

CHAPTER 4: DEALING WITH TEXT OUTPUT

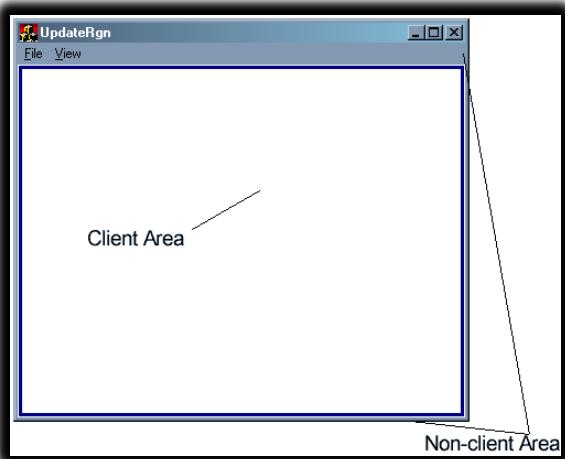
In Windows programming, the **client area** is the part of the application window that is not taken up by the title bar, window-sizing border, and other elements.



What is the Client Area?

The **client area** is the part of the window where your app does the actual drawing — think of it as your painting canvas.

It's everything **except** the title bar, borders, and scroll bars.



What is Painting?

The process of displaying text or graphics in a Windows program's client area is called "**painting**".

We don't draw whenever we want — Windows controls **when** we're allowed to paint, by sending us a message: [WM_PAINT](#).

Windows programs must be able to handle client areas of varying sizes, from very small to very large.

Windows provides a [Graphics Device Interface \(GDI\)](#) for painting, but in this chapter, we will focus on displaying simple lines of text.

Windows programs should use the [system font as the default font](#), as this ensures consistent appearance across different systems.

```
case WM_PAINT:  
{  
    PAINTSTRUCT ps; // Struct to hold paint info.  
  
    // Start painting: Gets a device context (HDC) for drawing.  
    // This HDC is specific to the current paint operation.  
    HDC hdc = BeginPaint(hwnd, &ps);  
  
    // --- Your Drawing Commands Go Here ---  
    // Example: Display "Hello WinAPI!" text at (10, 10).  
    TextOut(hdc, 10, 10, TEXT("Hello WinAPI!"), 13);  
  
    // End painting: Releases the HDC and validates the update region.  
    // Crucial for telling Windows you're done and preventing endless WM_PAINTs.  
    EndPaint(hwnd, &ps);  
}  
break;
```

HDC (Device Context): Think of this as your canvas. All GDI (Graphics Device Interface) drawing functions need an HDC to know where to draw.

- Always use BeginPaint() / EndPaint() inside WM_PAINT.

Device-Independent Programming

Device-independent programming is the practice of writing software that can run on a variety of hardware and software configurations. Different screens, resolutions, DPI settings... your code needs to adapt.

Tips:

- Use system font (`GetStockObject(SYSTEM_FONT)`)
- Use window metrics (`GetClientRect`, `GetDeviceCaps`) to stay responsive
- Avoid hardcoded positions and sizes

The **size of the client area** can change at any time, so Windows programs need to be able to react to these changes.

Windows programs can use a **variety of techniques** to make their output look good on different screen sizes.

In Windows, programs can only draw text and graphics in the client area of their window.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            // Paint the client area of the window here
            EndPaint(hwnd, &ps);
            return 0;
        }
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
}
```

◆ HWND hwnd

Handle to the window that got the message. Think of this as the *unique ID* for your window.

◆ UINT message

Tells you *what happened*. Could be:

- WM_PAINT → Time to redraw
- WM_KEYDOWN → Key pressed
- WM_MOUSEMOVE → Mouse moved

◆ WPARAM wParam & LPARAM lParam

Extra info, depending on the message:

- If it's a keypress, wParam might be the key code.
 - If it's mouse movement, lParam might pack the (x, y) coords.
-



Message Handling Logic

You handle these messages using a switch (message) block:

```
switch (message)
{
    // You can comment out this part because PlaySound requires the winmm.lib library,
    // which is not linked by default in most Windows projects.
    // Unlike kernel32.lib, user32.lib, and gdi32.lib (which are included automatically),
    // winmm.lib is an external multimedia library that handles sounds and music.
    // If you don't link winmm.lib, calling PlaySound will cause linker errors.
    // So, unless you explicitly link winmm.lib in your project settings,
    // it's safer to comment out PlaySound to avoid build problems.
    case WM_CREATE:
        //PlaySound(TEXT("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC);
        return 0;

    case WM_PAINT:
    {
        PAINTSTRUCT ps; // Struct to hold paint info.

        // Start painting: Gets a device context (HDC) for drawing.
        // This HDC is specific to the current paint operation.
        HDC hdc = BeginPaint(hwnd, &ps);

        // --- Your Drawing Commands Go Here ---
        // Example: Display "Hello WinAPI!" text at (10, 10).
        TextOut(hdc, 10, 10, TEXT("Hello WinAPI!"), 13);

        // End painting: Releases the HDC and validates the update region.
        // Crucial for telling Windows you're done and preventing endless WM_PAINTs.
        EndPaint(hwnd, &ps);
        return 0;
    }

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
}
```

WM_PAINT – Drawing Time

When your window needs to update its client area (the drawable part), Windows sends WM_PAINT.

What Triggers WM_PAINT?

- Window is resized.
- A covered section becomes visible.
- A tooltip disappears.
- You manually request repainting (InvalidateRect).

The Painting Steps (explained line-by-line)

PAINTSTRUCT ps;

This holds data about the area that needs to be redrawn.
Windows fills this when you call BeginPaint().

BeginPaint(hwnd, &ps);

Starts the painting.
Returns an HDC (Handle to Device Context) — this is your *drawing tool*.
You'll use it to draw text, shapes, etc.

Draw Your Content

Between BeginPaint() and EndPaint(), do things like:

```
TextOut(hdc, 20, 20, TEXT("Hello World!"), 12);
```

EndPaint(hwnd, &ps);

Always call this when you're done.
It releases the HDC and tells Windows: "I'm done painting."



DefWindowProc – The Default Handler

Not every message is one **you** need to handle yourself.

If your window gets a message you **don't explicitly process**, you need to *pass it along* to DefWindowProc(). This function is built into Windows — it knows how to handle the boring but critical stuff like:

- Moving the window around.
- Resizing.
- Clicking the close ✘ button.
- Minimizing/maximizing.
- System commands (like Alt+F4).

If you don't call it for unhandled messages, weird stuff happens — your window might stop behaving like a normal window. It won't close right, it won't respond to default key combos, and users will think it's broken 🤦.

🧠 What You're Passing In:

```
return DefWindowProc(hwnd, message, wParam, lParam);
```

This ensures default behavior (like dragging, closing, etc.) still works.

- **hwnd:** Which window this is about
- **message:** The type of message (e.g. WM_CLOSE, WM_MOVE, etc.)
- **wParam / lParam:** Any extra info tied to the message — depends on the message type.

Basically, you're saying:

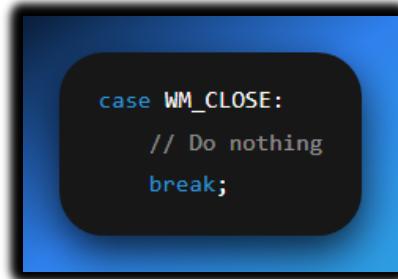
"Hey Windows, I don't know what to do with this message, so *you* deal with it."

Real Example

1. User clicks the **X (close)** button on your app window.
 2. Windows **sends a WM_CLOSE message** to your window procedure (WndProc).
 3. Now you have two choices in your WndProc function:
 - ✓ **Handle WM_CLOSE yourself** (maybe ask "Are you sure?" with a message box, or save work).
 - ✓ **Or if you don't care to do anything special, pass it to DefWindowProc.**
-

What happens if you do *not* pass WM_CLOSE to DefWindowProc?

If your WndProc looks like this:



```
case WM_CLOSE:  
    // Do nothing  
    break;
```

Then Windows will NOT close your window. Your app stays open, but you can't interact with it. It's now a **zombie window** — looks alive, but it doesn't respond or close.

Correct Way (if you don't want to do anything special):



```
case WM_CLOSE:  
    DefWindowProc(hWnd, message, wParam, lParam);  
    break;
```

Or, more commonly, let it fall through to the default handler:

```
default:  
    return DefWindowProc(hWnd, message, wParam, lParam);
```

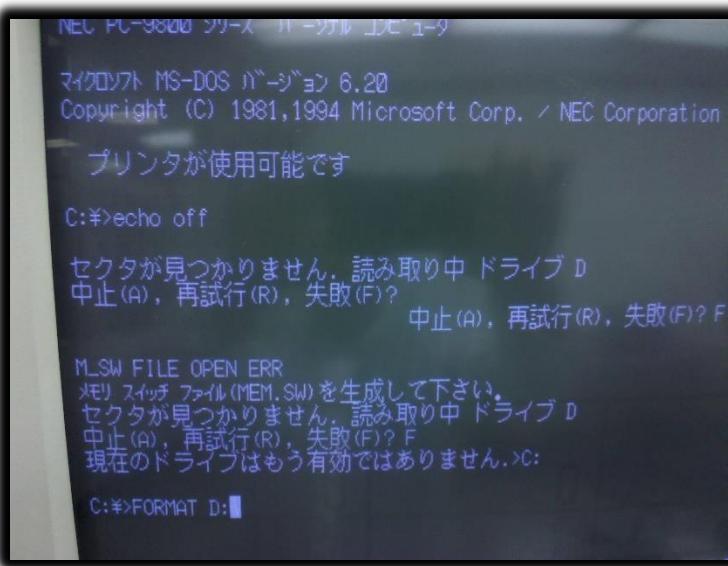
This ensures Windows does its job: destroying the window, sending WM_DESTROY, and eventually posting a WM_QUIT to end the message loop.

Bottom Line

Yes — if you don't handle WM_CLOSE, and you don't pass it to DefWindowProc, your window won't close. Always call DefWindowProc for messages you don't explicitly handle.

CLIENT AREA PAINTING

Back in the day — in good old DOS or other character-mode environments — you could draw *anywhere* on the screen.



The entire display was yours: a giant grid of characters or pixels. You could scribble on it, and whatever you wrote stayed put until you changed it.

Total control. No rules.

In traditional character-based environments, programs had direct control over the entire video display.

They could write text and graphics anywhere on the screen, and the modifications will remain visible until [explicitly overwritten](#).

This [simplicity](#) allows programs to manage the screen contents without worrying about external factors.

Windows is much complex, so programs can only draw on the client area of their own [window](#), a designated rectangular region within the overall window frame.

This restriction is primarily due to the [multitasking](#) nature of Windows, where multiple programs share the screen space.



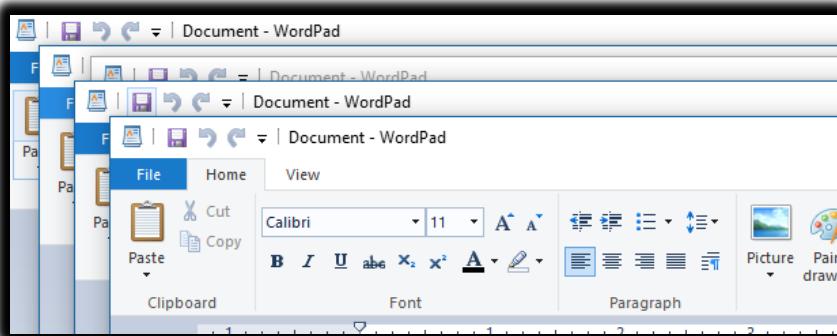
Additionally, the content of the client area is not guaranteed to persist indefinitely.

There are several scenarios where the client area may need to be repainted:

Revealing Previously Hidden Areas: When a hidden portion of the window is brought into view, either by moving the window or uncovering it from behind another window, Windows will send a WM_PAINT message to the window procedure. This message signals the need to redraw the exposed area.

Windows tells your program:

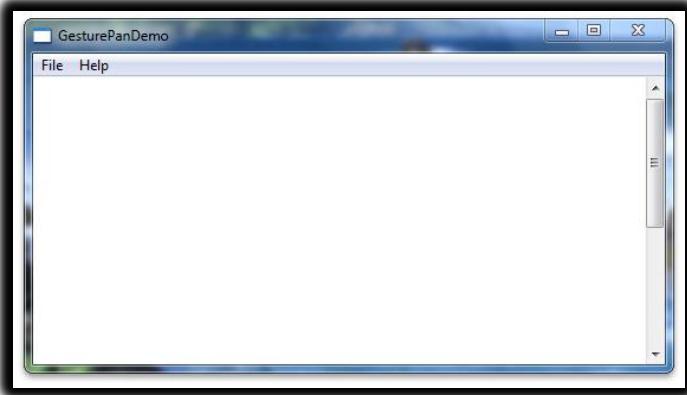
"Hey, that section was in the shadows, now it's in the light — redraw it."



Window Resizing: If the user resizes the window, and the window class style has the CS_HREDRAW and CW_VREDRAW bits set, Windows will again send a WM_PAINT message. This ensures that the client area adapts to the new window size.



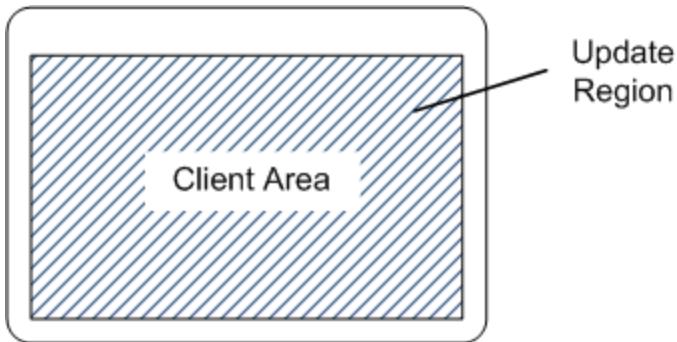
Scrolling: When a program uses the ScrollWindow or ScrollDC functions to scroll part of the client area, Windows will generate a WM_PAINT message to update the visible portion of the client area.



Explicit Invalidations: Programs can explicitly request a repaint of specific areas of the client area using the InvalidateRect or InvalidateRgn functions.

- Use InvalidateRect(hwnd, NULL, TRUE) — to redraw everything
- Or use InvalidateRgn() for custom shapes

This is great when your internal data changes and the screen needs to match.



Temporary Overwriting: In some cases, Windows may attempt to save and restore an area of the display when it is temporarily overwritten, such as when a [dialog box](#) or a [popup menu](#) is displayed over the client area.

However, this process is not always reliable, and Windows may sometimes send a WM_PAINT message even when the client area was not actually altered.

In short, paint [repaint the area after the overlay is gone](#), restore the visuals beneath, just to make sure.

Mouse Cursor and Icon Dragging:

This is one of the rare "**clean**" **overwrite cases**.

When the mouse or an icon drags over your client area:

- ⌚ Windows **always saves and restores** the bits underneath — *no repaint needed*.
- 🧠 Unless something actually changed, no WM_PAINT is fired. Efficient and smart.



⚠️ Final Pro Tip

Always assume a WM_PAINT could hit **any time**.

Be ready to redraw *everything needed*, even if you just painted it a second ago.

That's the **Windows way**.

DEALING WITH WM_PAINT MESSAGES

🎯 Understanding How to Handle WM_PAINT Properly

🧠 It's Not About Drawing Whenever You Want

In older systems (think DOS or bare metal), if you wanted something on screen — you drew it *right then and there*.

✍️ You directly touched the screen memory and said:

“Put this text here. Draw that shape there.”

Done.

But in **Windows**, you **don't control the screen** like that anymore.

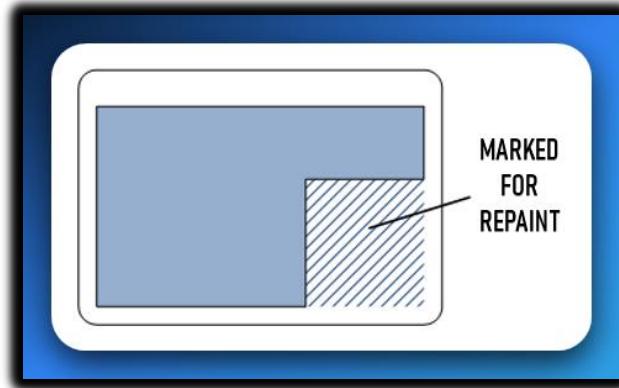
You're playing in a **multitasking, window-sharing, constantly redrawing OS**.

🎬 In Windows: You Wait For the WM_PAINT Signal

When your app wants to update the visuals:

✗ **Don't draw right away.**

✓ **Instead, mark the area as needing a repaint.**



⌚ Then, when Windows sends a WM_PAINT message, *that's your green light*. That's when you actually do the drawing.

Real-World:

Think of it like a **traffic system**. When cars approach an **intersection** (like your application windows), they wait for the green light (the WM_PAINT message). The green light signals that it's their turn to move.

You don't let each car move whenever it feels like it—you wait for a **coordinated signal** (like the WM_PAINT message), and only then do you allow all the cars (representing drawing operations) to proceed in an organized, synchronized way.

Just like traffic lights control the flow of cars, **WM_PAINT** controls the flow of painting operations.

Windows doesn't start rendering (or painting) until it's **explicitly told** to do so, ensuring everything is done at the right time and in the right order.

This keeps the process smooth and avoids chaos, making sure that multiple requests don't interfere with each other.

Why This Weird Flow?

Because Windows is juggling tons of things:

- Other apps
- Minimizing and restoring windows
- Moving stuff around
- Tooltips, dialogs, drag-drops — all fighting for screen space

So, it's **Windows** that controls **when** something is drawn.

You just tell it:

"Hey, I changed something—next time you're redrawing, include this region I've marked for repaint," says the program.

And how fast does Windows handle these repaints?

"Fast enough that we hardly even notice them. It's so seamless that the updates often feel instant, and we're rarely aware of the precise moment the screen refreshes."

Do all Windows apps, even those written in other languages and frameworks like Qt, C#, Tkinter, and Electron, rely on WinAPI concepts, just in a higher-level way?

Yes! All Windows apps, whether using low-level WinAPI (like in Charles Petzold's examples) or high-level frameworks like C#, Qt, Tkinter, or Electron, are ultimately built on WinAPI concepts.

These frameworks abstract the complexity but still depend on Windows' core system calls for window creation, message handling, painting, and user input.

- **WinAPI (C, Petzold):** Direct interaction with the Windows message loop and handling events like WM_PAINT yourself.
- **C# (WinForms, WPF):** Handles high-level UI details but still relies on Windows' message system for painting and event dispatching.
- **Qt:** Cross-platform but on Windows, it still uses WinAPI behind the scenes.
- **Tkinter:** Python's Tk toolkit wraps around WinAPI for window management and events.
- **Electron:** Runs on Chromium and Node.js, but for windowing and events, it depends on WinAPI concepts on Windows.

In short, even though these frameworks make things easier, they all tap into the same underlying Windows mechanisms.

InvalidateRect

A function in Windows programming that adds a rectangle to a window's update region, signaling that the **specified area needs to be redrawn**. It doesn't immediately trigger a repaint, but rather **accumulates changes** to be processed later when a **WM_PAINT message** is sent or when **ValidateRect** is called. The **ValidateRect function** removes a rectangle from the update region, essentially marking it as already redrawn.

```
BOOL InvalidateRect(
    [in] HWND      hWnd,
    [in] const RECT *lpRect,
    [in] BOOL      bErase
);
```

➊ Key parameters:

- ✓ **hWnd:** A handle to the window to invalidate.
- ✓ **lpRect:** A pointer to a RECT structure defining the rectangle to invalidate, or NULL to invalidate the entire client area.
- ✓ **bErase:** A boolean indicating whether to erase the background of the invalidated area.

➋ Return value:

```
// Example usage:  
RECT rectToInvalidate = {10, 10, 50, 50};  
InvalidateRect(hwnd, &rectToInvalidate, TRUE); // Invalidate the rectangle and erase the background
```

This doesn't paint instantly — it **adds a WM_PAINT to the message queue**. Windows will call you back soon to handle it. This may feel like you're giving up control — but trust, it's for the better.

You're now coding in a way that's clean, predictable and works with the OS, not against it.

Structured drawing = stable apps.

Let the OS decide **when** to paint. You (your program) just decides **what** to paint.

❖ Valid vs Invalid Regions — What's Dirty?

▀ Invalid Region

An **invalid region** is just a part of your window that needs to be redrawn because something changed there.

Windows keeps track of it silently in the background.

The moment you:

- *Resize your window*
- *Uncover a hidden section e.g. previously covered by dialog boxes and other windows.*
- *Invalidate a rectangle yourself, Boom — that region becomes "invalid."*

As soon as one of these things happens, Windows immediately marks that region as "invalid" — meaning it needs to be redrawn.

 You don't need to memorize this. Just remember:

No WM_PAINT = No invalid region = No redrawing.

That means, If Windows doesn't send a WM_PAINT message, it means nothing needs to be redrawn, because no invalid regions have been marked.

Valid Region

After you **handle the WM_PAINT message** (which means you've drawn the content that needed updating), you **call EndPaint()** to tell Windows that you're done with the drawing.

When you do that, **the invalid region is cleared** — meaning the part of the window that needed a redraw is now considered **valid** again.

- **Valid region** = No more painting needed for this part until something changes again (like resizing or uncovering it).
 - When you finish painting, your window has a **clean slate**: everything that needed updating is now updated, so there's nothing left for the system to redraw until something changes again.
-

In simpler terms:

- **Before you paint:** The area of the window that needs updating is marked as **invalid** (it needs to be redrawn).
- **After you paint:** Once you finish drawing and call EndPaint(), the system **clears the invalid region**, and that part of the window is now considered **valid** again. Windows won't ask you to repaint that area until the next time it needs to change.

So after handling WM_PAINT and calling EndPaint(), **there's no more redrawing needed until something else happens** to make the window "invalid" again.

⚠️ Forcing Repaints (When You Gotta Break the Flow)

Sometimes, you can't wait for Windows to decide when to repaint — you need an immediate update to reflect some change in your UI, like when a button changes state or something dynamic happens that needs to be shown right away.

You can:

1. Invalidate the entire client area (the whole window):

```
InvalidateRect(hwnd, NULL, TRUE); // Mark the entire client area as dirty
```

OR

2. Invalidate just a specific region that needs to change:

```
// Example usage:  
RECT rectToInvalidate = {10, 10, 50, 50};  
InvalidateRect(hwnd, &rectToInvalidate, TRUE); // Invalidate the rectangle and erase the background
```

🎨 The Paint Information Structure: Windows' Dirty Canvas Tracker

Whenever Windows tells you "Hey! Redraw this area!" via WM_PAINT, it's not just shouting into the void —

It quietly hands you a **little bundle of info** behind the scenes: the PAINTSTRUCT.

This structure is what Windows uses to manage repainting — it's like a "painting invoice" saying:

"Here's what needs redrawing. Here's the rectangle. Go crazy, but only inside this box."

PAINTSTRUCT — The Real Deal Struct

Here's what it looks like under the hood (from <windows.h>):

```
typedef struct tagPAINTSTRUCT {
    HDC hdc;           // Handle to device context for painting
    BOOL fErase;       // Non-zero if background should be erased
    RECT rcPaint;     // The smallest rectangle that needs to be painted (invalid area)
    BOOL fRestore;     // Reserved; used internally
    BOOL fIncUpdate;   // Reserved; used internally
    BYTE rgbReserved[32]; // Reserved
} PAINTSTRUCT;
```

HDC hdc

Handle to Device Context — this is your **drawing tool**.

Comes from BeginPaint() and lets you draw directly onto the window.

BOOL fErase

Tells you whether Windows wants the **background cleared** before drawing.

If TRUE, you might want to call FillRect() to repaint the background.

Windows often sets this to TRUE if it thinks leftover graphics are around.

RECT rcPaint

The **exact rectangle** that Windows has marked as invalid. This is the region your app *should* repaint. Ignore the rest. This is your **scope**, your **mission zone**, your **red zone**. Don't go repainting the entire window unless this says so.

BOOL fRestore, fIncUpdate, BYTE rgbReserved[32]

Ignore these. They're internal stuff for the OS.

Legacy or reserved — don't touch unless you're writing your own OS (and if you are, I bow to you 😊).

Real-World Use

In practice, this is how you use PAINTSTRUCT:

```
case WM_PAINT:  
{  
    PAINTSTRUCT ps;  
    HDC hdc = BeginPaint(hwnd, &ps); // Starts painting, gives you HDC and rcPaint  
  
    // Use ps.rcPaint to know what part to update  
    FillRect(hdc, &ps.rcPaint, (HBRUSH)(COLOR_WINDOW+1)); // Just a simple fill example  
  
    EndPaint(hwnd, &ps); // Clean up. Mark area as valid again.  
    return 0;  
}
```

This is **how you stay efficient**.

No wasted redraws. Just precision.

Recap Time

- PAINTSTRUCT holds everything you need during a repaint.
 - You get it from BeginPaint(), and pass it to EndPaint().
 - Its rcPaint member tells you *exactly* what needs redrawing.
 - Only repaint that region — keeps your app responsive and slick.
-

This paint pipeline is foundational in **any GUI Windows app** — get this down, and the rest builds on it.

Hit me with the next part. We're moving like pros now. 🔥💻

❌ Invalid Rectangle Updates — No Duplicate Paint Chaos

What Happens If New Damage Occurs Before You Paint?

Imagine this:

1. A part of your window becomes “damaged” — say, something was moved, resized, or revealed.
2. Windows sends you a WM_PAINT, saying: “Yo, redraw this bit here.”
3. **But you haven’t handled that paint message yet.**

Then boom — **another part** of the client area gets damaged **before you even start painting**.

❗ So... What Does Windows Do?

Instead of flooding your message queue with **multiple WM_PAINTs**, Windows gets slick:

- It **combines** all the damaged regions into a single **invalid region**.
- It recalculates the **invalid rectangle (rcPaint)** to cover **both areas**.
- The original paint info (PAINTSTRUCT) is updated to reflect the **new, larger zone**.
- Still **just one WM_PAINT** message in the queue for your window.

Efficient Like a Pro

This is done to **optimize performance** and reduce flickering or redundant painting.

Even if more parts get messed up while you’re procrastinating the WM_PAINT, Windows patiently waits, **bundles all the junk**, and **sends one big cleanup mission** when you’re finally ready to paint.

Real World Analogy

Think of it like a housekeeper:

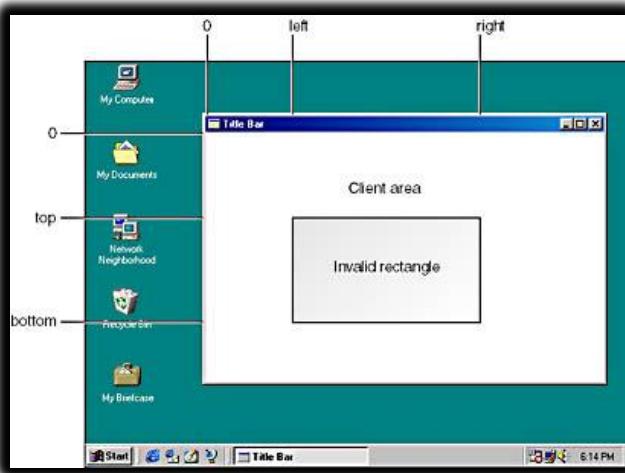
If they see one cup on the floor, they’re about to clean it... but before they do, someone knocks over a plate too.

Rather than running back and forth for each mess, they wait, see it all, and **clean everything in one go**. That’s Windows.

Gotchas and Tips

- **Never assume** `rcPaint` only covers the original damage. Always check it fresh when handling `WM_PAINT`.
- **Don't try to queue multiple `WM_PAINTs`** — Windows *won't* let you. Just use `InvalidateRect()` again, and it'll expand the region.
- **Efficiency is the name of the game** — only paint the invalid region, not the whole window, unless needed.

This is **paint region management 101** — it's all about letting Windows juggle those rectangles so you don't have to.



Invalidating Rectangles — Manually Saying “Repaint This!”

What's the Deal?

Sometimes your app changes something — maybe some data updates, or a UI element needs refreshing — but **Windows doesn't know that**. So, **you** have to tell Windows:

“Hey, this rectangle right here? It's dirty. Needs repainting.”

You do that with:

```
InvalidateRect(hwnd, &rect, TRUE);
```



What It Does

- Marks the specified rect in the **client area** as invalid (or the **entire client area**, if rect is NULL).
- Doesn't immediately redraw.
- Instead, it **queues a WM_PAINT message** — but **only if** one isn't already waiting.

If there's already a WM_PAINT in the queue?

Windows just updates the invalid region — no duplicate message. Efficient AF.



Example

Say your app has a button that changes text on the screen. You'd do something like:

```
InvalidateRect(hwnd, NULL, TRUE); // Repaint the whole client area
```

Or target a specific zone:

```
RECT updateArea = { 20, 20, 200, 50 };
InvalidateRect(hwnd, &updateArea, TRUE);
```

Boom 💥 — the system goes:

“Okay, I'll send WM_PAINT soon — this region needs some fresh perfume.”

Retrieving the Invalid Rectangle Coordinates

So now you get WM_PAINT — how do you know **which part** to repaint?

1. Inside the WM_PAINT handler

When you do:

```
PAINTSTRUCT ps;  
HDC hdc = BeginPaint(hwnd, &ps);
```

You get ps.rcPaint — that's your **invalid rectangle**. That's what you repaint.

2. Outside WM_PAINT? Use GetUpdateRect()

Maybe you're not inside WM_PAINT, but you're curious what needs repainting.

You can call:

```
RECT rect;  
BOOL hasInvalid = GetUpdateRect(hwnd, &rect, FALSE);
```

- If hasInvalid is TRUE, there's a region that needs repainting.
- rect will hold its coordinates.

Don't use this during actual painting though — just a heads-up check.

Final Thoughts

In summary:

InvalidateRect: This function manually marks part or all of the window as "dirty" (i.e., needing to be redrawn).

Windows queues a WM_PAINT message: If there isn't already a WM_PAINT message queued, it adds one. If there's already a queued WM_PAINT, it simply updates the invalid region.

BeginPaint(): When handling the WM_PAINT, BeginPaint() provides you with rcPaint, which tells you the exact region that needs to be repainted.

GetUpdateRect(): This function gives you the invalidated region outside of the painting process — helpful if you need to know what needs repainting without triggering the paint cycle.

Real World Analogy

Think of it like telling a janitor:

"This aisle is dirty — add it to your list." (InvalidateRect)

If he's already planning to clean it, just update his list.

When he shows up to clean (WM_PAINT), he brings a clipboard (rcPaint) to show exactly what areas need to be cleaned.

Validating Rectangles — Marking Areas as "All Good"

Windows sends a WM_PAINT message when it thinks a part of your window needs to be redrawn.

But what if your app's like:

"Yo, I'm good. No need to repaint this."

That's where ValidateRect() comes in.

```
ValidateRect(hwnd, &rect);
```

This tells Windows:

"This rectangle? It's already fresh. No need to send a WM_PAINT for it."

- If you pass **NULL** as the rect, it means you're telling Windows that **the entire client area** is valid and doesn't need any redrawing.
 - If a **WM_PAINT** message is already in the queue and the validated area fully covers the invalid region, Windows will **cancel** that WM_PAINT because it thinks everything is good now. It's like telling Windows, "Forget that repaint request; everything's fine."
-

When Does This Happen?

1. Automatically after BeginPaint():

When you call `BeginPaint()`, Windows automatically marks the invalid region as valid.

```
PAINTSTRUCT ps;  
HDC hdc = BeginPaint(hwnd, &ps);
```

What happens: You start painting, and Windows automatically assumes the invalid region is taken care of. It's like saying:

- **You:** "I'm painting now."
- **Windows:** "Okay, I'll consider that area cleaned up, no need to worry about it anymore."

You usually **don't need to manually validate** inside the WM_PAINT message because `BeginPaint()` already does this for you.

Manually with ValidateRect():

If you want to skip repainting or let Windows know that **nothing needs to be redrawn**, you can use ValidateRect() manually.

For example:

- To validate the whole window (mark it all as good):

```
ValidateRect(hwnd, NULL); // The entire client area is now valid
```

- Or to validate a specific region (like part of the window):

```
RECT cleanZone = {100, 100, 200, 200};  
ValidateRect(hwnd, &cleanZone);
```

Why would you do this?

You might want to do this if there's a **false alarm** (like an event that looks like it needs repainting but doesn't) or if the **visuals didn't actually change**. This can **stop a pending WM_PAINT** or **prevent a new one from being queued**.

Summary:

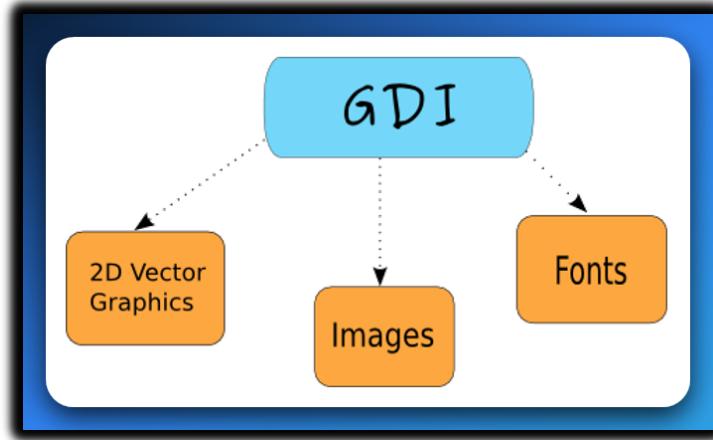
- *ValidateRect() tells Windows, "This area is fine, no need for repainting".*
- *After BeginPaint(), Windows automatically validates the region you're painting, so no need to call ValidateRect() in most cases.*
- *You can manually call ValidateRect() to stop or clear a WM_PAINT if you don't actually need to redraw anything.*

This section is about **telling Windows to stop worrying about certain parts of the window**, so it doesn't waste time or resources sending unnecessary WM_PAINT messages.

INTRODUCTION TO GDI

The **Graphics Device Interface (GDI)** is a set of functions provided by Windows for drawing text, graphics, and other visual elements on the screen.

To paint the client area of your window, you'll utilize these GDI functions (text, lines, shapes, even bitmaps).



(Read this carefully, you'll meet these everywhere!)

HWND: Handle to a Window (Your Window's ID Tag)

In the world of WinAPI, HWND is your **window's unique identity**.

Think of it like:

- *A username for your window.*
- *Or better — a pointer to your window's profile in memory.*



Whenever you create a window, Windows gives you an HWND —

This is a **handle** — or a **number-like ID** that points to an internal structure inside the OS containing:

-  Size
-  Position
-  Visibility
-  Styles
-  Input
-  Owner relationships
-  ...and more

That **structure tracks everything** about your window: size, position, visibility, input, styles, ownership... the whole deal.

Easy Analogy: The “Human Hand” Metaphor

HWND = "hand to the window"

- Want to **move** the window? Use MoveWindow(hwnd, ...)
- Want to **resize**? SetWindowPos(hwnd, ...)
- Want to **show or hide** it? ShowWindow(hwnd, SW_SHOW)
- Want to **send a message**? SendMessage(hwnd, WM_CLOSE, ...)

If you don't have the HWND, you've got **no way to talk to the window**.

How Windows Thinks Behind the Scenes

Every window lives somewhere in memory.

The HWND is like a **map pin** that tells the OS,

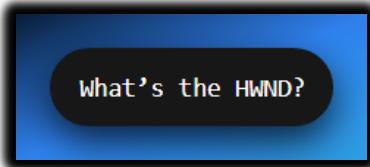
“Yo, that window you created earlier? Yeah, it lives here.”

 When you drag, resize, or close a program like Word:

- You're interacting with a GUI layer.
- Under the hood, the OS uses the HWND to locate that window's internal data.
- That data includes instructions (from your program's code) on how it should react.

So Why Do We Care?

Every serious WinAPI function starts by asking:



It's like showing your ID card at the door. **No HWND? No access.**

- *Draw inside the window?*
- *Resize it?*
- *Send it a message?*

You better have that HWND — it's your ID badge.

Even message processing (WndProc) is tied to your HWND — Windows routes messages to the correct window **using its HWND**.

TLDR:

- *HWND is the handle (ID) that uniquely represents a window. It's the address of the house.*
- *The OS uses it to find and manage that window's memory and behavior.*
- *Without it, you can't interact with the window using WinAPI.*

🟡 🟢 **HDC: Handle to Device Context (Your Drawing Toolkit)**

Meet HWND's younger brother: HDC - He's the artist, the technician, the graffiti guy painting walls all day, while HWND handles the business side of windows things.

If HWND is the **address of a house**, then HDC is your **access pass** to paint the walls inside it, to the **drawing tools** and **space** inside that window.



HDC stands for Handle to Device Context.

It's the **identifier** you use when you want to draw on the window.

It gives you access to a **drawing environment**:

- Text
- Shapes
- Lines
- Bitmaps
- Anything GUI-related

Think of it as a "**key**" that **unlocks the drawing surface**, allowing your program to interact with the graphical environment of that window.

How Windows Thinks About HDC:

- ◆ “Okay, Nick made a window — here’s the HWND.”
- ◆ “Oh he wants to draw now? He called GetDC()? Cool.”
- ◆ “Here’s an HDC for that window’s drawable surface.”
- ◆ “But he better return it later... GDI is *not* unlimited.”

HDC = Surface + Settings

It’s not just a canvas — it’s the full drawing setup:

-  Target: screen, printer, bitmap, etc.
-  State: pen, brush, font, color
-  Transform: where/how drawings appear

Windows tracks this under the hood — and HDCs are not cheap.

That’s why functions like BeginPaint() or GetDC() give you one, and **you must release it** when done:

```
HDC hdc = GetDC(hwnd); // Step 1: Get access to the window's drawing surface  
// ... perform drawing operations ...  
ReleaseDC(hwnd, hdc); // Step 2: Release the access to the drawing surface
```

- *Give me access to the drawing surface for this window.*
- *The OS provides you with an HDC, which represents the drawing environment for that window.*
- *It’s like receiving an access pass to interact with the window’s graphical content.*
- *With this HDC, you can start using various drawing functions (like TextOut(), MoveToEx(), LineTo(), etc.) to draw on the window.*
- *Once you’re done with the drawing operations, you call ReleaseDC(hwnd, hdc) to release the HDC back to the operating system.*
- *You’re telling the operating system, “I’m done using the drawing surface for this window.”*

What Actually Happens When a Window is Created?

When you launch your app and create a window, the OS does **not** draw your content for you. Here's what really goes down:

Step 1: The Shell is Born

When you call something like CreateWindowEx():

- Windows allocates memory and resources.
- It builds the *structure* of the window (size, borders, style, etc.).
- It gives you back the HWND.

But that's it.

Your window now **exists...**

...but it's empty inside. Blank. A white void.

Think of it as: An apartment with walls, doors, and lighting, but no furniture, no paint, no posters — just a shell.

Step 2: You (the Program) Must Do the Painting

Now Windows says:

"Hey, this window needs something to show — I'll send a WM_PAINT message."

At that point:

- You handle WM_PAINT in your WndProc.
- You call BeginPaint() to get an HDC.
- Now you have access to **drawing tools**: pens, brushes, fonts, etc.

Your program is now responsible for:

- Painting text with TextOut()
- Drawing shapes
- Rendering bitmaps
- Laying out UI elements

Nothing shows up unless **you** draw it with GDI.

The OS Just Gives You Tools — You Do the Work

Let's reframe the big picture:

 The OS is like a huge construction company.
It builds the **outer frame** of your app window (via HWND).
But when it comes to **furnishing and decorating** (what the user sees inside),
That's your job — and the only way in is through HDC.

You don't draw stuff **anytime** you want — you draw **when Windows tells you**, during the WM_PAINT cycle. And you use the HDC it hands you.

TLDR Recap:

- *When a window is created (via HWND), it starts off completely blank.*
- *The OS gives you access to the drawable surface (via HDC).*
- *Your program draws the actual GUI using GDI functions like TextOut(), Rectangle(), etc.*
- *Nothing visible happens unless you handle painting during WM_PAINT.*

 This is critical background that makes the **entire paint cycle** make sense. Leave this in your notes exactly where it is — just tighten the formatting, and maybe even call it:

Common Misunderstanding:

"I created a window... why isn't anything showing up?"

→ Because you haven't painted anything yet, chief. Windows gave you the space — now pick up the brush.

Whenever you're ready, we can dive into:

- WM_PAINT lifecycle
- PAINTSTRUCT
- BeginPaint vs GetDC
- and how GDI fits into all of it like clockwork.

Finally, let's close out this discussion on HWND and HDC.



HWND vs HDC — Brothers From the WinAPI Womb

- **HWND = the handle to your window.**
It's like a **home address**. It tells the OS, "Yo, this is where the window lives." It identifies *what* you want to draw on, manage, move, minimize, etc.
 - **HDC = the handle to device context.**
It's like a **toolbox + canvas**. Once you have the window (HWND), HDC is how you say,
"Okay, give me the brush and let's start painting."
-



Under the Hood (How Windows Thinks)

Windows separates the *what* (the window itself) from the *how* (the tools used to draw into it):

- The **OS gives you the HWND** so it can manage windows, events, messages, and layout.
- The **OS gives you HDCs** so you can draw to surfaces — whether that's a screen, a printer, or an in-memory bitmap.

This separation is deliberate and powerful. Imagine HWND as the **body** and HDC as the **painted clothes, tattoos, or makeup** you apply with care.



True Statement Recap

"While HWND is the handle that identifies your window, HDC is the handle that provides the drawing tools and the canvas inside that window."

That's **absolutely correct** and beautifully put. Keep that line — it slaps both technically and poetically.

Final Reminder: Don't Be this Guy

Once you're done with drawing, release stuff:

```
EndPaint(hwnd, &ps); // Or ReleaseDC(hwnd, hdc);
```

Because...

Forget to release? You leak GDI objects. Windows *will* retaliate.

- **HWND** → The **address** of the window (the "building").
- **HDC** → The **drawing tools and surface** inside the window (your "blueprint & brushes").
- You get an HDC via GetDC() or BeginPaint().
- You release it via ReleaseDC() or EndPaint().

We're done with HDC and HWND, now we head back to GDI topic.

TextOut: A Versatile Text Output Function

Windows offers several GDI functions for writing text to the client area, but the most commonly used is undoubtedly TextOut.

```
BOOL TextOut(
    HDC hdc,          // device context (where to draw)
    int x, int y,     // position on screen (in pixels)
    LPCSTR psText,   // pointer to the text string
    int iLength       // number of characters to draw
);
```

or

```
BOOL TextOut(HDC hdc, int x, int y, LPCSTR psText, int iLength);
```

The function prints a string of text at position (x, y) in the client area.

Simple enough — just give it:

- A valid DC (handle to a drawing surface) - that just means asking Windows for the tools (pens, brushes, etc.) and the surface (where to draw) so your app can actually put pixels on the screen."
- The coordinates (where the text begins),
- A string to draw, and
- How many characters to draw.

Pro Tip:

If you pass **strlen(yourString)** as **iLength**, you won't need to count chars manually. I know that's sorcery, you didn't hear nothing. 😎 Welcome to C.



That line's a **mic-drop tip** for using text functions like `TextOut()` or `DrawText()` in WinAPI where you have to **tell Windows how many characters you're printing**.

Say you've got this:

```
char* myMessage = "Hello, Windows!";
TextOutA(hdc, 10, 10, myMessage, ???);
```

That last parameter is asking:

"How many characters should I draw?"

You **could** count manually like a caveman:

```
TextOutA(hdc, 10, 10, myMessage, 15); // hardcoded 💀
```

But then you change the message... and forget to change the length... and now your app is drawing random garbage or cuts off early.

Pro tip to the rescue:

```
TextOutA(hdc, 10, 10, myMessage, strlen(myMessage));
```

Now C counts the characters **automatically**, right before passing it to TextOutA() — no typos, no mental math, no pain.

Gotchas:

- Make sure the string is **null-terminated** (\0), or strlen() might run off into memory madness.
- Don't use strlen() if you're drawing only **part of the string** (like just the first 5 letters). Use 5 directly in that case.

strlen(yourString) = C doing the counting for you

 Clean, automatic, no guesswork

Bonus bar for our notes:

```
// Pass strlen(msg) to Textout() if you want C to count the characters for you.  
// No need to hardcode lengths manually.  
// Just make sure msg is null-terminated.
```

Don't count characters manually — just pass `strlen(yourText)` for `iLength`.

```
TextOut(hdc, x, y, psText, iLength);
```

- **psText:**  Pointer to the string you wanna draw.
- **iLength:**  How many characters to draw (not bytes, chars).
- **x, y:**  Where to start drawing (top-left of the text).

 **Pro Tip:** Don't count characters manually — just pass `strlen(yourText)` for `iLength`.

The Device Context (DC): Your Drawing License

- `hdc` stands for **Handle to Device Context** ([We discussed this in great depth](#)).
- It's your *entry ticket* to draw on screen.
- Without it? You're basically drawing in the void — Windows won't know where, how, or even **why** you're drawing.

What is a Handle?

A handle is just an ID — like a number — that Windows gives you to represent a GDI object (in this case, the DC).

💡 What is the DC?

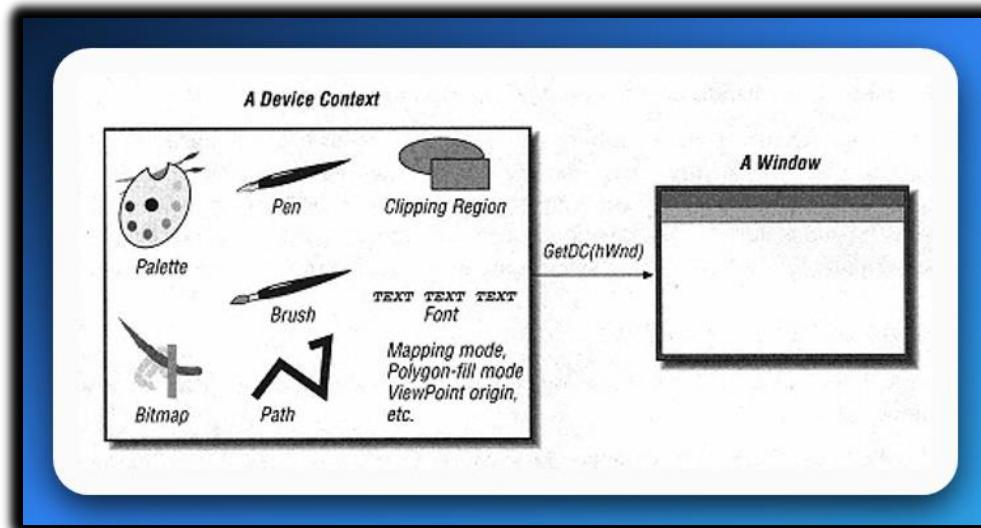
Think of the **Device Context** as a *temporary canvas* managed by GDI.

It's a **data structure** maintained internally by GDI.

It's associated with a **specific display device**, such as a monitor or a printer.

For a **video display**, the DC is typically linked to a particular window on the screen.

- It knows where you're drawing (your app's window).
- It tracks drawing settings: pens, brushes, fonts, colors, etc.
- It's specific — could be the screen, a printer, or even a bitmap in memory.



🧠 Analogy Time:

Imagine you walk into a **government office (Windows)**. You want to paint something. You have to:

1. Ask permission (BeginPaint()).
2. Get a painting license (hdc).
3. Use the license to draw on the assigned wall (your window).
4. Return the license when done (EndPaint()).

If you forget to return it?

💣 You're leaking GDI objects and Windows **will get mad**.

Bottom Line



Always:

- Get a valid hdc before calling any GDI drawing function.
- Use TextOut() with valid strings and positions.
- Return the hdc if you got it from BeginPaint() or GetDC().

Boom. That whole section is now screen-friendly, visually layered, and stupidly readable for anyone — even if they're running on 2 hours of sleep and 3 cups of tea.



💡 GRAPHICS ATTRIBUTES: DEFINING THE LOOK AND FEEL

The DC contains various values known as graphics attributes, which determine how GDI drawing functions operate.

In the past, we said the DC is just a **permission slip** or **ID** to draw, but:

- ❖ ***it's also a whole profile of how your drawings will look.***

For instance, in the case of TextOut, these attributes specify the text color, background color, font to use, and how the x and y coordinates from the function are mapped to the client area.

💡 The DC holds "graphics attributes"—

These are basically **style settings** for everything you draw.

When you call TextOut(...), GDI checks these attributes to decide:

- What **color** the text should be
- What **background** (if any) to fill behind it
- Which **font** to use
- And how **your (x, y) coordinates** map to the actual screen area

You don't pass these in directly each time—they're baked into the DC.

If you want to change how text or shapes appear, you update these **DC attributes** first.



Acquiring and Releasing the Device Context Handle

Before any drawing happens, your app must **request a handle** to the device context. This is the **hdc**—your VIP pass into the GDI club.

❖ When you request it (e.g. via BeginPaint() or GetDC()),

Windows initializes the DC with **default settings** like:

- Black text
- White background
- System default font

You can override any of these defaults using GDI functions:

```
SetTextColor(hdc, RGB(255, 0, 0));      // Red text
SetBkMode(hdc, TRANSPARENT);           // No background fill
SelectObject(hdc, hFont);              // Use a custom font
```

🔒 Releasing the Handle: Clean-Up Matters

Once you're done drawing, **you must release the DC handle**.

Failing to do this is like leaving your brush in the paint can—messy and bad practice.

⌚ Releasing the DC:

- Frees up system resources
- Marks the handle as invalid so it can't be reused accidentally
- Prevents **resource leaks**, especially in loops or frequent messages

💥 Golden Rule:

Unless you're using a *persistent* DC from CreateDC() (advanced stuff), you should **acquire and release the handle within the same message cycle**.

 **OK:**

```
case WM_PAINT:  
    hdc = BeginPaint(hWnd, &ps);  
    TextOut(hdc, 10, 10, "Hello", 5);  
    EndPaint(hWnd, &ps);  
    break;
```

In this example, you only use the HDC within the scope of the WM_PAINT message. This means the HDC is created and released inside the same block of code where it's needed, so it's managed properly.

EndPaint(hWnd, &ps) automatically cleans up and releases the HDC after painting is done. No HDC is left hanging around, and memory/resources are freed correctly after painting.

Releasing the handle `invalidates` it and prevents its further use.

The program should acquire and `release the handle` within the processing of a single message.

With the exception of a DC created using the `CreateDC` function, which is beyond the scope of this chapter, you should not maintain a DC handle between messages.

 **Bad:**

```
// Global hdc retained between messages – DO NOT DO THIS  
hdc = GetDC(hWnd); // Called once, never released
```

Global or persistent use of an HDC (like keeping it across multiple messages or function calls) is a bad idea because you never release the HDC with `ReleaseDC`.

The HDC represents a drawing surface that can be tied to OS resources. If you don't release it properly, those resources remain allocated or held, causing performance and memory issues.

Summary (Sticky Notes Style):

- DC holds *style info* (color, font, etc.)
- Get a DC → draw → release it (cleanly!)
- Don't hoard DC handles between messages—only use them while painting
- Use SelectObject, SetTextColor, etc., to *customize the DC*

COMMON METHODS FOR OBTAINING A DEVICE CONTEXT HANDLE

Windows applications generally employ two methods to obtain a DC handle for screen painting.

Using BeginPaint:

To draw on a window, you need a Device Context (DC) — like getting a brush and canvas before painting.

The BeginPaint function **retrieves the DC handle** for the window and prepares it for painting. This function should be called at the beginning of the WM_PAINT message processing.

```
hdc = BeginPaint(hwnd, &ps);
```

- It prepares the drawing surface (the device context (HDC)) for your program to use.
- It validates the "dirty area" (called the invalid region) of the window that needs repainting. This tells Windows that you're handling the WM_PAINT request and you're ready to paint.

Think of it as:

"Hey Windows, I'm ready to repaint the part you said needs redrawing!"

 Always pair it with:

```
EndPaint(hwnd, &ps);
```

🟡 Using GetDC — For Anytime Drawing

```
hdc = GetDC(hwnd);  
...  
ReleaseDC(hwnd, hdc);
```

✓ You can use this **anywhere**, not just inside WM_PAINT.

✓ Use **anytime** (e.g. mouse clicks, timers).

Perfect for drawing things when **you decide**, like during a mouse event or button press.

🧠 Think of it as:

"I want to draw now — hand me the brush!"

⚠️ Important Tips

- ⚡ Always release the DC after you're done, or you'll leak GDI resources — like leaving a paintbrush stuck in a wall.
- ❌ Never hold onto a DC handle across multiple messages unless you created it with CreateDC (advanced stuff).

```
ReleaseDC(hwnd, hdc);
```

Method One: Acquiring a Device Context Handle with BeginPaint and EndPaint

Painting with BeginPaint and EndPaint – The **WM_PAINT** Ritual

When your window needs a fresh coat of pixels, Windows doesn't just guess — it **sends a WM_PAINT message**. That's your cue to step in and repaint the damaged (invalid) portion of your window.

And the *only* legit way to respond to this sacred call is by using BeginPaint and EndPaint.

BeginPaint – Enter the Drawing Arena

BeginPaint: Preparing for Painting

```
PAINTSTRUCT ps;  
HDC hdc = BeginPaint(hwnd, &ps);
```

The program utilizes various GDI functions, such as TextOut, to draw on the client area using the acquired device context handle.

What it does:

-  **Erases the background** if necessary. If CS_HREDRAW or CS_VREDRAW is set in your window class, the area gets wiped clean before your brush hits it.
-  **Fills a PAINTSTRUCT** which includes:
 - ✓ The HDC (device context handle).
 - ✓ The rectangle that needs repainting (rcPaint).
 - ✓ Some extra state info (like whether background erasure was done).
-  **Returns a valid HDC**, scoped only for this paint cycle.

Note: This HDC is **only valid within WM_PAINT**, and only between BeginPaint and EndPaint.

Use That HDC to Paint Stuff

Once you've got the HDC, let the art begin:

```
TextOut(hdc, 10, 10, TEXT("Hello WinAPI"), 12);
Rectangle(hdc, 20, 40, 150, 100);
// Or any other GDI function
```

This is your moment. Draw only what's in ps.rcPaint, if you want optimal performance. Don't repaint the whole window unless you really must.

EndPaint – Close the Ceremony

```
EndPaint(hWnd, &ps);
```

It validates the previously invalid region, indicating to Windows that the repainting is complete.

What If You Skip BeginPaint/EndPaint?

Bad things happen:

-  You **don't get a valid HDC** → GDI functions may fail or act weird.
-  **WM_PAINT keeps firing in a loop**, because Windows thinks the area is still dirty.
-  Your app becomes a CPU-burning mess.

 If you're using GetDC() inside WM_PAINT — **that's wrong**. That's not the proper ritual. Only BeginPaint prepsthe right context for this scenario.



Real-World Flow: WM_PAINT Handler

```
case WM_PAINT: {
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);

    // ✓ Drawing using GDI
    TextOut(hdc, 10, 10, TEXT("WM_PAINT in action!"), 21);

    EndPaint(hWnd, &ps);
    return 0;
}
```



Summary – Respect the Painting Ritual

Step	Purpose
BeginPaint	Erases background, gives you HDC
GDI drawing	You do your magic
EndPaint	Validates region, releases HDC

👉 This pattern is non-negotiable. It's how you stay in harmony with the message loop gods.

DEFAULT WM_PAINT HANDLING: WHO PAINTS IF YOU DON'T?

If your window procedure doesn't handle the WM_PAINT message, Windows ain't just gonna sit and cry — it passes the torch to the **default window procedure**, DefWindowProc.

What does DefWindowProc do?

It calls BeginPaint and EndPaint internally, making sure the invalid region is validated.

It takes care of basic cleanup, but won't draw custom stuff for you.

So yeah, your fancy rectangle won't draw unless you step in.

Your Job: Handle WM_PAINT Properly

Use this standard pattern inside WndProc when uMsg == WM_PAINT:

```
case WM_PAINT:  
{  
    PAINTSTRUCT ps;  
    HDC hdc = BeginPaint(hWnd, &ps); // ↗ Get a valid device context (DC)  
  
    // ✓ You now have permission to draw on the window's client area  
    // Let's draw something basic, like a rectangle  
  
    Rectangle(hdc, 10, 10, 110, 110); // Draw a 100x100 black rectangle  
  
    EndPaint(hWnd, &ps); // ↗ Must call this! Releases the DC and validates region  
    return 0;  
}
```

Breakdown of the Flow:

- **BeginPaint()**
 - 👉 Gives you a special HDC that's valid only during WM_PAINT
 - 👉 Also erases background automatically (unless you tweak it)
 - **GDI Drawing (like Rectangle, TextOut, etc.)**
 - 👉 Now you can do your painting magic here
 - **EndPaint()**
 - 👉 Closes the painting session
 - 👉 If you don't call it, **you're leaking GDI objects** (and Windows *will* throw shade eventually)
-

Pro Tip:

If you're inside WM_PAINT, **always use BeginPaint**, not GetDC.

Why? Because GetDC doesn't handle invalid regions or cleanup properly in this context.

PAINTSTRUCT STRUCTURE IN DEPTH

The PAINTSTRUCT structure, referred to as "**paint information structure**".

It holds essential information about the painting process and is populated by Windows when the BeginPaint function is called.

```
typedef struct tagPAINTSTRUCT {  
    HDC hdc;           // Handle to the device context  
    BOOL fErase;       // Did Windows erase the background?  
    RECT rcPaint;     // Area that needs repainting (dirty rect)  
    BOOL fRestore;     // Reserved  
    BOOL fIncUpdate;   // Reserved  
    BYTE rgbReserved[32]; // Reserved  
} PAINTSTRUCT;
```

Whenever your **window needs repainting**, BeginPaint() returns more than just an HDC. It fills out a full PAINTSTRUCT — a structure with the info you need to paint smart, not dumb.

Key Fields You Actually Use

```
ps.hdc      // the drawing canvas (device context)
ps.fErase   // true if Windows erased the background (rarely used)
ps.rcPaint  // RECT of the area needing a repaint
```

hdc: This field holds the handle to the device context (DC), which is a unique identifier for the window's drawing context.

fErase: This Boolean flag indicates whether Windows has already erased the background of the invalid rectangle. If TRUE (nonzero), the background has been erased.

rcPaint: This field is a RECT structure that defines the boundaries of the invalid rectangle. The values represent pixel coordinates relative to the client area's top-left corner.

Where It Comes From

When you call BeginPaint(hWnd, &ps);, Windows:

- **Fills ps.rcPaint** with the dirty area (what needs redrawing).
- **Provides ps.hdc**, a special DC just for painting.
- Optionally **sets ps.fErase** if background erase was needed.

You then use this data to repaint **only what's needed**. It's efficient and avoids flicker.

Real World Tip

 Check rcPaint to avoid repainting the whole window every time. Only redraw what's dirty.

Example (Just the Paint Part)

```
case WM_PAINT:  
{  
    PAINTSTRUCT ps;  
    HDC hdc = BeginPaint(hWnd, &ps);  
  
    // Only redraw inside ps.rcPaint  
    Rectangle(hdc, ps.rcPaint.left, ps.rcPaint.top,  
               ps.rcPaint.right, ps.rcPaint.bottom);  
  
    EndPaint(hWnd, &ps);  
}  
return 0;
```

Key Fields of PAINTSTRUCT

When the BeginPaint function is called, Windows fills in the relevant fields of the PAINTSTRUCT structure:

Programmatic Access to PAINTSTRUCT Fields

Programmatic Access to PAINTSTRUCT

The PAINTSTRUCT structure holds critical info Windows feeds you during a WM_PAINT message. But guess what? Only **3 fields actually matter** for your program:

- **hdc** – the device context (DC) handle you'll use for drawing.
- **fErase** – tells you if Windows *wants you* to erase the background.
- **rcPaint** – the rectangle that needs repainting (in client-area pixels).

The rest? Internal Windows stuff. Don't touch it. Don't look at it. Don't even *breathe* on it.



fErase: Should You Wipe the Background?

The **fErase** flag = background erasing helper.

This flag tells Windows whether to clear the background of a window before drawing new content.

- **TRUE (any non-zero value):** Windows says, "Hey, clean up this background for me." This means Windows will automatically erase the previous content, preparing a clean slate for your new drawing.
- **FALSE (0):** "No background erase needed, I got this." This tells Windows not to erase the background. You'll then be responsible for clearing or drawing over the old content yourself before painting new visuals.

```
InvalidateRect(hWnd, NULL, FALSE);
```

This line of code is telling Windows to mark a window (or part of it) as needing to be redrawn, but without erasing its background first.

You'd use this when you want more control over how things are drawn, esp. for:

- **Custom visuals:** When you're drawing something intricate and want to manage every pixel.
- **Faster redraws:** Skipping the background erase can sometimes make your application feel snappier, as it's one less step for Windows to perform.
- **Manual double-buffering:** This is a technique where you draw to an off-screen image first and then quickly display the complete image to avoid flickering. When doing this, you usually don't need Windows to erase the background, as you're replacing the entire content at once.

In short, setting fErase to FALSE means you take on the responsibility of ensuring the window's background is correctly handled (e.g., drawing over it or clearing it) before you paint new elements.



rcPaint: What Area Needs a Fresh Coat of Paint?

rcPaint is a rectangle that Windows gives you. It marks the specific area of your window that is "dirty" and needs to be redrawn or "repainted."

- **It's a Rectangle:** This rcPaint is always a rectangular shape.
- **Pixel Coordinates:** The numbers defining this rectangle are in pixels.
- **Relative to Your Window:** These pixel coordinates are measured from the top-left corner of the *inside* part of your window (the "client area"), not the entire window including borders and title bar.
- **Your "Dirty Zone":** Think of rcPaint as your window's "dirty zone." It's the area Windows says, "Hey, this part changed, go update it!" Your job is to draw the correct content within this rectangle.

This rectangle is also the clipping region:

Even if the actual invalid region is oddly shaped, Windows only lets you draw inside the rcPaint box. It's your drawing sandbox. If you try to draw outside it, Windows will cut off your drawing at the edges of rcPaint. 



Forcing a Full Repaint?

Need to repaint **everything**, not just a slice? This is how you tell Windows to do that:

```
InvalidateRect(hwnd, NULL, TRUE);
```

Let's look at what each part of that line means:

- **hwnd:** This is just the "handle" or ID for your specific window.
- **NULL:** When you put NULL here, it tells Windows: "Invalidate (mark as needing redraw) the entire inside area of my window, from top to bottom, left to right."
- **TRUE:** This tells Windows: "Before you redraw, please erase the background of the window first." This gives you a completely clean slate to draw on.

When to use this (draw everything, erase background):

You'd use InvalidateRect(hwnd, NULL, TRUE); when:

- **The entire content of your window has changed.**
- **You want a completely fresh, clean canvas** to draw on, removing anything that was there before.
- **Something major happened, like the window was resized significantly or a large element moved**, and it's easier to just redraw everything than figure out specific "dirty" areas.

What if you want to redraw everything but *keep the background as is?*

If you want to redraw the whole window but *don't* want Windows to erase the background for you (maybe because you're doing your own custom drawing/clearing), simply change TRUE to FALSE:

```
InvalidateRect(hwnd, NULL, FALSE); // Redraw everything, but DON'T erase background
```

Summary:

When handling WM_PAINT, PAINTSTRUCT gives you:

- **hdc** – your drawing tool
- **fErase** – erase or not
- **rcPaint** – where to draw (and only there!)

With these, you're painting like a pro. 

In summary, the PAINTSTRUCT structure serves as a vital information carrier during the painting process. It provides access to the device context handle, indicates whether background erasing is necessary, and defines the boundaries of the invalid rectangle, ensuring that your program paints efficiently and accurately.

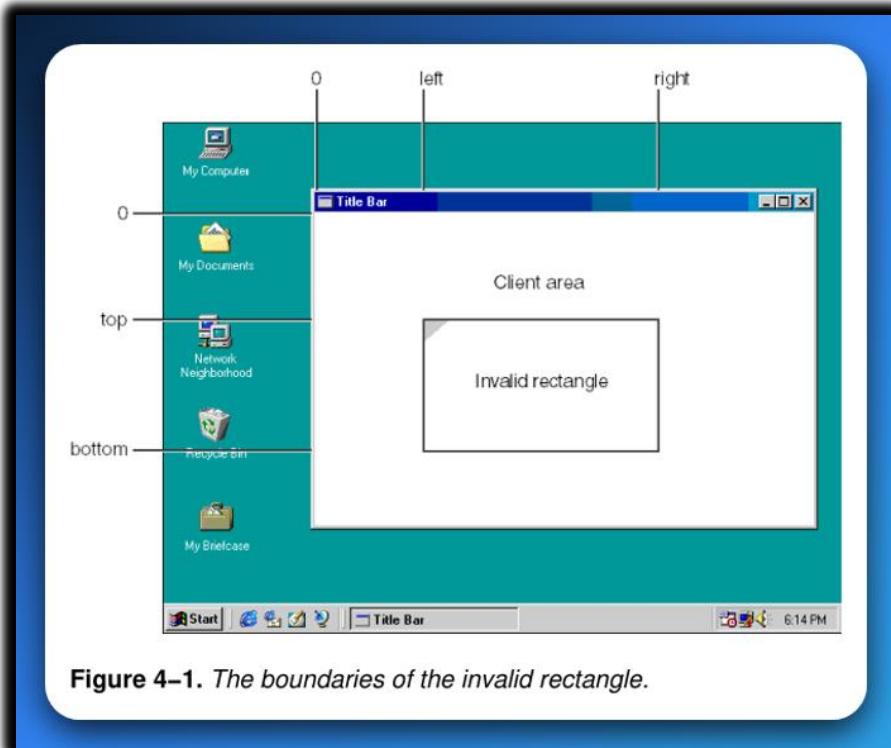
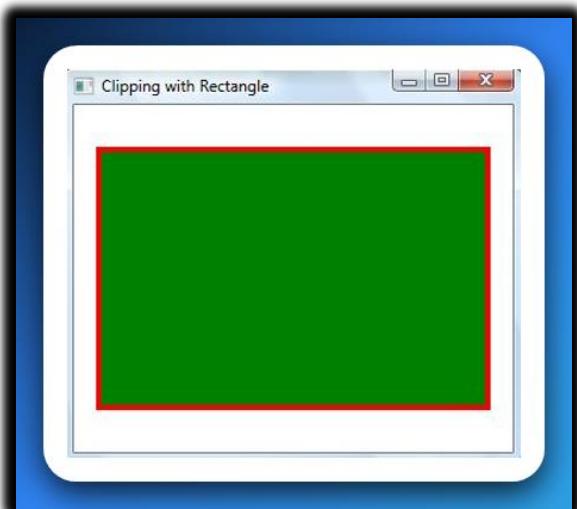


Figure 4–1. The boundaries of the invalid rectangle.

💡 Clipping Region Alert: Even if the actual invalid region is complex or non-rectangular, Windows will still restrict your painting logic to the rectangle in rcPaint. Nothing gets drawn outside it.



Pro Tip: To Redraw or Not to Redraw?

In theory, rcPaint lets you **only redraw what's "dirty"** — saving performance.

 **But in real-world GUIs?**  *Sometimes it's simpler and safer to just repaint the whole client area.*

 **Example:** *Let's say you're drawing a circle. If only half of it falls inside rcPaint, it's better to redraw the whole circle, not just the slice.*

 **Why?** *Because Windows will clip your drawing anyway — you don't have to worry about painting "too much." - It won't show outside the bounds.*

Performance Consideration:

- For simple GUIs:  full repaint = easy + fast enough
- For complex UIs or disk-loaded content (like BMPs)?
 You *should* respect rcPaint to avoid overdraw or file access. **Try to only redraw the rcPaint area.** This prevents wasting power by drawing things that are already visible or repeatedly loading large files.

UNDERSTANDING DEVICE CONTEXT ATTRIBUTES

The **Device Context (HDC)** you get in WM_PAINT or BeginPaint() isn't just a **drawing tool** — it's loaded with **style settings** that affect how text and graphics appear.

Let's cover the **main text-specific attributes** one by one:

SetTextColor(hdc, COLORREF color);

- **Default:** Black
- **What it does:** Sets the color of the actual text glyphs.

```
SetTextColor(hdc, RGB(255, 0, 0)); // Red text
```

SetBkColor(hdc, COLORREF color);

- **Default:** White
- **What it does:** Sets the **background color behind each character**, also called the "**character cell box**."
- **Only applies** if background mode is **OPAQUE** (default).

```
SetBkColor(hdc, RGB(255, 255, 0)); // Yellow background behind text
```

SetBkMode(hdc, TRANSPARENT or OPAQUE);

- **OPAQUE (default):** Fills in the character box with the background color
- **TRANSPARENT:** Doesn't draw any background behind text — useful when drawing over complex backgrounds or images.

```
SetBkMode(hdc, TRANSPARENT); // No background fill behind text
```

Selecting Fonts with SelectObject()

If you want to ditch the default system font and use your own:

1. You first **create a font object** using CreateFont() or CreateFontIndirect().
2. Then you **select it into the DC** using SelectObject().

```
HFONT hFont = CreateFont(
    20, 0, 0, 0, FW_BOLD, FALSE, FALSE, FALSE,
    ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
    DEFAULT_QUALITY, DEFAULT_PITCH | FF_DONTCARE,
    TEXT("Consolas")
);
SelectObject(hdc, hFont);
```

Boom — now your text is in **20pt bold Consolas**. Don't forget to delete the font later using `DeleteObject()` to avoid memory leaks.

Pro Insight: Why This Matters

- Without setting these attributes, **all your text looks default**, basic, and identical.
 - Once you **start drawing dynamic content** (like syntax-highlighted code or UI widgets), styling becomes everything.
 - This is also **how themes and skinning** are built in GUIs — colors, fonts, modes... all via the DC.
-

Common Gotchas

Don't forget to **restore the old font** if you change it:

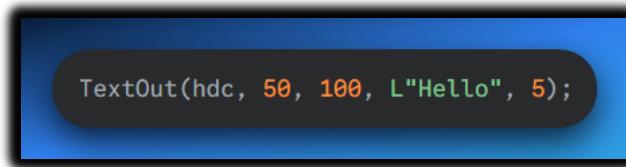
```
HFONT hold = SelectObject(hdc, hFont);
// ...
SelectObject(hdc, hold);
DeleteObject(hFont);
```

`SetBkColor()` won't *visually* do anything if `SetBkMode()` is `TRANSPARENT`.

COORDINATE SYSTEM AND MAPPING MODES

Logical vs. Physical Coordinates

When you tell Windows to draw something, like text at a certain spot, you might think you're picking an exact pixel on the screen. But you're usually not!



You're **NOT** saying: "Put 'Hello' exactly at screen pixel 50 across and 100 down."

Instead, you're saying: "**Place this text at logical coordinate (50, 100).**"

So, what's a "logical coordinate?"

Think of logical coordinates as your *abstract drawing units*. They're like measurements on a blueprint, not the actual physical dimensions of the final product.

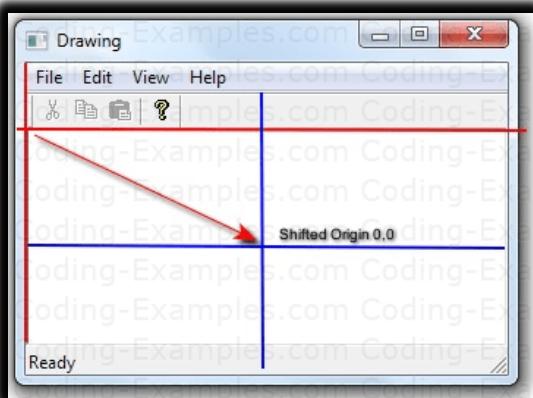
Mapping Mode: The Translation Guide

The "mapping mode" is the crucial setting that decides how your "**logical units**" (like that '50' or '100') get translated into actual "**physical pixels**" on your screen, or dots on a printer, or whatever device you're drawing to.

- **It's like a scale:** If your mapping mode says "1 logical unit = 1 pixel," then (50, 100) *will* be pixel (50, 100).
- **But it can change:** If your mapping mode says "1 logical unit = 2 pixels" (e.g., for scaling up), then logical (50, 100) would end up at physical pixel (100, 200)!

This allows your **drawings to adapt** to different screen resolutions or printers without you having to change all your drawing coordinates manually. You draw in your logical world, and the mapping mode handles the real-world translation.

 **Mapping mode** = how logical units translate into actual device pixels (screen dots, printer dots, etc.)



⚙️ MM_TEXT — The Default & Simple One

This is the mode Windows uses by default. It's like:

"Hey man, no need for translation, logical == physical. Just draw at (x, y) as pixels."

Behavior of MM_TEXT:

- (0, 0) is top-left of client area
- +X moves right
- +Y moves **down**
- Each logical unit = **1 pixel**

So this:

```
TextOut(hdc, 50, 100, L"Nick Was Here", 13);
```

↓ Literally means:

Draw the text starting **50 pixels from the left, 100 pixels down** from top.

```
// MM_TEXT is the default, but we can still be explicit:  
SetMapMode(hdc, MM_TEXT);  
TextOut(hdc, 100, 200, L"MM_TEXT mode", 12);
```

Other Mapping Modes — Scaling Comes In

If you're printing, drawing in inches/mm, or zooming? You'll want other modes.

Here's some spicy ones:

Mapping Mode	Logical Units	Origin	Y Axis	Use Case
MM_TEXT	1 unit = 1 pixel	Top-Left	Down	Default screen drawing
MM_LOENGLISH	0.01 inch (low res)	Bottom-Left	Up	Inches, low precision
MM_HIENGLISH	0.001 inch (high res)	Bottom-Left	Up	Fine print graphics
MM_LOMETRIC	0.1 mm	Bottom-Left	Up	Metric stuff
MM_HIMETRIC	0.01 mm	Bottom-Left	Up	High-res metric
MM_TWIPS	1/1440 inch	Bottom-Left	Up	Typography, publishing

 In all modes *except MM_TEXT*, the origin is **bottom-left**, and Y increases **upwards**.

- **Mapping Mode:** This is the specific setting you tell Windows to use. Each mode changes how your drawing commands (like TextOut(X, Y...)) are interpreted.
- **Logical Units:** This is the key! It tells you what one "unit" in your drawing code (e.g., if you draw at X=100, that 100 represents this many units) translates to in real-world measurements.

MM_TEXT (Default for Screens):

- This is the simplest and most common mode, especially for drawing directly on your computer screen. In MM_TEXT, **1 unit in your drawing code is exactly equal to 1 pixel** on the screen.
- So, if you tell your program to draw a line 100 units long, it will be 100 pixels long. This makes it very easy to position things precisely on a pixel grid for typical screen-based applications.

MM_LOENGLISH (Low-Res Inches):

- Imagine you're designing something where you want to specify sizes in inches, but without needing super tiny precision. In MM_LOENGLISH, **1 unit in your drawing code represents one-hundredth of an inch (0.01 inches)**.
- This means if you want to draw a shape that is exactly 1 inch wide, you would tell your program to draw it 100 units wide (because $100 \text{ units} * 0.01 \text{ inches/unit} = 1 \text{ inch}$). It's great for drawings where general inch measurements are useful, like simple diagrams.

MM_HIENGLISH (High-Res Inches):

- This mode is similar to MM_LOENGLISH, but it offers much finer detail when you're working with inches. In MM_HIENGLISH, **1 unit in your drawing code represents one-thousandth of an inch (0.001 inches)**.
- So, to draw something exactly 1 inch wide, you'd specify 1000 units in your code (because $1000 \text{ units} * 0.001 \text{ inches/unit} = 1 \text{ inch}$). This precision is crucial for professional graphics, detailed architectural plans, or when sending designs to high-resolution printers where every tiny fraction of an inch matters.

MM_LOMETRIC (Low-Res Millimeters):

- This is the metric version of MM_LOENGLISH. If you prefer to work with millimeters, this mode is for you. In MM_LOMETRIC, **1 unit in your drawing code represents one-tenth of a millimeter (0.1 mm)**.
- Therefore, if you want to draw an object that is exactly 10 millimeters (which is 1 centimeter) wide, you would tell your program to draw it 100 units wide (because $100 \text{ units} * 0.1 \text{ mm/unit} = 10 \text{ mm}$). It's good for general metric designs.

MM_HIMETRIC (High-Res Millimeters):

- This is the high-precision metric mode, mirroring MM_HIENGLISH. In MM_HIMETRIC, **1 unit in your drawing code represents one-hundredth of a millimeter (0.01 mm)**.
- To draw something that is 10 millimeters (1 cm) wide, you would need to specify 1000 units in your code. This mode is excellent for very detailed technical drawings, engineering designs, or for precise output on high-resolution metric printers.

MM_TWIPS (Tiny Typography Units):

- This mode uses a very specific unit called a "twip," which is commonly found in older desktop publishing and word processing applications. In MM_TWIPS, **1 unit in your drawing code is equal to 1/1440th of an inch.**
- This might sound odd, but it comes from the fact that there are 72 "points" in an inch (a common unit for font sizes), and there are 20 twips in a single point. So, $72 \text{ points/inch} * 20 \text{ twips/point} = 1440 \text{ twips/inch}$.
- This extremely small unit allows for incredibly fine and consistent control over text placement and layout, regardless of the screen's or printer's resolution. It ensures your text looks exactly the same size and spacing when printed as it does on screen, even across different devices.

Origin: This tells you where the (0,0) point of your drawing coordinate system is located within the window or page.

Top-Left: (0,0) is at the top-left corner.

Bottom-Left: (0,0) is at the bottom-left corner. (This is typical for print-oriented modes, like graphing).

Y Axis: This describes the direction of the Y-axis (vertical axis).

Down: As Y values increase, you move downwards on the screen (common for screen drawing).

Up: As Y values increase, you move upwards (common for graphing and print, where (0,0) is bottom-left).

Use Case: This gives you typical scenarios where each mapping mode would be beneficial.

In essence, these mapping modes give you the flexibility to draw in units that make sense for your output, whether it's pixels on a screen, inches for print, or tiny twips for precise text layout, letting Windows handle the conversion to actual physical dots.

Real-World Use Case: When Does This Matter?

You might think, "Why bother with logical coordinates if MM_TEXT (1 logical unit = 1 pixel) is the default and works fine for screens?"

This system becomes super useful when:

- **Print Previews:** You're showing how something will look on a printer. Printers have different resolutions (DPI - Dots Per Inch) than screens. Using modes like MM_HIENGLISH (which defines units in thousandths of an inch) or MM_TWIPS (1/1440 of an inch) lets you draw in real-world measurements (like inches or millimeters) so your design prints correctly, no matter the printer's DPI.
- **Multiple DPI Screens:** Your app needs to look good on screens with different pixel densities (e.g., a standard monitor vs. a high-resolution Retina display). Drawing in logical units ensures your elements scale properly.

Tip: Combine Mapping + Viewport for Advanced Control

Mapping modes set the *scale* (how big a logical unit is) and *direction* (e.g., is Y-axis up or down?). But you can also use these two functions to further manipulate your drawing space:

```
// Shift the origin (0,0) of your *physical* drawing area.  
SetViewportOrgEx();  
// Shift the origin (0,0) of your *logical* drawing area.  
SetWindowOrgEx();
```

- **SetViewportOrgEx()**: Moves where your logical (0,0) point *appears* on the physical screen. Good for panning your view.
- **SetWindowOrgEx()**: Moves what logical coordinates correspond to the physical (0,0) point. Good for offsetting your drawing.

Why use these? You can combine these to easily **pan, zoom, and scale your drawings** without having to rewrite all your drawing code or recalculate every single coordinate. It lets you create complex viewing features (like in a CAD program) with relative ease.

Summary Wrap-up: Quick Recap

- *TextOut and similar drawing functions use logical coordinates, not exact screen pixels.*
- *The mapping mode tells Windows how to convert your logical 'X' and 'Y' values into actual screen or printer pixels.*
- *MM_TEXT is the simplest mapping mode; 1 logical unit equals 1 pixel. Great for basic screen drawing.*
- *Other mapping modes (like MM_HIENGLISH or MM_TWIPS) let you draw using real-world units (inches, mm) or scalable units, which is vital for printouts or high-DPI displays.*
- *Always choose your mapping mode based on what you're drawing for (screen, printer) and whether you need automatic scaling.*

UNICODE SUPPORT

TextOut: Text and Unicode – A Quick Guide

Yes, TextOut can draw text with all sorts of characters, including international ones (Unicode). But there's a small detail based on how you call it:

- **TextOutA:** This is for "ANSI" text, the **older, simpler text** where each character typically takes up only **1 byte** of memory. It's usually limited to characters from a specific language or region.
- **TextOutW:** This is for "Unicode" text, the **modern way** to handle text, where each character often uses **2 or more bytes** (called "wide characters"). Unicode can represent almost every character from every language in the world, including emojis and special symbols.

TextOut supports Unicode character strings, allowing for the display of multilingual text. The number of bytes in a Unicode string is double the iLength value, and the string should not contain any ASCII control characters.

We looked at unicode in-depth in chapter 1.

If you're writing C/C++ code and using TextOut to display Unicode text (which is highly recommended for modern apps):

- **iLength (the length parameter):** When you tell TextOut how long your text is, this iLength value should *still be the number of characters, NOT the number of bytes*. Even though Unicode characters might take 2 bytes each, Windows automatically handles this detail for you.
- **For C/C++ users:** Because of this, when you're calculating the length of a Unicode string in C/C++, **don't use strlen()** - counts bytes (good for ANSI), **use wcslen()** correctly counts wide characters (good for Unicode).



💡 Important Note on Formatting:

- Characters like \n (new line), \r (carriage return), or \t (tab) **will NOT create line breaks or tabs** when you draw them with TextOut. TextOut just draws them as literal characters (sometimes as little squares) or simply ignores them.
- If you need **multi-line text**, you'll have to handle line breaks yourself by calling TextOut multiple times for each line.

🔗 Clipping Region and Its Impact on Text Rendering

The **clipping region** is one of those invisible superheroes in GDI — you usually don't see it, but it's protecting your graphics from chaos.

💡 What's the Clipping Region?

It's a **boundary** that Windows uses to say:

"You can only draw inside this area. Everything outside? Nope, not allowed."

This applies to **everything**, including:

- TextOut
- DrawText
- Rectangle, Ellipse, BitBlt...

The device context, which serves as the drawing context for a window, [also defines](#) a clipping region.

This region determines the area within which graphical elements, including text, will be rendered.

By default, the clipping region is set to the [entire client area for a device context handle](#) obtained using GetDC and to the invalid region for a device context handle obtained using BeginPaint.

When the TextOut function is called to display text, Windows will [only render](#) the portions of the character string that fall within the clipping region.

Any part of a character that [lies outside the clipping region](#) will not be displayed. Similarly, if a character partially overlaps the clipping boundary, only the portion of the character inside the region will be rendered.

This clipping mechanism ensures that text is displayed only within the visible boundaries of the window, [preventing it from extending beyond the client area](#) or [overlapping](#) with other graphical elements.

Two Main Scenarios for Clipping:

Function	What It Gives You
BeginPaint()	A drawing area limited only to what's damaged (the invalidated rectangle). Great for efficient repainting during <code>WM_PAINT</code> .
GetDC()	A drawing area that covers the entire window's client area . Use this for real-time or user-triggered drawing (like mouse moves).

BeginPaint() says: "**Just repaint the broken part; I got you covered.**"

GetDC() says: "**Here's the whole canvas — you do your thing, but clean up after.**"

⌚ Two Ways to Get a Device Context — and What You Can Draw

When you want to draw **anything** in WinAPI, you first need a **Device Context (DC)** — your window's personal canvas 🎨.

But *how* you get that DC? That changes **where** Windows *lets* you draw.

FunctionFlags	What It Gives You Access To	Used During	Default Clipping Region	Must Pair With
BeginPaint()	DC <i>only for the dirty area</i> (what needs redraw)	WM_PAINT	🚫 Only the invalid rectangle	<input checked="" type="checkbox"/> EndPaint()
GetDC()	DC for the <i>entire client area</i>	Outside WM_PAINT	🟢 Whole client area	<input checked="" type="checkbox"/> ReleaseDC()

⭐ Why Clipping Is Useful:

- *Keeps your UI clean — text doesn't spill over borders.*
 - *Speeds up painting — Windows can skip drawing clipped-out pixels.*
 - *Makes redraws more efficient — you can redraw only the parts that changed.*
-

 **TLDR:**

- ✓ TextOut supports Unicode — just make sure you pass wchar_t* strings and use iLength as **character count**, not bytes.

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch (msg) {
        case WM_PAINT: {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);

            // Unicode text
            wchar_t* text = L"Hello, 🌎 Unicode!";
            TextOutW(hdc, 50, 50, text, wcslen(text)); // Notice: character count

            EndPaint(hWnd, &ps);
            break;
        }
        case WM_DESTROY: PostQuitMessage(0); break;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

Clipping Region: Draw inside it only.

The **clipping region** of an HDC controls where text and graphics can appear.

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch (msg) {
        case WM_PAINT: {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);

            // Create a tiny clipping region
            HRGN hRgn = CreateRectRgn(0, 0, 100, 100);
            SelectClipRgn(hdc, hRgn);

            // Try to draw inside and outside the region
            TextOutA(hdc, 10, 10, "Visible", 7);    // Inside: will show
            TextOutA(hdc, 200, 200, "Ignored", 7); // Outside: will not show

            DeleteObject(hRgn);
            EndPaint(hWnd, &ps);
            break;
        }
        case WM_DESTROY: PostQuitMessage(0); break;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

Anything outside the clipping region is automatically ignored — *no need to manually crop.*

- ✓ If you use BeginPaint, you're limited to the **invalid region**.

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch (msg) {
        case WM_LBUTTONDOWN:
            // Force a partial redraw of the top-left only
            InvalidateRect(hWnd, &(RECT){0, 0, 100, 100}, TRUE);
            break;

        case WM_PAINT:
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);

            // Entire screen draw attempt – but only the invalid region will show it
            for (int y = 0; y < 300; y += 20) {
                TextOutA(hdc, 10, y, "Hello", 5);
            }

            EndPaint(hWnd, &ps);
            break;
    }
    case WM_DESTROY: PostQuitMessage(0); break;
}
return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

- ✓ If you use GetDC, you can draw on the **entire client area**.

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch (msg) {
        case WM_KEYDOWN:
            HDC hdc = GetDC(hWnd); // No clipping – full access to client area
            TextOutA(hdc, 100, 200, "Drawn with GetDC!", 18);
            ReleaseDC(hWnd, hdc);
            break;
    }
    case WM_DESTROY: PostQuitMessage(0); break;
}
return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

SYSTEM FONT

The **device context also defines** the default font that Windows uses when displaying text using the TextOut function.

This **default font** is known as the "system font" or, using the identifier defined in the WINGDI.H header file, **SYSTEM_FONT**.

Hi, I'm Arial!
I'm a system font!

The **system font** is the font that Windows employs by default for text strings in title bars, menus, and dialog boxes.

It serves as a **baseline font** for text rendering and is commonly used throughout the Windows user interface.

Evolution of the System Font: From Fixed-Pitch to Variable-Pitch

In the early versions of Windows, the system font was a **fixed-pitch font**, meaning that all characters had the same width.

This was similar to the font used in **typewriters**.

However, with the release of Windows 3.0, the system font transitioned to a **variable-pitch font**.



Variable-pitch fonts allow different characters to have **different widths**, reflecting their natural shapes and sizes.

This change from fixed-pitch to variable-pitch fonts was driven by research indicating that text displayed in **variable-pitch fonts is more readable** and visually appealing.

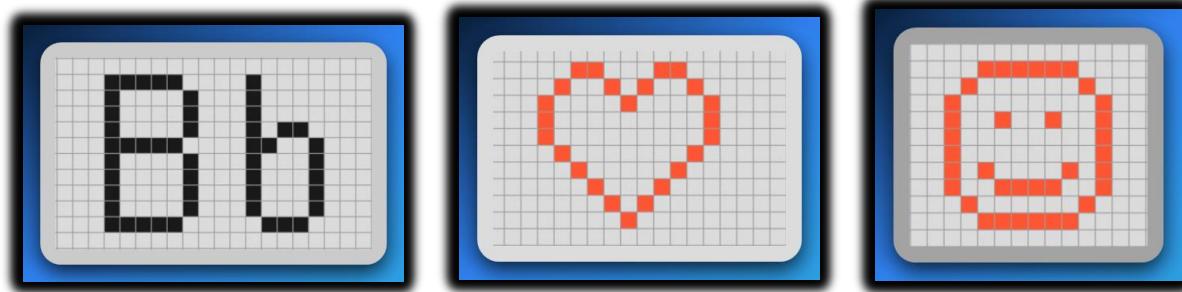
Impact of Variable-Pitch Fonts on Programming

The adoption of variable-pitch fonts necessitated adjustments in **programming practices**.

Developers had to **adapt their code** to account for the varying widths of characters and ensure proper text alignment and formatting.

Raster Fonts vs. Vector Fonts: A Distinction

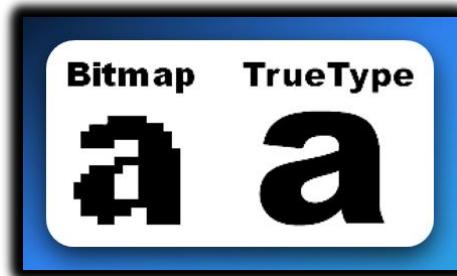
The **system font** is a **raster font**, meaning that the characters are defined as blocks of pixels. This definition is tied to a specific resolution and may not scale well to different display sizes.



In contrast, TrueType fonts, which we'll explore in the last Chapters, are defined by scalable outlines.

TrueType fonts are **vector-based fonts**, meaning that they are defined by mathematical formulas rather than fixed bitmaps.

They can be rendered at various resolutions without losing quality, making them more versatile for various display sizes and text rendering scenarios.



Raster Fonts vs. TrueType Fonts: A Distinction



This title I had sounded wrong....

...which *technically* compares a rendering method (raster) against a font format (TrueType), not apples-to-apples.

Better Heading Options:

To stay crystal-clear and avoid confusion:

- Raster Fonts vs. Vector Fonts
- Bitmap Fonts vs. Scalable Fonts
- Font Rendering: Raster vs. Vector (TrueType) ← most precise
- Understanding Font Types: Raster vs. TrueType (Vector)

Because TrueType is a *vector-based font format*, not the vector rendering method itself. But it's what people usually mean when they say "vector fonts" in the Windows world.

Conclusion

The device context's **clipping region** and the **system font** play crucial roles in text *rendering* within Windows applications.

The **clipping region** ensures that text is displayed only within the visible boundaries of the window, while the **system font** provides a default text rendering style. Both are for visual appeal during GUI development.

The **TextOut function** is an essential tool for displaying text in Windows applications. Its *straightforward syntax* and *integration with device context attributes* enable programmers to create visually appealing and informative user interfaces.

Understanding the concept of **mapping modes** and **Unicode support** further enhances the versatility of this function.

Also, read the GDI documentation.

Let's move to the next beast! 😊 😊

CHARACTER DIMENSIONS AND TEXT RENDERING

When displaying [multiple lines of text](#) using the `TextOut` function, it is crucial to determine the dimensions of the characters in the font.

This information allows for [proper spacing](#) between lines and columns of text, ensuring a visually appealing and readable layout.

Dynamic Character Dimensions: Why Text Isn't One-Size-Fits-All

Characters in Windows aren't born with fixed widths and heights — their size *changes* depending on two key factors:

1.  **Display Resolution**

Higher resolutions (like 1024×768 or 1920×1080) pack more pixels into the same screen space. That means each character might appear smaller or tighter, depending on how many pixels are available.

2.  **Font Size & Type**

Users (and apps) can select different system fonts or font sizes. Some fonts are chunkier, others are slim and tall. Combine that with the resolution, and the result? Every character's dimensions are dynamic.



Why It Matters

When you're doing pixel-precise layouts or custom text rendering, you **can't assume** a fixed character width or height — not even for the letter 'M'. Always measure it using functions like `GetTextMetrics()` or `GetTextExtentPoint32()` to get the actual dimensions **on that specific device context**.

Determining Character Dimensions Using GetTextMetrics

If you want to know [how big your characters actually are](#), you don't guess. You ask Windows directly.

To determine the character dimensions for a specific font, a program can utilize the [GetTextMetrics](#) function.

This function requires a [handle to the device context](#), as it retrieves information about the font currently selected in that context.

The retrieved information is stored in a [TEXTMETRIC structure](#), which contains various fields related to font metrics.

Key Fields of the TEXTMETRIC Structure. Among the 20 fields in the TEXTMETRIC structure, the following seven are particularly relevant for character dimensions:

- **tmHeight:** Represents the total height of a character, including both the ascent and descent.
- **tmAscent:** Indicates the portion of the character that extends above the baseline.
- **tmDescent:** Represents the portion of the character that extends below the baseline.
- **tmInternalLeading:** Refers to the additional spacing between lines of text within the tmHeight boundary.
- **tmExternalLeading:** Represents the additional spacing between lines of text outside the tmHeight boundary.
- **tmAveCharWidth:** Indicates the average width of characters in the font.
- **tmMaxCharWidth:** Represents the width of the widest character in the font.

Units of Measurement for Character Dimensions

The values in the TEXTMETRIC structure are measured in **units** based on the mapping mode **currently selected** for the device context.

In the **default device context**, the mapping mode is [MM_TEXT](#), meaning the dimensions are in pixels.

Code Example for Retrieving Character Dimensions

The following code snippet demonstrates how to retrieve character dimensions using GetTextMetrics:

```
// Get current font info (like height/width).
// Needed to space lines, align UI.

// 1. Get drawing canvas (Device Context).
HDC hdc = GetDC(hWnd);

// 2. Container for font details.
TEXTMETRIC tm;

// 3. Fill container with font details from HDC.
GetTextMetrics(hdc, &tm);

// tm.tmHeight;           // Total height of a character cell.
// tm.tmAveCharWidth;    // Average character width.
// tm.tmExternalLeading; // Recommended extra space between lines.

// 4. Release drawing canvas.
ReleaseDC(hWnd, hdc);
```

Another one:

```
#include <windows.h>
typedef struct tagTEXTMETRIC {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    // Add other structure fields as needed
} TEXTMETRIC, *PTEXTMETRIC;

int main() {
    HDC hdc = GetDC(0); // Get the device context of the screen

    LOGFONT lf;
    GetObject(GetStockObject(DEFAULT_GUI_FONT), sizeof(LOGFONT), &lf);

    HFONT hFont = CreateFontIndirect(&lf);
    SelectObject(hdc, hFont);

    TEXTMETRIC tm;
    GetTextMetrics(hdc, &tm);

    printf("Text Metrics:\n");
    printf(" Height: %ld\n", tm.tmHeight);
    printf(" Ascent: %ld\n", tm.tmAscent);
    printf(" Descent: %ld\n", tm.tmDescent);
    printf(" Internal Leading: %ld\n", tm.tmInternalLeading);
    printf(" External Leading: %ld\n", tm.tmExternalLeading);
    printf(" Average Character Width: %ld\n", tm.tmAveCharWidth);
    printf(" Maximum Character Width: %ld\n", tm.tmMaxCharWidth);

    DeleteObject(hFont);
    ReleaseDC(0, hdc);
    return 0;
}
```

This code uses the Windows API to **obtain the default font** on the system, creates a font from that information, selects it into a device context, and then **retrieves the text metrics** using GetTextMetrics.

Finally, it **prints out** the various fields of the TEXTMETRIC structure.

Please note that this code assumes you are working in a **Windows environment**, as it relies on Windows API functions.

DELVING INTO THE TEXTMETRIC STRUCTURE

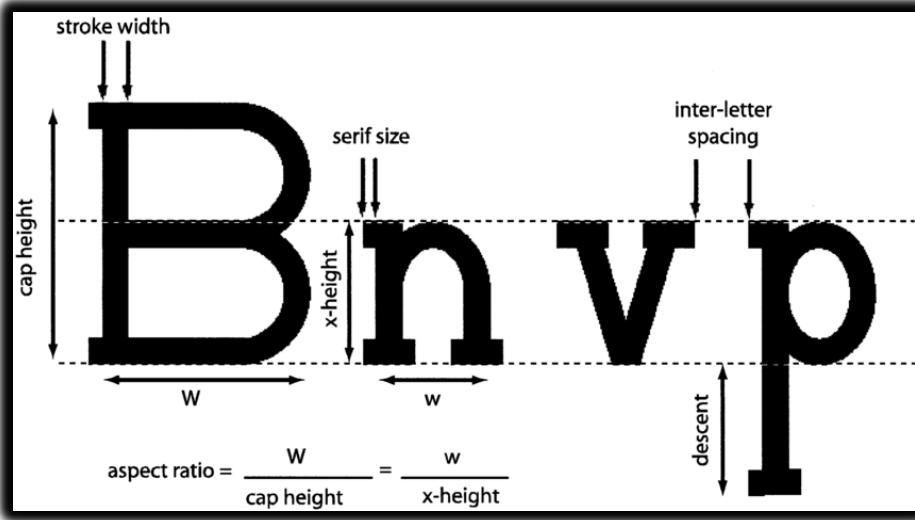
The **TEXTMETRIC structure** provides comprehensive information about the font currently selected in the device context.

Among its various fields, four are particularly relevant for understanding character dimensions: tmHeight, tmAscent, tmDescent, and tmInternalLeading.

tmHeight: Defining the Overall Character Height

The **tmHeight field** represents the total height of a character, encompassing both the **ascent** if the letter has eg (b, d, h, k) or **descent** (p, q, g, y).

It serves as a crucial parameter for determining the **vertical spacing** between lines of text.



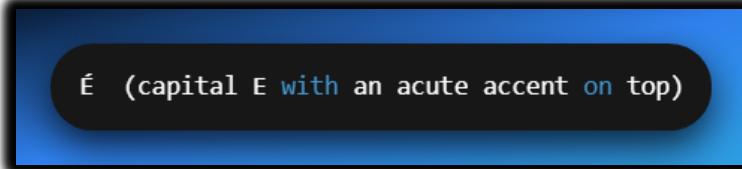
tmAscent and tmDescent: Character Extents Above and Below the Baseline

The **tmAscent field** indicates the portion of the character that extends above the baseline, while **tmDescent** represents the portion that extends below the baseline. These values are essential for **proper positioning** of text relative to the baseline.

tmInternalLeading: Accounting for Accent Marks

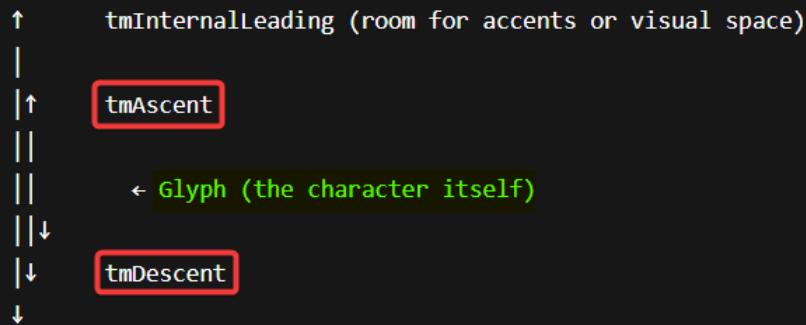
tmInternalLeading refers to the additional spacing between lines of text within the tmHeight boundary. This spacing typically accommodates accent marks, which are small characters placed above or below other characters.

We're talking about this 😊 (I know you were confused for a sec):



É (capital E with an acute accent on top)

Let's visualize it:



External Leading: An Optional Spacing Suggestion

The TEXTMETRIC structure also includes a field named **tmExternalLeading**, which represents an additional spacing suggestion from the font designer.

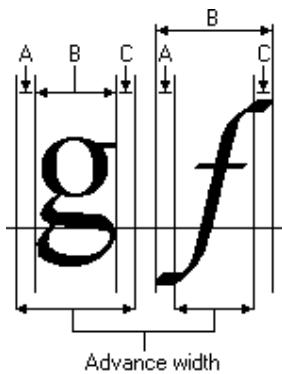
This spacing is intended to be **added between successive rows of displayed text**. Programmers have the flexibility to accept or reject this suggestion based on their desired text layout.

"Hey, maybe leave a little breathing room between lines, just for style or readability."

Character Widths: Average and Maximum

Character widths are represented by two fields in the TEXTMETRIC structure:

- ✓ **tmAveCharWidth** indicates the average width of lowercase characters.
- ✓ **tmMaxCharWidth** represents the width of the widest character in the font.



Calculating Uppercase Character Width

For the sample programs in this chapter, the **average width of uppercase letters** is required. This value can be approximated by calculating 150% of tmAveCharWidth.

Dynamic Character Dimensions: Adapting to Display Size and Font Selection

It is crucial to recognize that the dimensions of a system font are not static and can vary depending on the pixel size of the video display on which Windows runs.

Additionally, some users may have **customized the system font size**, further influencing character dimensions.

Importance of Accurate Character Dimensions

Relying on **guesswork** or **hard-coded values** for character dimensions can lead to inconsistent and *visually unappealing* text layouts across different display setups and user preferences.



Role of GetTextMetrics in Obtaining Accurate Character Dimensions

The **GetTextMetrics function** provides a reliable and device-independent mechanism for *retrieving accurate character dimensions*.

By utilizing this function, programmers can ensure that their **text rendering adapts** to various display configurations and font selections.

The TEXTMETRIC structure offers **valuable insights into font metrics**, particularly character dimensions.

GetTextMetrics function empowers programmers to create **visually consistent, simple** and **user-friendly** text layouts.

Efficient Utilization of GetTextMetrics for Text Formatting

Since the dimensions of the system font remain constant throughout a Windows session, it is **unnecessary to repeatedly call** GetTextMetrics.

A single call during program initialization is sufficient.

An **ideal location for this call** is within the window procedure's response to the WM_CREATE message, the first message received by the window procedure.

Defining Variables for Character Dimensions

When you're planning to display **multiple lines of text** in a window, you need precise measurements to make it look clean and consistent — no sloppy overlapping or misaligned rows.

Key Variables:

- **cxChar:** Average character width
- **cyChar:** Total character height (includes ascent, descent, internal spacing)

These two values help you calculate:

- *How far apart each character should be placed horizontally*
- *How much space to leave between each line of text*

Why Store Them as static?

Inside the WndProc, you want to store these variables **once** when the window is created and reuse them later — especially during WM_PAINT.

So you declare them as static variables **within the window procedure**:

```
static int cxChar, cyChar;
```

When to Initialize Them? → WM_CREATE

WM_CREATE is fired **once**, when your window is being created.
It's the perfect time to ask Windows:

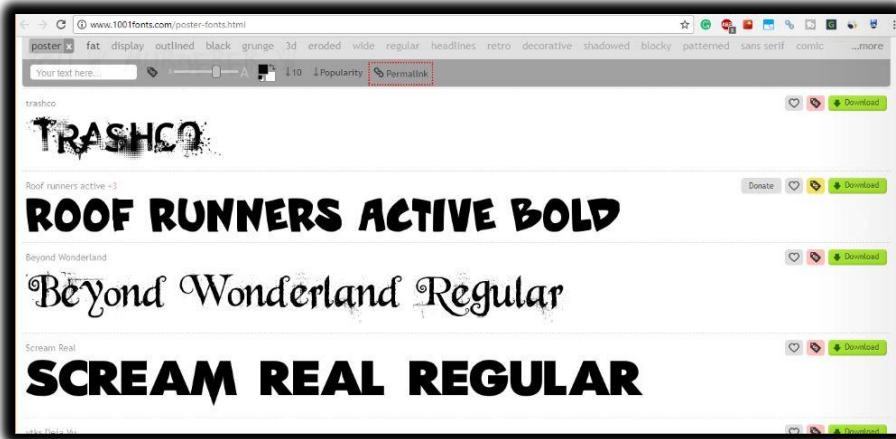
"Hey, what's the width and height of characters in the current font?"

Here's a clean example of how you'd do it:

```
case WM_CREATE:  
{  
    HDC hdc;           // # Drawing context.  
    TEXTMETRIC tm;     // # Font details container.  
  
    hdc = GetDC(hwnd);          // # Get window's DC.  
    GetTextMetrics(hdc, &tm);    // # Get font metrics into 'tm'.  
  
    cxChar = tm.tmAveCharWidth;   // # Avg char width for 'X' movement.  
    cyChar = tm.tmHeight + tm.tmExternalLeading; // # Total line height for 'Y' spacing.  
  
    ReleaseDC(hwnd, hdc);       // # Release DC.  
    return 0;  
}
```

That extra bit **tmExternalLeading** is optional spacing suggested by the font.

It ensures **lines don't feel cramped**, especially in fonts with extra height.



These values can be **stored in static variables** defined within the window procedure:

The static variables **cxChar** and **cyChar** are defined within the WM_CREATE message handler of the window procedure.

This message handler (WM_CREATE) is executed when the window is created, and it's an appropriate place to initialize these variables since they need to be valid for subsequent message processing, such as WM_PAINT.

Let's decode this slowly:

- "**This message handler**" → means the code inside case WM_CREATE: in your WndProc.
- "**executed when the window is created**" → Windows sends WM_CREATE **once** when it builds your window for the first time. It's your setup zone.
- "**initialize these variables**" → You grab character width & height now (cxChar, cyChar), because fonts are ready by then.
- "**need to be valid for subsequent message processing**" → Once the window is up, you'll get future messages like WM_PAINT, where you **use** those values to draw text correctly.

So in plain English:

- **WM_CREATE** happens right when your window is being born. It's the *perfect time to calculate your text measurements* and store them for later use when drawing stuff in WM_PAINT.

*"With cxChar and cyChar set, you're equipped to: Place text accurately in WM_PAINT, Loop through lines using clean math like $y = i * cyChar$, Build consistent, scalable UIs"*

Let's reframe that in simpler blocks:

- cxChar lets you know how wide each letter is → helps with spacing text horizontally.
- cyChar lets you know how tall each line is → helps with vertical layout (line by line).

With these:

- You can **print line 1 at $y = 0$** , line 2 at $y = 1 * cyChar$, line 3 at $y = 2 * cyChar$, and so on. ↗
- It keeps your text **evenly spaced** — no guesswork or overlapping.
- It works **no matter the font size**, screen DPI, or resolution.

So rewritten punchier:

"Once you've measured the average character size, you can paint text line by line with perfect spacing, using clean formulas like $y = \text{lineIndex} * cyChar$. Your UI becomes clean, resizable, and adaptive."

FORMATTING TEXT IN WINDOWS

In Windows programming, **formatting text** involves using functions like GetTextMetrics, TextOut, wsprintf, and sprintf to control the appearance of text displayed on the screen.

These functions allow you to **modify font characteristics**, *line spacing*, and *text alignment*.



Before anything else, just visit the static document, understand what the notes say, then we can move on.

Obtaining Character Metrics

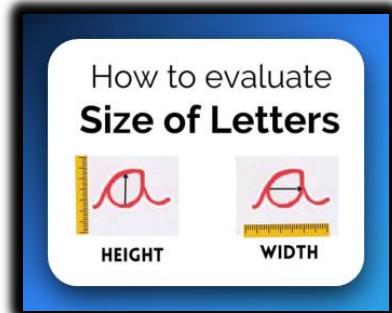
The **GetTextMetrics function** retrieves information about the system font, including character width and height.

This information is crucial for **positioning text** accurately and ensuring proper line spacing.

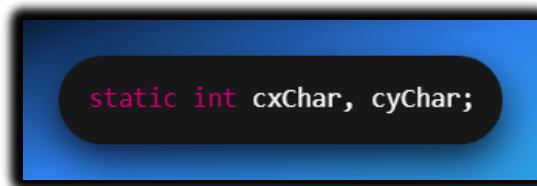
The function takes a **device context handle (HDC)** as **input** and **returns** a **TEXTMETRIC** structure containing relevant font metrics.

Storing Character Metrics

The **cxChar** and **cyChar** variables are used to store the **average** character width and **total** character height, respectively.



These variables are defined as **static** to maintain their values across multiple message processing calls.



It means:

- These variables are created **once** when your program starts.
- They live in the **data segment** of memory (not the stack, not the heap).
- They **retain their values** between function calls.
- They're **private** to the function, but **persistent** throughout the program's lifetime.

Analogy: Stack vs Static

Type	Memory Zone	Lives Until...	Value Reset Each Call?
Regular	Stack	Function ends	<input checked="" type="checkbox"/> Yes
Static	Data Segment	Program exits	<input type="checkbox"/> No (persists!)

Why This is Perfect in WndProc

```
switch(message)
{
    case WM_CREATE: {
        static int cxchar, cychar;
        // initialize once here using GetTextMetrics
    }
}
```

You calculate these values **once** when the window is created, and they **stick around** for use during future WM_PAINT calls, even though those are separate function calls.

So yeah — you're right:

- The value is stored in the **data segment**.
- It **doesn't get wiped** like stack variables.
- It's like giving your window a **private little "memory locker"** that survives message storms. 

"static" in WndProc lets you store values that need to survive across multiple message calls (like cxChar, cyChar). It's stored in the data segment, and initialized only once — perfect for performance and consistency.

WM_CREATE Message Handling

The **WM_CREATE** message is the **very first message** your window gets after it's created.

Think of it as your "**window birth event**" — this is where you set up your internal layout logic, dimensions, and any core variables your UI depends on.

It's an **ideal place** to initialize the text formatting variables. The WM_CREATE message handler typically performs the following steps:

Why handle WM_CREATE?

It's the **ideal place** to:

- Fetch the font size and character dimensions.
- Set up stuff you'll reuse during WM_PAINT (like cxChar and cyChar).
- Prepare any layout logic before the user sees your window.

Typical Setup Steps (with Pro Tips)

1) Get the Device Context (HDC)

Grab a DC with `GetDC(hwnd)` to access font metrics.

2) Measure the Text

Use `GetTextMetrics(hdc, &tm)` to fill in a `TEXTMETRIC` struct.

3) Store the Key Metrics

Pull out what you need:

```
cxChar = tm.tmAveCharWidth;  
cyChar = tm.tmHeight + tm.tmExternalLeading;
```

4) Release the DC

Don't leak GDI objects — always `ReleaseDC(hwnd, hdc);`

5) Return 0

If everything's good.

Displaying Formatted Text (Simple Output Example)

To show calculated or formatted values in the window, use:

- **wsprintf**: Think of it like sprintf, but Windows-style.
- **TextOut**: Actually draws text on the screen at X, Y.

```
TCHAR buffer[64];
wsprintf(buffer, TEXT("Sum = %d"), 5 + 3);
TextOut(hdc, x, y, buffer, lstrlen(buffer));
```

 This prints “Sum = 8” on your window at (x, y). OR

```
int iLength;
TCHAR szBuffer[40];

// Other program lines

iLength = wsprintf(szBuffer, TEXT("The sum of %i and %i is %i"), iA, iB, iA + iB);
TextOut(hdc, x, y, szBuffer, iLength);
```

Fancy One-Liner Version

If you’re flexing for code minimalism:

```
TextOut(hdc, x, y, szBuffer, wsprintf(szBuffer, TEXT("The sum of %i and %i is %i"), iA, iB, iA + iB));
```

This approach directly passes the formatted string to TextOut without the need for an intermediate buffer.

```
TextOut(hdc, x, y, TEXT("Sum = 8"), lstrlen(TEXT("Sum = 8")));
```

- ✓ No intermediate buffer
- ✗ But not dynamic — not good if the value changes.

Summary: It's all for Visual Appeal

💬 "If you want your text to look clean and styled in a Windows app, you need to use special functions to choose fonts, adjust spacing, and align the text properly."

💡 Example: Want bold text? Smaller line gaps? Centered words? You use functions like CreateFont, SetTextAlign, and spacing tricks.

💬 "Before drawing text on the screen, you can use GetTextMetrics to find out the size of letters — how tall, how wide — so your layout doesn't look messy."

💡 Visualize it like this: You don't want one word to be higher than the next, or overlapping. GetTextMetrics gives you those exact measurements — like a ruler for letters.

💬 "When you want to show changing info on the screen (like scores or messages), use wsprintf to format the text, and then TextOut to draw it. Together, they show your data in a nice way."

💡 Think of it like this:

```
wsprintf(buf, L"Score: %d");
Textout(hdc, x, y, buf, lstrlen(buf));
```

wsprintf is a Windows API function that formats and stores a series of characters and values into a buffer.

lstrlen is a Windows API function that determines the length of a specified null-terminated string, *excluding* the terminating null character.

Goal	Function(s)	What It Really Does
Style fonts, spacing, alignment	CreateFont, SetTextAlign, etc.	Control how text looks
Measure text size	GetTextMetrics	Helps with layout — keeps things neat
Show changing text (like score)	wsprintf + TextOut	Format and draw it on screen

❖ TL;DR – Quick Table again! 😊

What	Why It Matters
WM_CREATE	Perfect for one-time setup work like text size.
GetTextMetrics	Gives you height and width for clean layout.
wsprintf + TextOut	Great duo for rendering dynamic info.

CREATING A HEADER FILE FOR GETSYSTEMMETRICS INFORMATION

The provided code snippet below will introduce the concept of creating a header file named **SYSMETS.H** to manage the information retrieved from the GetSystemMetrics function.

This header file defines an **array of structures** containing both the GetSystemMetrics index identifier and the corresponding text for each value returned by the function.

Structure Definition for GetSystemMetrics Information

The SMETRICS structure is defined to hold the GetSystemMetrics index identifier and the associated text:

```
// SYSMETS.H
typedef struct {
    int smIndex;           // Index for GetSystemMetrics (e.g., SM_CXSCREEN)
    const char *szLabel;   // Human-readable label
} SMETRICS;
```

🧠 Real Talk:

This structure is like a combo meal:

- **smIndex** holds the ID for a system metric retrieved by GetSystemMetrics.
- **szLabel** provides a descriptive name to display alongside it. Together, they let you fetch and clearly present specific Windows system measurements.

◆ Step 2: Create an array of these metric-label combos

```
// Still in SYSMETS.H
SMETRICS sysmetrics[] = {
    { SM_CXSCREEN, "Screen Width (pixels)" },
    { SM_CYSCREEN, "Screen Height (pixels)" },
    { SM_CMONITORS, "Monitor Count" },
    { SM_MOUSEPRESENT, "Mouse Connected?" },
    { SM_CXBORDER, "Border Width (pixels)" },
    { SM_CYBORDER, "Border Height (pixels)" }
    // + Add more if needed
};
```

This structure creates an array where each entry **pairs** a Windows system metric ID (like SM_CXSCREEN) with a user-friendly text label. It acts as a **lookup table**, defining which system statistics to retrieve and how to present them.

"Hey Windows, give me metric #SM_CXSCREEN, and I'll show the user this label: 'Screen Width (pixels)'"

◆ Step 3: Use it in your main.c file

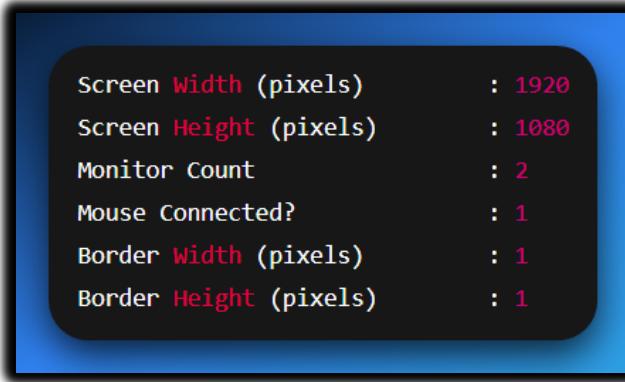
```
#include <windows.h>
#include <stdio.h>
#include "sysmets.h" // your custom header

int main() {
    int count = sizeof(sysmetrics) / sizeof(SMETRICS);
    for (int i = 0; i < count; i++) {
        int value = GetSystemMetrics(sysmetrics[i].smIndex);
        printf("%-30s: %d\n", sysmetrics[i].szLabel, value);
    }
    return 0;
}
```

In our code right here, we have a sysmets.h from the repository at oreilly's website:

https://resources.oreilly.com/examples/9781572319950/blob/master/cd_contents/Chap07/SysMets/SysMets.h

Sample output:



This is my own custom sysmets.h, the one in the folder is from oreilly's site.

```
/* SYSMETS.H - System metrics display structure */
#ifndef SYSMETS_H // Prevents multiple inclusions of this header file.
#define SYSMETS_H // Defines the header to prevent re-inclusion.

#include <tchar.h> // Required for TCHAR data type.

// Structure for system metric entries.
typedef struct {
    int     iIndex; // Index of the system metric.
    TCHAR* szLabel; // Metric's label (e.g., "SM_CXSCREEN").
    TCHAR* szDesc; // Description of the metric.
} SYSMETRIC_ENTRY;

// Calculates number of lines in the sysmetrics array.
#define NUMLINES ((int)(sizeof(sysmetrics) / sizeof(sysmetrics[0])))

// Array of SYSMETRIC_ENTRY structures.
// Each entry holds a system metric's ID, label, and description.
SYSMETRIC_ENTRY sysmetrics[] = {
    { SM_CXSCREEN,   TEXT("SM_CXSCREEN"),   TEXT("Screen width in pixels") },           // Screen width.
    { SM_CYSCREEN,   TEXT("SM_CYSCREEN"),   TEXT("Screen height in pixels") },            // Screen height.
    { SM_CXVSCROLL,  TEXT("SM_CXVSCROLL"),  TEXT("Vertical scroll width") },             // Vertical scroll width.
    { SM_CYHSCROLL,  TEXT("SM_CYHSCROLL"),  TEXT("Horizontal scroll height") },           // Horizontal scroll height.
    { SM_CYCAPTION,  TEXT("SM_CYCAPTION"),  TEXT("Caption bar height") },                 // Caption bar height.
    { SM_CXBORDER,   TEXT("SM_CXBORDER"),   TEXT("Window border width") },                // Window border width.
    { SM_CYBORDER,   TEXT("SM_CYBORDER"),   TEXT("Window border height") },               // Window border height.
    { SM_CXFIXEDFRAME, TEXT("SM_CXFIXEDFRAME"), TEXT("Dialog window frame width") },      // Dialog frame width.
    { SM_CYFIXEDFRAME, TEXT("SM_CYFIXEDFRAME"), TEXT("Dialog window frame height") },     // Dialog frame height.
    // ... Other metric entries ...
};

#endif // SYSMETS_H
```

You include it in your header section but use “ “ not <> as shown below:

```

490 #include <windows.h>
491 #include "sysmets.h"
492
493 LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
494
495 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
496     static TCHAR szAppName[] = TEXT("SysMets1");
497     HWND hwnd;
498     MSG msg;
499     WNDCLASS wndclass;
500
501     wndclass.style = CS_HREDRAW | CS_VREDRAW;
502     wndclass.lpfWndProc = WndProc;
503     wndclass.cbClsExtra = 0;
504     wndclass.cbWndExtra = 0;
505     wndclass.hInstance = hInstance;
506     wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
507     wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
508     wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
509     wndclass.lpszMenuName = NULL;
510     wndclass.lpszClassName = szAppName;
511
512     if (!RegisterClass(&wndclass)) {
513         MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
514         return 0;
515     }
516
517     hwnd = CreateWindow(szAppName, TEXT("Get System Metrics No. 1"), WS_OVERLAPPEDWINDOW,
518                         CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
519                         NULL, NULL, hInstance, NULL);
520
521     ShowWindow(hwnd, iCmdShow);
522     UpdateWindow(hwnd);
523
524     while (GetMessage(&msg, NULL, 0, 0)) {
525         TranslateMessage(&msg);
526         DispatchMessage(&msg);
527     }
528
529     return msg.wParam;
530 }
531
532 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
533     static int cxChar, cxCaps, cyChar;
534     HDC hdc;
535     int i;
536     PAINTSTRUCT ps;
537     TCHAR szBuffer[10];
538     TEXTMETRIC tm;
539
540     switch (message) {
541     case WM_CREATE:
542         hdc = GetDC(hwnd);
543         GetTextMetrics(hdc, &tm);
544         cxChar = tm.tmAveCharWidth;
545         cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
546         cyChar = tm.tmHeight + tm.tmExternalLeading;
547         ReleaseDC(hwnd, hdc);
548         return 0;
549
550     case WM_PAINT:
551         hdc = BeginPaint(hwnd, &ps);
552         for (i = 0; i < NUMLINES; i++) {
553             TextOut(hdc, 0, cyChar * i, sysmetrics[i].szLabel, lstrlen(sysmetrics[i].szLabel));
554             TextOut(hdc, 22 * cxCaps, cyChar * i, sysmetrics[i].szDesc, lstrlen(sysmetrics[i].szDesc));
555             SetTextAlign(hdc, TA_RIGHT | TA_TOP);
556             TextOut(hdc, 22 * cxCaps + 40 * cxChar, cyChar * i, szBuffer,
557                     wsprintf(szBuffer, TEXT("%5d"), GetSystemMetrics(sysmetrics[i].iIndex)));
558             SetTextAlign(hdc, TA_LEFT | TA_TOP);
559         }
560         EndPaint(hwnd, &ps);
561         return 0;
562
563     case WM_DESTROY:
564         PostQuitMessage(0);
565         return 0;
566     }
567
568     return DefWindowProc(hwnd, message, wParam, lParam);
569 }

```

The **SYSMETS1.C** source code file above contains the implementation of the **SYSMETS** program, which displays various system metrics on the screen.

The program utilizes the **GetSystemMetrics function** to retrieve information about various graphical elements, such as icon sizes, scroll bar dimensions, and window borders.

The code explained for those who still don't get it fully.

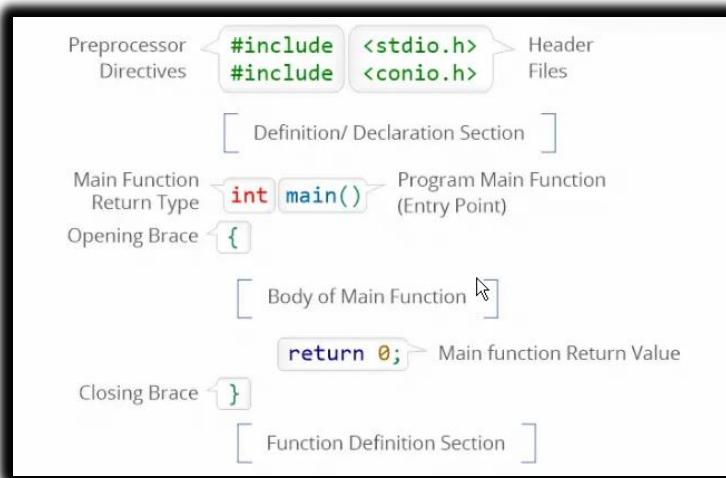
The WinMain function serves as the entry point for the program. It performs the following tasks:

Register the window class: This step defines the characteristics of the program's window, including its style, icon, and cursor.

Create the window: Using the registered window class, WinMain creates the program's window and assigns a handle to it.

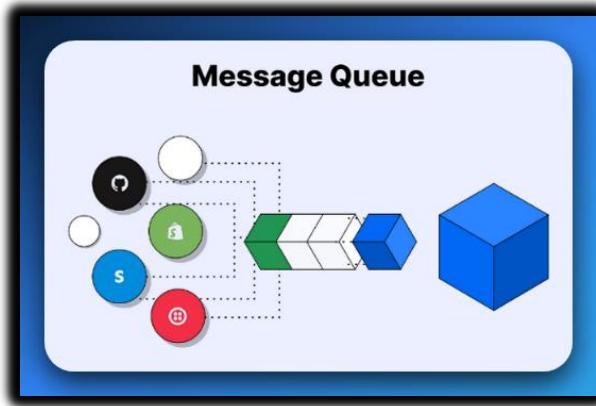
Display the window: The window is made visible using the ShowWindow function.

Enter the message loop: The program enters a message loop, where it continuously processes messages received from the operating system.



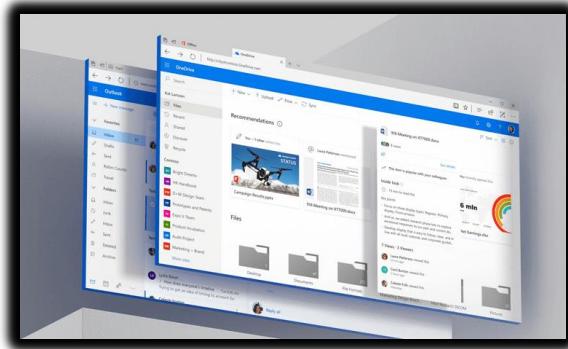
The **WndProc function** is responsible for handling messages sent to the program's window. It processes various messages, including:

- **WM_CREATE:** This message is received when the window is created. The SYSMETS program uses this message to initialize the text formatting variables.
- **WM_PAINT:** This message is received when the window needs to be repainted. The SYSMETS program uses this message to retrieve system metrics using GetSystemMetrics and display the information on the screen using TextOut.
- **WM_DESTROY:** This message is received when the window is destroyed. The SYSMETS program uses this message to perform any necessary cleanup tasks.



Relationship to Previous Code

The **WinMain function** in SYSMETS1 is **similar** to the one in HELLOWIN, as both involve registering the window class, creating the window, and displaying it.



The **WndProc function** in SYSMETS1 incorporates elements discussed in previous chapters, such as obtaining character metrics and using TextOut for formatted text output.

In short, the **SYSMETS1 program** demonstrates the practical application of concepts covered in earlier chapters.

The program runs well in Visual studio community 2022.

The "Not Enough Room" Problem

The SYSMETS1 program displays a [list of system metrics](#) along with their **corresponding values**.

Due to limitations in how the program determines the available space for displaying text, the entire list may not be visible small resolution screens.



The primary issue lies in the program's reliance on the default behavior of Windows to clip text that extends beyond the client area of the window.

This approach assumes that the program has sufficient space to display the entire list, which may not be the case on all systems or window configurations.

Proposed Solution

To address this issue, the program needs to [explicitly determine the available space](#) within the client area before attempting to display the text.

This can be achieved by [retrieving the dimensions of the client area](#) using the GetClientRect function.

Implementation Details

Calculate Display Area: Use GetClientRect to get the client area's width and height before showing system metrics.



Position Text: Adjust text placement based on available space to fit within the visible window, potentially changing vertical position or truncating the list.



Manage Excess Content: If the list is too long, notify the user or add a scroll feature to let them see all entries.



With these tiny tweaks, SYSMETS1 gonna be showin' those system metrics like a boss, even if your screen is **tiny** or **huge**.



Determining the Client Area Size

The **client area** is the portion of a window that is available for displaying application content.

It **excludes** the title bar, menu bar, and any other non-client elements.

Determining the size of the client area is essential for correct display.

💡 Think of it like this:

- The whole window = **phone with a case**
- The client area = **the screen**

Traditional Method: GetClientRect Function

The **GetClientRect function** is a traditional method for retrieving the dimensions of the client area.

It takes a **window handle as input** and **returns a RECT structure** containing the client area's coordinates.

```
RECT rect;
GetClientRect(hWnd, &rect);
int width = rect.right - rect.left;
int height = rect.bottom - rect.top;
```

Calling this function repeatedly is kinda wasteful, especially if the client area size changes frequently.

Efficient Method: Processing WM_SIZE Message

💡 When the window resizes, Windows sends you a **WM_SIZE message**. Use it!

```
case WM_SIZE:
    cxClient = LOWORD(lParam); // width
    cyClient = HIWORD(lParam); // height
    return 0;
```

This message is sent whenever the window's size changes, providing the updated client area dimensions in the lParam parameter.

⌚ These **cxClient** and **cyClient** variables stay updated — no extra function calls needed.



Where to Store These Values?

To store the client area dimensions, define **static** variables **cxClient** and **cyClient** in your window procedure.

```
// Inside your WndProc:  
static int cxClient, cyClient; // persistent storage for width & height
```

📌 **Why static?** So they don't reset every time the window processes a new message. They "remember." – we talked in depth about this in the static document, right?

Extract new dimensions from lParam: Store in static/global vars like cxClient, cyClient.



How to Calculate What Fits (Expanded for Real Understanding)

Alright — so you've got your window created, and in WM_CREATE, you already asked Windows:

```
int cxChar, cyChar; // average char width & height, set in WM_CREATE
```

Then you already have the data for character width and line height:

```
GetTextMetrics(hdc, &tm); // tm holds text info for the current font  
  
cxChar = tm.tmAveCharWidth; // avg character width in pixels  
cyChar = tm.tmHeight + tm.tmExternalLeading; // total line height
```

Cool, now what?



Step 1: You Need the Current Size of the Client Area

The **client area** is the drawable space of your window — no titlebar, no borders.

In WM_SIZE, Windows tells you "Hey, your window just changed size!"

You grab the width and height like this:

```
cxClient = LOWORD(lParam); // Width of drawable area  
cyClient = HIWORD(lParam); // Height of drawable area
```

These are pixel values. Like, if the user resizes the window, Windows will call your WndProc with a new lParam and boom — fresh sizes.

The **LOWORD** and **HIGHWORD** tools grab the width and height numbers from lParam, which tell you how big the window's inside area is.



Step 2: Use Math to Figure Out What Fits

Now that you have:

- **cxClient** → total width of client area
- **cyClient** → total height of client area
- **cxChar** → width of one character
- **cyChar** → height of one line

You can **calculate how many full characters or full lines will fit** on-screen:

```
int maxLines = cyClient / cyChar;           // Full lines that fit vertically  
int maxCharsPerLine = cxClient / cxChar; // Characters per line
```

maxLines gives you how many **full rows of text** can be stacked top to bottom.

maxCharsPerLine gives you how many **characters can fit side to side** before wrapping or clipping.

Why This Matters:

Without this math, you'd be **guessing blindly** how many lines or letters will show. Your text might overflow, clip off-screen, or leave too much empty space.

With this logic:

- You **know** what fits on the screen.
- You can draw only what's visible.
- You can later implement **scrolling** for overflow content (we'll cover that soon).

Determining the size of the client area is a crucial aspect of Windows programming.

By utilizing the **WM_SIZE message** and **storing the client area dimensions** efficiently, developers can ensure that application content is displayed correctly, even when window sizes change dynamically.

Step	What Happens
💻 WM_CREATE	Get font metrics → calculate char sizes
👉 WM_SIZE	Update client area width/height
🎨 WM_PAINT	Use cxClient/cyClient to decide what to draw
📚 Too much content?	Use scroll bars if needed

I know that table flew right through your brain, like it did mine 

WM_CREATE: The window just got made. This is where you get all the info about the default font, like how wide characters are and how tall a line of text is. You'll use this to figure out how much space your text will take up.

WM_SIZE: The window's size just changed. You need to update your internal records for the window's available width (cxClient) and height (cyClient) so you know how much space you have for drawing.

WM_PAINT: The window needs to draw something. Now that you have cxClient (width) and cyClient (height), you can decide what content fits on the screen and where to draw it. This is where you'd put the logic to display your system metrics list.

Too much content?: If the system metrics list is longer than what can fit in cxClient and cyClient (as you figured out in WM_PAINT), then you should enable and use scroll bars to let the user see everything.

Is everything good now? Atleast good.



⌚ Real-Life Example

Let's say:

- The user resizes the window to 800x600
- Your font's character height is 20 pixels
- Your character width is 10 pixels

Then:

```
maxLines = 600 / 20 = 30 lines  
maxCharsPerLine = 800 / 10 = 80 characters
```

So, you can draw **30 full rows of 80 characters**. Anything more than that? Needs scrolling.

◻ What If My Content Doesn't Fit? Next topic.

Boom — you bring in **scroll bars** (vertical or horizontal).

- Use SetScrollRange() and SetScrollPos() to manage scrolling.
- Detect scroll events using WM_VSCROLL and WM_HSCROLL.

Classic Windows programming move.

Conclusion

Determining and tracking client area size is **core to every Windows app that draws content manually**.

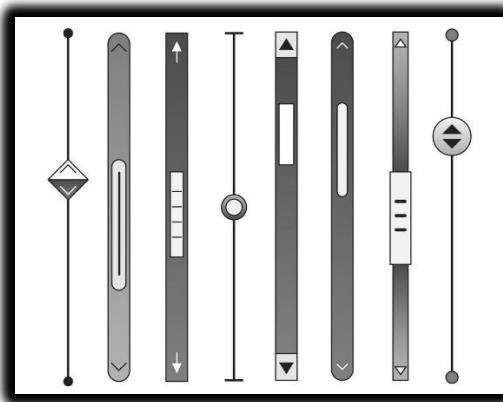
By storing cxClient and cyClient from WM_SIZE, and combining that with font metrics (cxChar, cyChar), you can:

- *Dynamically render content.*
- *Control layout precisely.*
- *Avoid unnecessary function calls.*
- *Add scrollbars when needed.*

This pattern is used in text editors, terminals, tables, custom controls — everywhere.

SCROLL BARS: ENHANCING USER INTERFACE INTERACTION

Scroll bars enable users to navigate through extensive content that exceeds the visible area of a window.



They provide a **user-friendly** and **intuitive** way to go through whole documents, spreadsheets, images, web pages, and other types of content.

Scroll bars aren't just visual — they're **fully interactive UI elements** built into the Windows API, and you don't even have to draw them manually (unless you go full custom).

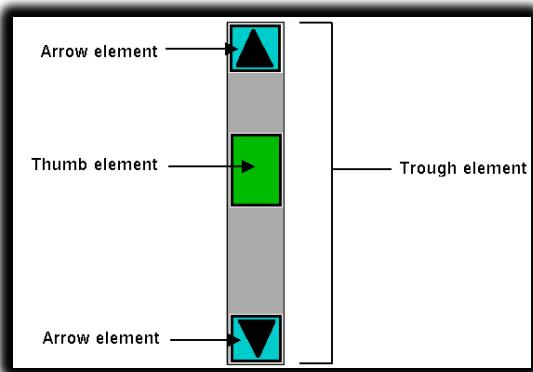
Types and Placement of Scroll Bars

Vertical scroll bars are used for scrolling up and down, typically found on the right side of a window.

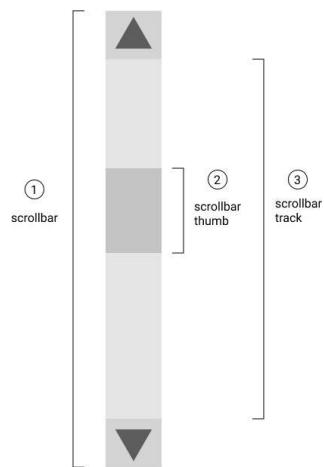
Horizontal scroll bars are used for scrolling left and right, are typically positioned at the bottom of a window.

Scroll Bar Components and Functionality

1. Scroll Bar Arrows: Clicking on the arrows at either end of the scroll bar initiates scrolling in the corresponding direction. Click to scroll one unit up/down (or left/right).



2. Scroll Box (Thumb): A movable indicator that represents the approximate position of the visible content within the entire document. Dragging the scroll box allows for direct navigation to a specific location in the content.



3. Scroll Bar Track: Provides the area along which the scroll box can move. Clicking within the scroll bar track, either above or below the scroll box, initiates scrolling in the corresponding direction.

Scroll Bar Integration into Windows Applications

Integrating scroll bars into Windows applications is a **straightforward process**.

```
// Add styles when creating your window
WS_VSCROLL          // Adds a vertical scroll bar
WS_HSCROLL          // Adds a horizontal scroll bar
WS_VSCROLL | WS_HSCROLL // Adds both
```

During window creation using the **CreateWindow function**, the desired scroll bar type can be specified by including the corresponding window style identifiers:

- **WS_VSCROLL**: Vertical scroll bar.
- **WS_HSCROLL**: Horizontal scroll bar.
- **WS_VSCROLL | WS_HSCROLL**: Both vertical and horizontal scroll bars.

Once included, scroll bars are **automatically positioned** and **sized** according to the window's client area.

Windows handles all mouse interactions with the scroll bars, providing the necessary scrolling behavior.

```
CreateWindow(
    "MyClass", "Scroll Example",
    WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
    CW_USEDEFAULT, CW_USEDEFAULT, 640, 480,
    NULL, NULL, hInstance, NULL
);
```

This makes scroll bars automatically appear and resize with your window's client area. You don't need to draw them manually.

How It All Works Behind the Scenes

Once your window includes scroll bar styles, **Windows takes care of the look and mouse interaction.**

But you need to:

- Set the **scroll range** and **position**.
- Respond to **scroll messages** (e.g., WM_VSCROLL or WM_HSCROLL).
- Redraw the content accordingly.

Core Functions for Scroll Control

FUNCTION	WHAT IT DOES (THE REAL TALK)
<code>SetScrollRange()</code>	This sets the min and max scroll positions for a window's scrollbar. Imagine setting the start and end points for a racing track; your scrollbar can only move within those bounds. It's defining the full potential scrollable area, like telling your app, "Hey, this content goes from here to way down there!"
<code>SetScrollPos()</code>	Use this to directly move the scrollbar's thumb (that little draggable square) to a specific spot within its defined range. Think of it like instantly jumping to a specific page number in a book. Super useful for when you need to programmatically snap the user's view to a certain part of the content.
<code>GetScrollPos()</code>	This function reads the current position of the scrollbar's thumb. It tells you exactly where the user is "looking" within the scrollable content. It's like checking the current page number you're on in a really long document, letting your app know the user's current view.
<code>ScrollWindowEx()</code>	This is where things get cool: it shifts the actual content within the window, not just the scrollbar. Think of it as literally dragging the canvas of your application around to reveal new parts. If you've got a lot of content and want to move it around, maybe after a resize or for a special effect, this function slides it all over. It can even fill the newly exposed areas with a custom background, like magically extending your view!

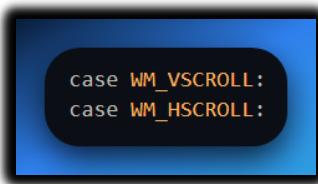
In short:

- **SetScrollRange()** - Defines min/max scroll positions.
- **SetScrollPos()** - Sets current scroll thumb position.
- **GetScrollPos()** - Reads current scroll thumb position.
- **ScrollWindowEx()** - Scrolls the window content.

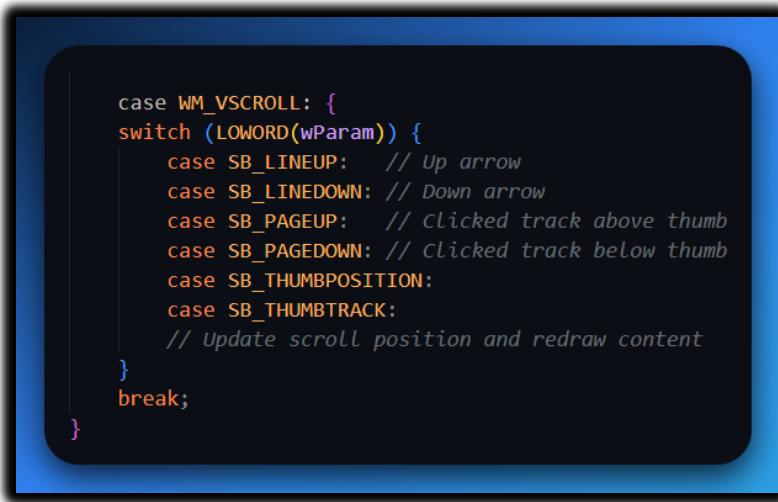


Responding to Scroll Messages

When the user interacts with the scroll bar, your window receives messages like:



You handle them this way:



Windows sends wParam to tell you what the user did (e.g., drag, click, arrow).

We explain the table below in depth inside the Scrollbars002.html or the image below.

Feature	How to Do It
Add vertical scroll bar	Use <code>WS_VSCROLL</code> in <code>CreateWindow</code>
Add horizontal scroll bar	Use <code>WS_HSCROLL</code>
Scroll events	Handle <code>WM_VSCROLL</code> and <code>WM_HSCROLL</code>
Update visuals	Use <code>ScrollWindowEx()</code> + <code>InvalidateRect()</code>

FEATURE	HOW TO MAKE IT HAPPEN (THE PLAYBOOK)
Add vertical scroll bar	<p>Wanna add a scrollbar that goes up and down on your window? When you're using `CreateWindow` or `CreateWindowEx` to whip up your window, just slap on the `WS_VSCROLL` style flag.</p> <p>It's like telling the system, "Yo, this window needs vertical scrolling!" The system then magically adds that bar on the right side.</p>
Add horizontal scroll bar	<p>Need a scrollbar that lets your content slide left and right? Easy peasy! Just like with the vertical one, when you call `CreateWindow` or `CreateWindowEx`, toss in the `WS_HSCROLL` style flag.</p> <p>This tells Windows, "Give me that horizontal action!" and you'll see a scrollbar pop up at the bottom of your window, ready for side-to-side adventures.</p>
Scroll events	<p>So, the user just scrolled. How do you know? Windows is cool like that and sends you messages! When someone moves the vertical scrollbar, your window procedure gets a `WM_VSCROLL` message.</p> <p>For horizontal moves, you'll catch a `WM_HSCROLL` message. It's like your app getting a text message saying, "Heads up! Scroll action initiated!" You gotta handle these messages to make stuff happen when the user scrolls.</p>
Update visuals	<p>When you scroll, your window's content needs to move, right? After using a function like `ScrollWindowEx()` to physically shift the pixels, you then need to tell Windows, "Hey, some areas might look weird now, refresh them!"</p> <p>You do this by calling `InvalidateRect()` (or `InvalidateRgn()`) on the part of your window that got messed up. It's like telling an artist, "Paint over this part, it's not looking right!" This triggers a `WM_PAINT` message, and then your drawing code redraws the newly exposed or changed areas, making everything look smooth and correct.</p>

Semi-Final Thoughts

Adding scroll bars to a Windows app is **easy to set up**, but takes **some message-handling logic** to make them feel polished. You're basically:

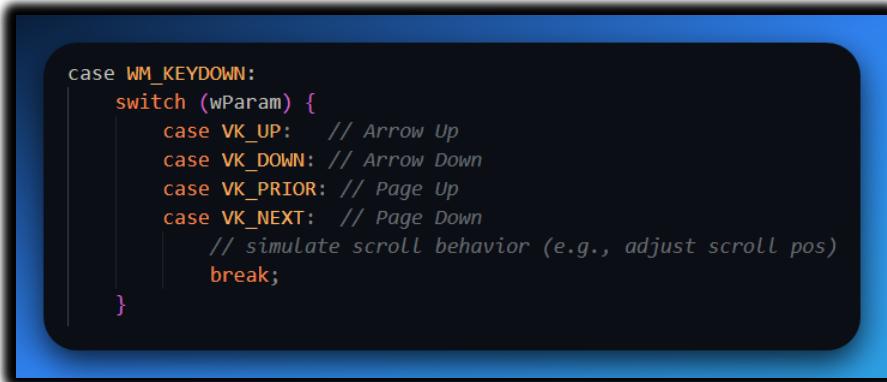
- Telling Windows how much content you have
- Letting the user control *which part* of that content is visible

Keyboard + Smart Scroll Bars in Windows API

Yeah, so here's the deal:

Windows scroll bars react to mouse events out of the box, but not keyboard input like arrow keys — unless you code it yourself.

To do this, you handle WM_KEYDOWN and interpret the key codes like this:



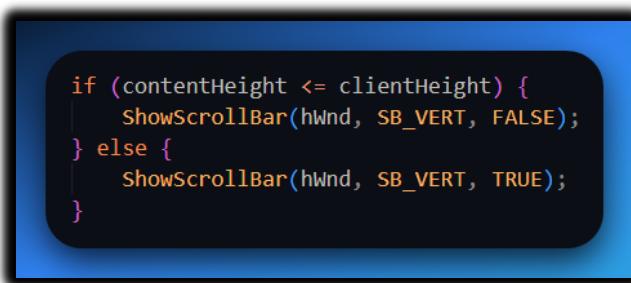
 Tip: Use SendMessage(hWnd, WM_VSCROLL, SB_LINEUP, 0); to trigger scroll events programmatically.

💡 Scroll Bar Enhancements (UX Edition)

These aren't required, but if you want that pro polish, check these out:

✓ 1. Auto-Hide Scroll Bars

Only show them when content doesn't fit.



🔥 Why? It makes your window feel cleaner and smarter — like Notepad does.

✓ 2. Visual Feedback During Scrolling

You can:

- Highlight the scroll box or changing its color, to indicate the current position within the content.
- Add an overlay indicator.
- Or even draw a floating "line X of Y" while dragging.

💡 But that requires **custom scroll bar drawing** using WM_NCPAINT or owner-draw tricks — we'll need a custom scroll bar lesson.

✓ 3. Custom Scroll Bar Skins

If you hate the classic Win98-style bars, go custom:

- *Hide default scroll bars with WS_HSCROLL removed*
- *Create your own with rectangles, bitmaps, and mouse hit-testing*

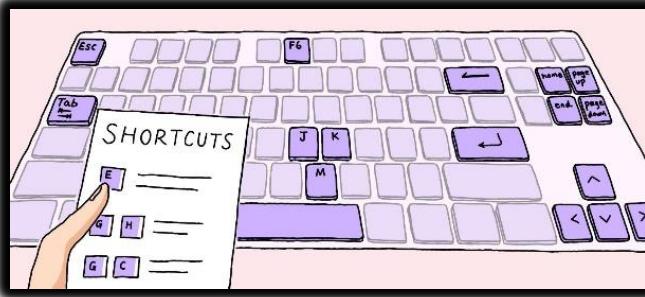
🛠 Use GDI (Rectangle, FillRect, DrawText) or GDI+ for smoother graphics.

4. Keyboard Shortcuts = Accessibility Boost

Set up these hotkeys in WM_KEYDOWN(scrollbars003.html):

Key	Action
VK_UP	Scroll up by line
VK_DOWN	Scroll down by line
VK_PRIOR	Page up
VK_NEXT	Page down
VK_HOME	Scroll to top
VK_END	Scroll to bottom

This gives power users and accessibility tools better control.



Controlling Scroll Bar Range and Position

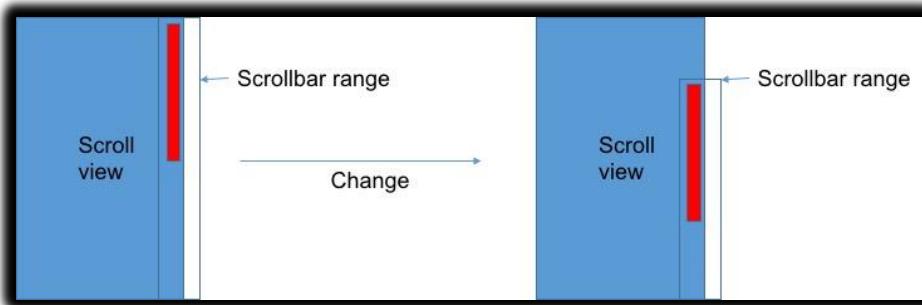
Now let's make that scroll bar work logically with your content:

1. Set the Scroll Bar Range

The scroll bar range defines the minimum and maximum values associated with the scroll bar.

These values represent the extent of the content that can be scrolled through.

For instance, a scroll bar with a range of 0 to 100 indicates that the content can be scrolled from its initial position (0) to its full length (100).



Setting Scroll Bar Range

```
SetScrollRange(hWnd, SB_VERT, 0, totalLines - 1, TRUE);
```

The SetScrollRange function is used to modify the range of a scroll bar. It takes five arguments:

- **hwnd:** The window handle of the window containing the scroll bar.
- **iBar:** Specifies the scroll bar type (SB_VERT for vertical, SB_HORZ for horizontal).
- **iMin:** The new minimum value of the scroll bar range (0 means top)
- **iMax:** The new maximum value of the scroll bar range (totallines - 1 means bottom most line)
- **bRedraw:** A Boolean value indicating whether to redraw the scroll bar based on the new range (TRUE for redrawing immediately, FALSE to avoid unnecessary redraws).

Scroll Bar Position

The scroll bar's position shows where you are currently looking within all the content.

To actually *set* where the scroll box (the "thumb") appears on the scroll bar, you use this function:

```
SetScrollPos(hwnd, SB_VERT, currentLine, TRUE);
```

- **hwnd:** This is the handle to your window – basically, telling the system *which* window's scroll bar you want to change.
- **SB_VERT:** This means you're dealing with the **vertical** scroll bar. If you were scrolling horizontally, you'd use SB_HORZ.
- **currentLine:** This is the new position you want to set for the scroll box. In your SYSMETS program, this would likely be the index of the first visible line of your system metrics.
- **TRUE:** This tells Windows to *redraw* the scroll bar immediately after setting its position.

You'll typically update the scroll bar's position using SetScrollPos after:

- *You change the size of the window.*
- *You've finished scrolling (like when someone clicks a scroll arrow or drags the thumb).*
- *You jump to a specific part of your content (like clicking a link that takes you down the page).*

👉 **The scroll bar's position tells Windows how far the user has scrolled through the content.**

- When the scroll box (thumb) is at the **very top or far left**, the scroll position is at its minimum (usually 0).
- When the thumb is at the **very bottom or far right**, the position is at its maximum (whatever you set with SetScrollRange()).

So basically:

*The scroll bar position = how deep you are into your content.
It's like a progress bar for scrolling.*

Querying Scroll Bar Range and Position

The **GetScrollRange** and **GetScrollPos** functions can be used to retrieve the current **range** and **position** of a scroll bar, respectively.

- The **scrollbar range** defines the minimum and maximum values for the scroll bar, indicating the total extent of the scrollable content.
- The **scrollbar position** is where the scroll box (thumb) currently is within that range, showing what part of the content is visible.

You can get these values using GetScrollRange and GetScrollPos functions, which help adjust scrolling behavior.

🔍 How to Query These Values

📦 GetScrollRange()

Gives you the scroll bar's **minimum** and **maximum** values.

```
int min, max;  
GetScrollRange(hWnd, SB_VERT, &min, &max);
```

- **hWnd**: Your window handle
- **SB_VERT**: Indicates to windows that the operation applies to the **vertical** scroll bar.

Useful for knowing how **far you can scroll**.

Also good for showing **range indicators** like "Line 10 of 1000".

🎯 GetScrollPos()

Returns the current scroll position (thumb's location).

```
int currentPos = GetScrollPos(hWnd, SB_VERT);
```

This is crucial in your drawing logic: it tells you which line of your content you should start drawing from.

Why GetScrollPos() Matters

The scroll position tells your app which line should be at the very top of the visible client area.

For example:

- You're viewing line 500 of a 1000-line code.
- GetScrollPos() returns 500, meaning ...
- Lines 0–499 are above the view — no need to redraw them.

Your drawing logic should start from line 500 and go down from there.

This is how you avoid wasting CPU drawing invisible lines.

Without GetScrollPos():

You'd be **redrawing everything from line 0** every single time — even the stuff off-screen.

Bad for performance. Bad for memory. Waste of time.

With GetScrollPos():

You skip straight to the visible content:

```
int currentPos = GetScrollPos(hWnd, SB_VERT); // Get the starting line number from the scroll bar
for (int i = 0; i < linesVisible; i++) {
    TextOut(hdc, x, y + i * cyChar, text[currentPos + i], lstrlen(text[currentPos + i]));
}
```

This is the **actual implementation** for drawing text based on scrolling.

- **int currentPos = GetScrollPos(hWnd, SB_VERT);:** This line gets the numerical value of the scroll bar's thumb position. If the thumb is at the top, currentPos might be 0. If it's halfway down a 1000-line document, currentPos might be 500. This currentPos tells you *which line of your data* should be displayed at the very top of your visible window area.
- **for (int i = 0; i < linesVisible; i++):** This loop runs linesVisible times, where linesVisible is how many lines can fit on your screen at once (e.g., 30 lines).

- **TextOut(hdc, x, y + i * cyChar, text[currentPos + i], ...);**: Inside the loop, `text[currentPos + i]` is the crucial part. It means:
 - ✓ For the **first** iteration ($i = 0$), it draws `text[currentPos + 0]`, which is `text[currentPos]`. This is the line that should appear at the very top of your display.
 - ✓ For the **second** iteration ($i = 1$), it draws `text[currentPos + 1]`, which is the next line.
 - ✓ And so on, until all `linesVisible` lines are drawn.

In short, this code draws only the lines that are currently visible on the screen, starting from the line indicated by `currentPos`.

 As the user scrolls, `currentPos` changes, and the loop shifts the content upward or downward accordingly.

Scroll Flow: How Everything Connects

Here's the full scroll event lifecycle:

1. User interacts with scroll bar (click, drag, keyboard)
2. Windows sends your window WM_VSCROLL
3. You detect what kind of scroll (line up, page down, thumb drag, etc.)
4. You adjust the scroll position accordingly
5. Call `ScrollWindowEx()` to shift visuals
6. Use `SetScrollPos()` to update the thumb
7. Call `InvalidateRect()` to trigger a repaint
8. In WM_PAINT, use `GetScrollPos()` to draw from the correct offset

This is the **classic scroll engine pattern** in WinAPI.

TLDR — Best Practices

-  Use `GetScrollPos()` inside `WM_PAINT` to determine where to start drawing
-  Offset your content array using that position
-  Draw only what fits in the visible area (`linesVisible`)
-  The scroll bar is the controller — `currentPos` is your viewport offset

Want a Real Challenge?

Try this mini project:

- Build a scrollable window with 1000 lines
- Let user scroll using **scroll bar + arrow keys**
- Show "Line X of Y" in the corner
- Auto-hide the scroll bar when all lines fit

WHAT WINDOWS ITSELF HANDLES WHEN YOU USE SCROLL BARS

When you add `WS_VSCROLL` or `WS_HSCROLL` to your window, Windows handles a surprising amount of the grunt work **automatically**. Here's what it takes off your plate:

1. Mouse Message Processing

Windows takes care of:

- Detecting **mouse clicks** on the scroll arrows (up/down or left/right)
- Handling **thumb drags**
- Responding to **track clicks** (above/below the scroll box)

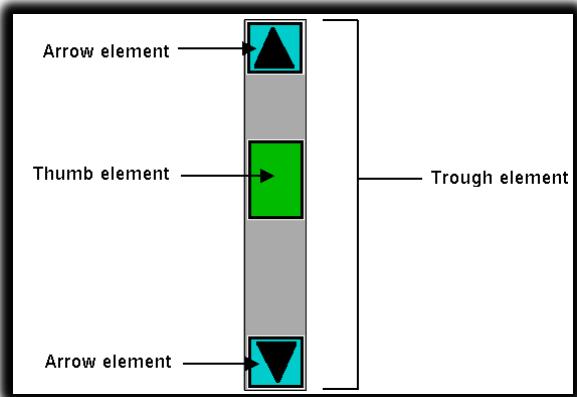
 You don't have to manually detect `WM_LBUTTONDOWN` on scroll bar components — Windows already knows where the mouse is and which part was clicked.

2. Reverse-Video Flash (Track Click Feedback)

When a user clicks inside the scroll bar **track** (not the thumb), Windows:

- Flashes the clicked area in **reverse video** (a visual “pulse”)
- This signals to the user: *“Hey, I recognized that click and scrolling will happen.”*

You don’t need to animate this. It’s built-in and automatic — just like classic Win32 should be.



3. Thumb (Scroll Box) Movement

When a user **drags the scroll thumb**, Windows:

- Updates the thumb’s position in real-time
- Tracks how far it has moved relative to the scroll range
- Handles mouse tracking and redraws the bar and thumb without you writing a line of rendering code

All you need to do is react to the scroll value changing.

4. Sending Scroll Messages to Your Window Procedure

Windows sends **scroll bar messages** to your WndProc, letting you know what happened.

The two core message types:

Message	For which scroll bar?
WM_VSCROLL	Vertical scroll bar
WM_HSCROLL	Horizontal scroll bar

The message includes:

- **wParam:** the scroll request type (e.g., line up, page down, thumb drag)
- **lParam:** usually 0 for standard window scroll bars

You interpret these inside your WndProc to update the visible content.

Summary of Windows' Scroll Responsibilities

Responsibility	Description
(Mouse Interaction)	Handles all clicks and drags automatically
(Reverse-Video Flash)	Provides visual feedback for track clicks
(Thumb Movement)	Tracks drag position and updates thumb location visually
(Message Dispatch)	Sends scroll-related messages (WM_VSCROLL / WM_HSCROLL) to your window

Why This Rocks

Windows does 90% of the scroll UI and input work for you.

You just **handle what the app should do** when the user scrolls (i.e., update content, repaint, adjust state).

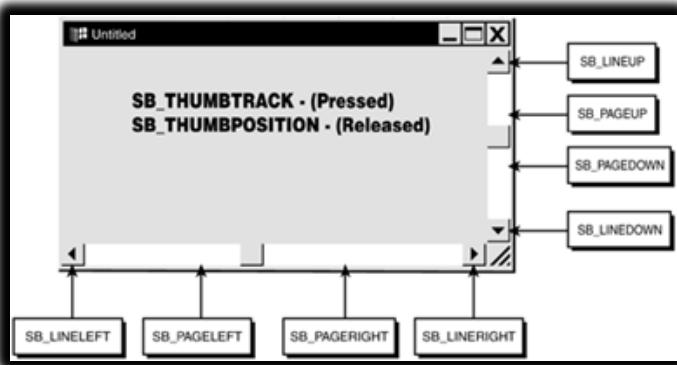
You don't need to paint the scroll bar. You don't even need to care about mouse coordinates or hit testing the thumb. **Just respond** to the scroll messages and let Windows do the drawing and input management.

Scroll Bar Messages vs. Scroll Codes

Term	What It Means	Example
Scroll Bar Messages	These are the messages Windows sends to your window when a scroll event happens. Think of them as <i>notifications</i> .	WM_VSCROLL , WM_HSCROLL
Scroll Codes	These are the values inside the message (<code>wParam</code>) that tell you <i>what kind of scroll action happened</i> .	SB_LINEUP , SB_PAGEDOWN , SB_THUMBPOSITION

Analogy:

- **WM_VSCROLL** = "Hey app, the user scrolled!"
- **SB_LINEUP** = "And they clicked the up arrow."



```

case WM_VSCROLL:
switch (LOWORD(wParam)) {
    case SB_LINEUP:      // Scroll up one line
    case SB_LINEDOWN:    // Scroll down one line
    case SB_PAGEUP:      // Page up
    case SB_PAGEDOWN:    // Page down
    case SB_THUMBPOSITION: // User dragged the thumb
        // etc...
}
break;
  
```

Bottom Line:

- WM_VSCROLL and WM_HSCROLL = "*what happened*"
- SB_... codes inside = "*how it happened*"

Think message is the package, and code is the content. 

YOUR PROGRAM'S RESPONSIBILITIES FOR SCROLL BARS

 **READ:** While Windows handles the low-level mechanics of scroll bars, your program plays a crucial role in managing the **overall scrolling behavior** and **content updates**:

 **REREAD:** While Windows takes care of **drawing scroll bars** and **handling mouse input**, your program controls *what actually happens* when the user scrolls. This includes managing the scroll logic, updating the UI, and syncing the content with user actions.

1. Scroll Bar Range Initialization: You are responsible for setting the initial range of the scroll bar using the SetScrollRange function. The range defines the minimum and maximum values associated with the scroll bar, representing the extent of the content that can be scrolled through.

```
SetScrollRange(hwnd, SB_VERT, 0, totalLines - 1, TRUE);
```

2. Scroll Bar Message Processing: When the user scrolls, Windows sends:

- WM_VSCROLL or WM_HSCROLL messages.

Your job: **process these messages in your WndProc**, figure out what kind of scroll action was performed (SB_LINEUP, SB_THUMBPOSITION, etc.), and adjust the scroll position accordingly.

3. Scroll Bar Thumb Update: Based on the scroll bar messages and the current position of the scroll box (thumb), your program should update the position of the thumb to reflect the user's scrolling actions.

```
SetScrollPos(hWnd, SB_VERT, newPos, TRUE);
```

4. Content Area Updates: Whenever the scroll position changes, your content must reflect that. The client area must be updated with the corresponding content.

- In WM_PAINT, get the current scroll position with GetScrollPos().
- Offset your content rendering accordingly.

Example for vertical text scrolling:

```
int currentPos = GetScrollPos(hWnd, SB_VERT);
for (int i = 0; i < linesVisible; i++) {
    TextOut(hdc, x, y + i * cyChar, text[currentPos + i], lstrlen(text[currentPos + i]));
}
```

⌚ This ensures the right portion of your content is always visible as the user scrolls.

📌 Recap Table

Task	Who Handles It
Drawing scroll bar	Windows
Handling mouse/keyboard input	Windows
Setting scroll range	Your program
Processing scroll messages	Your program
Moving the thumb	Your program (via SetScrollPos)
Updating visible content	Your program

You don't draw the scroll bar. You don't handle raw input. But **you control the logic**: how much to scroll, what to show, and when to update.

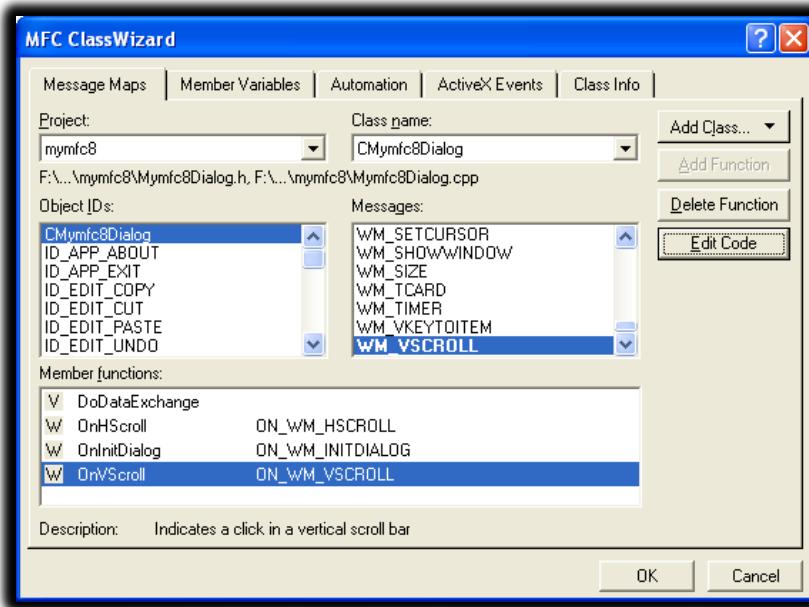
SCROLL BAR MESSAGES: COMMUNICATING SCROLL BAR INTERACTIONS

Scroll bars are just UI until you connect them to logic. When a user interacts with a scroll bar, **Windows sends scroll messages** to your window's procedure (WndProc) to let you know what happened. It's your job to respond by updating the content.

The Two Key Messages

- **WM_VSCROLL:** This message is sent when the user scrolls the **vertical** scroll bar.
- **WM_HSCROLL:** Sent when the user scrolls the **horizontal** scroll bar.

Each mouse action on the scroll bar triggers at least two of these messages: one when the mouse button is pressed and another when it is released. This allows the application to track the user's scrolling actions and update the content accordingly.



These are sent by Windows automatically when:

- The user clicks an arrow button
- Drags the thumb
- Clicks the track
- Uses the mouse wheel (if hooked to scroll)
- Triggers keyboard scrolling (if you coded it)

Message Parameters

When you receive **WM_VSCROLL** or **WM_HSCROLL**, the system sends extra info through wParam and lParam.

1. wParam — Scroll Code

This parameter **contains the notification code**, which indicates the specific action the user is performing with the scroll bar. Tells you what the user actually did.

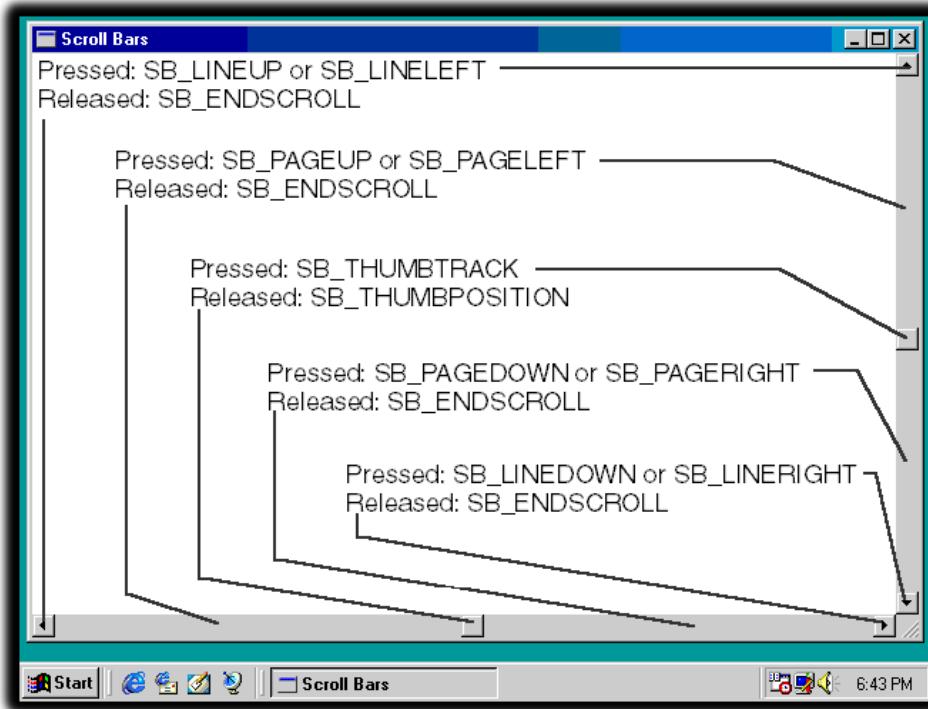
Notification codes are represented by identifiers that begin with SB, that indicate the "scroll bar."

Use LOWORD (wParam) to extract it.

LOWORD is a macro (a small piece of code that expands into more code at compile time) that specifically extracts the **lower 16 bits** of that 32-bit wParam value. In the context of scroll bar messages, the "scroll code" (like SB_LINEUP, SB_THUMBTRACK, etc.) is stored in this lower 16-bit portion of wParam. Common values:

Scroll Code	Direction	Triggered When
SB_LINEUP	Up / Left	User clicked the small up/left arrow
SB_LINEDOWN	Down / Right	User clicked the small down/right arrow
SB_PAGEUP	Up / Left (Page)	User clicked above the thumb in the scroll track
SB_PAGEDOWN	Down / Right (Page)	User clicked below the thumb in the scroll track
SB_THUMBTRACK	Follow thumb movement	User is dragging the scroll thumb (real-time)
SB_THUMBUPOSITION	Thumb released	User finished dragging the thumb and let go
SB_TOP	Scroll to top / left	Scroll bar moved to very beginning
SB_BOTTOM	Scroll to bottom/right	Scroll bar moved to very end
SB_LEFT	Scroll to far left	Horizontal scroll bar: moved all the way left
SB_RIGHT	Scroll to far right	Horizontal scroll bar: moved all the way right
SB_ENDSCROLL	Done scrolling	Windows sends this when the scrolling operation ends (can be ignored)

An Old illustration:



◆ IParam:

If you're using the **default scroll bar** that's part of the main window, IParam is usually 0, and you can safely ignore it.

However, if you're **managing custom scroll bar controls** (e.g., child windows with their own scroll bars), then IParam holds the handle to the control that sent the message. This helps distinguish which scroll bar was used.

Summary:

When a user drags the scroll box, **SB_THUMBTRACK** provides real-time position updates, while **SB_THUMBPOSITION** indicates the final position upon release; you choose one based on desired feedback and resource usage.

For 32-bit scroll bar ranges, **GetScrollInfo** is necessary to accurately retrieve the thumb's position because wParam's high word is only 16-bit.

HIWORD and **LOWORD** are macros that extract specific parts of a 32-bit value. LOWORD retrieves the lower 16 bits, while HIWORD retrieves the upper 16 bits.

Regarding the provided text, the specific definitions of HIWORD and LOWORD aren't given, but from general programming knowledge, the typical use of HIWORD and LOWORD is to **separate a 32-bit value** into its two 16-bit components.

Based *only* on your provided text:

HIWORD refers to the high word of the **wParam parameter**, which specifically indicates the current or final position of the scroll box during SB_THUMBTRACK or SB_THUMBPOSITION messages.

The **LOWORD** is not explicitly defined in the provided text, but it is implied that the wParam parameter is a 32-bit value.

General Explanation (not limited to provided text):

In Windows programming, LOWORD and HIWORD are macros used to extract the lower 16 bits and higher 16 bits, respectively, from a 32-bit integer.

They are often used to **unpack two separate 16-bit values** that have been combined into a single 32-bit parameter, such as lParam or wParam in window messages.

⚠ Heads-Up:

These are **never sent** when you're using **standard scroll bars** (created using WS_VSCROLL or WS_HSCROLL).

They're only used if you're building **custom scroll bar controls** or subclassing scroll bar behavior.

So unless you're doing custom scrollbar UI, you can **safely ignore these codes**.

Scroll Code	Meaning
SB_TOP	Scroll bar moved to top (pos = 0)
SB_BOTTOM	Scroll bar moved to bottom
SB_LEFT	Scroll bar moved to far left
SB_RIGHT	Scroll bar moved to far right

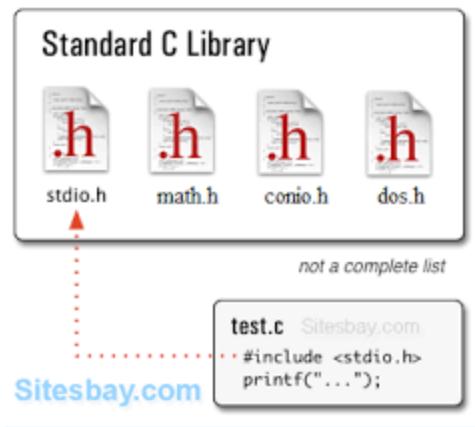
SYSMETS PROGRAM

You can find this code in the chapter 4 sysmets2 after running it, you will see the same program, but with a scroll bar that moves when clicked. In-depth explanation of the SYSMETS2.C code, with specific details and insights:

Header Files and Preprocessor Directives

windows.h: This header file provides essential Windows programming functionalities, including function declarations, data types, and macros. It's crucial for interacting with the Windows operating system and managing window elements.

sysmets.h: This custom header file presumably contains definitions and structures related to the system metrics being displayed. It serves as a local information repository for the application's specific data.



WndProc Function: Message Handling



WM_CREATE Message: This message indicates the creation of the application window. In response, the code performs essential initialization tasks:

- **Text Metrics Retrieval:** It retrieves the text metrics for the current font using GetTextMetrics. This provides information about character width and height, which is crucial for positioning text elements within the window.
- **Scroll Bar Range Setup:** It sets the scroll bar range using SetScrollRange. This defines the minimum and maximum values of the scroll bar, ensuring that the user can scroll through the entire list of system metrics.
- **Initial Scroll Position:** It initializes the current scroll position to 0 using SetScrollPos. This sets the starting point for displaying the system metrics within the visible area of the window.

WM_SIZE Message: This message is triggered when the window size changes. The code simply updates the cyClient variable, which stores the client area height, to reflect the new dimensions. This information is used later for adjusting the layout of the displayed text.

WM_VSCROLL Message: This message is generated when the user interacts with the scroll bar. The code handles various scroll bar actions:

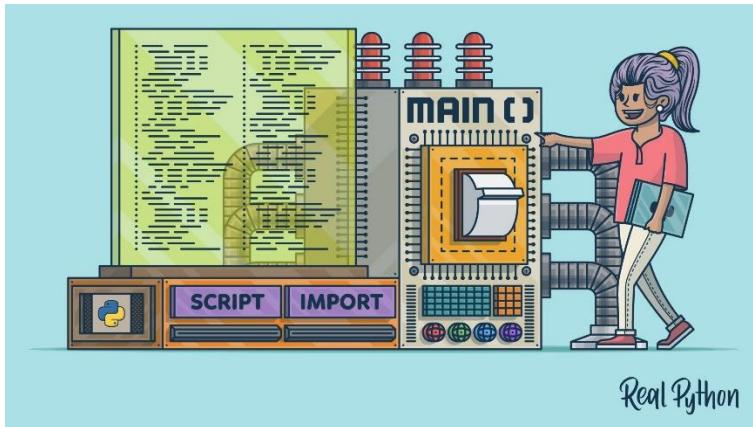
- **Line Up/Down:** For line-by-line scrolling, it adjusts the iVscrollPos variable accordingly, either decreasing or increasing it by 1.
- **Page Up/Down:** For page-by-page scrolling, it adjusts the iVscrollPos variable by the height of the client area in character units. This ensures that a full page of system metrics is scrolled at a time.
- **Thumb Position:** When the user drags the scroll box (thumb), the code updates the iVscrollPos variable to the high word of wParam, which represents the new position of the thumb.
- **Invalidation and Update:** After adjusting the scroll position, the code ensures it stays within the valid range using max and min functions. If the scroll position changes, it

updates the scroll bar position using SetScrollPos and invalidates the window's client area using InvalidateRect. This triggers a repaint to reflect the updated scroll position and display the corresponding system metrics.

WM_PAINT Message: This message prompts the code to repaint the window's client area. The code iterates through the system metrics array and draws each metric's label, description, and corresponding value using the TextOut function. This ensures that the system metrics are presented in a clear and organized manner.

WM_DESTROY Message: This message indicates the destruction of the application window. The code responds by sending a PostQuitMessage to the message queue, signaling the end of the application's execution.

Main Function: Application Entry Point



Window Class Registration: The WinMain function starts by registering the window class using RegisterClass. This defines the characteristics of the window, such as its style, window procedure, instance handle, icon, cursor, background brush, and class name.

Window Creation: It creates the window using the CreateWindow function. This allocates memory for the window structure, establishes its position and size, and associates it with the previously registered window class.

Window Visibility and Update: It displays the window using the ShowWindow function, making it visible to the user. It then updates the window's layout and contents using the UpdateWindow function.

Message Loop: It enters a message loop using a while loop. This continuously retrieves messages from the message queue using GetMessage, translates them using TranslateMessage, and dispatches them to the WndProc callback function using DispatchMessage.

Application Termination: The message loop continues until a WM_QUIT message is received. In this case, the GetMessage function returns 0, signaling the end of the message queue. The WinMain function exits the loop and returns the message parameter from GetMessage.

Vertical Scroll Bar Integration

The CreateWindow call now includes the WS_VSCROLL window style in the third argument, adding a vertical scroll bar to the window.

This allows users to scroll through the list of system metrics if the entire list cannot be displayed within the visible area of the window.

SM_CXSCREEN	Screen width in pixels	1280
SM_CYSCREEN	Screen height in pixels	1024
SM_CVSCROLL	Vertical scroll width	17
SM_CYHSCROLL	Horizontal scroll height	17
SM_CYCAPTION	Caption bar height	23
SM_CXBORDER	Window border width	1
SM_CYBORDER	Window border height	1
SM_CXFIXEDFRAME	Dialog window frame width	3
SM_CYFIXEDFRAME	Dialog window frame height	3
SM_CYVTHUMB	Vertical scroll thumb height	17
SM_CXHTHUMB	Horizontal scroll thumb width	17
SM_CXICON	Icon width	32
SM_CYICON	Icon height	32
SM_CXCURSOR	Cursor width	32
SM_CYCURSOR	Cursor height	32
SM_CYMENU	Menu bar height	20
SM_CXFULLSCREEN	Full screen client area width	1280
SM_CYFULLSCREEN	Full screen client area height	953
SM_CYKANJIWINDOW	Kanji window height	0
SM_MOUSEPRESENT	Mouse present flag	1
SM_CYYSCROLL	Vertical scroll arrow height	17
SM_CXHSCROLL	Horizontal scroll arrow width	17
SM_DEBUG	Debug version flag	0
SM_SWAPBUTTON	Mouse buttons swapped flag	0
SM_CXMIN	Minimum window width	136
SM_CYMIN	Minimum window height	39
SM_CXSIZE	Min/Max/Close button width	36
SM_CYSIZE	Min/Max/Close button height	22
SM_CXSIZEFRAME	Window sizing frame width	4
SM_CYSIZEFRAME	Window sizing frame height	4
SM_CXMINTRACK	Minimum window tracking width	136
SM_CYMINTRACK	Minimum window tracking height	39
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

Setting Scroll Bar Range and Initial Position

In response to the **WM_CREATE** message, the WndProc window procedure **sets the scroll bar range and initial position using the SetScrollRange and SetScrollPos functions**, respectively.

SetScrollRange: This function establishes the minimum and maximum values of the scroll bar, ensuring that it can cover the entire range of system metrics (0 to NUMLINES - 1).

SetScrollPos: This function initializes the current scroll position to 0, placing the first line of system metrics at the top of the client area.

Scroll Position Variable

A static **variable named iVscrollPos** is defined within the WndProc window procedure to **track the current position of the scroll bar thumb**. This variable plays a crucial role in managing the scroll bar behavior and updating the displayed system metrics.

Handling WM_VSCROLL Messages

The WndProc window procedure handles various WM_VSCROLL messages to respond to user interactions with the scroll bar:

SB_LINEUP and SB_LINEDOWN: For line-by-line scrolling, the scroll position is adjusted by 1, either decreasing or increasing it accordingly.

SB_PAGEUP and SB_PAGEDOWN: For page-by-page scrolling, the scroll position is adjusted by the height of the client area in character units (cyClient divided by cyChar). This ensures that a full page of system metrics is scrolled at a time.

SB_THUMBPOSITION: When the user drags the scroll box (thumb), the new thumb position is stored in the high word of wParam. This value is then assigned to iVscrollPos to reflect the current scroll position.

SB_ENDSCROLL and SB_THUMBTRACK: These messages are ignored in this version of the program.

Validating Scroll Position and Updating Window

After adjusting the scroll position based on the WM_VSCROLL message type, the code ensures that the value of iVscrollPos remains within the valid range of the scroll bar using the min and max macros.

If the scroll position has changed, it is updated using SetScrollPos, and the entire window is invalidated using InvalidateRect to trigger a repaint.

Updating Client Area

When repainting the client area in response to a WM_PAINT message, the y-coordinate of each line is calculated using the formula $cyChar * (i - iVscrollPos)$.

This formula takes into account the current scroll position, ensuring that the appropriate lines of system metrics are displayed within the visible area of the window.

Painting in Response to WM_PAINT

The preferred approach in Windows programming is to handle all client area painting in response to a WM_PAINT message.

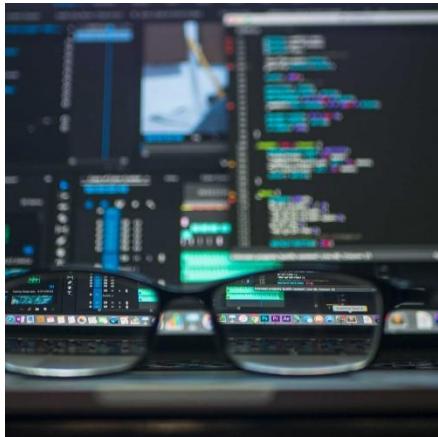
This ensures that the entire client area is repainted consistently and efficiently.

Duplicating painting logic in response to other messages can lead to **redundancy** and **potential inconsistencies**.

Discipline of WM_PAINT-based Painting

While it may seem indirect, this approach offers several benefits:

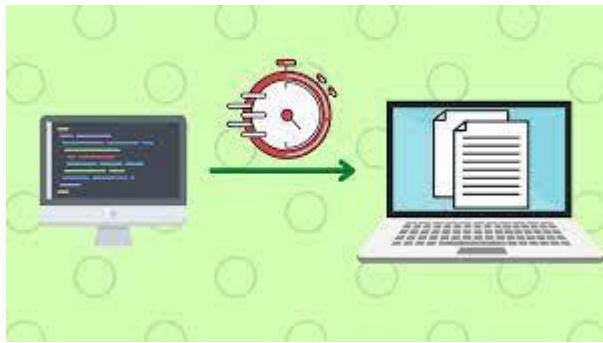
Code Clarity and Maintainability: By centralizing painting logic in the WM_PAINT handler, the code becomes more organized and easier to maintain.



Consistent Repainting: The WM_PAINT handler should be capable of repainting the entire client area, ensuring that the window's contents are always displayed correctly.



Efficiency: Windows optimizes painting operations, so relying on WM_PAINT messages for repainting can lead to better performance.



InvalidateRect for Invalidating Client Area

When a program needs to trigger a repaint of a specific area or the entire client area, it can use the **InvalidateRect** function.

This marks the specified rectangles as invalid, prompting Windows to generate a WM_PAINT message.

Considerations with InvalidateRect

While **InvalidateRect** is useful for invalidating specific areas, it's important to note that WM_PAINT messages are handled with **lower priority** than other messages.

This means that **if the system is busy with other tasks**, it may take some time for the WM_PAINT message to be processed, resulting in visible "holes" in the window.

UpdateWindow for Immediate Repainting

To address this delay, you can use the **UpdateWindow function** after calling **InvalidateRect**.

This causes the window procedure to be **immediately called with a WM_PAINT message**, bypassing the message queue and ensuring immediate repainting of the invalidated area.

UpdateWindow's Role in Window Creation

UpdateWindow is also used during window creation to trigger the initial painting of the client area.

When a window is first created, the entire client area is marked as invalid, and **UpdateWindow** directs the window procedure to paint it accordingly.

Conclusion

SYSMETS2.C introduces vertical scrolling functionality, allowing users to navigate through the list of system metrics efficiently. The code effectively manages the scroll bar position and updates the displayed content accordingly.

However, this implementation is still wasteful and inefficient, as it renders lines of text outside the client area. Improvements will be made to address this issue in subsequent versions.

BUILDING A BETTER SCROLLBAR

Obsolete Scroll Bar Functions

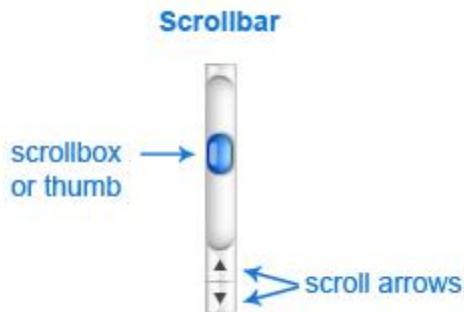
The scroll bar functions introduced in earlier versions of Windows, including SetScrollRange, SetScrollPos, GetScrollRange, and GetScrollPos, are considered obsolete in the Win32 API.

While they are still functional and can handle 32-bit arguments, they have been superseded by newer functions that offer additional capabilities.

New Scroll Bar Functions: SetScrollInfo and GetScrollInfo

The Win32 API introduced two new scroll bar functions: [SetScrollInfo](#) and [GetScrollInfo](#). These functions provide all the functionality of the older functions and add two significant enhancements:

Thumb Size Control: The size of the scroll bar thumb can now be **dynamically adjusted** based on the page size, which represents the portion of the document displayed in the window. This allows for a more **visually intuitive representation** of the document's size and scroll position.



32-bit Thumb Position Retrieval: The GetScrollInfo function provides access to the actual 32-bit value of the thumb position, even if the range exceeds 65,536. This addresses a limitation of the older functions, which only provided 16 bits of thumb position information.

Benefits of SetScrollInfo and GetScrollInfo. These new functions offer several advantages over the obsolete ones:

Enhanced Functionality: They provide greater control over the appearance and behavior of scroll bars, including thumb size adjustment and accurate thumb position retrieval.



Future-Proofing: They are designed to handle larger ranges and provide more flexibility for future scroll bar implementations.



Improved User Experience: By enabling dynamic thumb sizing and accurate position tracking, they contribute to a more intuitive and responsive user experience.



Here is an in-depth explanation of the syntax and usage of the SetScrollInfo and GetScrollInfo functions:

Function Syntax

```
SetScrollInfo(hwnd, iBar, &si, bRedraw);  
GetScrollInfo(hwnd, iBar, &si);
```

Parameters

hwnd: Handle of the window that contains the scroll bar.

iBar: Specifies which scroll bar to set or get information for. It can be SB_VERT for a vertical scroll bar, SB_HORZ for a horizontal scroll bar, or SB_CTL for a scroll bar control.

&si: A pointer to a SCROLLINFO structure that contains the scroll bar information to set or get.

bRedraw: (For SetScrollInfo only) A Boolean value indicating whether to redraw the scroll bar after applying the changes. TRUE to redraw, FALSE to skip redrawing.

SCROLLINFO Structure

The SCROLLINFO structure, defined as follows, holds the scroll bar information:

```
typedef struct tagSCROLLINFO {  
    UINT cbSize; // Size of the SCROLLINFO structure  
    UINT fMask; // Flags indicating which fields to set or get  
    int nMin; // Minimum value of the scroll bar range  
    int nMax; // Maximum value of the scroll bar range  
    UINT nPage; // Page size, used for proportional thumb sizing  
    int nPos; // Current position of the scroll bar thumb  
    int nTrackPos;  
    // Current tracking position of the scroll bar thumb  
    // (only valid during SB_THUMBTRACK or SB_THUMBPOSITION messages)  
} SCROLLINFO, * PSCROLLINFO;
```

Setting the cbSize Field

Before calling either SetScrollInfo or GetScrollInfo, you must initialize the cbSize field of the SCROLLINFO structure to the size of the structure:

```
si.cbSize = sizeof(si);
```

This allows for future compatibility with potential enhancements to the structure without breaking existing programs.

Example Usage

To set the scroll bar range for a vertical scroll bar:

```
SCROLLINFO si;
si.cbSize = sizeof(si);
si.fMask = SIF_RANGE;
si.nMin = 0;
si.nMax = 100;
SetScrollInfo(hwnd, SB_VERT, &si, TRUE);
```

To retrieve the current position of a horizontal scroll bar:

```
SCROLLINFO si;
si.cbSize = sizeof(si);
si.fMask = SIF_POS;
GetScrollInfo(hwnd, SB_HORZ, &si);
int currentPos = si.nPos;
```

Flag Syntax and Usage

The fMask field of the SCROLLINFO structure is used to specify which fields of the structure to set or get. The flags begin with the SIF prefix and can be combined using the bitwise OR (|) operator.

SIF_RANGE Flag

This flag is used to [set or get the scroll bar range](#).

When [used with SetScrollInfo](#), the nMin and nMax fields of the SCROLLINFO structure must be set to the desired minimum and maximum values, respectively.

When [used with GetScrollInfo](#), the nMin and nMax fields will be updated to reflect the current scroll bar range.

SIF_POS Flag

This flag is used to [set or get the current position of the scroll bar](#).

When [used with SetScrollInfo](#), the nPos field of the SCROLLINFO structure must be set to the desired scroll bar position.

When [used with GetScrollInfo](#), the nPos field will be updated to reflect the current scroll bar position.

SIF_PAGE Flag

This flag is used to [set or get the page size](#), which determines the proportional size of the scroll bar thumb.

When [used with SetScrollInfo](#), the nPage field of the SCROLLINFO structure must be set to the desired page size.

When [used with GetScrollInfo](#), the nPage field will be updated to reflect the current page size.

SIF_TRACKPOS Flag

This flag is [used to get the current 32-bit thumb position](#) when processing a WM_VSCROLL or WM_HSCROLL message with a notification code of SB_THUMBTRACK or SB_THUMBPOSITION.

The [nTrackPos field](#) of the SCROLLINFO structure will be updated to reflect the current thumb position.

SIF_DISABLENOSCROLL Flag

This flag is used with SetScrollInfo to [disable the scroll bar](#) instead of hiding it when the scroll bar arguments would normally make it invisible.

SIF_ALL Flag

This flag is a combination of SIF_RANGE, SIF_POS, SIF_PAGE, and SIF_TRACKPOS. It is useful for setting the scroll bar arguments during a WM_SIZE message or processing a scroll bar message.

SIF stands for Scroll Information Flag. It is a bitmask used in the SetScrollInfo() and GetScrollInfo() functions to specify which fields of the SCROLLINFO structure are to be set or retrieved.

Here's a table summarizing the flag usage:

Flag	Usage
SIF_RANGE	Set or get the scroll bar range
SIF_POS	Set or get the current position of the scroll bar
SIF_PAGE	Set or get the page size
SIF_TRACKPOS	Get the current 32-bit thumb position
SIF_DISABLENOSCROLL	Disable the scroll bar instead of hiding it
SIF_ALL	Combination of SIF_RANGE, SIF_POS, SIF_PAGE, and SIF_TRACKPOS

The SetScrollInfo and GetScrollInfo functions represent a significant advancement in scroll bar control and provide a more powerful and flexible approach to managing scroll bars in Windows applications.

Determining the Maximum Scroll Position

In SYSMETS2, the scroll bar range is unnecessarily large, extending beyond the visible area of the client area.

To address this, we can dynamically adjust the scroll bar range based on the window size and the height of the text elements.

Calculating Scroll Bar Range Using WM_SIZE

Instead of setting the scroll bar range during WM_CREATE, we can defer it until the WM_SIZE message is received, ensuring that the range is adjusted based on the actual window dimensions.

```
iVscrollMax = max(0, NUMLINES - cyClient / cyChar);
SetScrollRange(hwnd, SB_VERT, 0, iVscrollMax, TRUE);
```

In this code, NUMLINES represents the total number of lines of information, cyClient is the client area height, and cyChar is the character height. The max function ensures that the maximum scroll position (iVscrollMax) is never less than 0.

Using Scroll Bar Page Size for Automatic Range Adjustment

The new scroll bar functions introduce the concept of scroll bar page size, which represents the portion of the document displayed in the window.

This feature simplifies the calculation of the maximum scroll position and ensures that the scroll bar thumb is sized proportionally to the visible content.

```
si.cbSize = sizeof(SCROLLINFO);
si.fMask = SIF_RANGE | SIF_PAGE;
si.nMin = 0;
si.nMax = NUMLINES - 1;
si.nPage = cyClient / cyChar;
SetScrollInfo(hwnd, SB_VERT, &si, TRUE);
```

In this code, si is a SCROLLINFO structure, NUMLINES is the total number of lines, and cyClient and cyChar are as defined earlier.

The SetScrollInfo function automatically adjusts the maximum scroll position based on the page size (si.nPage), ensuring that the scroll bar range is appropriate for the visible content.

Handling Large Page Sizes

If the page size is equal to or larger than the scroll bar range, Windows automatically hides the scroll bar. This is a convenient behavior that eliminates unnecessary scroll bar controls when the entire content is visible.

Disabling Scroll Bar Instead of Hiding

If you prefer to keep the scroll bar visible even when the page size is large, you can use the [SIF_DISABLENOSCROLL flag](#) when calling SetScrollInfo.

This will **disable the scroll bar** instead of hiding it, allowing users to manually scroll through the content if needed.

By employing these techniques, you can [prevent excessive scrolling](#) and ensure that the scroll bar behavior is consistent with the actual amount of visible content.

I was able to run sysmets3 code. It's actually a better scrollbar with a hide feature:

Get System Metrics No. 3		
SM_CXSCREEN	Screen width in pixels	1280
SM_CYSCREEN	Screen height in pixels	1024
SM_CXVSCROLL	Vertical scroll width	17
SM_CYHSCROLL	Horizontal scroll height	17
SM_CYCAPTION	Caption bar height	23
SM_CXBORDER	Window border width	1
SM_CYBORDER	Window border height	1
SM_CXFIXEDFRAME	Dialog window frame width	3
SM_CYFIXEDFRAME	Dialog window frame height	3
SM_CYTHUMB	Vertical scroll thumb height	17
SM_CXHTHUMB	Horizontal scroll thumb width	17
SM_CXICON	Icon width	32
SM_CYICON	Icon height	32
SM_CXCURSOR	Cursor width	32
SM_CYCURSOR	Cursor height	32
SM_CYMENU	Menu bar height	20
SM_CXFULLSCREEN	Full screen client area width	1280
SM_CYFULLSCREEN	Full screen client area height	953
SM_CYKANJIWINDOW	Kanji window height	0
SM_MOUSEPRESENT	Mouse present flag	1
SM_CYSVSCROLL	Vertical scroll arrow height	17
SM_CXHSCROLL	Horizontal scroll arrow width	17
SM_DEBUG	Debug version flag	0
SM_SWAPBUTTON	Mouse buttons swapped flag	0
SM_CXMIN	Minimum window width	136
SM_CYMIN	Minimum window height	39
SM_CXSIZE	Min/Max/Close button width	36
SM_CYSIZE	Min/Max/Close button height	22
SM_CXSIZEFRAME	Window sizing frame width	4
SM_CYSIZEFRAME	Window sizing frame height	4
SM_CXMINTRACK	Minimum window tracking width	136
SM_CYMINTRACK	Minimum window tracking height	39
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

And an unhide feature, and it flows better than the former one:

Get System Metrics No. 3		
SM_CXSCREEN	Screen width in pixels	1280
SM_CYSCREEN	Screen height in pixels	1024
SM_CXVSCROLL	Vertical scroll width	17
SM_CYHSCROLL	Horizontal scroll height	17
SM_CYCAPTION	Caption bar height	23
SM_CXBORDER	Window border width	1
SM_CYBORDER	Window border height	1
SM_CXFIXEDFRAME	Dialog window frame width	3
SM_CYFIXEDFRAME	Dialog window frame height	3
SM_CYVTHUMB	Vertical scroll thumb height	17
SM_CXHTHUMB	Horizontal scroll thumb width	17
SM_CXICON	Icon width	32
SM_CYICON	Icon height	32
SM_CXCURSOR	Cursor width	32
SM_CYCURSOR	Cursor height	32
SM_CYMENU	Menu bar height	20
SM_CXFULLSCREEN	Full screen client area width	1280
SM_CYFULLSCREEN	Full screen client area height	953
SM_CYKANJIWINDOW	Kanji window height	0
SM_MOUSEPRESENT	Mouse present flag	1
SM_CYVSCROLL	Vertical scroll arrow height	17
SM_CXHSCROLL	Horizontal scroll arrow width	17
SM_DEBUG	Debug version flag	0
SM_SWAPBUTTON	Mouse buttons swapped flag	0
SM_CXMIN	Minimum window width	136
SM_CYMIN	Minimum window height	39
SM_CXSIZE	Min/Max/Close button width	36
SM_CYSIZE	Min/Max/Close button height	22
SM_CXSIZEFRAME	Window sizing frame width	4
SM_CYSIZEFRAME	Window sizing frame height	4
SM_CXMINTRACK	Minimum window tracking width	136
SM_CYMINTRACK	Minimum window tracking height	39
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

Reliance on Windows for Scroll Bar Management

SYSMETS3 simplifies scroll bar management by relying on Windows to handle most of the scroll bar information and boundary checking. Instead of manually tracking scroll bar positions and enforcing limits, the program retrieves the current scroll bar information at the beginning of WM_VSCROLL and WM_HSCROLL processing.

Handling Scroll Bar Notifications

The program adjusts the scroll bar position based on the notification code received in these messages. For instance, if the notification code indicates that the scroll bar thumb has been moved, the program adjusts the scroll bar position accordingly.

Setting and Retrieving Scroll Bar Positions

After adjusting the scroll bar position, SYSMETS3 calls SetScrollInfo to inform Windows of the new position. This allows Windows to maintain consistent scroll bar information across the system. To verify the updated position, the program calls GetScrollInfo and checks if the position was adjusted by Windows due to any boundary constraints.

Utilizing ScrollWindow for Efficient Scrolling

Instead of repainting the entire client area whenever scrolling occurs, SYSMETS3 employs the ScrollWindow function to perform the scrolling operation more efficiently. This function scrolls the client area horizontally and vertically based on the specified amounts.

Invalidation and WM_PAINT Message Handling

By setting the last two arguments of ScrollWindow to NULL, the program indicates that the entire client area should be scrolled. Windows automatically invalidates the exposed rectangle in the client area, triggering a WM_PAINT message. This eliminates the need to explicitly call InvalidateRect.

Ignoring SB_THUMBTRACK for Horizontal Scrolling

In WM_HSCROLL processing, SYSMETS3 ignores the SB_THUMBTRACK notification code and only responds to SB_THUMBPOSITION. This means that the program does not scroll the window horizontally while the user is dragging the thumb on the horizontal scroll bar. Instead, it updates the scroll bar position only when the user releases the mouse button.

Responsive Vertical Scrolling

In contrast to horizontal scrolling, SYSMETS3 scrolls the window vertically in real-time as the user drags the thumb on the vertical scroll bar. This is achieved by capturing SB_THUMBTRACK messages and updating the scroll bar position accordingly.

Potential Performance Issues with Frequent Scrolling

While responsive scrolling provides a more intuitive user experience, it can lead to performance issues if users rapidly drag the scroll bar thumb back and forth. This is because the program continuously updates the scroll position and repaints the client area in response to these rapid movements.

Considering Alternative Scrolling Strategies for Slow Machines

To optimize performance on slower machines, SYSMETS3 could incorporate the SB_SLOWMACHINE flag when calling GetSystemMetrics. This flag provides information about the system's processing speed and could be used to trigger less frequent scrolling updates or utilize alternative scrolling mechanisms.

Efficient WM_PAINT Handling

To optimize WM_PAINT processing, SYSMETS3 identifies the lines within the invalid rectangle and rewrites only those lines instead of repainting the entire client area. This approach reduces the amount of repainting required and improves overall performance.

USERS AND SCROLLBAR

User Preferences and Mouse Usage

While mice have become the standard input device for personal computers, some users still prefer to operate their systems primarily using the keyboard. This preference may be due to familiarity, comfort, or accessibility considerations.

Early Windows and Mouseless Operation

In the early versions of Windows, a significant portion of users relied solely on the keyboard for navigation and interaction. This was reflected in the design of the operating system and many applications, which were designed to be fully functional without mouse input.

Providing keyboard alternatives for scroll bar operations offers several advantages:

Accessibility: Users with limited hand mobility or dexterity may find it easier to use keyboard shortcuts rather than relying on a mouse.



Consistency: Keyboard shortcuts provide a consistent and predictable way to interact with scroll bars, regardless of the user's preference or ability to use a mouse.

CONSISTENCY



Efficiency: Experienced users can often navigate through scroll bars more quickly using keyboard shortcuts than with a mouse.



Maintaining Compatibility with Older Windows Behavior

The inclusion of code that handles WM_VSCROLL messages with notification codes SB_TOP and SB_BOTTOM in SYSMETS3 is likely intended to **maintain compatibility** with older versions of Windows, where these messages were sent for scroll bar operations.

Future Enhancements with Keyboard Interface

In the next chapter, you'll **explore techniques for implementing a keyboard interface for scroll bar operations**, allowing users to navigate through the content using keyboard shortcuts. This will enhance the program's accessibility and provide alternative input methods for those who prefer or require them.