INTRODUCTION: WHAT ARE CHILD WINDOW CONTROLS?

Up until now, we have been building one big, blank window. But real apps need buttons, checkboxes, lists, and scrollbars. In Windows API terms, these are called **Child Window Controls**.

The Big Secret: A "Button" is not some special graphical drawing. **It is a Window.** It has a Window Class, a Window Procedure, and it processes messages just like your main application window. The only difference is:

1. It is small.
2. It lives *inside* the client area of a "Parent" window.
3. It is pre-programmed to look and act like a button.

1. How to Create Them

You have two options:

- **Option A: The Hard Way (Custom Controls)** You write your own Window Procedure, draw the button yourself using GDI, handle mouse clicks, and register a new class. (We rarely do this unless we want something totally unique).
- **Option B: The Easy Way (Predefined Controls)** Windows comes with built-in Window Classes that are already registered for you. You just call CreateWindow and pass the specific class name.
    - "button"
    - "edit" (Text box)
    - "listbox"
    - "scrollbar"
    - "static" (Labels)

Example: To make a button, you don't call RegisterClass. You just say:
CreateWindow("button", "Click Me", ...)

2. Communication: The "Parent-Child" Talk

Since the button is a separate window, it needs a way to talk to your main window (the Parent).

- **Child to Parent (Notifications):** When you click the button, it sends a message to the Parent's WndProc saying, "Hey! I was clicked!" *(Technical Note: This is usually a WM_COMMAND message).*
- **Parent to Child (Commands):** If the Parent wants to change the button's text, it sends a message to the button saying, "Change your text to 'Submitted'." *(Technical Note: This is done via SendMessage or SetWindowText).*

3. Where do they live?

You will use controls in two main environments:

A. On a Normal Window (The "Manual" Way)

- **What it is:** You place a button directly on your main app screen.
- **The Catch:** You have to do everything yourself. You must calculate the X/Y coordinates. If the user resizes the window, the button stays stuck in place unless you write code to move it. You also have to manage "Focus" (which window receives keyboard input).

B. In a Dialog Box (The "Manager" Way)

- **What it is:** A special popup window (like "File > Open" or "Settings").
- **The Benefit:** Windows includes a **Dialog Manager**. It handles the layout, the tab order, and the focus for you. It is much easier to set up.

4. "Standard" vs. "Common" Controls

- **Standard Controls:** The basics that have been in Windows since version 1.0 (Buttons, Edit boxes, Scrollbars). This chapter focuses on these.
- **Common Controls:** The fancy modern ones (Progress Bars, Tree Views, Sliders). These live in a separate library and are more complex to set up.

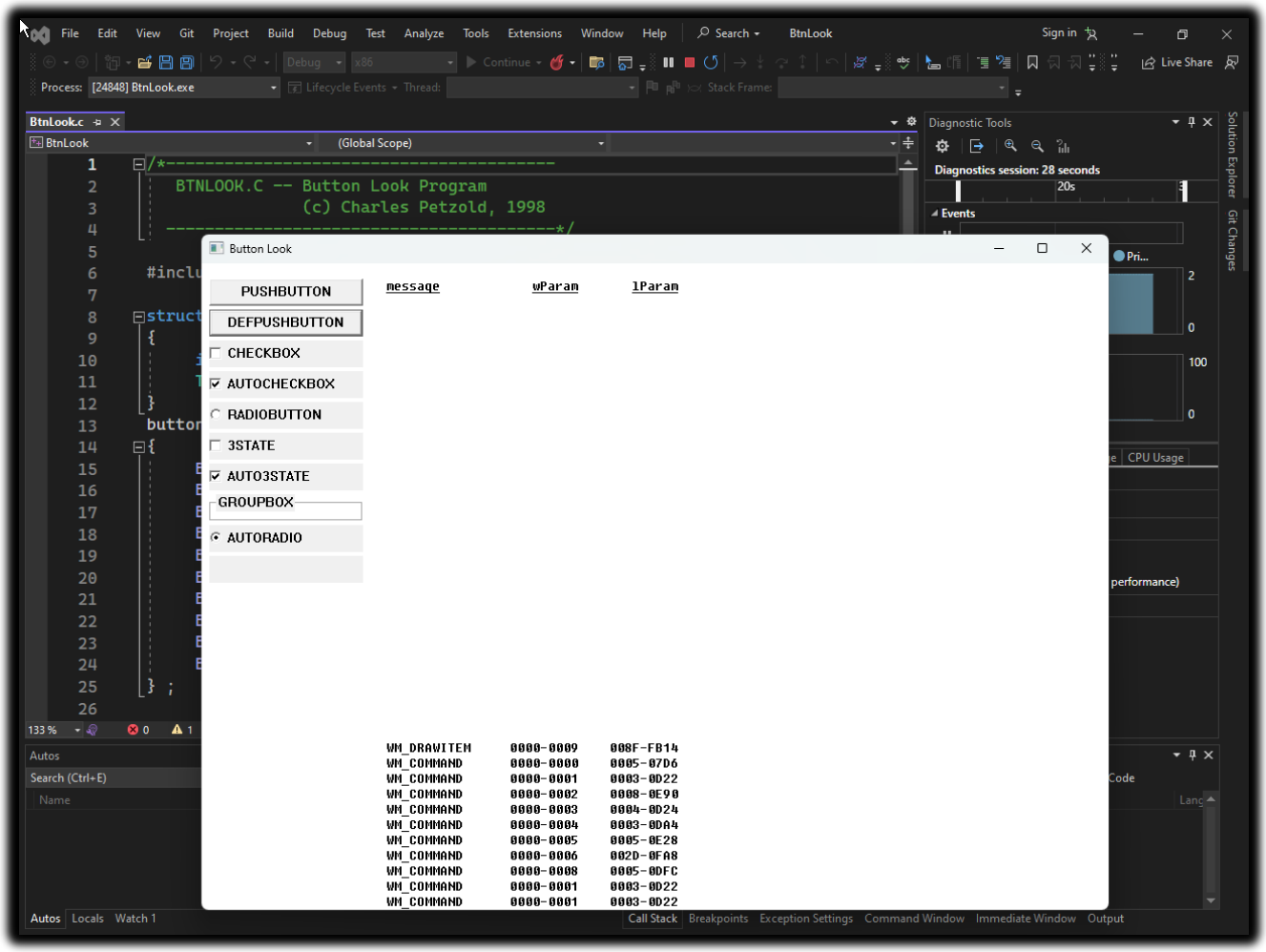
4. Quick Review

Question 1: Is a "Button" inside your application a completely different object type than the Application Window itself? *(Answer: No! They are both just "Windows." The button is just a child window of the application window.)*

Question 2: If you want to create a standard push button, do you need to write a WNDCLASS and register it? *(Answer: No. You use the pre-defined class name "button" inside CreateWindow.)*

Question 3: Why is putting controls on a "Normal Window" harder than in a "Dialog Box"? *(Answer: In a normal window, you have to manually calculate positions and handle resizing code. In a Dialog, the Dialog Manager handles much of that for you.)*

BtnLook program in chapter 9...



The video illustration…

BtnLook
program.mp4

The BTNLOOK Program: A "Button Zoo"

The **BTNLOOK** program is essentially a showcase. It doesn't do any useful work; instead, it displays 10 different types of buttons on the screen so you can see how they look and behave.

1. The Goal

To demonstrate the **10 Standard Button Styles** available in Windows. It also acts as a "spy" tool: whenever you click a button, the program prints the exact details of the message (wParam and lParam) that the button sent to the parent.

2. Key Mechanics

The "Owner-Draw" Button: One of the buttons has the style BS_OWNERDRAW. This means Windows *won't* paint it. The program itself must listen for WM_DRAWITEM and manually draw the button's face (using GDI functions).

Message Spy: When a button is clicked, it sends a WM_COMMAND message. The main window catches this and draws the message details on the right side of the screen.

3. Message Handling Breakdown

MESSAGE	ACTION IN BTNLOOK
`WM_CREATE`	Initializes the UI. Calls `CreateWindow` 10 times to build the various button controls (Push, Radio, Checkbox, etc.).
`WM_SIZE`	Responsive Design. If you stretch the window, this code runs to reposition the buttons so they stay organized and neat.
`WM_PAINT`	Draws the descriptive text labels (e.g., "Push Button", "Checkbox") next to the actual controls.
`WM_DRAWITEM`	Specific to the **Owner-Draw** button. Contains the custom GDI drawing logic for that specific button's appearance.
`WM_COMMAND`	**The Click Handler.** It grabs the ID of the clicked button and displays it to the user.

Creating Child Windows: The Recipe

A child window is just a window that lives inside another one. This includes buttons, text boxes, and lists.

1. The CreateWindow Call

You use the exact same function as you did for your main window, but the parameters are tweaked.

```
hwndButton = CreateWindow(
    TEXT("button"),       // 1. Class Name (Predefined by Windows)
    TEXT("Click Me"),     // 2. Window Text (The label on the button)
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, // 3. Styles
    x, y,                 // 4. Position (Relative to parent's client area)
    width, height,        // 5. Size
    hwndParent,           // 6. Parent Window Handle (Crucial!)
    (HMENU) iButtonID,    // 7. Child ID (Cast to HMENU type)
    hInstance,            // 8. Instance Handle
    NULL                  // 9. Extra Params
);
```

2. Critical Parameters Explained

Class Name ("button"): This tells Windows to use its built-in logic for drawing and handling clicks. You don't need to write a WndProc for this!

Styles (WS_CHILD | WS_VISIBLE):

- WS_CHILD: "I am attached to a parent. If the parent moves, I move. If the parent minimizes, I hide."
- WS_VISIBLE: "Show me immediately." (Without this, the button is created invisible).

Parent Window (hwndParent): This links the button to your main window.

Child ID: In a main window, this is the Menu Handle slot. Since child windows don't have menus, we reuse this slot to store a unique integer ID (like 1, 2, 100). This ID is what you check later in WM_COMMAND.

Understanding the "Button Types" (Styles)

When you create a button, you add a flag to the style parameter to tell Windows what *kind* of button it is.

- **BS_PUSHBUTTON:** A normal "OK" or "Cancel" button.
- **BS_DEFPUSHBUTTON:** A button with a thick black border (usually the "Enter" key trigger).
- **BS_CHECKBOX:** A square box with text.
- **BS_AUTOCHECKBOX:** Same as above, but Windows handles the "check mark" toggling automatically.
- **BS_RADIOBUTTON:** A circle. Used for "one of many" choices.
- **BS_GROUPBOX:** A rectangular frame with a title, used to group other controls visually.
- **BS_OWNERDRAW:** A blank slate. You draw whatever you want.

Quick Review

Question 1: If you create a button but forget WS_VISIBLE, what happens? *(Answer: The button exists in memory and can receive messages, but the user cannot see it on the screen.)*

Question 2: Where does the button send its notification messages (like "I was clicked")? *(Answer: To the Parent Window's WndProc, specifically as a WM_COMMAND message.)*

Question 3: What is the difference between BS_CHECKBOX and BS_AUTOCHECKBOX? *(Answer: BS_CHECKBOX requires you to manually write code to draw the checkmark when clicked. BS_AUTOCHECKBOX toggles the checkmark automatically without extra code.)*

The Button Creation Loop (WM_CREATE)

Instead of writing CreateWindow 10 separate times, the program uses a for loop to create all 10 buttons efficiently. It pulls the text and styles from a data array (button[]).

1. The Code Logic

The goal is to stack the buttons vertically on the left side of the window.

```
for (i = 0; i < 10; i++)
{
    hwndChild[i] = CreateWindow(
        TEXT("button"),                    // 1. Class Name (Always "button")
        button[i].szText,                  // 2. Text (e.g., "Push Button")
        WS_CHILD | WS_VISIBLE | button[i].iStyle, // 3. Style flags
        cxChar,                            // 4. X Position (1 character indent)
        cyChar * (1 + 2 * i),              // 5. Y Position (Calculated row height)
        20 * cxChar,                       // 6. Width (20 characters wide)
        7 * cyChar / 4,                    // 7. Height (1.75 characters tall)
        hwnd,                              // 8. Parent Window Handle
        (HMENU) i,                         // 9. Child ID (0 to 9)
        ((LPCREATESTRUCT) lParam)->hInstance, // 10. Instance Handle
        NULL                               // 11. Extra Params
    );
}
```

2. Analyzing the Parameters

TEXT("button"): This is the magic word. It tells Windows, "Use your internal code to make a button." If you typo this (e.g., "Button" with a capital B in older versions), it fails.

The Style (WS_CHILD | WS_VISIBLE):

- **WS_CHILD**: Mandatory. Without this, the button tries to be a standalone desktop window (and usually fails or looks weird).
- **WS_VISIBLE**: Crucial. If you forget this, the button is created but remains invisible until you manually call ShowWindow.

The Position Math (y coordinate):

- cyChar * (1 + 2 * i)
- This formula spaces them out. i=0 is at line 1. i=1 is at line 3. i=2 is at line 5. It leaves a gap between each button.

The ID ((HMENU) i):

- Notice we cast the integer i to HMENU.
- **Why?** The function expects a Menu Handle here. But for child windows, this slot is repurposed to hold the **Control ID**. We will use this ID (0 through 9) later to identify which button was clicked.

Important Concepts

1. System Metrics (cxChar / cyChar)

The code relies heavily on cxChar and cyChar. These represent the average width and height of a character in the system font.

- **Why?** By using these instead of hard pixels (e.g., "100 pixels"), the buttons automatically scale up if the user has a larger font size or high-DPI screen.

2. The Instance Handle

((LPCREATESTRUCT) lParam)->hInstance

- In WinMain, hInstance is easy to get.
- In WndProc, it's harder. When WM_CREATE fires, lParam points to a structure containing the creation data. We extract hInstance from there to pass it to the child window.

Quick Review

Question 1: Why do we cast i to (HMENU) in the CreateWindow call? *(Answer: Because the function signature demands an HMENU type in that position, even though we are actually passing an integer ID.)*

Question 2: If you change 20 * cxChar to 5 * cxChar, what happens? *(Answer: The buttons become very narrow (5 characters wide), likely cutting off the text inside them.)*

Question 3: Does this loop draw the buttons? *(Answer: No. CreateWindow creates the button logic. Windows then automatically generates a WM_PAINT message for the buttons, causing them to draw themselves.)*

GETTING THE INSTANCE HANDLE (HINSTANCE)

When creating child windows (like buttons), the CreateWindow function demands the **Instance Handle** of your application (hInstance). But how do you get it inside your WndProc? The user's notes outline two popular ways to solve this.

The Problem: In WinMain, you have easy access to hInstance because it is passed as a parameter. In WndProc, however, you are isolated. You don't automatically have access to that variable. Here are the two ways to fix this:

Method 1: The Global Variable (The "Quick & Dirty" Way)

This is the method described in your notes. It is very common in older C programs (like Petzold's examples) because it is simple.

1. Create a Global Variable: Put this at the very top of your .c file, outside any function.

```
HINSTANCE hInst; // Global variable visible to all functions
```

2. Initialize it in WinMain: As soon as the program starts, save the handle.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
    hInst = hInstance; // Save the local parameter to the global variable
    // ... rest of your code ...
}
```

3. Use it in WndProc: Now you can use hInst anywhere.

```
hwndButton = CreateWindow(TEXT("button"), ..., hInst, NULL);
```

Method 2: The API Approach (GetWindowLongPtr)

If you dislike global variables (which is good practice in modern coding), you can ask Windows to look it up for you using the Window Handle (hwnd).

Note: Your notes had a typo "Gggyy.iLdgyyLgng". This refers to GetWindowLong.

The Modern Code (64-bit safe):

```
// Ask Windows: "Who created this window?"
HINSTANCE hInst = (HINSTANCE)GetWindowLongPtr(hwnd, GWLP_HINSTANCE);
```

- **hwnd**: The handle of your main window.
- **GWLP_HINSTANCE**: A flag telling Windows to fetch the Instance Handle associated with this window.

Summary Table

METHOD	PROS	CONS
Global Variable	Very easy to understand. Highly efficient (direct memory access).	"Pollutes" the global namespace. Discouraged in strict/modular C++ architecture.
`GetWindowLongPtr`	Clean. No globals needed. Data is tied directly to the window handle (`HWND`).	Requires a function call (slightly slower). Requires the `HWND` to be valid to access data.
`LPCREATESTRUCT`	The "Purest" way. Allows passing initial data directly during window creation.	Only accessible during the `WM_CREATE` message processing.

Quick Review

Question 1: Why can't you just write hInstance inside WndProc without doing anything else? *(Answer: Because hInstance is a local variable inside WinMain. WndProc cannot see inside WinMain.)*

Question 2: Your notes mentioned GetModuleHandle(NULL). What does that do? *(Answer: It retrieves the instance handle of the currently running file (.exe). It's another way to initialize the global variable if you didn't save the one from WinMain.)*

Question 3: Which method is better if you are writing a large, complex application? *(Answer: Method 2 (API) or passing it via a class structure. Avoiding global variables prevents bugs where different parts of the program overwrite each other's data.)*

Handling Button Clicks (WM_COMMAND): The Notification

When a user clicks a button, the button itself doesn't launch a missile or save a file. It is just a dumb window. Instead, it picks up a telephone and calls its Parent Window.

- **The Caller:** The Child Button.
- **The Receiver:** The Parent Window (WndProc).
- **The Message:** WM_COMMAND.

Decoding the Message Parameters

The WM_COMMAND message packs three vital pieces of information into two variables (wParam and lParam). You have to "unpack" them to understand what happened.

PARAMETER	PART	WHAT IT HOLDS	MEANING
`wParam`	Low Word	Child ID	*"Who called?" (The ID you assigned in `CreateWindow`, e.g., 0 through 9).*
	High Word	Notification Code	*"What happened?" (e.g., `BN_CLICKED`, `BN_DOUBLECLICKED`).*
`lParam`	Full Value	Child Handle	*The actual `HWND` (Window Handle) of the button control.*

The Code Pattern: To extract these values, you use macros:

```
case WM_COMMAND:
    int id = LOWORD(wParam);          // Which button?
    int code = HIWORD(wParam);        // What did it do?
    HWND hButton = (HWND) lParam;     // The button handle

    if (code == BN_CLICKED) {
        // Do something!
    }
    break;
```

The Notification Codes (The "Action" Types)

The "Notification Code" tells you exactly what the user did to the button. In the BTNLOOK program, these are displayed so you can see the internal mechanics.

Here are the standard Button Notifications (BN_):

Notification Code Identifier	Value	Description
BN_CLICKED	0	The button has been clicked.
BN_PAINT	1	The button needs to be repainted.
BN_HILITE or BN_PUSHED	2	The button has been highlighted or pushed.
BN_UNHILITE or BN_UNPUSHED	3	The button has been unhighlighted or unpushed.
BN_DISABLE	4	The button has been disabled.
BN_DOUBLECLICKED or BN_DBLCLK	5	The button has been double-clicked.
BN_SETFOCUS	6	The button has received the input focus.
BN_KILLFOCUS	7	The button has lost the input focus.

Why do we need BN_CLICKED vs BN_PUSHED?

- **BN_PUSHED** happens the moment your finger goes *down*.
- **BN_CLICKED** happens only after your finger goes *down AND up* while still over the button.
- *Rule of Thumb:* Always listen for BN_CLICKED. It allows the user to change their mind (by dragging the mouse away before releasing).

Quick Review

Question 1: If you receive a WM_COMMAND message, how do you know which button triggered it? *(Answer: Check the Low Word of wParam (LOWORD(wParam)). It contains the ID number you assigned in CreateWindow.)*

Question 2: Does lParam contain the ID of the button? *(Answer: No. lParam contains the **Handle** (HWND) of the button. The ID is in wParam.)*

Question 3: In the BTNLOOK program, why do we see BN_PAINT notifications? *(Answer: Because one of the buttons was created with the BS_OWNERDRAW style. This tells Windows "I will draw this button myself," so Windows sends BN_PAINT whenever that button needs updating.)*

The "Focus" Shift

Before looking at the code, it is important to understand what happens to the keyboard when you click a button.

- **Stealing Focus:** When you click a child button, the button grabs the **Input Focus**.
- **The Consequence:** Your main window stops receiving keyboard messages (WM_KEYDOWN). Instead, the *Button* gets them.
- **Button Behavior:** The button control is programmed to ignore most keys, but it listens for the **Spacebar**. Pressing Space while a button has focus pushes the button (same as a mouse click).

The Code: Decoding WM_COMMAND

When the parent receives WM_COMMAND, it needs to unpack the data to answer: "Who called?" and "What did they do?"

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_COMMAND:
        {
            // 1. Extract the Data
            int iID   = LOWORD(wParam);           // The Child ID (e.g., 0-9)
            int iCode = HIWORD(wParam);           // The Notification (e.g., BN_CLICKED)
            HWND hwndChild = (HWND) lParam;       // The Button's Handle

            // 2. Format the Message (Safe String Handling)
            // We use TCHAR arrays to support both Unicode and ANSI
            TCHAR szBuffer[256];

            // wsprintf is the standard Windows way to format strings (like printf)
            wsprintf(szBuffer,
                     TEXT("Child Window ID: %d\nNotification Code: %d"),
                     iID, iCode);

            // 3. Display it
            // Note: MessageBox pauses the program until you click OK.
            MessageBox(hwnd, szBuffer, TEXT("Button Notification"), MB_OK);
        }
        return 0;

    // ... handle other messages ...

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

Code Breakdown (Why we fixed it this way)

LOWORD & HIWORD Macros: The wParam is a 32-bit integer. Windows packs two 16-bit integers inside it to save space.

- **Low 16 bits:** The ID.
- **High 16 bits:** The Notification Code.
- *Correction from your notes:* You had LOKORD and HIWRD. The correct macros are LOWORD and HIWORD.

wsprintf vs snprintf: Your snippet tried to use snprintf with TEXT() macros. This often causes errors because snprintf is for standard ASCII C, while TEXT often implies Unicode (Wide Characters).

- **Best Practice:** In Windows API, use wsprintf (simple) or StringCchPrintf (safe) which automatically handle the TCHAR type logic.

The Formatting: We put both values into *one* buffer.

- *Why?* Calling MessageBox twice is annoying for the user. They would have to click "OK" for the ID, and then click "OK" again for the Code. Doing it in one shot is much cleaner.

A Critical Note on MessageBox vs. Real Apps

In the BTNLOOK program described in Petzold's book, the program does **not** actually use MessageBox.

- **Why?** MessageBox is "Modal"—it freezes the entire application until you close the popup.
- **The Better Way:** The actual BTNLOOK program simply saves the values to variables and calls InvalidateRect(hwnd, NULL, TRUE). This triggers a repaint, and the program draws the text directly on the window background. This allows you to click buttons rapidly without being interrupted by popups.

Quick Review

Question 1: When a button has focus, what does the Spacebar do? *(Answer: It triggers a click event (BN_CLICKED).)*

Question 2: Why did we use curly braces { ... } inside the case WM_COMMAND: block? *(Answer: In C/C++, if you declare new variables (like szBuffer) inside a switch case, you must wrap that case in braces to define the scope.)*

Question 3: What is HIWORD(wParam) used for in this message? *(Answer: It retrieves the **Notification Code** (e.g., seeing if the button was clicked vs. double-clicked).)*

HOW PARENT WINDOW TALKS TO ITS CHILD WINDOW IN BTNLOOK:

Think of the **parent window** as a manager and the **child windows (buttons)** as workers.

The manager doesn't click the buttons itself. Instead, it **talks to them by sending messages**.

1. Sending Messages to Child Windows

A parent window can send messages to its child windows to tell them what to do or to ask them questions. For example, it can:

- Ask a button what state it's in
- Tell a button to change how it looks
- Turn a button on or off

It does this by sending messages directly to the child window, kind of like sending a short note that says, *"Change this"* or *"What's your status?"*

2. Button-Specific Messages

Buttons also understand **special messages** that only buttons know how to respond to.

These messages start with **BM**, which stands for **Button Message**. You can think of them as button-only commands—like a remote control that works only on buttons.

Windows defines eight of these button messages in WINUSER.H. Each one lets the parent window control or query a button in a specific way, such as checking it, unchecking it, or changing its behavior.

Button Message	Value	Description
BM_GETCHECK	0x00F0	Retrieves the check mark state of a check box or radio button.
BM_SETCHECK	0x00F1	Sets the check mark state of a check box or radio button.
BM_GETSTATE	0x00F2	Retrieves the state of a button (normal, pushed, or disabled).
BM_SETSTATE	0x00F3	Sets the state of a button (normal, pushed, or disabled).
BM_SETSTYLE	0x00F4	Changes the style of a button.
BM_CLICK	0x00F5	Simulates a mouse click on a button.
BM_GETIMAGE	0x00F6	Retrieves the image associated with a button.
BM_SETIMAGE	0x00F7	Sets the image associated with a button.

3. Check Marks (Check Boxes and Radio Buttons)

Think of a check box or radio button like a light switch.

- **BM_GETCHECK** is how the parent asks:
  "Is this switch on or off?"
- **BM_SETCHECK** is how the parent says:
  "Turn this switch on" or *"Turn it off."*

The parent window sends these messages to the button to check or change whether the mark is there.

---

4. Button State (Normal, Pressed, Disabled)

A button can be in different moods:

- **Normal** (ready to be clicked), **Pressed** (being pushed) or **Disabled** (grayed out and unusable)
- **BM_GETSTATE** is the parent asking:
  "What mood are you in right now?"
- **BM_SETSTATE** is the parent telling the button:
  "Look pressed" or *"Go disabled."*

---

4. Changing How a Button Looks (Style)

The **style** controls how a button looks and behaves.

BM_SETSTYLE is used when the parent wants to change the button's look or behavior.

Think of it like changing a button's outfit.

---

5. Fake a Mouse Click on a Button

Sometimes you want a button to act like it was clicked, even if the user didn't touch the mouse.

BM_CLICK is like the parent saying: *"Pretend you were clicked right now."*

This makes the button run its normal click action automatically.

---

6. Button Images (Icons or Pictures)

Buttons can have pictures on them.

- **BM_GETIMAGE** asks:
  "What picture are you showing?"
- **BM_SETIMAGE** says:
  "Show this new picture instead."

This is useful when you want the button's appearance to change.

---

7. Child Window ID (Name Tag)

Every child window has a unique ID, like a **name tag**.

You can get this ID by using:

- GetWindowLong, or
- GetDlgCtrlID

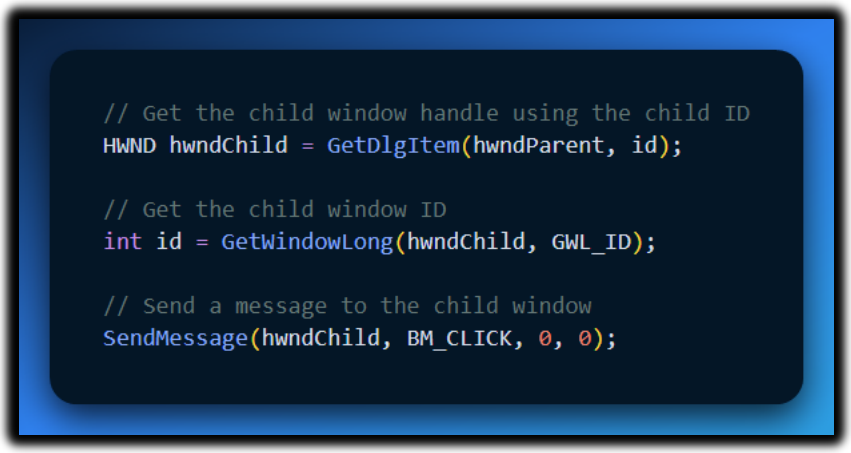
8. Child Window Handle (Phone Number)

If the ID is the name tag, the **handle** is like the phone number.

Once you know:

- the parent window, and
- the child window's ID,

you can get the child's handle using **GetDlgItem**, so you can talk to it directly.



```
// Get the child window handle using the child ID
HWND hwndChild = GetDlgItem(hwndParent, id);

// Get the child window ID
int id = GetWindowLong(hwndChild, GWL_ID);

// Send a message to the child window
SendMessage(hwndChild, BM_CLICK, 0, 0);
```

This example code:

- Finds the child
- Identifies it
- Sends it a message it understands

In short, the parent finds a child window using its ID, gets its handle, and then communicates with it by sending a message.

PUSH BUTTON DEFINITION AND APPEARANCE

A **push button** is the most common kind of button you see in Windows programs. It's a rectangle with some text on it, like **OK**, **Cancel**, or **Submit**.

When a push button is created with CreateWindow, the text you give it becomes the label on the button. The button fills the entire width and height you specify, and the text is automatically centered inside the rectangle.

Push buttons are mainly used to do **one immediate action**. You click the button, the action happens, and that's it. The button does **not** stay on or off. This is why they are often used in dialog boxes for things like accepting or canceling something.

Types of Push Buttons

There are two kinds of push buttons:

- **BS_PUSHBUTTON**
- **BS_DEFPUSHBUTTON**

The **DEF** in BS_DEFPUSHBUTTON means **default**.

In dialog boxes, the default push button is special. It's usually the button that activates when the user presses **Enter** on the keyboard.

However, when these buttons are used as **child window controls**, both types behave the same way. The only visual difference is that a BS_DEFPUSHBUTTON has a **thicker, darker border**, making it stand out more.

How a Push Button Should Look

A push button looks best when its height is about **1¾ times the height of the text** inside it. This gives the button enough space so it doesn't look squished.

The width of the button should be wide enough to hold:

- the text, plus
- a little extra space on both sides

The BTNLOOK program follows these rules so the buttons look clean and balanced.

What Happens with the Mouse

When the mouse cursor is over a push button and the user presses the mouse button, the push button redraws itself with a **3D effect**. This makes it look like the button is being pushed inward.

When the mouse button is released:

- the button goes back to its normal look
- the button sends a WM_COMMAND message to the parent window
- the notification code sent is BN_CLICKED

This is how the parent window knows the button was clicked.

What Happens with the Keyboard

When a push button has keyboard focus, you'll see a **dashed rectangle** around the text on the button.

If the user presses and releases the **Spacebar**, it works exactly the same as clicking the button with the mouse. The button visually presses down and then sends the same BN_CLICKED message to the parent window.

Simple Way to Remember

- Mouse click = action happens
- Spacebar = same action
- Button doesn't stay on or off
- Parent window is notified with BN_CLICKED

Simulating Push Button States

You can simulate a push-button flash by sending the window a BM_SETSTATE message. This causes the button to appear depressed:

```
SendMessage(hwndButton, BM_SETSTATE, 1, 0);
```

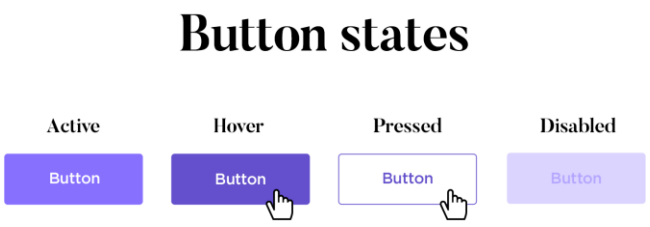
To restore the button to its normal state, use the following SendMessage call:

```
SendMessage(hwndButton, BM_SETSTATE, 0, 0);
```

In both cases, hwndButton is the window handle returned by the CreateWindow call.

Retrieving Push Button State

You can send a BM_GETSTATE message to a push button to retrieve its current state. The child window control returns TRUE if the button is depressed and FALSE if it is not depressed. However, most applications do not require this information.



Additional Notes

- Push buttons do not retain any on/off information, so the BM_SETCHECK and BM_GETCHECK messages are not used.
- Push buttons are typically used in conjunction with event handlers to perform actions when clicked.

CHECK BOX DEFINITION AND APPEARANCE

A check box is a square box with text typically appearing to the right of it. Check boxes are commonly used in applications to allow users to select options. They function as toggle switches: clicking the box once causes a check mark to appear; clicking again toggles the check mark off.

Types of Check Boxes

There are two main types of check boxes:

- BS_CHECKBOX: This style requires the programmer to control the check mark state using BM_SETCHECK and BM_GETCHECK messages.
- BS_AUTOCHECKBOX: This style automatically toggles the check mark state when clicked and doesn't require any manual intervention.

BS_CHECKBOX Handling

To manage the check mark state of a BS_CHECKBOX check box, you can use the following code:

```
250 int isChecked = (int)SendMessage(hwndButton, BM_GETCHECK, 0, 0); // Get current check state
251 SendMessage(hwndButton, BM_SETCHECK, isChecked ? 0 : 1, 0); // Toggle check state
```

This code retrieves the current check state using BM_GETCHECK and then toggles the state using BM_SETCHECK.

BS_AUTOCHECKBOX Handling

For BS_AUTOCHECKBOX check boxes, you can simply ignore WM_COMMAND messages and use BM_GETCHECK to retrieve the check state:

```
int isChecked = (int)SendMessage(hwndButton, BM_GETCHECK, 0, 0);
```

Additional Check Box Styles

BS_3STATE: This style allows a third state, indicated by a grayed-out check mark, which occurs when you send WM_SETCHECK with wParam equal to 2. This state indicates an indeterminate or irrelevant selection.

BS_AUTO3STATE: This style automatically toggles the check mark state between the three states (unchecked, checked, indeterminate) when clicked.

Check Box Alignment and Dimensions

The check box is aligned with the rectangle's left edge and centered within the top and bottom dimensions specified during the CreateWindow call. The minimum height for a check box is one character height, and the minimum width is the number of characters in the text, plus two.

User Interaction and Messages

Clicking anywhere within the check box rectangle sends a WM_COMMAND message to the parent window. The parent window can use this message to handle the check box selection and update its state accordingly.

RADIO BUTTONS

Radio buttons are a type of control that allows users to select one of a group of mutually exclusive options. They are commonly used in dialog boxes to present a set of choices, where only one choice can be selected at a time.

Radio buttons resemble check boxes, but instead of a square box, they have a small circle. A filled circle indicates that the radio button is selected.

Radio buttons typically have the window style BS_RADIOBUTTON or BS_AUTORADIOBUTTON. The latter style is specifically designed for use in dialog boxes.

Radio Button Behavior

Unlike check boxes, radio buttons do not function as toggles. Clicking a selected radio button does not deselect it. Instead, selecting one radio button automatically deselects any other radio buttons in the same group.

Radio Button State Management

When you receive a WM_COMMAND message from a radio button, you should update the state of all radio buttons in the same group.

To select the radio button that sent the WM_COMMAND message, send it a BM_SETCHECK message with wParam equal to 1:

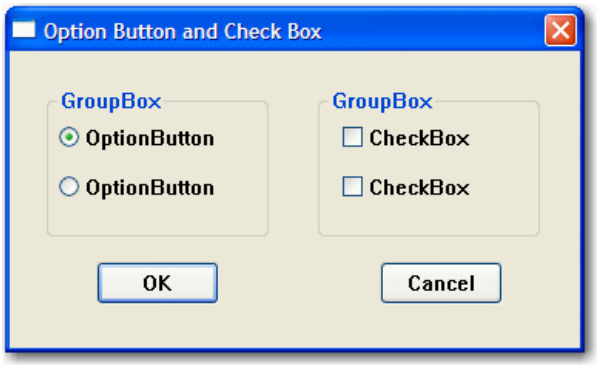
```
SendMessage(hwndButton, BM_SETCHECK, 1, 0);
```

To deselect all other radio buttons in the same group, send them BM_SETCHECK messages with wParam equal to 0:

```
for (int i = 0; i < numRadioButtons; i++) {
    HWND otherRadioButton = GetDlgItem(hwndParent, i);
    if (otherRadioButton != hwndButton) {
        SendMessage(otherRadioButton, BM_SETCHECK, 0, 0);
    }
}
```

GROUP BOXES

Group boxes, which have the BS_GROUPBOX style, are non-interactive controls that serve to visually group related control elements. They are commonly used to enclose other button controls, such as radio buttons or check boxes, to provide a clear visual distinction between different groups of options.



Group Box Appearance

Group boxes consist of a rectangular outline with their window text displayed at the top. They do not have any associated check mark or other visual indication of their state.

Group Box Function

Group boxes do not process mouse or keyboard input, nor do they send WM_COMMAND messages to their parent window. Their primary purpose is to organize and group related controls to enhance the user interface's clarity and usability.

CHANGING BUTTON TEXT

To change the text displayed on a button, you can use the SetWindowText function. This function takes two arguments:

- hwnd: The handle to the button window you want to modify.
- pszString: A pointer to a null-terminated string containing the new text for the button.

Here's an example of how to change the text of a button:

```
255  HWND hwndButton = CreateWindow(TEXT("BUTTON"), TEXT("Original Text"), WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 0, 0, 100, 30, hwndParent, NULL, NULL, NULL);
256
257  // Change the button text
258  SetWindowText(hwndButton, TEXT("New Text"));
```

Obtaining Button Text

You can retrieve the current text displayed on a button using the GetWindowText function. This function takes three arguments:

- hwnd: The handle to the button window you want to get the text from.
- pszBuffer: A pointer to the buffer where the retrieved text will be stored.
- iMaxLength: The maximum number of characters to copy into the buffer.

The function returns the length of the copied string, or zero if an error occurred.

Here's an example of how to get the current text of a button:

```
TCHAR pszText[256];
int iLength = GetWindowText(hwndButton, pszText, sizeof(pszText));
```

Visible and Enabled Buttons

For a button to respond to mouse and keyboard input, it must be both visible and enabled. When a button is visible but not enabled, its text is displayed in gray.

Making a Button Visible

To make a button visible, you can include the WS_VISIBLE style in the window class when creating the button. Alternatively, you can call the ShowWindow function with the SW_SHOWNORMAL flag after creating the button.

Here's an example of making a button visible using the window class style:

```
HWND hwndButton = CreateWindow(TEXT("BUTTON"), TEXT("Button Text"), WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 0, 0, 100, 30, hwndParent, NULL, NULL, NULL);
```

Here's an example of making a button visible using ShowWindow:

```
HWND hwndButton = CreateWindow(TEXT("BUTTON"), TEXT("Button Text"), WS_CHILD | BS_PUSHBUTTON, 0, 0, 100, 30, hwndParent, NULL, NULL, NULL);
ShowWindow(hwndButton, SW_SHOWNORMAL);
```

Hiding a Button

To hide a button, you can call the ShowWindow function with the SW_HIDE flag.

Here's an example of hiding a button:

```
ShowWindow(hwndButton, SW_HIDE);
```

Enabling and Disabling Buttons

By default, a button is enabled. To disable a button, you can call the EnableWindow function with the FALSE flag. When a button is disabled, its text appears in gray, and it does not respond to mouse or keyboard input.

Here's an example of disabling a button:

```
EnableWindow(hwndButton, FALSE);
```

Enabling a Button

To enable a disabled button, you can call the EnableWindow function with the TRUE flag.

Here's an example of enabling a disabled button:

```
EnableWindow(hwndButton, TRUE);
```

Checking Button Visibility and Enabled State

You can determine whether a button is visible using the IsWindowVisible function. This function takes one argument:

- hwnd: The handle to the button window you want to check.

The function returns TRUE if the button is visible and FALSE if it is hidden.

Here's an example of checking whether a button is visible:

```
BOOL isVisible = IsWindowVisible(hwndButton);
```

You can determine whether a button is enabled using the IsWindowEnabled function. This function takes one argument:

- hwnd: The handle to the button window you want to check.

The function returns TRUE if the button is enabled and FALSE if it is disabled.

Here's an example of checking whether a button is enabled:

```
BOOL isEnabled = IsWindowEnabled(hwndButton);
```

INPUT FOCUS AND BUTTONS

How buttons interact with input focus and how to prevent them from stealing focus from the parent window:

When a push button, check box, radio button, or owner-draw button is clicked with the mouse, it gains input focus. This is indicated by a dashed line that surrounds the text of the control.

When a child window control gains input focus, it receives all keyboard input instead of the parent window. However, most button controls only respond to the Spacebar, which acts as a simulated mouse click.

Preventing Buttons from Stealing Focus

To prevent a button from taking input focus away from the parent window, you can process WM_KILLFOCUS messages in the parent window's message handling function.

When a WM_KILLFOCUS message is received, it indicates that the parent window is about to lose input focus.

You can check if the window losing focus is one of the child window controls by comparing it to the handles stored in an array. If it is, you can call SetFocus to restore the input focus to the parent window.

Code Example 1

```
case WM_KILLFOCUS:
for (i = 0; i < NUM; i++) {
    if (hwndChild[i] == (HWND) wParam) {
        SetFocus(hwnd);
        break;
    }
}
return 0;
```

In this code, the parent window checks each child window handle in the array and restores focus to itself if the losing focus window matches one of the child window handles.

Alternative Code Example:

```
case WM_KILLFOCUS:
if (hwnd == GetParent((HWND) wParam)) {
    SetFocus(hwnd);
}
return 0;
```

This alternative approach directly compares the parent window handle to the window losing focus. It is simpler but less obvious than the first method.

Limitations of Preventing Focus Stealing

Both of these methods have a drawback: they prevent the button from responding to the Spacebar keypress. This is because the button never gains input focus. A better solution would allow the button to receive input focus while also enabling tab navigation between buttons.

Window Subclassing for Improved Focus Handling

A technique called "window subclassing" can be used to achieve this. Subclassing allows you to intercept and modify the behavior of an existing window procedure. By subclassing the button window procedure, you can capture keyboard events, including the Tab key, and handle them appropriately.

Window Subclassing Implementation

The COLORS1 program in the later part of the chapter demonstrates how to implement window subclassing to handle button focus and tab navigation. It involves creating a subclass procedure that overrides the default button procedure and handles keyboard events accordingly.

Buttons can interfere with keyboard input by stealing focus from the parent window. Techniques like WM_KILLFOCUS processing and window subclassing can be employed to prevent this and maintain control over keyboard input while still allowing buttons to function as expected.

We've come this far but i want you to explain to me like a teenager, what is input focus, with illustrations, and what is a handle?

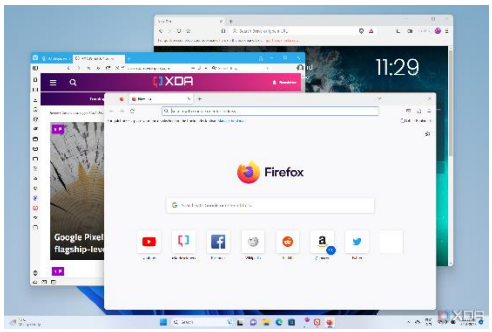
Input Focus

Imagine you're sitting at a computer with multiple open programs, each with its own window. When you click on a particular window, it becomes the active window, and that's where your keyboard input goes. That's what input focus is – it's the ability of a window to receive keyboard input.

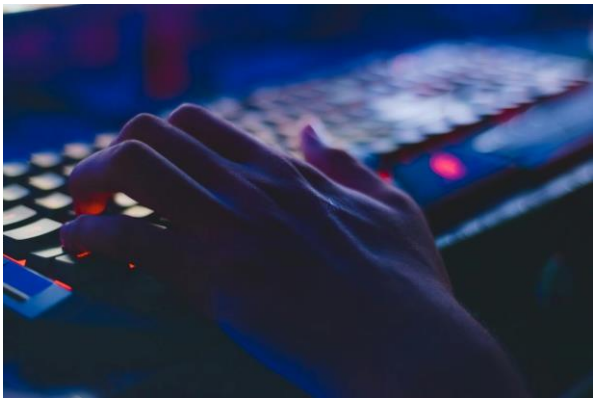
Think of it like a spotlight. When you shine the spotlight on a particular window, that window is in focus, and it's like you're talking directly to that window. Other windows might be open, but they're not paying attention to your keyboard input.

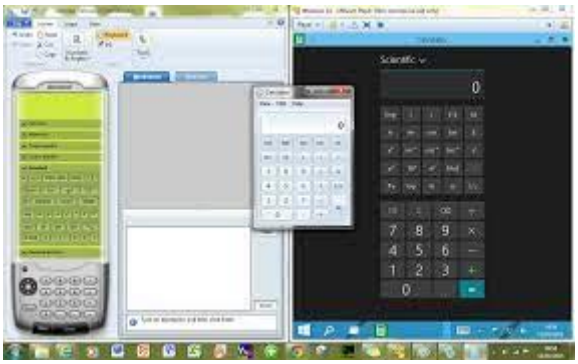
Illustration

Let's say you have a web browser window open, and you're typing a search query. The web browser window has the input focus, so all your keystrokes go towards entering the search term.



If you switch to a different window, like a calculator app, and start pressing buttons, the calculator app gets the input focus, and your keystrokes now control the calculator instead.





Window Handles

Every window has a unique identifier called a window handle. It's like a special address that lets your computer identify and keep track of all the different windows you have open.

Think of it like a house address. Each house has a unique address that allows the postman to deliver mail to the right place. Similarly, window handles allow your computer to send messages to the correct windows.

Relationship between Input Focus and Window Handles

The window handle is the behind-the-scenes mechanism that allows input focus to be assigned to specific windows. When you click on a window, your computer uses the window handle to identify that window and give it the input focus.

So, input focus is like the spotlight that highlights the currently active window, and window handles are like the unique addresses that let your computer identify and control those windows.

System Colors in Windows

System colors are a set of predefined colors that Windows uses to paint various elements of the graphical user interface (GUI), such as window borders, titles, buttons, and text. These

colors are stored by the system and can be accessed using the GetSysColor and SetSysColors functions.

Table of System Colors

GetSysColor and SetSysColors Identifier	Registry Key or WIN.INI Value	Default RGB Value	Description
COLOR_SCROLLBAR	Scrollbar	C0-C0-C0	The color of scrollbars.
COLOR_BACKGROUND	Background	00-80-80	The color of the background behind windows.
COLOR_ACTIVECAPTION	ActiveTitle	00-00-80	The color of the title bar of the active window.
COLOR_INACTIVECAPTION	InactiveTitle	80-80-80	The color of the title bar of inactive windows.
COLOR_MENU	Menu	C0-C0-C0	The color of the background of menus.
COLOR_WINDOW	Window	FF-FF-FF	The color of the background of windows.
COLOR_WINDOWFRAME	WindowFrame	00-00-00	The color of the border of windows.
COLOR_MENUTEXT	MenuText	C0-C0-C0	The color of text in menus.
COLOR_WINDOWTEXT	WindowText	00-00-00	The color of text in windows.
COLOR_CAPTIONTEXT	TitleText	FF-FF-FF	The color of text in title bars.
COLOR_ACTIVEBORDER	ActiveBorder	C0-C0-C0	The color of the border of the active window.
COLOR_INACTIVEBORDER	InactiveBorder	C0-C0-C0	The color of the border of inactive windows.
COLOR_APPWORKSPACE	AppWorkspace	80-80-80	The color of the background of non-client areas, such as the desktop and the Start menu.
COLOR_HIGHLIGHT	Highlight	00-00-80	The color of the highlight when selecting text or items.
COLOR_HIGHLIGHTTEXT	HighlightText	FF-FF-FF	The color of text in the highlight.
COLOR_BTNFACE	ButtonFace	C0-C0-C0	The color of the face of buttons.
COLOR_BTNSHADOW	ButtonShadow	80-80-80	The color of the shadow of buttons.
COLOR_GRAYTEXT	GrayText	80-80-80	The color of grayed-out text.
COLOR_BTNTEXT	ButtonText	00-00-00	The color of text on buttons.

COLOR_INACTIVECAPTIONTEXT	InactiveTitleText	C0-C0-C0	The color of text in inactive title bars.
COLOR_BTNHIGHLIGHT	ButtonHighlight	FF-FF-FF	The color of the highlight on buttons when the mouse is over them.
COLOR_3DDKSHADOW	ButtonDkShadow	00-00-00	The color of the darkest shadow of buttons.
COLOR_3DLIGHT	ButtonLight	C0-C0-C0	The color of the lightest light of buttons.
COLOR_INFOTEXT	InfoText	00-00-00	The color of text in message boxes.
COLOR_INFOBK	InfoWindow	FF-FF-FF	The color of the background of message boxes.
No identifier	ButtonAlternateFace	B8-B4-B8	The color of the alternate face of buttons.
COLOR_HOTLIGHT	HotTrackingColor	00-00-FF	The color of hot tracked items.
COLOR_GRADIENTACTIVECAPTION	GradientActiveTitle	00-00-80	The gradient color of the active title bar.
COLOR_GRADIENTINACTIVECAPTION	GradientInactiveTitle	80-80-80	The gradient color of the inactive title bar.

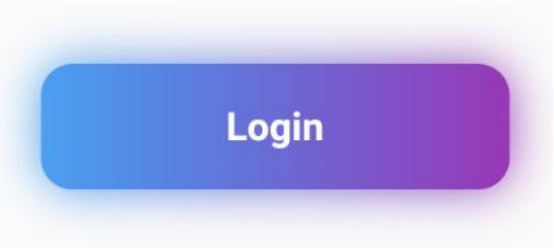
The default RGB values for these colors can vary slightly depending on the display driver.

CHALLENGES WITH SYSTEM COLORS FOR BUTTONS

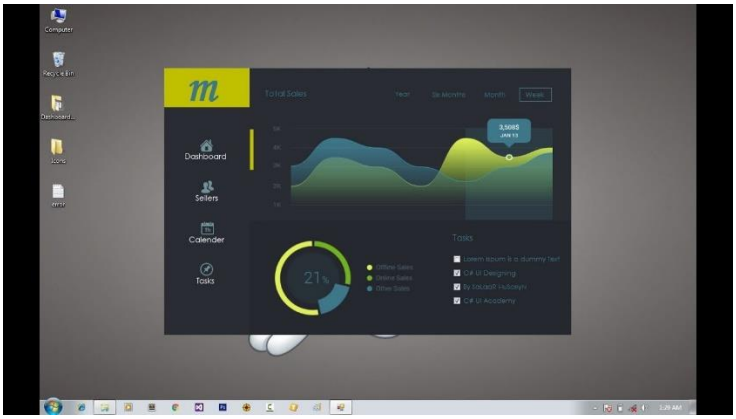
In recent versions of Windows, the use of system colors for buttons has become increasingly complex due to the growing visual complexity of controls and the introduction of three-dimensional appearances. This poses several challenges for programmers:

Inconsistent Color Usage: While some system colors have intuitive names that match their intended purpose, others have become less consistent, making it difficult to predict the exact color behavior.

Multiple Colors per Button: Each button requires multiple system colors for its various elements, such as the face, shadow, text, and border. This increases the complexity of managing button colors.



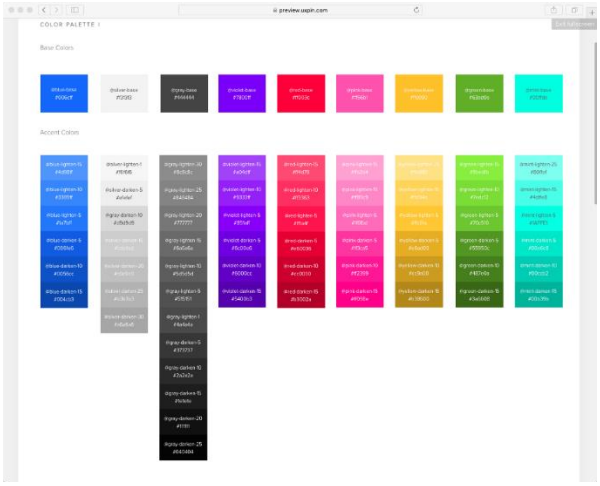
Color Clash with Client Area: If the client area background color is set to the default white, it clashes with the system colors used for buttons, creating an inconsistent visual appearance.



Solutions to Address Color Issues

To address these challenges, programmers can employ several strategies:

Yield to System Colors: By setting the client area background color to COLOR_BTNFACE, the client area matches the default button face color, eliminating the color clash.



Explicitly Set Text Colors: Since the default text colors in the device context are white (background) and black (text), programmers need to explicitly set the text background color to COLOR_BTNFACE and the text color to COLOR_WINDOWTEXT to match the button colors.

Handle System Color Changes: If the user changes system colors while the program is running, the client area needs to be invalidated to reflect the new colors. This can be done using the WM_SYSCOLORCHANGE message.

Alternative Approach: Custom Colors

An alternative approach is to avoid using system colors altogether and define custom colors for the client area, buttons, and text.

This provides more control over the visual appearance and eliminates the need to handle system color changes. However, this approach requires managing multiple custom colors and ensuring consistency across the application.

Code examples:

```
wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1);
```

This code sets the background color of the client area to COLOR_BTNFACE, which is the system color used for dialog boxes and message boxes. This helps to avoid color clash with the buttons.

```
SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));
SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));
```

This code sets the text background color and text color to the system colors COLOR_BTNFACE and COLOR_WINDOWTEXT, respectively. This ensures that the text is consistent with the button colors.

```
case WM_SYSCOLORCHANGE:
InvalidateRect(hwnd, NULL, TRUE);
break;
```

This code handles the WM_SYSCOLORCHANGE message, which is sent when the system colors change. The code invalidates the client area, which causes Windows to redraw it using the new system colors.

Here's an explanation of the code:

- wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1);: This line sets the background color of the client area to COLOR_BTNFACE, which is a system color defined by Windows. The + 1 is necessary because Windows expects the value of hbrBackground to be one more than the system color identifier.
- SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));: This line sets the background color of the current device context to COLOR_BTNFACE. The device context is used for drawing text and graphics.
- SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));: This line sets the text color of the current device context to COLOR_WINDOWTEXT. This is the system color used for window text.
- case WM_SYSCOLORCHANGE: InvalidateRect(hwnd, NULL, TRUE); break;: This code handles the WM_SYSCOLORCHANGE message, which is sent when the system colors change. The InvalidateRect function causes Windows to redraw the client area of the window.

- The NULL parameter specifies that the entire client area should be redrawn. The TRUE parameter tells Windows to send a WM_PAINT message to the window when the redrawing is complete. This message is necessary to trigger the window's paint handling function, which will redraw the window with the new system colors.

WM_CTLCOLORBTN Message

The WM_CTLCOLORBTN message is sent to the parent window of a button control before the button is about to paint its client area. This gives the parent window the opportunity to customize the colors used to paint the button.

Message Parameters

- wParam: The handle to the button's device context.
- lParam: The button's window handle.

Processing WM_CTLCOLORBTN

When the parent window procedure receives a WM_CTLCOLORBTN message, it can perform the following actions:

- Set Text Color: Use SetTextColor to set the text color of the button.
- Set Text Background Color: Use SetBkColor to set the text background color of the button.
- Return Brush Handle: Return a handle to a brush that will be used to paint the button's background.

Limitations of WM_CTLCOLORBTN

- Limited Scope: Only push buttons and owner-drawn buttons send WM_CTLCOLORBTN to their parent windows.

- Ineffective for Owner-Drawn Buttons: Owner-drawn buttons are already responsible for drawing their own backgrounds, so processing WM_CTLCOLORBTN for them is redundant.

Alternative Approaches

- SetSysColors: Use SetSysColors to change the system colors for buttons. However, this affects all buttons in the system, which may not be desirable.
- Custom Controls: Create custom controls that handle their own drawing and color management.

While WM_CTLCOLORBTN offers a mechanism for customizing button colors, its limitations make it less useful for practical applications. Alternative approaches, such as using SetSysColors or creating custom controls, may be more suitable for achieving specific color customizations.

OnDraw.mp4

Owner-Draw Buttons

The OWNDRAW program demonstrates the use of owner-draw buttons, which provide complete control over the visual appearance of buttons.

The program consists of two main parts: the WinMain function and the WndProc window procedure.

The WinMain function performs the following tasks:

- Register the Window Class: Registers the window class that defines the appearance and behavior of the window.
- Create the Main Window: Creates the main window of the application using the registered window class.
- Show the Window: Displays the main window on the screen.
- Enter the Message Loop: Enters the message loop, which processes messages sent to the window until the window is closed.

WndProc Window Procedure

The WndProc window procedure handles messages sent to the window. The program handles the following messages:

- WM_CREATE: Initializes the window by creating two owner-draw buttons.
- WM_SIZE: Resizes the buttons when the window size changes.
- WM_COMMAND: Handles button clicks by resizing the window.
- WM_DRAWITEM: Draws the owner-draw buttons.

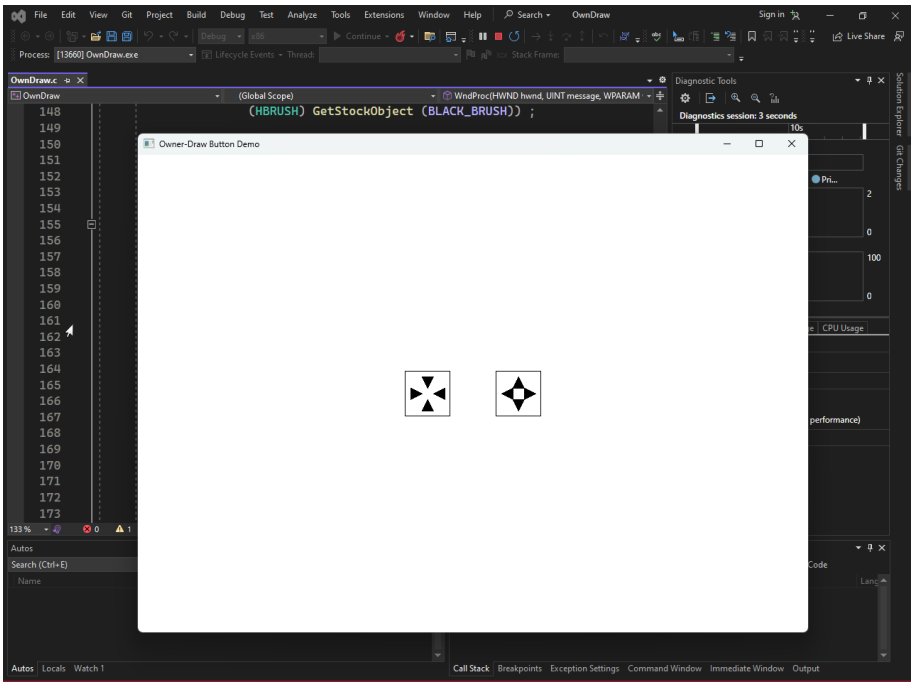
Drawing Owner-Draw Buttons

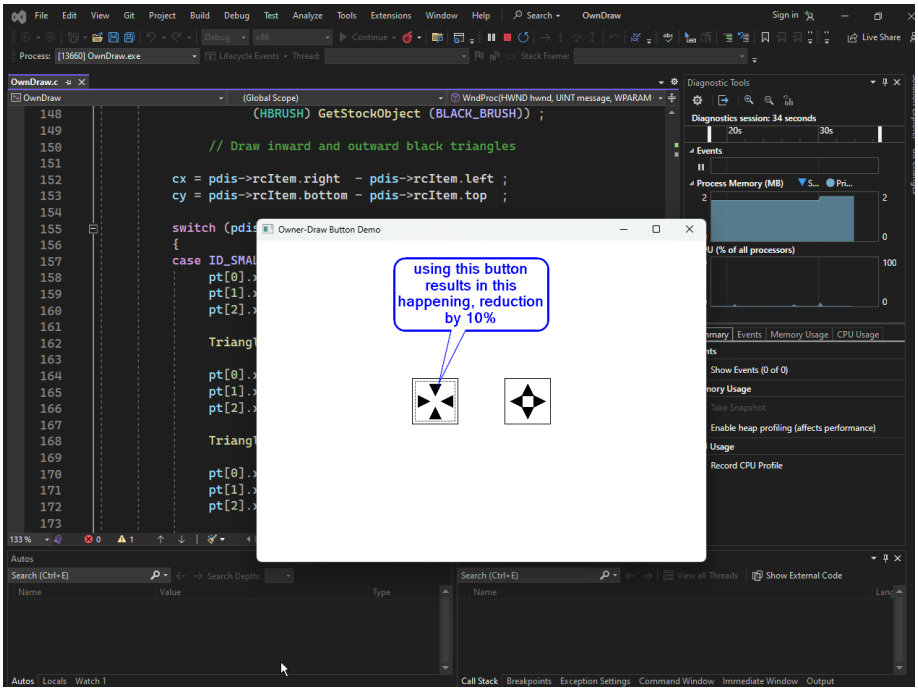
The WM_DRAWITEM message handler is responsible for drawing the owner-draw buttons. It uses the Triangle function to draw triangles on the buttons and the InvertRect and DrawFocusRect functions to handle button selection and focus.

Button Functionality

The button on the left decreases the window size by 10% when clicked, while the button on the right increases the window size by 10%. This is achieved by modifying the window's rectangle and calling the MoveWindow function to update the window's position and size.

The OWNDRAW program demonstrates the use of owner-draw buttons to create custom button appearances. Owner-draw buttons provide flexibility in designing buttons but require more programming effort compared to standard buttons.





And vice versa...

Button Creation and Positioning

- During the WM_CREATE message, OWNDRAW creates two buttons with the BS_OWNERDRAW style.
- The buttons are given a width of eight times the system font and four times the system font height.
- This creates buttons that are approximately 64 by 64 pixels on a VGA monitor.
- The buttons are not yet positioned at this stage.
- During the WM_SIZE message, OWNDRAW positions the buttons in the center of the client area by calling MoveWindow.

Button Click Handling

- When the left button is clicked, it generates a WM_COMMAND message.
- OWNDRAW processes the WM_COMMAND message by calling GetWindowRect to store the position and size of the entire window in a RECT structure.
- The position is relative to the screen.
- OWNDRAW then adjusts the fields of this rectangle structure to decrease the window size by 10%.
- The program then repositions and resizes the window by calling MoveWindow.
- This generates another WM_SIZE message, and the buttons are repositioned in the center of the client area.
- Similarly, when the right button is clicked, OWNDRAW processes the WM_COMMAND message by increasing the window size by 10%.

Button Drawing

- A button created with the BS_OWNERDRAW style sends its parent window a WM_DRAWITEM message whenever the button needs to be repainted.
- The lParam message parameter is a pointer to a structure of type DRAWITEMSTRUCT.
- The OWNDRAW program stores this pointer in a variable named pdis.
- This structure contains the information necessary for a program to draw the button.
- (The same structure is also used for owner-draw list boxes and menu items.)

The structure fields important for working with buttons are:

- hDC: The device context for the button.
- rcItem: A RECT structure providing the size of the button.
- CtlID: The control window ID.
- itemState: Which indicates whether the button is pushed or has the input focus.
- OWNDRAW begins WM_DRAWITEM processing by calling FillRect to erase the surface of the button with a white brush.
- It then calls FrameRect to draw a black frame around the button.
- Next, OWNDRAW draws four black-filled triangles on the button by calling Polygon.
- This is the normal button appearance.
- If the button is currently being pressed, a bit of the itemState field of the DRAWITEMSTRUCT will be set.
- OWNDRAW tests this bit using the ODS_SELECTED constant.
- If the bit is set, OWNDRAW inverts the colors of the button by calling InvertRect.
- This creates a pressed button effect.
- If the button has the input focus, the ODS_FOCUS bit of the itemState field will be set.
- In this case, OWNDRAW draws a dotted rectangle just inside the periphery of the button by calling DrawFocusRect.
- This indicates that the button has the input focus.

Considerations for Owner-Draw Buttons

- When using owner-draw buttons, make sure to leave the device context in the same state you found it.
- Any GDI objects selected into the device context must be unselected before returning from the WM_DRAWITEM message handler.
- Be careful not to draw outside the rectangle defining the boundaries of the button.

STATIC CLASSES IN C AND WINAPI

In C and WinAPI, static classes are used to create child window controls that are drawn but do not interact with the user. They are typically used to display text, images, or other static content. Static controls are created by using the CreateWindow function with the "static" window class.

Characteristics of Static Classes

Static classes have the following characteristics:

- Do not accept mouse or keyboard input: Static controls do not have a focus rectangle and do not respond to mouse clicks or keyboard presses.
- Do not send WM_COMMAND messages: Static controls do not send WM_COMMAND messages to their parent windows.
- Trap WM_NCHITTEST messages: When the mouse moves over a static control, the control traps the WM_NCHITTEST message and returns HTTRANSPARENT. This allows mouse clicks to pass through the static control to the underlying window.

Types of Static Classes

There are three main types of static classes:

- Rectangular static controls: These controls draw a solid rectangle or a frame in the client area of the child window. The color of the rectangle or frame is based on the system colors.
- Text static controls: These controls display text in the client area of the child window. The text can be left-justified, right-justified, or centered.
- Icon static controls: These controls display an icon in the client area of the child window. Icon static controls are not commonly used.

Creating Static Classes

To create a static class, you use the CreateWindow function with the "static" window class. The CreateWindow function takes the following parameters

- Parent window handle: The handle of the parent window for the static control.
- Window style: The style of the static control. The style can be one of the rectangular static control styles, one of the text static control styles, or the SS_ICON style.
- Window text: The text to display in the static control. This parameter is ignored for rectangular static controls.
- X-coordinate: The x-coordinate of the upper-left corner of the static control.
- Y-coordinate: The y-coordinate of the upper-left corner of the static control.
- Width: The width of the static control.
- Height: The height of the static control.

Customizing Static Classes

You can customize the appearance of static classes by intercepting the WM_CTLCOLORSTATIC message.

This message is sent to the parent window of the static control before the static control is painted.

You can use the SetTextColor and SetBkColor functions to change the text color and background color of the static control, respectively.

You can also return a handle to a custom brush to change the background pattern of the static control.

Example of Static Classes

The following code snippet creates a static control that displays the text "Hello, world!" in the client area of a parent window:

```
HWND hStaticControl = CreateWindow(
  "static",
  "Hello, world!",
  WS_VISIBLE | WS_CHILD,
  10,
  10,
  100,
  25,
  hParentWindow,
  (HMENU) 1,
  hInstance,
  NULL
);
```

Static classes are a versatile tool for adding text and images to your WinAPI applications. They are easy to create and customize, and they do not interfere with the user's ability to interact with other controls in your application. Here’s the full table:

Static Window Style	Description
SS_BLACKRECT	Draws a black rectangle in the client area of the child window.
SS_BLACKFRAME	Draws a black frame in the client area of the child window.
SS_GRAYRECT	Draws a gray rectangle in the client area of the child window.
SS_GRAYFRAME	Draws a gray frame in the client area of the child window.
SS_WHITERECT	Draws a white rectangle in the client area of the child window.
SS_WHITEFRAME	Draws a white frame in the client area of the child window.
SS_ETCHEDHORZ	Creates a shadowed-looking frame with the white and gray colors, with a horizontal emphasis.
SS_ETCHEDVERT	Creates a shadowed-looking frame with the white and gray colors, with a vertical emphasis.
SS_ETCHEDFRAME	Creates a shadowed-looking frame with the white and gray colors, with both horizontal and vertical emphasis.
SS_LEFT	Creates left-justified text.
SS_RIGHT	Creates right-justified text.
SS_CENTER	Creates centered text.
SS_ICON	Not applicable to child window controls.
SS_USERITEM	Not applicable to child window controls.

SCROLL BAR CLASS

The scroll bar class is used to create child window scroll bars that can appear anywhere in the client area of the parent window. Unlike button controls, scroll bar controls do not send WM_COMMAND messages to the parent window. Instead, they send WM_VSCROLL and WM_HSCROLL messages.

Creating Scroll Bar Controls

To create a scroll bar control, you use the CreateWindow function with the predefined window class "scrollbar" and one of the two scroll bar styles SBS_VERT and SBS_HORZ. The CreateWindow function takes the following parameters:

- Parent window handle: The handle of the parent window for the scroll bar control.
- Window style: The style of the scroll bar control. The style can be SBS_VERT or SBS_HORZ.
- Window text: The text to display in the scroll bar control. This parameter is ignored.
- X-coordinate: The x-coordinate of the upper-left corner of the scroll bar control.
- Y-coordinate: The y-coordinate of the upper-left corner of the scroll bar control.
- Width: The width of the scroll bar control.
- Height: The height of the scroll bar control.

Understanding lParam Parameter

When processing the scroll bar messages, you can differentiate between window scroll bars and scroll bar controls by the lParam parameter. It will be 0 for window scroll bars and the scroll bar window handle for scroll bar controls.

Setting Scroll Bar Range and Position

You can set the range and position of a scroll bar control with the same calls used for window scroll bars:

- SetScrollRange: Sets the minimum and maximum positions of the scroll bar.
- SetScrollPos: Sets the current position of the scroll bar.
- SetScrollInfo: Sets the minimum, maximum, and page size of the scroll bar, as well as the current position and an optional scroll bar info structure.

Colorizing Scroll Bar Controls

You can trap WM_CTLCOLORSCROLLBAR messages to override the color used for the large area between the two end buttons. This allows you to customize the appearance of the scroll bar control.

Colors chapter 9.mp4

The program structure:

1. Creating Child Window Controls

The COLORS1 program creates 10 child window controls: 3 scroll bars, 6 windows of static text, and 1 static rectangle. Child window controls are created using the CreateWindow function, which takes the following parameters:

- Parent window handle: The handle of the parent window for the child window.
- Window class: The window class name of the child window.
- Window style: The window style of the child window.
- X-coordinate: The x-coordinate of the upper-left corner of the child window.
- Y-coordinate: The y-coordinate of the upper-left corner of the child window.
- Width: The width of the child window.
- Height: The height of the child window.

2. Trapping WM_CTLCOLORSCROLLBAR Messages

The COLORS1 program traps WM_CTLCOLORSCROLLBAR messages to color the interior sections of the three scroll bars red, green, and blue. This is done by returning a handle to a brush from the message. The brush is created using the CreateSolidBrush function, which takes the desired color as a parameter.

3. Trapping WM_CTLCOLORSTATIC Messages

The COLORS1 program traps WM_CTLCOLORSTATIC messages to color the static text. This is done by returning a handle to a brush from the message. The brush is created using the CreateSolidBrush function, which takes the desired color as a parameter.

4. Using VK_TAB to Switch Focus

The COLORS1 program uses the VK_TAB key to switch focus between the three scroll bars. This is done by using the SetFocus function to set the focus to the next scroll bar in the tab order.

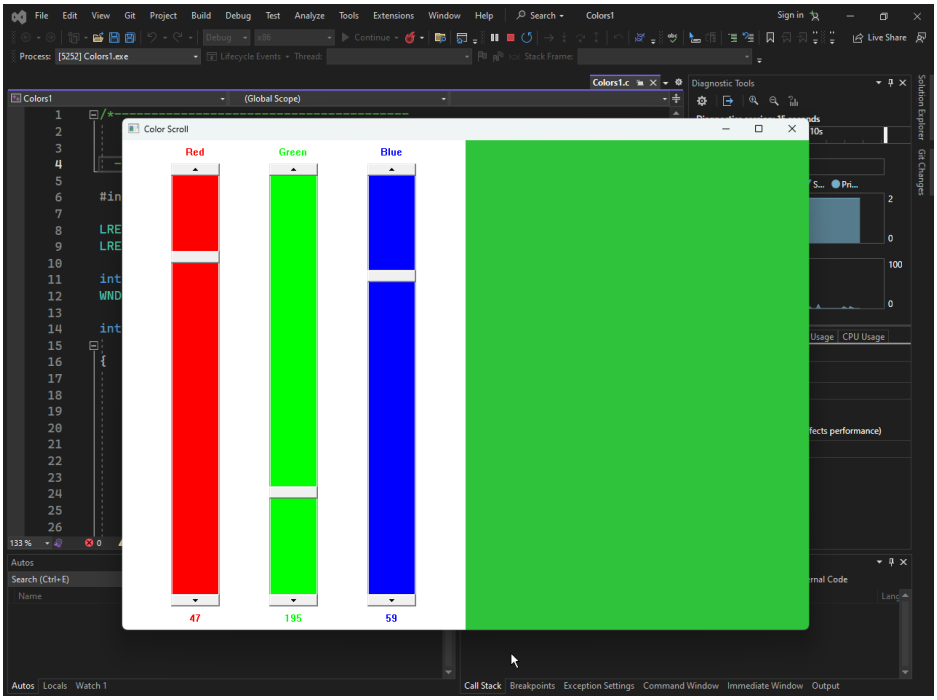
5. Using GetKeyState to Determine Shift Key State

The COLORS1 program uses the GetKeyState function to determine whether the Shift key is pressed when the VK_TAB key is pressed. This is done to change the direction of the tab order.

6. Using DefWindowProc for Unhandled Messages

The COLORS1 program uses the DefWindowProc function for unhandled messages. This is done to ensure that the default window procedure is called for messages that the program does not handle.

These are just a few of the crucial concepts in the COLORS1 code. The code is a good example of how to use child window controls, trap messages, and handle input in a WinAPI application.



1. COLORS1's Window Procedure

The COLORS1 program's window procedure, WndProc, handles most of the work for the program. It receives messages from the operating system and performs the appropriate actions. Here are some of the key messages that WndProc handles:

WM_CREATE: This message is sent when the window is created. WndProc uses this message to create the child windows.

WM_SIZE: This message is sent when the window is resized. WndProc uses this message to resize the child windows.

WM_VSCROLL: This message is sent when a scroll bar is scrolled. WndProc uses this message to update the color of the client area.

WM_CTLCOLORSCROLLBAR: This message is sent when the operating system needs to draw the interior of a scroll bar. WndProc uses this message to color the interior of the scroll bars red, green, and blue.

WM_CTLCOLORSTATIC: This message is sent when the operating system needs to draw the text of a static control. WndProc uses this message to color the text of the static controls.

2. Child Windows

COLORS1 uses a number of child windows to implement its functionality. Child windows are windows that are created within another window. In COLORS1, the child windows are used to display the scroll bars, the color labels, and the color values.

Window ID	Window Class	Style	Function
0	scrollbar	SBS_VERT	Red scroll bar
1	scrollbar	SBS_VERT	Green scroll bar
2	scrollbar	SBS_VERT	Blue scroll bar
3	static	SS_CENTER	Red label
4	static	SS_CENTER	Green label
5	static	SS_CENTER	Blue label
6	static	SS_CENTER	Red value
7	static	SS_CENTER	Green value
8	static	SS_CENTER	Blue value
9	static	SS_WHITERECT	White rectangle

3. WM_VSCROLL Message Handling

The WM_VSCROLL message is sent when a scroll bar is scrolled. WndProc uses this message to update the color of the client area. Here are the steps that WndProc takes to handle this message:

- Get the ID of the scroll bar that sent the message.
- Get the new value of the scroll bar.
- Update the color of the client area based on the new values of the scroll bars.
- Update the text of the static control that displays the value of the scroll bar.

4. WM_CTLCOLORSCROLLBAR Message Handling

The WM_CTLCOLORSCROLLBAR message is sent when the operating system needs to draw the interior of a scroll bar. WndProc uses this message to color the interior of the scroll bars red, green, and blue. Here are the steps that WndProc takes to handle this message:

- Get the ID of the scroll bar that sent the message.
- Create a brush of the appropriate color.
- Return the brush handle to the operating system.

5. WM_CTLCOLORSTATIC Message Handling

The WM_CTLCOLORSTATIC message is sent when the operating system needs to draw the text of a static control. WndProc uses this message to color the text of the static controls. Here are the steps that WndProc takes to handle this message:

- Get the ID of the static control that sent the message.
- Set the text color of the static control to the appropriate color.
- Set the background color of the static control to the appropriate color.
- Return the brush handle to the operating system.

Conclusion

Scroll bar controls are a useful tool for adding scroll functionality to your WinAPI applications. They are easy to create and customize, and they can be used to control the position of a variety of controls, such as text boxes, list boxes, and edit controls.

WINDOW SUBCLASSING

Window subclassing is a technique in Windows programming that allows you to intercept and modify the behavior of an existing window procedure.

This can be useful for a variety of purposes, such as adding new functionality to a window or changing the way it handles certain messages.

To subclass a window, you first need to obtain the address of the original window procedure. This can be done using the GetWindowLong function with the GWL_WNDPROC parameter.

Once you have the address of the original window procedure, you can call the SetWindowLong function to replace it with your own subclassing procedure.

Your subclassing procedure should call the original window procedure to handle messages that it does not care about. For messages that it does care about, it can modify the behavior of the window as needed.

Using Window Subclassing to Jump Between Scroll Bars

In the case of COLORS1, we can use window subclassing to intercept the WM_KEYDOWN message and check if the Tab key was pressed. If the Tab key was pressed, we can then set the input focus to the next scroll bar in the array of scroll bar handles.

Here is the code for the subclassing procedure:

```
350  LRESULT CALLBACK SubclassProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
351  {
352      switch (msg)
353      {
354          case WM_KEYDOWN:
355          {
356              if (wParam == VK_TAB)
357              {
358                  int idFocus = GetWindowLong(hwnd, GWL_ID);
359                  int nextFocus = (idFocus + 1) % MAX_SCROLLBARS;
360
361                  SetFocus(hwndScroll[nextFocus]);
362                  return 0;
363              }
364
365              break;
366          }
367
368          default:
369              return CallWindowProc((WNDPROC)GetWindowLong(hwnd, GWL_WNDPROC), hwnd, msg, wParam, lParam);
370      }
371
372      return 0;
373  }
```

To install the subclassing procedure, we need to call the SetWindowLong function with the GWL_WNDPROC parameter and the address of the subclassing procedure.

We should also save the address of the original window procedure so that we can call it from our subclassing procedure.

Here is the code for installing the subclassing procedure:

```
WNDPROC oldWndProc = (WNDPROC)SetWindowLong(hwndScroll[i], GWL_WNDPROC, (LONG)SubclassProc);
```

We should also remove the subclassing procedure when we are done with it. This can be done by calling the SetWindowLong function with the GWL_WNDPROC parameter and the address of the original window procedure.

Here is the code for removing the subclassing procedure:

```
SetWindowLong(hwndScroll[i], GWL_WNDPROC, (LONG)oldWndProc);
```

Window subclassing is a powerful technique that can be used to modify the behavior of existing windows.

In the case of COLORS1, we used window subclassing to add a facility to jump from one scroll bar to another using the Tab key. This is just one example of how window subclassing can be used to add new functionality to Windows applications.

Here is a summary of the key points from the section you provided:

- Scroll bars can only process keystrokes if they have the input focus.
- The Tab key can be used to cycle between scroll bars by using window subclassing to intercept the WM_KEYDOWN message and check if the Tab key was pressed.
- The original window procedure should be called from the subclassing procedure to handle messages that the subclassing procedure does not care about.
- Window subclassing is a technique that allows you to intercept and modify the behavior of an existing window procedure.
- This can be done by calling the GetWindowLong function to get the address of the original window procedure and then calling the SetWindowLong function to replace it with your own subclassing procedure.
- Your subclassing procedure should call the original window procedure to handle messages that it does not care about.
- For messages that it does care about, it can modify the behavior of the window as needed.

SETTING THE BACKGROUND BRUSH

When COLORS1 defines its window class, it sets the background color of the client area to black.

This is done by creating a solid black brush and assigning it to the hbrBackground member of the WNDCLASSEX structure.

The CreateSolidBrush function is used to create a new brush, and the 0 parameter specifies that the brush should be black.

Updating the Background Color

When the settings of COLORS1's scroll bars are changed, the program needs to update the background color of the client area.

This is done by creating a new brush of the desired color and assigning it to the hbrBackground member of the WNDCLASSEX structure.

The CreateSolidBrush function is used to create a new brush, and the RGB macro is used to create a color value from the RGB values of the three scroll bars. The SetClassLong function is used to set the value of the hbrBackground member.

Deleting the Old Brush

After the new brush has been set, the old brush should be deleted. This is done by calling the DeleteObject function with the handle of the old brush.

Invalidating the Client Area

After the new brush has been set and the old brush has been deleted, the client area of the window needs to be invalidated.

This is done by calling the InvalidateRect function. The InvalidateRect function tells Windows that the client area of the window needs to be repainted.

The first parameter to the InvalidateRect function is the handle of the window, the second parameter is a pointer to a RECT structure that specifies the area of the client area that needs to be repainted, and the third parameter is a Boolean value that specifies whether the background should be erased before repainting.

In this case, the TRUE value is passed to the third parameter to specify that the background should be erased.

Processing the WM_PAINT Message

The WM_PAINT message is sent to a window when it needs to be repainted. The WndProc function for COLORS1 does not process the WM_PAINT message, but instead passes it to the DefWindowProc function.

The DefWindowProc function will simply call the BeginPaint and EndPaint functions to validate the window.

Processing the WM_ERASEBKGND Message

The WM_ERASEBKGND message is sent to a window when its background needs to be erased.

The WndProc function for COLORS1 does not process the WM_ERASEBKGND message, but instead ignores it.

Windows will process the WM_ERASEBKGND message by erasing the background of the client area using the brush specified in the window class.

Cleaning Up

Before terminating, the WM_DESTROY message is sent to the window. The WndProc function for COLORS1 processes the WM_DESTROY message by deleting the old brush. This is done by calling the DeleteObject function with the handle of the old brush.

Conclusion

COLORS1 colors its background by creating a new brush of the desired color and assigning it to the hbrBackground member of the WNDCLASSEX structure.

The old brush is then deleted.

The client area of the window is invalidated, and Windows repaints the client area using the new brush.

COLORING THE SCROLL BARS

COLORS1 colors the scroll bars by creating three brushes, one for each primary color (red, green, and blue).

These brushes are created during the WM_CREATE message processing.

The CreateSolidBrush function is used to create the brushes, and the crPrim array is used to specify the RGB values of the brushes.

When the WndProc function receives a WM_CTLCOLORSCROLLBAR message, it returns one of the three brushes based on the ID of the scroll bar.

The ID of the scroll bar is obtained using the GetWindowLong function with the GWL_ID parameter.

The brushes are destroyed during the WM_DESTROY message processing to prevent memory leaks.

Coloring the Static Text

COLORS1 colors the static text by setting the text color using the SetTextColor function and the background color using the SetBkColor function.

The text color is set to the color of the corresponding scroll bar, and the background color is set to the system color COLOR_BTNHIGHLIGHT.

To prevent the background color of the static text from changing if the system color COLOR_BTNHIGHLIGHT is changed, COLORS1 creates a brush of the COLOR_BTNHIGHLIGHT color during the WM_CREATE message processing and uses this brush when handling the WM_CTLCOLORSTATIC message. The brush is destroyed during the WM_DESTROY message processing to prevent memory leaks.

Handling WM_SYSCOLORCHANGE Message

COLORS1 also processes the WM_SYSCOLORCHANGE message to recreate the hBrushStatic brush with the new value of the COLOR_BTNHIGHLIGHT color.

This ensures that the background color of the static text is always up-to-date.

Here is the code for creating and destroying the brushes:

```
// Create the brushes
for (i = 0; i < 3; i++) {
  hBrush[i] = CreateSolidBrush(crPrim[i]);
}

// Destroy the brushes
for (i = 0; i < 3; i++) {
  DeleteObject(hBrush[i]);
}
```

Here is the code for handling the WM_CTLCOLORSCROLLBAR message:

```
case WM_CTLCOLORSCROLLBAR:
  i = GetWindowLong((HWND)lParam, GWL_ID);
  return (LRESULT)hBrush[i];
```

Here is the code for handling the WM_CTLCOLORSTATIC message:

```
case WM_CTLCOLORSTATIC:
  SetTextColor(hdc, GetTextColor(hdc));
  SetBkColor(hdc, COLOR_BTNHIGHLIGHT);
  return (LRESULT)hBrushStatic;
```

Here is the code for handling the WM_SYSCOLORCHANGE message:

```
case WM_SYSCOLORCHANGE:
  DeleteObject(hBrushStatic);
  hBrushStatic = CreateSolidBrush(COLOR_BTNHIGHLIGHT);
  return 0;
```

POPAD1 PROGRAM INSIDE THE CHAPTER 9 FOLDER...

POPPAD1 Code

POPPAD1 is a simple multiline editor that demonstrates the use of the edit control window class. The program is less than 100 lines of C code and does not perform any file I/O. However, it allows the user to type text, move the cursor, select portions of text, delete selected text, copy text, and insert text from the clipboard.

POPPAD1.mp4

The POPPAD1 code is divided into two main parts: the WinMain function and the WndProc function.

The WinMain function is responsible for initializing the window class and creating the main window of the program. The WndProc function is responsible for processing messages sent to the main window.

WinMain Function

The WinMain function first registers the window class. The window class specifies the style of the window, the window procedure, the instance handle, the icon, the cursor, the background brush, and the class name.

The WinMain function then creates the main window using the CreateWindow function. The CreateWindow function specifies the window class name, the window title, the window style, the window position, the window size, the parent window, the menu, the instance handle, and the parameter data.

The WinMain function then shows the window using the ShowWindow function and updates the window using the UpdateWindow function.

WndProc Function

The WndProc function processes messages sent to the main window. The WndProc function handles the following messages:

WM_CREATE: This message is sent when the window is created. The WndProc function creates the edit control window using the CreateWindow function. The CreateWindow function specifies the window class name, the window title, the window style, the window position, the window size, the parent window, the menu, the instance handle, and the parameter data.

WM_SETFOCUS: This message is sent when the window receives the input focus. The WndProc function sets the input focus to the edit control window using the SetFocus function.

WM_SIZE: This message is sent when the window is resized. The WndProc function resizes the edit control window using the MoveWindow function.

WM_COMMAND: This message is sent when a command is sent to the window. The WndProc function handles the EN_ERRSPACE and EN_MAXTEXT notifications from the edit control window. These notifications are sent when the edit control is out of space.

WM_DESTROY: This message is sent when the window is destroyed. The WndProc function posts a quit message to the message queue using the PostQuitMessage function.

Edit Control Styles

The edit control window class has a number of styles that can be used to control its appearance and behavior. These styles are specified in the CreateWindow function call that creates the edit control.

Text Justification

The edit control supports three types of text justification: left-justified, right-justified, and centered. The ES_LEFT, ES_RIGHT, and ES_CENTER styles are used to specify the desired justification.

Multi-line Editing

The edit control can be used to enter either single-line or multi-line text. The ES_MULTILINE style is used to specify that the edit control should support multi-line editing.

Horizontal Scrolling

For single-line edit controls, the ES_AUTOHSCROLL style can be used to enable automatic horizontal scrolling. This means that the text will be automatically scrolled to the left or right as the user types, so that the entire line of text is always visible.

Vertical Scrolling

For multi-line edit controls, the ES_AUTOVSCROLL style can be used to enable automatic vertical scrolling. This means that the text will be automatically scrolled up or down as the user types, so that the entire text is always visible.

Scroll Bars

For multi-line edit controls, the WS_HSCROLL and WS_VSCROLL styles can be used to add horizontal and vertical scroll bars, respectively. This allows the user to scroll the text even if automatic scrolling is not enabled.

Border

The edit control can have a border drawn around it. The WS_BORDER style is used to add a border to the edit control.

Selection Highlighting

When text is selected in an edit control, Windows normally displays the selected text in reverse video. However, when the edit control loses the input focus, the selected text is no longer highlighted. The ES_NOHIDESEL style can be used to prevent the selection from being hidden when the edit control loses the input focus.

POPPAD1 Edit Control

The POPPAD1 program creates an edit control with the following styles:

Style	Description
WS_CHILD	Specifies that the window is a child window.
WS_VISIBLE	Specifies that the window is initially visible.
WS_HSCROLL	Specifies a horizontal scroll bar.
WS_VSCROLL	Specifies a vertical scroll bar.
WS_BORDER	Specifies a border window style.
ES_LEFT	Specifies a left-aligned edit control.
ES_MULTILINE	Specifies a multiline edit control.
ES_AUTOHSCROLL	Automatically scrolls horizontally when needed.
ES_AUTOVSCROLL	Automatically scrolls vertically when needed.

This means that the edit control is a child window, is visible, has horizontal and vertical scroll bars, has a border, is left-justified, supports multi-line editing, has automatic horizontal and vertical scrolling, and does not hide the selection when the edit control loses the input focus.

The size of the edit control is set to the size of the main window when the WndProc function receives a WM_SIZE message. This is done by calling the MoveWindow function:

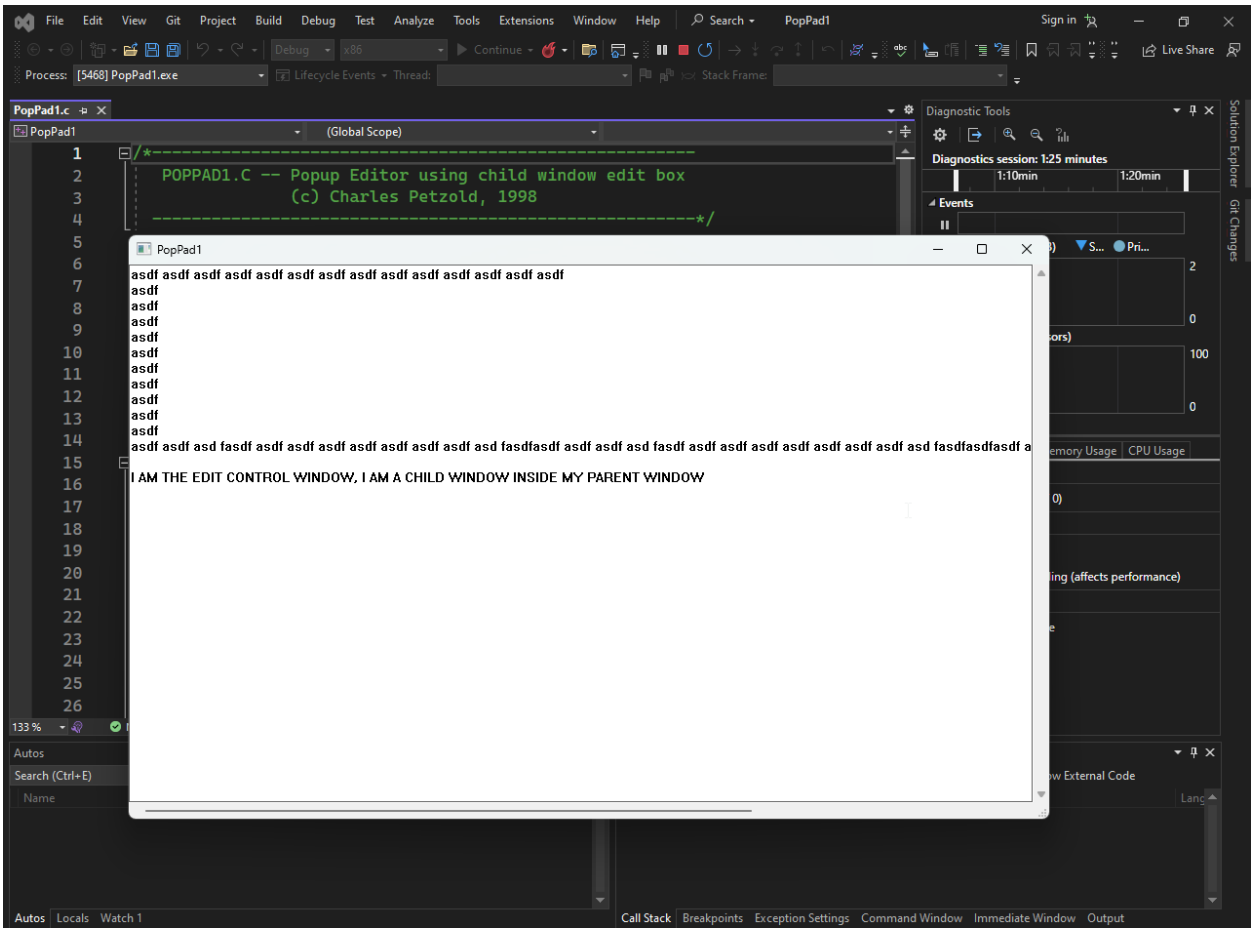
```
MoveWindow(hwndEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
```

This code sets the position of the edit control to (0, 0) and the size of the edit control to the width and height of the main window.

The TRUE parameter tells Windows to repaint the edit control after it has been resized.

The edit control window class is a powerful tool for creating text editing controls in Windows applications.

The various styles available for the edit control allow you to control its appearance and behavior in a wide variety of ways.



Yes, the entire window that you see in the POPPAD1 program is the edit control window. This is because the program creates a single edit control window as its child window and then resizes the edit control window to fill the entire client area of the main window. As a result, the edit control window appears to be the same as the main window.

Edit Control Notifications

Edit controls send WM_COMMAND messages to their parent window to notify the parent window of various events, such as changes to the edit control's contents or scroll bars. The wParam and lParam parameters of the WM_COMMAND message contain information about the notification.

Notification Code

The notification code is specified in the high-order word (HIWORD) of the wParam parameter. The following table lists the notification codes and their meanings:

Notification Code	Meaning
EN_SETFOCUS	The edit control has gained the input focus.
EN_KILLFOCUS	The edit control has lost the input focus.
EN_CHANGE	The edit control's contents will change.
EN_UPDATE	The edit control's contents have changed.
EN_ERRSPACE	The edit control has run out of space.
EN_MAXTEXT	The edit control has run out of space on insertion.
EN_HSCROLL	The edit control's horizontal scroll bar has been clicked.
EN_VSCROLL	The edit control's vertical scroll bar has been clicked.

lParam Parameter

The lParam parameter of the WM_COMMAND message contains the handle of the edit control that sent the notification.

POPPAD1 Notification Handling

The POPPAD1 program traps only the EN_ERRSPACE and EN_MAXTEXT notification codes and displays a message box in response. This means that the program will only notify the user when the edit control is out of space.

Here is the code for handling the EN_ERRSPACE and EN_MAXTEXT notification codes in POPPAD1:

```
case WM_COMMAND:
  if (LOWORD(wParam) == ID_EDIT) {
    if (HIWORD(wParam) == EN_ERRSPACE ||
      HIWORD(wParam) == EN_MAXTEXT) {
      MessageBox(hwnd, TEXT("Edit control out of space."),
            szAppName, MB_OK | MB_ICONSTOP);
      return 0;
    }
  }
  return 0;
```

This code checks the low-order word (LOWORD) of the wParam parameter to make sure that the notification is coming from the edit control.

Then, it checks the high-order word (HIWORD) of the wParam parameter to see if it is the EN_ERRSPACE or EN_MAXTEXT notification code. If it is, the code displays a message box to notify the user.

Edit control notifications are a powerful way to keep track of events in an edit control and to respond to those events accordingly. The POPPAD1 program demonstrates how to use edit control notifications to handle out-of-space errors.

Explained:

case WM_COMMAND: This line indicates the start of the case block for handling the WM_COMMAND message.

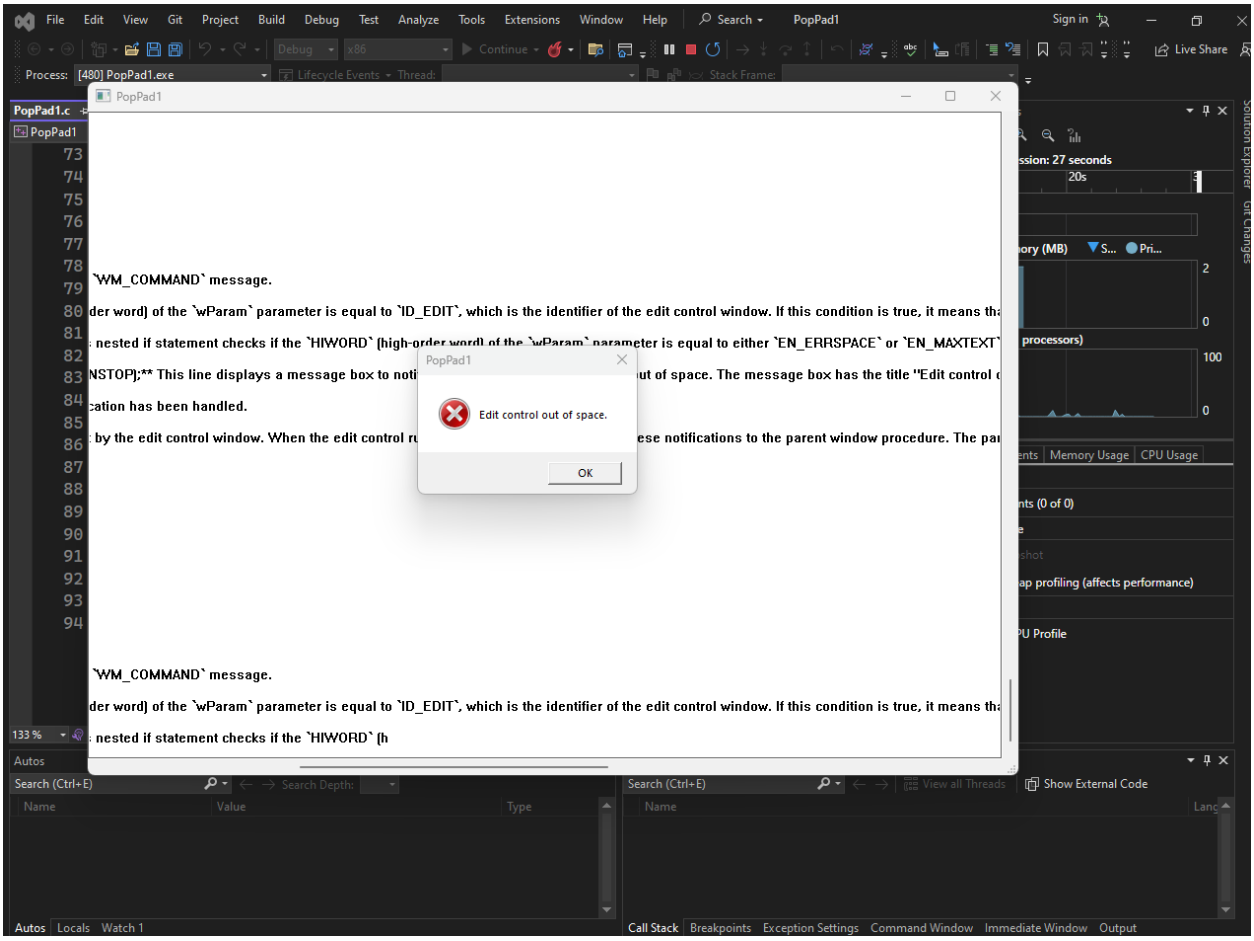
if (LOWORD(wParam) == ID_EDIT): This if statement checks if the LOWORD (low-order word) of the wParam parameter is equal to ID_EDIT, which is the identifier of the edit control window. If this condition is true, it means that the notification is coming from the edit control window.

if (HIWORD(wParam) == EN_ERRSPACE || HIWORD(wParam) == EN_MAXTEXT): This nested if statement checks if the HIWORD (high-order word) of the wParam parameter is equal to either EN_ERRSPACE or EN_MAXTEXT. These are notification codes that indicate that the edit control has run out of space. If either of these conditions is true, it means that the edit control is out of space.

MessageBox(hwnd, TEXT("Edit control out of space."), szAppName, MB_OK | MB_ICONSTOP); This line displays a message box to notify the user that the edit control is

out of space. The message box has the title "Edit control out of space.", the text "Edit control out of space.", and the buttons OK and Stop.

return 0; This line returns 0 to the parent window procedure, indicating that the notification has been handled.



In summary, this code handles the EN_ERRSPACE and EN_MAXTEXT notifications sent by the edit control window.

When the edit control runs out of space, it sends one of these notifications to the parent window procedure. The parent window procedure checks the notification code and, if it is EN_ERRSPACE or EN_MAXTEXT, displays a message box to notify the user.

USING EDIT CONTROLS

Edit controls are versatile tools that allow users to enter and edit text.

Tab and Shift-Tab Key Handling

If you use multiple single-line edit controls on a window, you can use window subclassing to move the input focus from one control to another when the user presses the Tab or Shift-Tab keys. This is similar to how the COLORS1 program handles tabbing between color names.

Here's an example of how to subclass an edit control to handle tabbing:

```
LRESULT CALLBACK EditSubclassedProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_KEYDOWN:
        if (wParam == VK_TAB)
        {
            HWND hNextEdit = GetNextDlgItem(hwnd, wParam);
            if (hNextEdit)
            {
                SetFocus(hNextEdit);
                return 0;
            }
        }
        break;
    }

    return DefSubclassProc(hwnd, msg, wParam, lParam);
}
```

This code defines a subclass procedure EditSubclassedProc that handles the WM_KEYDOWN message. If the user presses the Tab key, the code gets the next edit control in the window and sets the input focus to that control.

Handling the Enter Key

You can handle the Enter key in different ways. You can use it to move the input focus to the next edit control, or you can use it as a signal to your program that all the edit fields are ready.

Inserting Text

To insert text into an edit control, you can use the SetWindowText function. This function sets the text of the entire edit control.

```
SetWindowText(hwndEdit, TEXT("This is some text."));
```

This code sets the text of the edit control with the handle hwndEdit to "This is some text.".

Getting Text

To get text from an edit control, you can use the GetWindowTextLength and GetWindowText functions. The GetWindowTextLength function gets the length of the text in the edit control, and the GetWindowText function gets the text itself.

```
int length = GetWindowTextLength(hwndEdit);
if (length > 0)
{
    TCHAR buffer[length + 1];
    GetWindowText(hwndEdit, buffer, length + 1);
    MessageBox(hwnd, buffer, TEXT("Text from edit control"), MB_OK);
}
```

This code gets the length of the text in the edit control with the handle hwndEdit. If the length is greater than zero, it allocates a buffer of the appropriate size and gets the text into the buffer. Finally, it displays the text in a message box.

Messages to Edit Controls

There are many messages that you can send to an edit control using the SendMessage function. Here are some of the most common messages:

WM_CUT: This message removes the current selection from the edit control and sends it to the clipboard.

WM_COPY: This message copies the current selection to the clipboard but leaves it intact in the edit control.

WM_CLEAR: This message deletes the current selection from the edit control without passing it to the clipboard.

WM_PASTE: This message inserts the text from the clipboard at the cursor position in the edit control.

```
406    SendMessage(hwndEdit, WM_CUT, 0, 0);
407
408    SendMessage(hwndEdit, WM_COPY, 0, 0);
409
410    SendMessage(hwndEdit, WM_CLEAR, 0, 0);
411
412    SendMessage(hwndEdit, WM_PASTE, 0, 0);
```

EM_GETSEL: This message gets the starting and ending positions of the current selection in the edit control.

```
int start, end;
SendMessage(hwndEdit, EM_GETSEL, (WPARAM)&start, (LPARAM)&end);
```

EM_SETSEL: This message sets the selection in the edit control to the specified starting and ending positions.

```
SendMessage(hwndEdit, EM_SETSEL, 0, 10);
```

EM_REPLACESEL: This message replaces the current selection with the specified text.

```
SendMessage(hwndEdit, EM_REPLACESEL, 0, (LPARAM)"This is some text.");
```

The **EM_REPLACESEL message** is a Windows message that replaces the current selection in an edit control with the specified text. The EM_REPLACESEL message has two parameters:

wParam: Specifies whether the replacement operation can be undone. If wParam is TRUE, the operation can be undone. If wParam is FALSE, the operation cannot be undone.

lParam: A pointer to a null-terminated string containing the replacement text. The EM_REPLACESEL message does not return a value.

Undoing the Replacement Operation: If the wParam parameter of the EM_REPLACESEL message is set to TRUE, the replacement operation can be undone by sending the EM_UNDO message to the edit control.

```
SendMessage(hwndEdit, EM_UNDO, 0, 0);
```

Using EM_REPLACESEL to Insert Text: The EM_REPLACESEL message can also be used to insert text into an edit control. To do this, you can set the start parameter to the position where you want to insert the text and the end parameter to the position after the inserted text.

Here is an example of how to insert the text "This is some text." at the cursor position in the edit control with the handle hwndEdit:

```
int cursorPos = SendMessage(hwndEdit, EM_GETSEL, 0, 0);
SendMessage(hwndEdit, EM_REPLACESEL, cursorPos, cursorPos, (LPARAM)"This is some text.");
```

Multiline Edit Controls

Multiline edit controls provide additional functionality for editing multi-line text. Here are some of the messages that you can send to a multiline edit control:

- EM_GETLINECOUNT: This message gets the number of lines in the edit control.
- EM_LINEINDEX: This message gets the offset from the beginning of the edit buffer text for the specified line.
- EM_LINELENGTH: This message gets the length of the specified line in the edit control.
- EM_GETLINE: This message copies the specified line from the edit control into a buffer.

WHAT IS A LISTBOX?

A listbox is a control that displays a list of items. The user can select one or more items from the list. Listboxes are commonly used in graphical user interfaces (GUIs) to allow users to choose from a set of options.

Types of Listboxes

There are two types of listboxes: single-selection and multiple-selection.

- Single-selection listboxes: In a single-selection listbox, the user can only select one item at a time.
- Multiple-selection listboxes: In a multiple-selection listbox, the user can select multiple items at a time.

How to Use a Listbox

To use a listbox, you first need to create it and add items to it.

You can do this by sending messages to the listbox window procedure.

Once you have added items to the listbox, you can set its selection mode (single-selection or multiple-selection) and handle the WM_COMMAND messages that the listbox sends to its parent window when an item is selected.

Features of Listboxes

Listboxes have a number of features that make them useful for displaying and selecting items:

- Scrolling: Users can scroll through the listbox to see all of the items.
- Highlighting: The selected item is highlighted by displaying it in reverse video.
- Keyboard navigation: Users can use the arrow keys, Page Up, Page Down, and letter keys to navigate through the listbox.
- Mouse selection: Users can select items by clicking or double-clicking them.

Creating List Boxes

List boxes are created using the CreateWindow function with the "listbox" window class and the WS_CHILD window style.

This default list box style, however, does not send WM_COMMAND messages to its parent window, meaning that the parent window would need to constantly poll the list box to determine the selected item.

To address this, list boxes typically include the LBS_NOTIFY style, which enables the parent window to receive WM_COMMAND messages when an item is selected.

Single-Selection vs. Multiple-Selection

List boxes can be either single-selection or multiple-selection. Single-selection list boxes allow users to select only one item at a time, while multiple-selection list boxes allow users to select multiple items. The LBS_MULTIPLESEL style is used to create multiple-selection list boxes.

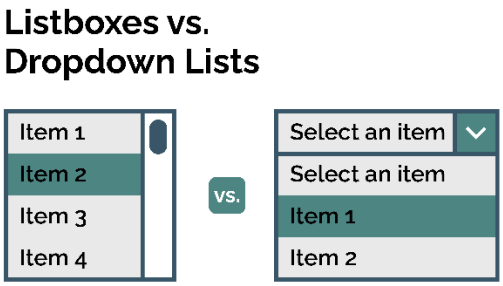
Preventing List Box Updates

By default, list boxes automatically update themselves when a new item is added. To prevent this automatic update, the LBS_NOREDRAW style can be used.

However, this style is generally not recommended, as it can lead to visual inconsistencies. Instead, the WM_SETREDRAW message can be used to temporarily prevent the repainting of a list box.

Borders and Scroll Bars

By default, list boxes do not have borders or scroll bars. To add a border, the WS_BORDER style can be used. To add a vertical scroll bar for scrolling through the list with the mouse, the WS_VSCROLL style can be used.



Standard List Box Style

The Windows header files define a list box style called LBS_STANDARD that includes the most commonly used styles. It is defined as:

Style	Description
LBS_NOTIFY	Enables the list box to send notifications to the parent window.
LBS_SORT	Sorts strings in the list box alphabetically.
WS_VSCROLL	Adds a vertical scroll bar to the list box.
WS_BORDER	Creates a border window style.

The combinations above indicates that the list box should notify its parent window of certain events (LBS_NOTIFY), display items in sorted order (LBS_SORT), and include vertical scrolling functionality (WS_VSCROLL). Additionally, it specifies that the list box should have a border (WS_BORDER).

Resizing and Moving List Boxes

The WS_SIZEBOX and WS_CAPTION styles can be used to allow users to resize and move list boxes within their parent window's client area. However, these styles are typically not used for list boxes, as they can make the user interface less consistent.

Calculating List Box Width and Height

The width of a list box should accommodate the width of the longest string plus the width of the scroll bar. The width of the vertical scroll bar can be obtained using:

```
GetSystemMetrics(SM_CXVSCROLL);
```

The height of the list box can be calculated by multiplying the height of a character by the number of items you want to appear in view.

Adding Strings to a List Box

After creating a list box, you can add strings to it using the SendMessage function and the LB_ADDSTRING message.

This message takes two parameters: the handle of the list box window and a pointer to a null-terminated string.

For instance, to add the string "This is a string" to the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_ADDSTRING, 0, (LPARAM)"This is a string");
```

If you want to add strings to the list box in a specific order, you can use the LB_INSERTSTRING message.

This message takes three parameters: the handle of the list box window, the index of the item to insert the string after, and a pointer to a null-terminated string.

For example, to insert the string "This is another string" after the second item in the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_INSERTSTRING, 2, (LPARAM)"This is another string");
```

Deleting Strings from a List Box

To delete a string from a list box, you can use the LB_DELETESTRING message.

This message takes two parameters: the handle of the list box window and the index of the item to delete.

For example, to delete the third item from the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_DELETESTRING, 3, 0);
```

Clearing the List Box

To clear the entire list box, you can use the LB_RESETCONTENT message. This message takes no parameters.

For example, to clear the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_RESETCONTENT, 0, 0);
```

Temporarily Inhibiting Redrawing

If you have a large number of strings to add or delete to a list box, you may want to temporarily inhibit redrawing to improve performance.

To do this, you can send the WM_SETREDRAW message to the list box window with a wParam value of FALSE.

For example, to disable redrawing for the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, WM_SETREDRAW, FALSE, 0);
```

Once you have finished adding or deleting strings, you can re-enable redrawing by sending the WM_SETREDRAW message with a wParam value of TRUE.

For example, to re-enable redrawing for the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, WM_SETREDRAW, TRUE, 0);
```

Handling Errors

The SendMessage function can return an error code if there is a problem adding or deleting a string from the list box.

For example, if the list box is full, the SendMessage function will return LB_ERRSPACE. You can check the return value of the SendMessage function to ensure that the operation was successful.

Adding, deleting, and clearing strings are common tasks when working with list boxes. By understanding the different messages and techniques involved, you can effectively manage the contents of your list boxes.

Getting the Number of Items in a List Box

To get the number of items in a list box, you can use the SendMessage function and the LB_GETCOUNT message.

This message takes no parameters and returns the number of items in the list box. For example, to get the number of items in the list box with the handle hwndList, you would use the following code:

```
int iCount = SendMessage(hwndList, LB_GETCOUNT, 0, 0);
```

Setting the Default Selection

To set the default selection in a single-selection list box, you can use the SendMessage function and the LB_SETCURSEL message.

This message takes two parameters: the handle of the list box window and the index of the item to set as the default selection.

An index of -1 deselects all items. For example, to set the second item as the default selection in the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_SETCURSEL, 1, 0);
```

Selecting an Item Based on Initial Characters

To select an item in a single-selection list box based on its initial characters, you can use the SendMessage function and the LB_SELECTSTRING message.

This message takes three parameters: the handle of the list box window, the index of the item to start the search from, and a pointer to a null-terminated string containing the initial characters to match.

For example, to select the first item that starts with the letter "A" in the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_SELECTSTRING, -1, (LPARAM)"A");
```

Getting the Index of the Current Selection

To get the index of the current selection in a list box, you can use the SendMessage function and the LB_GETCURSEL message.

This message takes no parameters and returns the index of the selected item. If no item is selected, the message returns LB_ERR.

For example, to get the index of the current selection in the list box with the handle hwndList, you would use the following code:

```
int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
```

Getting the Length of a List Box Item

To get the length of a list box item, you can use the SendMessage function and the LB_GETTEXTLEN message.

This message takes two parameters: the handle of the list box window and the index of the item.

For example, to get the length of the third item in the list box with the handle hwndList, you would use the following code:

```
int iLength = SendMessage(hwndList, LB_GETTEXTLEN, 2, 0);
```

Copying a List Box Item to the Text Buffer

To copy a list box item to the text buffer, you can use the SendMessage function and the LB_GETTEXT message.

This message takes three parameters: the handle of the list box window, the index of the item, and a pointer to a buffer to hold the copied text.

The buffer must be large enough to hold the length of the item plus a terminating NULL character.

For example, to copy the fifth item in the list box with the handle hwndList to a buffer named szBuffer, you would use the following code:

```
int iLength = SendMessage(hwndList, LB_GETTEXT, 4, (LPARAM)szBuffer);
```

Setting the Selection State of a Multiple-Selection List Box Item

To set the selection state of an item in a multiple-selection list box, you can use the SendMessage function and the LB_SETSEL message.

This message takes three parameters: the handle of the list box window, a wParam parameter that specifies whether to select or deselect the item, and the index of the item.

For example, to select the third item in the list box with the handle hwndList, you would use the following code:

```
SendMessage(hwndList, LB_SETSEL, TRUE, 2);
```

Getting the Selection State of a Multiple-Selection List Box Item

To get the selection state of an item in a multiple-selection list box, you can use the SendMessage function and the LB_GETSEL message.

This message takes two parameters: the handle of the list box window and the index of the item. The message returns a non-zero value if the item is selected and 0 if it is not selected.

For example, to get the selection state of the second item in the list box with the handle hwndList, you would use the following code:

```
int iSelect = SendMessage(hwndList, LB_GETSEL, 1, 0);
```

Selecting and extracting entries from list boxes are fundamental tasks when working with these controls. By understanding the different messages and techniques involved, you can effectively manage the selection and retrieval of items from your list boxes.

Receiving Messages from List Boxes

List boxes send WM_COMMAND messages to their parent windows to inform them of user interactions. These messages contain information about the selected item and the type of interaction that occurred.

Understanding the WM_COMMAND Message

The WM_COMMAND message has two parameters: wParam and lParam. The wParam parameter contains two parts:

- LOWORD(wParam): The child window ID
- HIWORD(wParam): The notification code

The lParam parameter contains the child window handle.

Notification Codes

List boxes send several notification codes to their parent windows. The following table lists the notification codes and their meanings:

Notification Code	Meaning
`LBN_ERRSPACE`	The list box has run out of space.
`LBN_SELCHANGE`	The current selection has changed.
`LBN_DBLCLK`	A list box item has been double-clicked with the mouse.
`LBN_SELCANCEL`	The current selection has been canceled.
`LBN_SETFOCUS`	The list box has received the input focus.
`LBN_KILLFOCUS`	The list box has lost the input focus.

Handling Selection Changes

The LBN_SELCHANGE notification code is sent whenever the current selection in the list box changes.

This includes when the user moves the highlight through the list box, toggles the selection state with the Spacebar, or clicks an item with the mouse.

To handle selection changes, you can add a case statement to your WM_COMMAND message handler that checks for the LBN_SELCHANGE notification code.

For example, the following code snippet shows how to get the index of the selected item when the LBN_SELCHANGE notification code is received:

```
switch (HIWORD(wParam)) {
  case LBN_SELCHANGE:
    int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
    // Handle selection change
    break;
}
```

Handling Double-Clicks

The LBN_DBLCLK notification code is sent when a list box item is double-clicked with the mouse.

To handle double-clicks, you can add another case statement to your WM_COMMAND message handler that checks for the LBN_DBLCLK notification code.

For example, the following code snippet shows how to get the index of the double-clicked item when the LBN_DBLCLK notification code is received:

```
switch (HIWORD(wParam)) {
  case LBN_SELCHANGE:
    int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
    // Handle selection change
    break;
  case LBN_DBLCLK:
    int iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
    // Handle double-click
    break;
}
```

Receiving and handling messages from list boxes is essential for creating interactive applications. By understanding the different notification codes and how to handle them, you can effectively respond to user interactions and provide a rich user experience.

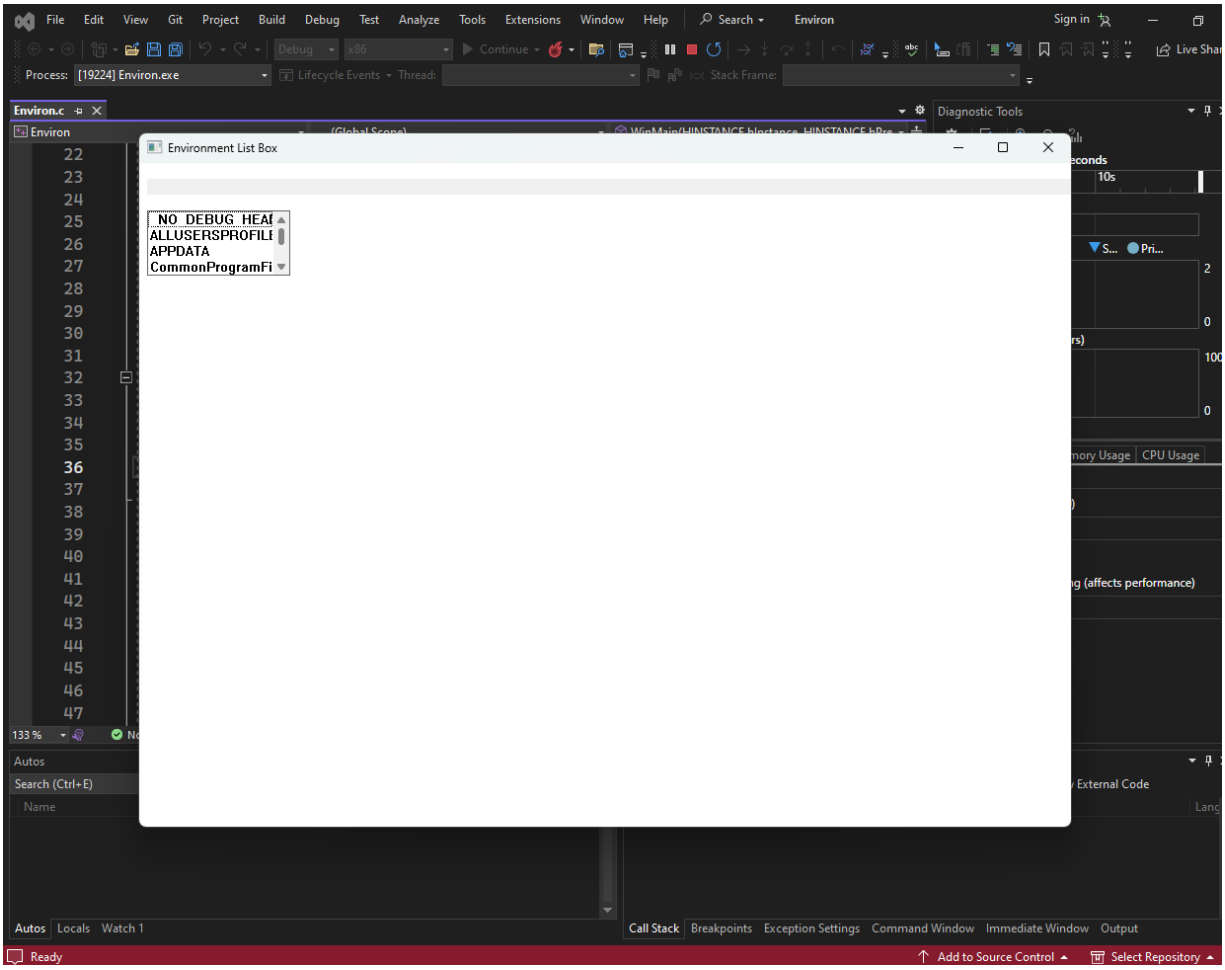
The ENVIRON Program

The ENVIRON program is a simple application that displays a list of environment variables and their corresponding values. It uses a list box to display the environment variable names and a static text window to display the values. The program is written in C and uses the Windows API to create and manage the windows and handle user interactions.

The Main Function

- The main function of the ENVIRON program is WinMain. This function is responsible for initializing the application, creating the main window, and entering the main message loop. The first step in WinMain is to register the window class for the main window. This is done by calling the RegisterClass function. The window class defines the style of the window, the window procedure, and other attributes.
- Next, WinMain creates the main window by calling the CreateWindow function. The CreateWindow function takes a number of parameters, including the name of the window class, the window title, the window style, the window position, the window size, the parent window handle, the menu handle, the instance handle, and a parameter that can be used to pass data to the window procedure.

- After the main window is created, WinMain shows the window by calling the ShowWindow function. This function displays the window on the screen and sets the initial focus to the window. Finally, WinMain enters the main message loop by calling the GetMessage function. The main message loop retrieves messages from the queue and dispatches them to the appropriate window procedure.



The Window Procedure

- The window procedure for the ENVIRON program is WndProc. This function is responsible for handling all of the messages that are sent to the main window. The WndProc function switches on the message value to determine what action to take.
- The WM_CREATE message is sent when the window is created. In response to this message, WndProc creates the two child windows: the list box and the static text window. The list box is created with the style LBS_STANDARD, which means that it will display a single selection at a time. The static text window is created with the style SS_LEFT, which means that the text will be left-justified.
- The WM_SETFOCUS message is sent when the window receives the input focus. In response to this message, WndProc sets the input focus to the list box. This means that the user can use the keyboard to navigate through the list of environment variables.
- The WM_COMMAND message is sent when the user interacts with a control in the window. In response to this message, WndProc checks to see if the control that sent the message is the list box and if the notification code is LBN_SELCHANGE. This notification code is sent when the user changes the selection in the list box.
- If the selection has changed, WndProc obtains the index of the selected item using the LB_GETCURSEL message. Then, it obtains the text of the selected item using the LB_GETTEXT message. The text of the selected item is the name of the environment variable.
- Next, WndProc uses the GetEnvironmentVariable function to obtain the value of the environment variable. The value of the environment variable is the string that is displayed in the static text window. Finally, WndProc uses the SetWindowText function to set the text of the static text window to the value of the environment variable.
- The WM_DESTROY message is sent when the window is destroyed. In response to this message, WndProc posts a WM_QUIT message to the message queue. This message causes the main message loop to terminate and the program to exit.
- The ENVIRON program is a simple but useful application that demonstrates how to use list boxes and static text windows in a Windows application. The program also demonstrates how to use the GetEnvironmentVariable and SetWindowText functions.

LISTING FILES WITH LB_DIR

The LB_DIR message is a powerful list box message that allows you to fill a list box with a file directory list. This message can be used to list files, directories, and drives.

Understanding the iAttr Parameter

The iAttr parameter is a file attribute code that specifies which files and directories to include in the list. The least significant byte of iAttr is a file attribute code that can be a combination of the following values:

Value	Attribute
0x0000	Normal file
0x0001	Read-only file
0x0002	Hidden file
0x0004	System file
0x0010	Subdirectory
0x0020	File with archive bit set

The next highest byte of iAttr provides some additional control over the items desired:

Value	Option
0x4000	Include drive letters
0x8000	Exclusive search only

The DDL prefix stands for "dialog directory list."

Using File Attribute Codes

Here are some examples of how to use file attribute codes with the LB_DIR message:

To list all normal files, read-only files, and files with the archive bit set, use the following code:

```
SendMessage(hwndList, LB_DIR, DDL_READWRITE, (LPARAM) szFileSpec);
```

To list all subdirectories, use the following code:

```
SendMessage(hwndList, LB_DIR, DDL_DIRECTORY, (LPARAM) szFileSpec);
```

To list all valid drives and their subdirectories, use the following code:

```
SendMessage(hwndList, LB_DIR, DDL_DRIVES | DDL_DIRECTORY, (LPARAM) szFileSpec);
```

To list only files that have been modified since the last backup, use the following code:

```
SendMessage(hwndList, LB_DIR, DDL_EXCLUSIVE | DDL_ARCHIVE, (LPARAM) szFileSpec);
```

The LB_DIR message is a versatile tool for listing files and directories in a list box. By understanding the iAttr parameter and how to use file attribute codes, you can effectively control the contents of your list box.

Ordering File Lists

The LBS_SORT message is used to order file lists in a list box. When this message is sent to a list box with a file list, the list box will first list files satisfying the file specification and then (optionally) list subdirectory names.

Understanding the File Specification

The file specification is a string that specifies which files to list. The file specification can contain wildcards, such as *.*, to match multiple files. The file specification does not affect the subdirectories that the list box includes.

The "Double-Dot" Subdirectory Entry

The "double-dot" subdirectory entry, [..], is a special entry that lets the user back up one level toward the root directory. This entry will not appear if you are listing files in the root directory.

Subdirectory Names

Subdirectory names are listed in the form [SUBDIR], where SUBDIR is the name of the subdirectory.

Valid Disk Drives

Valid disk drives are listed in the form [−A−], where A is the drive letter.

The HEAD Program

The HEAD program is a simple tool that displays the beginning of a file. It uses a list box to display a directory list and a text box to display the file contents.

Ordering the File List

The HEAD program uses the LBS_SORT message to order the file list in the list box. This ensures that the files are listed in alphabetical order.

Handling Double-Clicks

The HEAD program handles double-clicks on the list box by displaying the contents of the selected file in the text box.

Handling Enter Key Presses

The HEAD program handles Enter key presses when the filename is selected by displaying the contents of the selected file in the text box.

Changing the Subdirectory

The HEAD program allows the user to change the subdirectory by double-clicking on a subdirectory name in the list box or by pressing the Enter key when a subdirectory name is selected.

Displaying File Contents

The HEAD program displays up to 8 KB of the beginning of the file in the text box. This is done by opening the file and reading the first 8 KB of data.

Conclusion

The HEAD program is a simple but useful tool that demonstrates how to order file lists and display file contents.

THE HEAD PROGRAM

The HEAD program is a simple tool that displays the beginning of a file. It uses a list box to display a directory list and a text box to display the file contents.

Main Function (WinMain)

The WinMain function is the entry point of the program. It initializes the application, creates the main window, and enters the main message loop.

The RegisterClass function registers the window class for the main window. This defines the style of the window, the window procedure, and other attributes.

The CreateWindow function creates the main window. This function takes a number of parameters, including the name of the window class, the window title, the window style, the window position, the window size, the parent window handle, the menu handle, the instance handle, and a parameter that can be used to pass data to the window procedure.

The ShowWindow function displays the window on the screen and sets the initial focus to the window.

The GetMessage function retrieves messages from the queue and dispatches them to the appropriate window procedure.

Window Procedure (WndProc)

The WndProc function is the window procedure for the main window. This function is responsible for handling all of the messages that are sent to the main window.

The WM_CREATE message is sent when the window is created. In response to this message, WndProc creates the two child windows: the list box and the static text window.

The SendMessage function sends a message to the list box to fill it with a directory list.

The WM_SIZE message is sent when the window is resized. In response to this message, WndProc updates the position of the child windows.

The WM_SETFOCUS message is sent when the window receives the input focus. In response to this message, WndProc sets the input focus to the list box.

The WM_COMMAND message is sent when the user interacts with a control in the window. In response to this message, WndProc checks to see if the control that sent the message is the list box and if the notification code is LBN_DBLCLK. This notification code is sent when the user double-clicks on an item in the list box.

If the user double-clicks on an item in the list box, WndProc attempts to open the file and display the beginning of the file in the static text window.

The WM_PAINT message is sent when the window needs to be repainted. In response to this message, WndProc retrieves the contents of the file and displays them in the static text window.

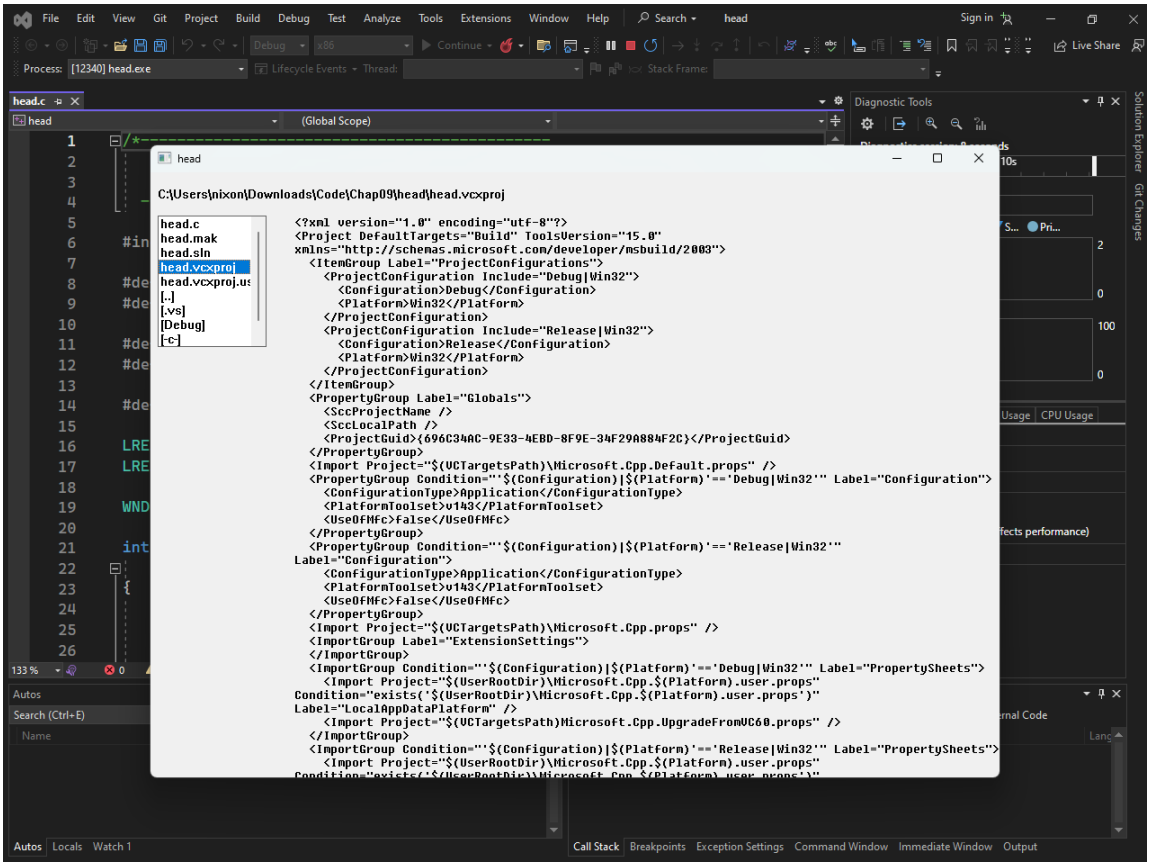
The WM_DESTROY message is sent when the window is destroyed. In response to this message, WndProc posts a WM_QUIT message to the message queue. This message causes the main message loop to terminate and the program to exit.

List Box Procedure (ListProc)

The ListProc function is the window procedure for the list box. This function is responsible for handling messages that are sent to the list box.

The WM_KEYDOWN message is sent when the user presses a key while the list box has the input focus. In response to this message, ListProc checks to see if the key that was pressed is the Enter key.

If the user presses the Enter key, ListProc sends a WM_COMMAND message to the parent window. This message tells the parent window that the user has double-clicked on the selected item in the list box.



Handling Double-Clicks

The ENVIRON program allows users to select an environment variable and display the corresponding value.

This is done by simply clicking on the desired environment variable in the list box. However, this approach would not be suitable for the HEAD program, as it would require continuously opening and closing files as the user moves the selection through the list box. This would make the program very slow and unresponsive.

To address this issue, the HEAD program requires users to double-click on the desired file or subdirectory in the list box.

This presents a challenge, as list box controls do not have an automatic keyboard interface that corresponds to a mouse double-click. To provide a keyboard alternative, the HEAD program uses window subclassing.

Window Subclassing

Window subclassing is a technique that allows you to intercept and modify the behavior of a window by creating a subclass of the original window class.

In the HEAD program, the list box is subclassed by creating a subclass procedure named ListProc.

This procedure intercepts the WM_KEYDOWN message and checks if the pressed key is the Enter key (VK_RETURN).

If it is, ListProc sends a WM_COMMAND message to the parent window with an LBN_DBLCLK notification code, simulating a double-click.

Processing WM_COMMAND Message

The WndProc procedure handles the WM_COMMAND message and uses the CreateFile function to check if the selected item is a file.

If CreateFile returns an error, the item is not a file and is likely a subdirectory. In this case, SetCurrentDirectory is used to change the current directory to the selected subdirectory.

Handling Drive Letter Selection: If SetCurrentDirectory fails after removing the preliminary dash and adding a colon, it means the user has selected an invalid drive letter. In this case, the program simply ignores the selection and does not update the list box.

Processing WM_PAINT Message

The WndProc procedure also handles the WM_PAINT message, which is responsible for repainting the window.

When this message is received, the program opens the selected file using the CreateFile function.

This function returns a handle to the file, which can be passed to the ReadFile and CloseHandle functions.

Unicode Considerations

The HEAD program assumes that all text files contain ASCII text and uses the DrawTextA function to display the file contents.

However, this is not always the case. Some text files may contain Unicode text, and using DrawTextA on Unicode text will result in garbled characters.

To properly handle Unicode text files, the program should determine the encoding of the file before displaying the contents.

This can be done by checking the byte order mark (BOM) at the beginning of the file. If the BOM is present, it indicates that the file is Unicode and the DrawTextW function should be used.

Otherwise, the file is assumed to be ASCII and the DrawTextA function can be used.

Simplified Approach

The HEAD program takes a simpler approach and always uses the DrawTextA function, regardless of the file encoding. This may result in garbled characters for Unicode files, but it is a simpler and more lightweight solution.

Conclusion

The HEAD program demonstrates how to use window subclassing to handle double-clicks in a list box and how to open and display files. It also highlights the issue of Unicode text and how to properly handle it.

End of Chapter 9...