
            }
            else
            {
                // User said No: Do NOTHING.
                // By returning 0 here, we stop DefWindowProc from closing the app.
                return 0;
            }
            break; // Break will fall through to default if structured differently,
                   // but usually we return 0 directly above.

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;

        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

4. Summary

- **Message Chaining** allows Windows to break complex actions into small steps.
- WM_NCHITTEST starts almost every mouse chain.
- WM_CLOSE is your opportunity to say "No" to the operating system.

Manual Hit-Testing

This section teaches you the math behind converting a mouse click (Pixel X, Y) into a logical selection (Item #5). While modern controls (like ListView) do this for you, understanding the math is essential for game development, custom UI, or grids (like Excel).

1. What is Hit-Testing?

Hit-testing is the reverse of drawing.

- **Drawing:** You have Data Item #5 \rightarrow You calculate where to put pixels on screen.
- **Hit-Testing:** User clicks pixels on screen \rightarrow You calculate which Data Item is there.

2. The Scenario: A Multi-Column File List

Imagine you are writing a custom File Explorer. You draw file names in columns to fill the screen. Now the user clicks somewhere. Which file did they click?

The Variables:

- cxColWidth: How wide one column is (in pixels).
- cyChar: How tall one line of text is (in pixels).
- iNumInCol: How many files fit vertically in one column.

3. The Math (Step-by-Step)

You need to convert the Mouse Coordinates (`cxMouse`, `cyMouse`) into an **Array Index**.

Coordinate-to-Array Conversion

A Which Column? (The X-Axis)

Divide the mouse X position by the column width. Integer division in C/Asm automatically **truncates** the remainder.

```
ColumnIndex = cxMouse / cxColWidth
```

Scenario: Columns are 100px wide, Mouse X is at 250px.

Calculation: $250 / 100 = \text{Column Index } 2$

B Which Row? (The Y-Axis)

Divide the mouse Y position by the character/text height.

```
RowIndex = cyMouse / cyChar
```

Scenario: Text is 20px high, Mouse Y is at 50px.

Calculation: $50 / 20 = \text{Row Index } 2$

C The Final Index

Combine them into a linear index to find the exact item in your buffer.

```
FinalIndex = (ColIndex * ItemsPerCol) + RowIndex
```

Result: If there are 50 items per column:

$(2 \times 50) + 2 = \text{Array Index } 102$

Note: This math assumes a 0-based index, which is standard for C arrays and pointer arithmetic.

4. The Safety Check

The math above is "blind." It will calculate an index even if the user clicks in empty white space at the bottom right of the screen.

Critical Step: You must verify the index exists.

```
if (iIndex < TotalNumberOfFiles) {  
    // Valid click! Open the file at szFileNames[iIndex]  
} else {  
    // The user clicked blank space. Ignore.  
}
```

Handling Non-List View Scenarios

Hit-testing can become more complex when dealing with graphical images or variable font sizes. It requires understanding the layout and mapping mouse coordinates to the underlying data or objects.

This code acts as the "Controller" logic. It translates a physical mouse click into a logical data change.

```
HRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        case WM_LBUTTONDOWN:
        {
            // 1. EXTRACT COORDINATES
            int cxMouse = LOWORD(lParam); // Mouse X
            int cyMouse = HIWORD(lParam); // Mouse Y

            // 2. CALCULATE GRID POSITION (The "Hit Test")
            // "cxColWidth" is the width of one column/block
            int iColumn = cxMouse / cxColWidth;

            // "cyChar" is the height of one row/block
            int iFromTop = cyMouse / cyChar;

            // 3. CALCULATE ARRAY INDEX (Linear Index)
            // Formula: (Current Column * Items Per Column) + Current Row
            int iIndex = (iColumn * iNumInCol) + iFromTop;

            // 4. VALIDATE (Safety Check)
            // Ensure the user didn't click outside the list of valid items
            if (iIndex < 0 || iIndex >= numFiles)
            {
                return 0; // Clicked on empty area; ignore.
            }

            // 5. ACTION
            // The user successfully clicked item #iIndex.
            // Example: SelectFile(iIndex);
            // Example: ToggleCheckerBoard(iColumn, iFromTop);

            return 0;
        }

        default:
        {
            return DefWindowProc(hwnd, msg, wParam, lParam);
        }
    }
    return 0;
}
```

The Logic Explained: CHECKER1

In the CHECKER1 program, this exact logic is used to manage a 5x5 grid of rectangles.
How CHECKER1 works:

The Window Setup (WM_SIZE):

- Windows tells you the window size is 500x500 pixels.
- You divide by 5: Each block is 100x100.
- You save cxBlock = 100 and cyBlock = 100.

The Click (WM_LBUTTONDOWN):

- User clicks at X=250, Y=150.
- **Math:**
 - ✓ Column = $250 / 100 = 2$ (3rd Column).
 - ✓ Row = $150 / 100 = 1$ (2nd Row).
- **Result:** You know the user clicked Grid Square [2][1].

The Action (Toggle State):

- The program has a 2D array storing "True/False" for every square.
- It flips the value: $fState[2][1] = !fState[2][1]$.

The Repaint (InvalidateRect):

- Crucially, changing the array variable **does not** update the screen.
- You must call `InvalidateRect(hwnd, ...)` to tell Windows: "*My data has changed. Please send me a WM_PAINT message so I can draw the new X.*"

Handling Complexity (Non-List Scenarios)

The notes mention "Variable Font Sizes" or "Graphical Images."

The math above (Mouse / Width) only works if every item is the exact same size (Uniform Grid).

If items are different sizes (e.g., a list of images with different heights):

- **Simple Division Fails:** You cannot just divide Y / Height.
- **The Loop Solution:** You must loop through your items, adding up their heights, until the total exceeds cyMouse.

```
int currentY = 0;
for (int i = 0; i < totalItems; i++) {
    currentY += itemHeight[i];
    if (cyMouse < currentY) {
        clickedItem = i;
        break;
}
```

The first hittest program...



Hittesting program
1-l was the one click

CHECKER2 PROGRAM LOGIC.

This version is a significant upgrade over CHECKER1. It introduces a "Keyboard Interface" for a mouse-driven program. This means you can play the game using **Arrow Keys** to move the mouse and **Spacebar** to click.

1. The Strategy: "Fake" the Mouse

In CHECKER1, everything relied on WM_LBUTTONDOWN. Instead of rewriting that logic for the keyboard, CHECKER2 simply moves the actual mouse cursor and tricks the window into thinking the mouse was clicked.

2. The Logic Steps (WM_KEYDOWN)

Step A: Where is the mouse right now?

When you press an arrow key, the program first needs to know which square you are currently hovering over.

1. **GetCursorPos:** Get the **Screen** coordinates (GPS) of the mouse.
2. **ScreenToClient:** Convert them to **Window** coordinates (Seat Number).
3. **Calculate Grid Index:** Divide by the block size ($x / cxBlock$) to get the current column/row (0 to 4).

Step B: The Move (Arrow Keys)

The switch statement modifies the grid index:

- **VK_LEFT:** $x--$
- **VK_RIGHT:** $x++$
- **VK_UP:** $y--$
- **VK_DOWN:** $y++$

Step C: The Safety Clamp (min / max)

Because it is a 5x5 grid, valid indices are **0, 1, 2, 3, 4**. If the user is at $x=4$ and presses Right, x becomes 5. This would crash or break the logic.

- **The Fix:** The code forces the numbers to stay in bounds.
 - ✓ $x = \max(0, \min(4, x))$
 - ✓ *Translation:* "If x is less than 0, make it 0. If x is more than 4, make it 4."

Step D: Moving the Physical Mouse

This is the unique part of CHECKER2. It doesn't just move an imaginary highlight; it snaps the actual white arrow cursor to the **center** of the new square.

1. **Calculate Center:** $\text{pixelX} = (x * \text{width}) + (\text{width} / 2)$
2. **ClientToScreen:** Convert that pixel back to Global Screen coordinates.
3. **SetCursorPos:** Teleport the mouse to that exact pixel.

Step E: The "Fake" Click (Spacebar/Enter)

If the user presses **Space** or **Enter**, the program reuses the mouse logic.

- **SendMessage:** This function calls your own Window Procedure.
- **The Message:** It sends WM_LBUTTONDOWN to itself at the current position.
- **Result:** The code you wrote for CHECKER1 runs automatically, toggling the X.

3. The Code Implementation

Here is how that logic looks in C:

```
case WM_KEYDOWN:
    point.x = x; // Current column
    point.y = y; // Current row

    switch (wParam)
    {
        case VK_UP:   y--; break;
        case VK_DOWN: y++; break;
        case VK_LEFT: x--; break;
        case VK_RIGHT: x++; break;

        case VK_HOME: x = y = 0; break; // Bonus: Go to top-left

        case VK_RETURN: // Enter key
        case VK_SPACE: // Spacebar
            // TRICK: Send a "fake" mouse click message to ourselves
            SendMessage(hwnd, WM_LBUTTONDOWN, MK_LBUTTON,
                        MAKELONG(x * cxBlock, y * cyBlock));
            return 0;
    }

    // 1. Clamp the values so we don't go off the board
    x = max(0, min(4, x));
    y = max(0, min(4, y));

    // 2. Calculate the pixel center of the new square
    point.x = (x * cxBlock) + (cxBlock / 2);
    point.y = (y * cyBlock) + (cyBlock / 2);

    // 3. Convert to Screen Coordinates (Global GPS)
    ClientToScreen(hwnd, &point);

    // 4. Teleport the actual mouse cursor
    SetCursorPos(point.x, point.y);
    return 0;
```

Summary of CHECKER2

- It demonstrates **Coordinate Conversion** (ScreenToClient vs ClientToScreen).
- It shows how to **control the system cursor** (SetCursorPos).
- It teaches **Code Reuse** via SendMessage (using the keyboard to trigger mouse events).

EXPLANATION OF CHECKER3 AND THE CONCEPT OF CHILD WINDOWS

This marks a major shift in how you structure a Windows program. In CHECKER1 and CHECKER2, you had to do all the math yourself. In CHECKER3, you let Windows do the heavy lifting by chopping your application into pieces.

1. The "Paint" Analogy

Think about Microsoft Paint. It has a drawing canvas, a toolbar on the left, and a color palette at the bottom.

- **Without Child Windows:** The programmer has to say, "If the user clicks between pixel 0 and 50 on the X-axis, they clicked a tool."
- **With Child Windows:** The toolbar is its own distinct window. The canvas is a separate window.
 - ✓ If the user clicks the toolbar, the **Toolbar Window** receives the message.
 - ✓ If the user clicks the canvas, the **Canvas Window** receives the message.
 - ✓ **No coordinate math is required.**

2. How CHECKER3 Works: Divide and Conquer

Instead of drawing a grid of lines on one big window, CHECKER3 creates **25 separate small windows** arranged in a 5x5 grid.

A. The Hierarchy

The Parent: The main application window. Its only job is to hold the children.

The Children: 25 tiny windows.

- They have no title bars or borders (they look like simple rectangles).
- They sit inside the Parent's Client Area.

B. Automatic Hit-Testing

This is the biggest advantage.

- **In CHECKER1:** You had to calculate Column = MouseX / Width.
- **In CHECKER3:** You don't calculate anything.
 - ✓ If the user clicks the square at Row 2, Column 3, **only that specific Child Window** receives the WM_LBUTTONDOWN message.
 - ✓ The Child Window just says: "Ouch! I was clicked. I will draw an X on myself now."

C. Relative Coordinates (IParam)

This is a critical technical detail.

- **In Parent Window:** (0,0) is the top-left of the main application.
- **In Child Window:** (0,0) is the top-left of **that specific child window**. This means drawing code is simple. You always draw the 'X' at the center of *yourself*, regardless of where you are on the screen.

3. Advantages of this Approach

FEATURE	SINGLE WINDOW (CHECKER1)	CHILD WINDOWS (CHECKER3)
Hit Testing	Complex Math (Division & Remainder checks)	Automatic (<i>Windows identifies which child was clicked</i>)
Drawing	Complex Loop (Manually draw all 25 squares)	Simple (<i>Each child window paints only itself</i>)
Code Structure	Monolithic (One massive WndProc)	Modular (<i>Clean, separate window procedures</i>)

4. CHECKER3 Specifics

Creation: It uses a loop to call CreateWindow 25 times.

The Missing Piece: The notes mention it **lacks a Keyboard Interface**.

Why? Because keyboard focus is tricky with child windows. When you press a key, *which* of the 25 windows gets the message? CHECKER3 doesn't solve this yet (that is for CHECKER4).

CHECKER3: DUAL WINDOW PROCEDURES AND CLASS REGISTRATION.

This section gets technical. It explains *how* you actually create 25 separate little windows inside one big one. The secret lies in registering **two** different types of window classes in your WinMain function.

1. The Tale of Two Procedures

In previous programs (like CHECKER1), you only had one WndProc that did everything. In CHECKER3, you split the work:

WndProc (The Boss):

- Manages the Main Window.
- Handles resizing the grid.
- **Does NOT handle mouse clicks.** (It delegates this to the children).

ChildWndProc (The Worker):

- Manages *one specific square* of the grid.
- **Handles WM_LBUTTONDOWN:** "I was clicked! I'll toggle my state".
- **Handles WM_PAINT:** "I need to draw an X on myself".
- *Note:* All 25 child windows share this **same** single procedure code, but they have different data (separate handles).

2. Registering the Classes

Before you can create these windows, you must register their "blueprints" ie. Window Classes.

a) The Shortcut

Instead of writing two massive structs from scratch, the code is lazy (efficient). It fills out the wndclass struct for the Main Window first, registers it, and then just **tweaks 4 fields** to recycle it for the Child Window.

FIELD CHANGED	NEW VALUE	REASON
<code>lpfnWndProc</code>	<code>ChildWndProc</code>	<i>The child needs its own logic (e.g., clicking toggles an 'X').</i>
<code>cbWndExtra</code>	<code>sizeof(long)</code> (4 bytes)	Crucial. <i>Each child needs 4 bytes of "Personal Memory" to remember its state (0 or 1).</i>
<code>hIcon</code>	<code>NULL</code>	<i>Child windows don't appear in the Taskbar; they don't need icons.</i>
<code>lpszClassName</code>	<code>"Checker3_Child"</code>	<i>Needs a unique name to identify this specific type of child.</i>

3. The CreateWindow Call (The Table)

Your provided table highlights the differences when actually *spawning* the windows. Key Differences to Note:

Style:

Main: `WS_OVERLAPPEDWINDOW` (Standard window with borders/title).

Child: `WS_CHILDWINDOW`. (Must be a child; has no border).

Parent Handle:

Main: `NULL` (It has no boss).

Child: `hwnd` (The Main Window is its boss).

Menu / ID:

Main: `NULL` (Uses the standard menu).

Child: (`HMENU`) ($y << 8$). **This is a clever trick.** Instead of a Menu Handle, we store the Child's ID here. By combining x and y into a single number, we give each child a unique ID (like "Row 2, Col 3").

4. Message Handling in ChildWndProc

The child logic is incredibly simple compared to CHECKER1.

WM_CREATE: Set my personal memory (State) to 0.

WM_LBUTTONDOWN:

- Get my current State (0 or 1).
- Flip it (1 or 0).
- Save it back to cbWndExtra.
- Call InvalidateRect (trigger a repaint).

WM_PAINT:

- Check my State.
- If 1: Draw the X.
- If 0: Draw nothing.

Note: No coordinate math needed! Drawing from (0,0) to (width, height) works perfectly because (0,0) is the top-left of *this specific square*.

Argument	Main Window	Child Window
Window class	"Checker3"	"Checker3_Child"
Window caption	"Checker3"	NULL
Window style	WS_OVERLAPPEDWINDOW	WS_CHILDWINDOW
Horizontal position	CW_USEDEFAULT	0
Vertical position	CW_USEDEFAULT	0
Width	CW_USEDEFAULT	0
Height	CW_USEDEFAULT	0
Parent window handle	NULL	hwnd
Menu handle/child ID	NULL	(HMENU) (y << 8)
Instance handle	hInstance	(HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE)
Extra parameters	NULL	NULL

5. Dynamic Sizing (The WM_SIZE Strategy)

In the CreateWindow calls for the child windows, the width, height, X, and Y were all set to 0.

Why? Because when the program first starts, you don't yet know how big the user's screen or the main window will be.

The Fix: The main window waits for the WM_SIZE message (which is sent immediately after creation).

- It calculates: $\text{blockWidth} = \text{clientWidth} / 5$
- It loops through all 25 child windows.
- It calls MoveWindow on each one to snap it into the correct grid position.

Benefit: If the user drags the corner to resize the app, WM_SIZE runs again and the grid automatically stretches to fit the new window size.

6. The "ID Card" (Composite ID)

How does the main window talk to a specific square?

It uses a **child ID**.

Instead of naming them "Window1", "Window2", etc., the program uses math to pack the square's grid coordinates into a single number (the ID).

The Formula:

$$\text{ID} = (\text{y} \ll 8) | \text{x}$$

- Take the row number (y)
- Shift it left by 8 bits
- Combine it with the column number (x)

The Result:

- Row 0, Column 2 → ID = 2
- Row 1, Column 0 → ID = 256

Usage:

When a child window sends a message, the main window reads the ID and instantly knows:
"Ah, that came from Row 1, Column 0."

7. “Backpack” Memory (State Tracking)

This is the most crucial part:

How does a specific square remember whether it has an “X” drawn on it?

The Old Way (CHECKER1):

A global 2D array:

```
int state[5][5];
```

The CHECKER3 Way:

Window extra bytes (cbWndExtra).

When the window class is registered, it requests 4 bytes of extra memory:

```
cbWndExtra = sizeof(long)
```

Think of this as a tiny “**backpack**” attached to every single child window.

To Save State:

```
SetWindowLong(hwnd, 0, 1)
```

(Put a “1” in the backpack.)

To Read State:

```
GetWindowLong(hwnd, 0)
```

(Check what’s stored in the backpack.)

Benefit:

- No global arrays
- Each window is completely self-contained
- If a window is destroyed, its memory disappears with it

This concludes **CHECKER3**.

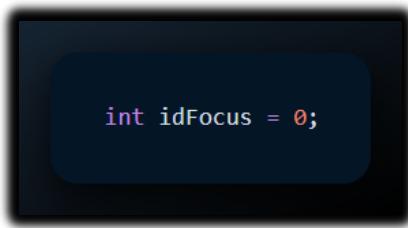
The only flaw in CHECKER3 is that, because there are 25 separate windows, the keyboard doesn’t know which one to “talk” to — resulting in **focus issues**.

CHECKER4

This program solves the "Focus Problem." In CHECKER3, you couldn't use the keyboard because Windows didn't know which of the 25 little squares should receive the keystrokes. CHECKER4 fixes this by manually managing the **Input Focus**.

1. The Global Tracker: idFocus

The program introduces a global integer variable:



This acts as the memory for the system. It stores the **ID** of the specific child window (square) that currently has the "talking stick" (Input Focus).

- If idFocus is 0, the top-left square is active.
- If idFocus is 12, the middle square is active.

2. The "Relay" Logic (Parent Window)

The Main Window now acts like a traffic controller for focus.

WM_SETFOCUS: When the user clicks the Main Window title bar to activate the app, the Main Window receives this message.

- *Action:* It immediately calls SetFocus(GetDlgItem(hwnd, idFocus)).
- *Translation:* "Don't look at me! Give the focus to the child square we last used."

WM_KILLFOCUS: When the user Alt-Tabs away.

- *Action:* It triggers a repaint so the "Focus Rectangle" (the little dotted line highlighting the active square) disappears.

3. The Child Window Logic (ChildWndProc)

The child windows now have to handle keyboard inputs intelligently.

A. Handling Keys (WM_KEYDOWN)

When a key is pressed, the *Focused Child* gets the message. It decides what to do:

1. **Action Keys (Space / Enter):**

- ✓ "This is for me."
- ✓ Toggle the X mark (State 0 \rightarrow 1).
- ✓ Redraw myself.

2. **Navigation Keys (Arrows):**

- ✓ "I don't know who my neighbor is. The Parent knows the grid layout."
- ✓ **Pass the Buck:** It sends the message up to the Parent window.
- ✓ *Code:* `SendMessage(GetParent(hwnd), msg, wParam, lParam);`

B. Handling Clicks (WM_LBUTTONDOWN)

When you click a square with the mouse:

1. **Toggle X:** Draw/Undraw the checkmark.
2. **Steal Focus:** It calls `SetFocus(hwnd)` to tell Windows: "Make ME the active keyboard target now."
3. **Update Global:** It updates `idFocus` to its own ID so the Parent remembers.

4. The Parent's Navigation Logic

(This is implied in the "Pass to Parent" step).

When the Parent receives the forwarded Arrow Key message:

1. It checks the current idFocus.
2. **Math:**
 - **Left:** idFocus--
 - **Right:** idFocus++
 - **Up:** idFocus -= 5 (Move back one row).
 - **Down:** idFocus += 5 (Move forward one row).
3. **Execute:** It calls SetFocus on the new ID.

5. Summary of CHECKER4 Features

FEATURE	DESCRIPTION
<code>idFocus</code>	A variable that remembers which square (ID) is currently active or selected.
<code>GetDlgItem</code>	Converts an ID (integer) back into a Window Handle (<code>HWND</code>) so you can send messages to it.
<code>WM_SETFOCUS</code>	A message handler that ensures focus always lands on a child window , never on the empty parent background.
<code>GetDlgCtrlID</code>	Allows a child window to ask the system: "What is my own unique ID number?"

Chapter 7: The Evolution of Mouse & Input

You have just walked through the complete evolution of Windows Input logic:

1. **CONNECT:** Basic **Mouse Tracking** (drawing lines).
2. **CHECKER1:** Basic **Hit-Testing** (Math to find grid squares).
3. **CHECKER2:** Adding **Keyboard** support to a single-window app (Cursor simulation).
4. **CHECKER3:** Using **Child Windows** to automate Hit-Testing (No math needed).
5. **CHECKER4:** Managing **Focus** across multiple Child Windows for full keyboard/mouse support.

6. CHECKER4: The Visuals of Focus

We established that CHECKER4 uses a global variable idFocus to track which square is active. But how does the *user* know?

A. The "Focus Rectangle" (Dotted Line)

In Windows, the active button or item usually has a tiny dotted line around its text. CHECKER4 manually draws this.

- **WM_SETFOCUS:**
 - ✓ The child wakes up.
 - ✓ It updates the global idFocus = MyID.
 - ✓ It repaints itself.
- **WM_PAINT:**
 - ✓ Inside the painting logic, it checks: if (hwnd == GetFocus()).
 - ✓ If true, it selects a **PS_DASH** (Dashed Style) pen and draws a rectangle around its border.
- **WM_KILLFOCUS:**
 - ✓ The child goes to sleep.
 - ✓ It repaints itself (this time *without* the dashed line).

B. The "Tag Team" Keyboard Logic

This section confirms the parent-child relationship for keys.

- **Child's Job:** Handle "Action" keys (Space/Enter). If it sees an Arrow key, it says "I don't know who is next to me," and sends the message to the Parent.
- **Parent's Job:** It receives the passed Arrow key message. It does the grid math (e.g., currentX + 1) and moves the focus to the new neighbor.

BLOKOUT1: CAPTURING THE MOUSE

This program solves a specific problem in UI design: **Dragging**. When you click and drag to draw a box (like selecting icons on your desktop), what happens if your mouse accidentally slips *outside* the window while you are still holding the button?

- **Normal Behavior:** The window stops receiving WM_MOUSEMOVE messages because the mouse is no longer inside it.
- **The Solution: Mouse Capture** (SetCapture). This forces Windows to send *all* mouse messages to your window, even if the mouse is floating over the Start Button or another app, until you release the button.

A. The "Rubber Band" Effect

BLOKOUT1 draws a rectangle that expands and contracts as you move the mouse. This is often called "Rubber Banding."

The Drawing Logic (WM_PAINT vs. WM_MOUSEMOVE):

While Dragging (WM_MOUSEMOVE):

- You don't want to fill the rectangle with color yet (too slow/messy).
- You only draw the **Outline**.
- **The Trick (SetROP2):** The notes mention SetROP2 (Raster Operation). This usually sets the drawing mode to **XOR** (Exclusive OR).
 - ✓ *Draw Line Once:* Line appears.
 - ✓ *Draw Line Again:* Line disappears (it inverts the pixels back).
- This allows the program to "erase" the old rectangle outline just by drawing over it again, without needing to redraw the entire background.

B. The Timeline of Events

MESSAGE	ACTION
WM_LBUTTONDOWN	Start. Save start coordinate (<code>ptBeg</code>). Change cursor to Crosshair. Capture Mouse (<code>SetCapture</code>).
WM_MOUSEMOVE	Update. (If button is held): Erase old outline → Update <code>ptEnd</code> → Draw new outline.
WM_LBUTTONUP	Finish. Save final coordinate. Release Mouse Capture (<code>ReleaseCapture</code>). Reset Cursor. <code>InvalidateRect</code> (Trigger a full real paint).
WM_PAINT	Permanent. If finalized, draw a real, filled rectangle using the <code>Rectangle</code> function.
WM_CHAR (Esc)	Abort. If the user hits Escape while dragging, cancel the capture and erase the temporary outline.

C. Summary

- **CHECKER4** taught us how to manage **Focus** manually using WM_SETFOCUS and WM_KILLFOCUS to draw dotted lines.
- **BLOKOUT1** introduces **Mouse Capture**, which ensures you don't lose control of a drag operation if the user moves the mouse too fast or off-screen. It also uses **ROP2 (XOR) drawing** for temporary "rubber band" lines.



BlockOut 1.mp4

Mouse Capturing and the difference between BLOKOUT1 and BLOKOUT2.

This section addresses a specific frustration in UI programming: What happens when the user drags the mouse too fast and the cursor flies off the edge of your window?

1. The Problem: "The Leash Breaks"

In BLOKOUT1 (the previous version), if you clicked inside the window to start drawing a rectangle, held the button down, and moved the mouse **outside** the window border:

- Windows stopped sending WM_MOUSEMOVE messages to your program (because the mouse was no longer over it).
- The rectangle stopped updating.
- If you released the button outside, your program never received the WM_LBUTTONDOWN message. It would get "stuck" thinking the button was still down.

2. The Solution: SetCapture (The Leash)

Mouse Capturing is the ability of a window to "claim" the mouse input, regardless of where the cursor physically is on the screen.

- **To Activate:** Call SetCapture(hwnd) (usually in WM_LBUTTONDOWN).
- **The Effect:** Windows will redirect **all** mouse messages to your window procedure, even if the user is hovering over the Start Button or a different application.
- **To Deactivate:** Call ReleaseCapture() (usually in WM_LBUTTONUP).

3. Coordinate Behavior ("Negative Coordinates")

When you capture the mouse, the coordinate system behaves interestingly.

- Normally, (0,0) is the top-left of your client area.
- With Capture, if the user moves the mouse to the left of your window, you will receive **negative coordinates** (e.g., x = -50).
- If they move above the window, y will be negative.
- **Why this matters:** Your drawing logic usually needs to handle this (e.g., using $\max(0, x)$) so you don't try to draw rectangles in "negative space" which might look weird or crash.

4. The 32-Bit Safety Valve

In the old days (16-bit Windows), a buggy program could capture the mouse and never release it, effectively freezing the computer because you couldn't click anything else. In modern (32-bit/64-bit) Windows, there is a safety rule:

- **The Rule:** You can generally only keep the capture **while a mouse button is held down.**
- If the button is up and you try to keep capture while moving over another window, Windows might force-break the capture to let the other window receive input.

5. BLOKOUT2 vs. BLOKOUT1

BLOKOUT2 is the "Corrected" version of the rectangle drawing program.

FEATURE	BLOKOUT1 (LOCAL)	BLOKOUT2 (GLOBAL CAPTURE)
Start Drag	Sets <code>fBlocking = TRUE</code>	Calls <code>SetCapture(hwnd)</code>
During Drag	<i>Stops working if mouse leaves window</i>	Works everywhere (Screen-wide tracking)
End Drag	Sets <code>fBlocking = FALSE</code>	Calls <code>ReleaseCapture()</code>
Abort (Esc)	Just resets variables	Calls <code>ReleaseCapture()</code> to clean up OS state

The Golden Rule of Capture:

- "If you SetCapture on WM_LBUTTONDOWN, you **must** ReleaseCapture on WM_LBUTTONUP."
- If you enlarge the window after capturing the mouse, the full rectangle you drew becomes visible.
- Mouse capture isn't just for unusual or special-case applications—it should be used anytime you need reliable mouse tracking after a button press.
- Handling mouse capture correctly leads to simpler code and behavior that matches what users naturally expect.



BlockOut 2.mp4

Mouse Wheel Support

This is the final piece of the mouse puzzle. Unlike clicking (which is a simple Yes/No switch), the mouse wheel is an *analog* control—it measures "how much" and "how fast."

1. The Core Concept: The "Click" vs. The "Flow"

In the early days, mouse wheels physically clicked as you turned them. Each "click" sent a signal. Modern mice (and trackpads) are smooth. They don't just send "Scroll Down"; they send "Scroll Down by 3.5 units."

Windows handles this via the **WM_MOUSEWHEEL** message.

2. The Magic Number: 120 (WHEEL_DELTA)

When you receive WM_MOUSEWHEEL, the most important data is hidden in the wParam. This number is called the **Delta**.

- **Why 120?** Microsoft chose 120 as the "Standard Unit" because it is divisible by many numbers ($120 / 3 = 40$, $120 / 4 = 30$, etc.).
- **Standard Mouse:** One physical "click" of the wheel sends a Delta of **120**.
- **High-Precision Mouse:** Might send smaller Deltas like **40** or **30** more frequently for smoother feeling.

3. The "Piggy Bank" Logic (Accumulation)

This is the smartest part of the SYSMETS code. You don't just scroll every time you get a message. You **accumulate** the data.

The Scenario: Imagine a high-precision mouse sends three rapid messages with a Delta of 40 each.

- *Bad Logic:* Scroll immediately on 40? No, that's not enough for a full line.
- *Good Logic (Accumulation):*
 1. Message 1 (Delta 40): Save it. Total = 40. (Do nothing).
 2. Message 2 (Delta 40): Save it. Total = 80. (Do nothing).
 3. Message 3 (Delta 40): Save it. Total = 120. **Action!** That equals one standard "Unit." Scroll now!

The Variable: iAccumDelta is the variable acting as the "Piggy Bank." It collects small movements until they add up to enough force to push the scrollbar.

4. Respecting User Settings (SystemParametersInfo)

You cannot hard-code "Scroll 3 lines." Some users set their Windows settings to scroll one page at a time; others want 1 line at a time.

- **Function:** SystemParametersInfo(SPI_GETWHEELSCROLLLINES, ...)
- **Purpose:** It asks Windows, "Hey, when the user rotates the wheel one 'click' (120 delta), how many text lines should I move?" **Calculation:**

$$\text{DeltaPerLine} = \frac{120}{\text{User's Preferred Lines}}$$

5. Summary of the Flow

Event: User rolls wheel.

Message: WM_MOUSEWHEEL arrives with a value (e.g., 120).

Math: Add value to iAccumDelta.

Check: Is iAccumDelta big enough to cover a line?

Action: If yes, send WM_VSCROLL and subtract the used amount from the piggy bank.

You have now completed the entire Mouse Input chapter! Here is the roadmap:

1. **Basics:** WM_LBUTTONDOWN and Client vs. Screen coordinates.
2. **Hit-Testing:** The math to find *what* was clicked (CHECKER1).
3. **Child Windows:** Using windows as buttons (CHECKER3).
4. **Input Focus:** Handling Keyboard + Mouse together (CHECKER4).
5. **Capture:** dragging without losing control (BLOKOUT).
6. **The Wheel:** Handling smooth scrolling (SYSMETS).

The creation of customized mouse cursors is covered in Chapter 10 along with other Windows resources. The video...



Sysmets with
mousewheel.mp4