KEYBOARD ACCELERATORS

Keyboard accelerators are key combinations that allow users to quickly access frequently used commands or actions in a software application.

These shortcuts are typically represented as a combination of two or more keys, often including a modifier key like Ctrl, Alt, or Shift, and a non-modifier key like A, B, C, or D.

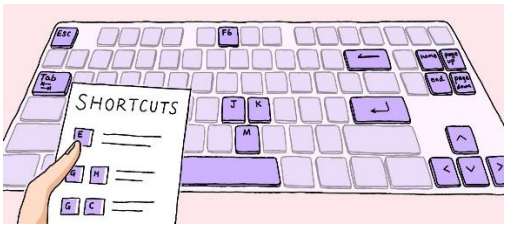
Purpose of Keyboard Accelerators

Keyboard accelerators offer several advantages over traditional menu-based navigation:

Increased Efficiency: Keyboard accelerators allow users to perform actions without switching between the keyboard and mouse, significantly improving efficiency and workflow speed.

Reduced Eye Strain: By keeping users focused on the keyboard, keyboard accelerators minimize the need for constant eye movement between the keyboard and the screen, reducing eye strain.

Accessibility Enhancements: Keyboard accelerators provide an alternative input method for users with limited hand or mouse control, enhancing accessibility and usability.



Common Keyboard Accelerator Usage

Keyboard accelerators are widely used in various software applications, including:

- Word Processors: Copy (Ctrl+C), Paste (Ctrl+V), Undo (Ctrl+Z), Redo (Ctrl+Y)
- Web Browsers: Open New Tab (Ctrl+T), Close Tab (Ctrl+W), Switch Tabs (Ctrl+Tab/Ctrl+Shift+Tab), Save Page (Ctrl+S)
- Operating Systems: Cut (Ctrl+X), Copy (Ctrl+C), Paste (Ctrl+V), Undo (Ctrl+Z), Redo (Ctrl+Y), Save (Ctrl+S), Print (Ctrl+P)

Implementing Keyboard Accelerators

Software developers can implement keyboard accelerators using various methods, including:

- Windows API: The Windows API provides functions like TranslateAccelerator and CreateAcceleratorTable to manage keyboard accelerators in Windows applications.
- Cross-Platform Toolkits: Cross-platform toolkits like Qt and GTK+ offer native support for keyboard accelerators, allowing consistent implementation across different platforms.
- Application Frameworks: Application frameworks like .NET Framework and Electron provide built-in functionality for defining and handling keyboard accelerators.

Benefits of Keyboard Accelerators

Keyboard accelerators offer numerous benefits to both users and developers:

- User Benefits: Increased efficiency, reduced eye strain, improved accessibility, enhanced productivity
- Developer Benefits: Simplified code, reduced menu clutter, improved user experience

Encouraging Keyboard Accelerator Use

To encourage users to adopt keyboard accelerators, developers can implement strategies like:

- Prominent Display: Display keyboard shortcuts alongside menu items or provide a dedicated cheat sheet.
- Training and Documentation: Include clear instructions and tutorials on using keyboard accelerators in the application's documentation or help system.
- Customizability: Allow users to customize keyboard shortcuts to suit their preferences and accessibility needs.

Guidelines for Assigning Keyboard Accelerators

Keyboard accelerators, also known as hotkeys, are key combinations that allow users to quickly access frequently used commands or actions in a software application.

When assigning keyboard accelerators, it's crucial to consider consistency, accessibility, and potential conflicts with system functions. Here are some general guidelines to follow:

Consistency with Common Applications: Strive for consistency with keyboard accelerators used in popular applications. This helps users maintain familiarity and avoid confusion when switching between programs.

Avoid Conflicts with Windows Functions: Refrain from using keys like Tab, Enter, Esc, and Spacebar for keyboard accelerators, as these are often reserved for system functions.

Use Modifier Keys Effectively: Utilize modifier keys like Ctrl, Shift, and Alt to create unique and memorable keyboard shortcuts without overloading individual keys.

Consider Old and New Accelerators: When applicable, support both the old and new keyboard accelerators for a specific function, as users may be accustomed to either convention.

Reserve F1 for Help: Dedicate the F1 key to invoke help or context-sensitive assistance.

Avoid F4, F5, and F6: Refrain from using the F4, F5, and F6 keys for keyboard accelerators, as these are often reserved for special functions in Multiple Document Interface (MDI) applications.

Examples of Recommended Keyboard Accelerators

Here's a table of common keyboard accelerators and their associated functions:

Function	Recommended Accelerators
Undo	Ctrl+Z
Redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	Ctrl+A
Find	Ctrl+F
Replace	Ctrl+H
Save	Ctrl+S
Print	Ctrl+P
Help	F1

THE ACCELERATOR TABLE

An accelerator table is a data structure that defines keyboard shortcuts for menu items and other actions in a Windows application. Each entry in the table specifies an ID, a keystroke combination, and the corresponding menu item or action.

Defining Accelerators in Developer Studio

- You can define accelerator tables using the Accel Properties dialog box in Developer Studio. To create an accelerator table:
- Select the menu item or action for which you want to define a shortcut.
- Right-click and select "Properties" from the context menu.
- In the Properties dialog box, click the "Accel" button.
- In the Accel Properties dialog box, enter a keystroke combination in the "Keystroke" field. You can use virtual key codes, ASCII characters, or a combination of both in conjunction with the Shift, Ctrl, or Alt keys.
- Click "OK" to save the accelerator.
- Loading the Accelerator Table in Your Program

To load an accelerator table into your program, you use the LoadAccelerators function. This function takes two parameters:

- hInstance: The handle to the program's instance.
- lpAcceleratorName: The name of the accelerator table resource. The resource name can be a string or a number.

Here's an example of how to load an accelerator table named MyAccelerators:

```
HANDLE hAccel = LoadAccelerators(hInstance, TEXT("MyAccelerators"));
```

Once the accelerator table is loaded, you can use it to translate keystrokes into menu IDs or actions. The TranslateAccelerator function takes three parameters:

- hWindow: The handle to the window that receives the keystroke.
- hMsg: The handle to the message that contains the keystroke.
- wParam: The wParam value of the message.

The TranslateAccelerator function returns a menu ID if the keystroke matches an accelerator in the table. If the keystroke does not match an accelerator, it returns 0.

Here's an example of how to use the TranslateAccelerator function:

```
int menuID = TranslateAccelerator(hWindow, hMsg, wParam);
```

If menuID is not 0, it is the ID of the menu item that corresponds to the keystroke. You can then use this ID to perform the corresponding action.

Tips for Defining Accelerators

When defining accelerators, keep the following tips in mind:

- Use consistent keystrokes for similar actions. For example, you might use Ctrl+Z for undo and Ctrl+X for cut.
- Avoid using keystrokes that are already used by Windows. For example, you should not use Ctrl+C for copy, as this is already used by Windows.
- Use descriptive keystrokes. For example, you might use Ctrl+F for find and Ctrl+H for replace.

Loading the Accelerator Table

The LoadAccelerators function is used to load an accelerator table into memory and obtain a handle to it. The syntax of the LoadAccelerators function is as follows:

```
HANDLE LoadAccelerators(
  HINSTANCE hInstance,
  LPCTSTR lpAcceleratorName
);
```

The hInstance parameter is the handle to the program's instance. The lpAcceleratorName parameter is the name of the accelerator table resource. The resource name can be a string or a number.

Here's an example of how to load an accelerator table named MyAccelerators:

```
HANDLE hAccel = LoadAccelerators(hInstance, TEXT("MyAccelerators"));
```

Translating Keystrokes

The TranslateAccelerator function is used to translate a keystroke message into a menu ID or action. The syntax of the TranslateAccelerator function is as follows:

```
int TranslateAccelerator(
  HWND hWnd,
  HACCEL hAccel,
  LPMSG lpMsg
);
```

The hWnd parameter is the handle to the window that receives the keystroke. The hAccel parameter is the handle to the accelerator table. The lpMsg parameter is a pointer to the message structure that contains the keystroke.

The TranslateAccelerator function returns a menu ID if the keystroke matches an accelerator in the table. If the keystroke does not match an accelerator, it returns 0.

Here's an example of how to use the TranslateAccelerator function:

```
int menuID = TranslateAccelerator(hWindow, hAccel, &msg);
```

If menuID is not 0, it is the ID of the menu item that corresponds to the keystroke. You can then use this ID to perform the corresponding action.

Integrating Keyboard Accelerators into the Message Loop

To integrate keyboard accelerators into the message loop, you can modify the standard message loop as follows:

```
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(hwnd, hAccel, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

This code will first check whether the keystroke can be translated using the accelerator table. If it can, the TranslateAccelerator function will send the corresponding message to the window procedure. Otherwise, the code will continue with the normal message loop processing.

Understanding the hwnd Parameter

The hwnd parameter is used to specify the window that should receive the keyboard accelerator messages. If you omit the hwnd parameter, the messages will be sent to the window that currently has the input focus.

Handling Modal Dialog Boxes and Message Boxes

The TranslateAccelerator function does not translate keyboard messages when a modal dialog box or message box has the input focus. This is because messages for these windows do not come through the program's message loop.

If you want keyboard accelerators to be translated for modal dialog boxes or message boxes, you will need to use a different technique.

One technique is to use the SetWindowsHookEx function to install a hook that intercepts keyboard messages before they are sent to the dialog box or message box.

Types of Accelerator Messages

Message Type	Description
WM_SYSCOMMAND	Sent when a keyboard accelerator corresponds to a menu item in the system menu.
WM_COMMAND	Sent when a keyboard accelerator corresponds to a menu item outside the system menu or when a menu item is selected.
WM_INITMENU	Sent before a menu is displayed, allowing for menu customization.
WM_INITMENUPOPUP	Sent before a popup menu is displayed, allowing for dynamic menu configuration.
WM_MENUSELECT	Sent when a menu item is highlighted, providing the option to cancel or modify the selection.

The TranslateAccelerator function translates keystrokes into corresponding messages, either WM_SYSCOMMAND or WM_COMMAND, depending on whether the accelerator corresponds to a menu item in the system menu or not.

WM_SYSCOMMAND Messages for System Menu Items

When a keyboard accelerator corresponds to a menu item in the system menu, the TranslateAccelerator function sends the window procedure a WM_SYSCOMMAND message. This message indicates that a system menu command has been invoked using the keyboard.

WM_COMMAND Messages for Non-System Menu Items

For keyboard accelerators that correspond to menu items outside the system menu, the TranslateAccelerator function sends the window procedure a WM_COMMAND message. This message indicates that a non-system menu command has been invoked using the keyboard.

WM_COMMAND Message Parameters

The WM_COMMAND message contains information about the invoked command, including:

- LOWORD(wParam): The accelerator ID or menu ID of the command.
- HIWORD(wParam): A notification code specific to the command.
- lParam: The handle of the child window control associated with the command, if applicable.

Additional Messages for Menu Items

When a keyboard accelerator corresponds to a menu item, the window procedure also receives the following messages, just as if the menu option had been chosen:

- WM_INITMENU: Sent before the menu is displayed, allowing for menu customization.
- WM_INITMENUPOPUP: Sent before a popup menu is displayed, allowing for dynamic menu configuration.
- WM_MENUSELECT: Sent when a menu item is highlighted, providing the option to cancel or modify the selection.

Handling Disabled Menu Items

If the keyboard accelerator corresponds to a disabled or grayed menu item, the TranslateAccelerator function does not send the window procedure a WM_COMMAND or WM_SYSCOMMAND message. This prevents users from activating unavailable menu options using keyboard shortcuts.

Accelerator Behavior for Minimized Windows

When the active window is minimized, the TranslateAccelerator function sends the window procedure WM_SYSCOMMAND messages for keyboard accelerators that correspond to enabled system menu items. This allows users to access essential system commands even when the window is minimized.

Handling Non-System Menu Accelerators for Minimized Windows

For keyboard accelerators that do not correspond to any menu items, the TranslateAccelerator function sends the window procedure WM_COMMAND messages even when the window is minimized. This ensures that users can still access other commands using keyboard shortcuts.

Accelerator Type	Description
System Menu Accelerators	Keyboard shortcuts that correspond to menu items in the system menu, typically accessed using the Alt key and a function key.
Non-System Menu Accelerators	Keyboard shortcuts that correspond to menu items outside the system menu, often used for frequently executed actions or to navigate menus quickly.
Control Accelerators	Keyboard shortcuts associated with child window controls within a program's window, allowing users to interact with specific elements directly.
Global Accelerators	Keyboard shortcuts that can be invoked regardless of which application has the input focus, typically used for system-wide actions or context-sensitive functions.

Additional Points:

- Accelerator IDs are unique identifiers assigned to each keyboard shortcut, allowing the window procedure to distinguish between different accelerators.
- Notification codes provide additional information about the type of command or action triggered by the accelerator.
- Child window handles identify the specific control associated with a control accelerator.
- Global accelerators are registered using the RegisterHotKey function and require elevated privileges in some cases.

Accelerator Table Element	Description
Accelerator ID	A unique identifier assigned to each keyboard shortcut, allowing the window procedure to distinguish between different accelerators.
Keystroke Combination	The specific key combination associated with the accelerator, typically represented as a combination of virtual key codes or ASCII characters with modifier keys (Ctrl, Shift, Alt).
Menu ID	The identifier of the menu item corresponding to the accelerator, used for menu-related accelerators.
Action ID	The identifier of the action triggered by the accelerator, used for non-menu accelerators.
Notification Code	Additional information about the type of command or action triggered by the accelerator, such as whether the accelerator is for a menu item or a control.
Child Window Handle	The handle of the child window control associated with a control accelerator, if applicable.
Flags	Optional flags that modify the behavior of the accelerator, such as whether it should be disabled or global.

In summary, the TranslateAccelerator function plays a crucial role in translating keystrokes into corresponding messages, enabling users to efficiently interact with applications using keyboard shortcuts.

Popad2 program in chapter 10 folder….

POPPAD2: A Rudimentary Notepad with Menus and Accelerators

This document details the POPPAD2 program, a rudimentary notepad application that builds upon the previous POPPAD1 program. POPPAD2 introduces several new functionalities, including:

Menus: The program incorporates File and Edit menus, providing access to various options like New, Open, Save, Undo, Cut, Copy, Paste, Clear, and Select All.

Accelerators: Keyboard shortcuts are associated with specific menu items, enabling faster access to frequently used functions.

Edit Control Functionality: POPPAD2 utilizes a child window edit control to handle text editing, offering features like undo, cut, copy, paste, clear, and select all.

Functionality Breakdown

Menus:

File Menu:

Currently, these options are non-functional and will generate a beep sound upon selection.

Future chapters will implement functionalities for New, Open, Save, Save As, and Print.

Edit Menu:

Undo: Sends a WM_UNDO message to the child window edit control, enabling undo functionality.

Cut: Sends a WM_CUT message to the edit control, copying the selected text to the clipboard and removing it from the document.

Copy: Sends a WM_COPY message to the edit control, copying the selected text to the clipboard without removing it from the document.

Paste: Sends a WM_PASTE message to the edit control, inserting the text from the clipboard into the document at the current cursor position.

Clear: Sends a WM_CLEAR message to the edit control, deleting all text from the document.

Select All: Sends a EM_SETSEL message to the edit control, selecting all text in the document.

Additional Features:

The program dynamically updates the enabled state of menu items based on the current context. For example, the "Cut" and "Copy" options are only enabled when there is text selected.

Keyboard shortcuts provide quick access to specific menu items, improving user experience and efficiency.

The program confirms with the user before closing, ensuring any unsaved data is not lost accidentally.

POPPAD2.RC:

This file defines the program's resources, including icons, menus, and keyboard shortcuts.

Icons:

The file specifies the use of "poppad2.ico" as the program's icon.

Menus:

The POPPAD2 menu consists of three main categories: File, Edit, and Help.

File Menu:

This menu provides options for creating new files (IDM_FILE_NEW), opening existing files (IDM_FILE_OPEN), saving files (IDM_FILE_SAVE), saving files with a different name (IDM_FILE_SAVE_AS), printing (IDM_FILE_PRINT), and exiting the program (IDM_APP_EXIT).

Edit Menu:

This menu offers options for undoing actions (IDM_EDIT_UNDO), cut (IDM_EDIT_CUT), copy (IDM_EDIT_COPY), paste (IDM_EDIT_PASTE), deleting text (IDM_EDIT_CLEAR), and selecting all text (IDM_EDIT_SELECT_ALL).

Help Menu:

This menu provides access to help information (IDM_HELP_HELP) and an about dialogue for the program (IDM_APP_ABOUT).

Accelerators:

The POPPAD2 resource file defines keyboard shortcuts for various functions:

Undo: Ctrl+Z (IDM_EDIT_UNDO)

Delete: Del (IDM_EDIT_CLEAR)

Cut: Ctrl+X, Shift+Del (IDM_EDIT_CUT)

Help: F1 (IDM_HELP_HELP)

Copy: Ctrl+Insert (IDM_EDIT_COPY)

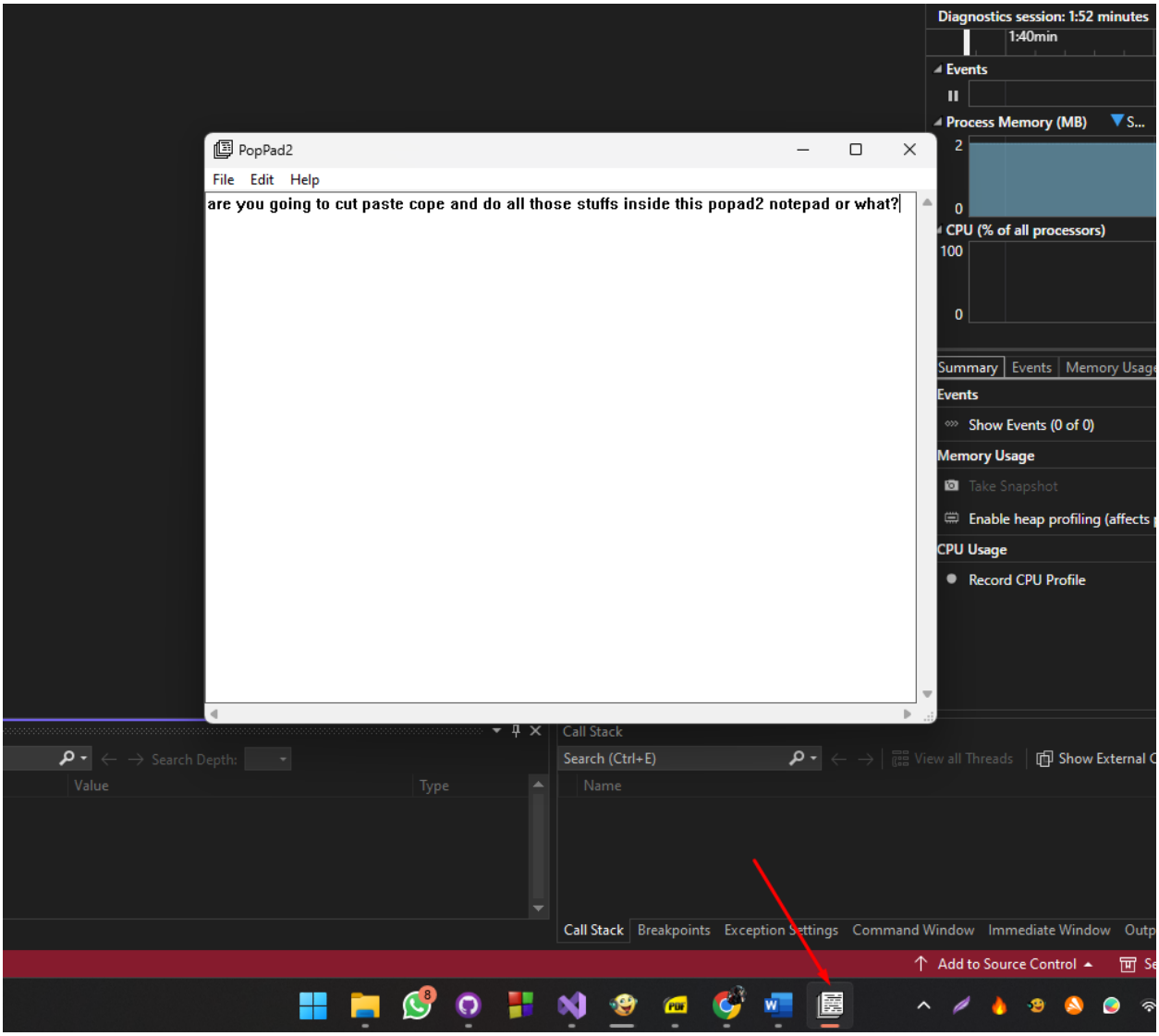
Paste: Shift+Insert, Ctrl+V (IDM_EDIT_PASTE)

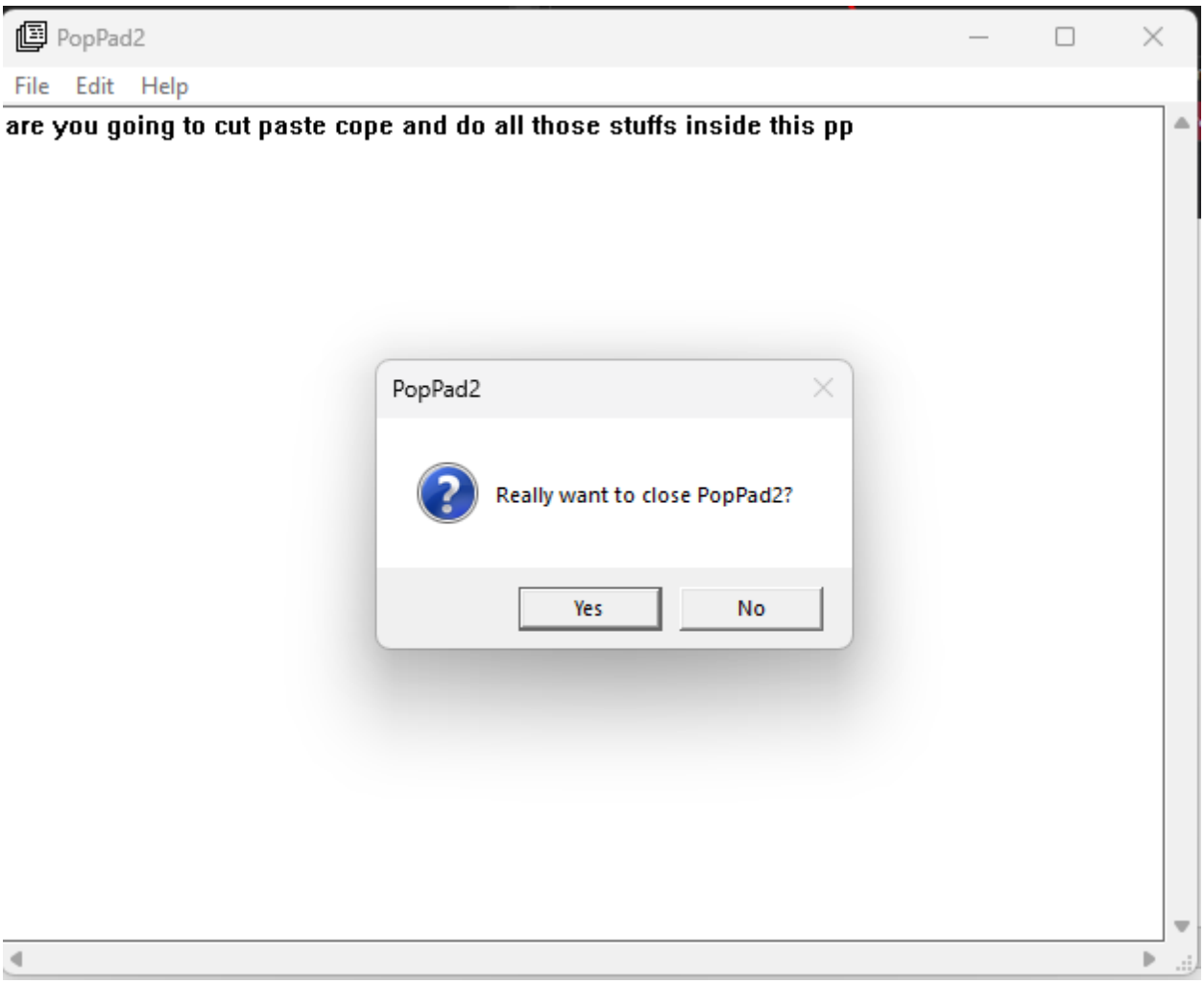
Select All: Ctrl+A (IDM_EDIT_SELECT_ALL)

RESOURCE.H:

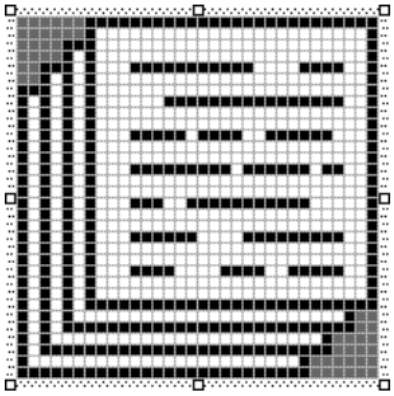
This file provides symbolic names for the various menu options and keyboard shortcuts defined in POPPAD2.RC.

This allows the application code to refer to these resources by name instead of using numerical identifiers, making the code more readable and maintainable.




PopPad2
File Edit Help
are you going to cut paste cope and do all those stuffs inside this pp
PopPad2
Really want to close PopPad2?
Yes
No

POPPAD2.ICO



POPPAD2: Menu and Accelerator Handling in Depth

Menu and Accelerator Resources

The POPPAD2.RC resource script defines the program's user interface elements, including menus and keyboard shortcuts. These are crucial for user interaction and program accessibility.

Menus:

The script defines three main menus: File, Edit, and Help.

Each menu item has a unique identifier (IDM_FILE_NEW, IDM_EDIT_UNDO, etc.) for referencing in the program code.

The Edit menu items have their associated keyboard shortcuts defined within the character strings, separated by a tab (\t) character.

Accelerators:

The resource script defines keyboard shortcuts for most Edit menu options:

- Undo: Ctrl+Z (IDM_EDIT_UNDO)
- Cut: Ctrl+X, Shift+Del (IDM_EDIT_CUT)
- Copy: Ctrl+Insert (IDM_EDIT_COPY)
- Paste: Shift+Insert, Ctrl+V (IDM_EDIT_PASTE)

- Select All: Ctrl+A (IDM_EDIT_SELECT_ALL)

Enabling and Graying Menu Items

WM_INITMENUPOPUP Message: This message is sent to the window procedure when a popup menu is about to be displayed. POPPAD2 uses this message to dynamically enable or disable menu items based on the current context.

Edit Menu Item Control:

Here's how Edit menu items are enabled/disabled:

Undo:

The program sends an EM_CANUNDO message to the edit control.

If the call returns non-zero, signifying undo possibility, the menu item is enabled.

Otherwise, it's grayed out.

Paste:

The IsClipboardFormatAvailable function checks if the clipboard contains text.

If text is present, the menu item is enabled, else it's grayed out.

Cut, Copy, and Delete:

The program sends an EM_GETSEL message to the edit control to retrieve selection information.

If the low and high words of the returned value are the same, no text is selected.

In this case, the menu items are grayed out.

If text is selected, the menu items are enabled.

This dynamic behavior ensures the menu reflects the current state of the edit control and provides a more intuitive user experience.

1. Menu Options Processing:

The code discusses the implementation of various menu options, specifically focusing on the Edit menu. The use of a child window edit control (hwndEdit) simplifies the process, as each menu option corresponds to sending a specific message to this control.

Undo, Cut, Copy, Paste, Delete, Select All:

```
case IDM_UNDO:
    SendMessage(hwndEdit, WM_UNDO, 0, 0);
    return 0;
// Similar cases for Cut, Copy, Paste, Delete, and Select All
```

Each option in the Edit menu (Undo, Cut, Copy, Paste, Delete, Select All) is implemented by sending the corresponding message to the edit control (hwndEdit).

For instance, to perform an undo operation, the code sends the WM_UNDO message to hwndEdit.

These operations are streamlined due to the use of the edit control.

About Option:

The "About" option in the File menu triggers the display of a simple message box using MessageBox.

It shows information about the application, such as its name and copyright.

```
96  case IDM_ABOUT:
97      MessageBox(hwnd, TEXT("POPPAD2 (c) Charles Petzold, 1998"), szAppName, MB_OK | MB_ICONINFORMATION);
98      return 0;
```

Exit Option:

Choosing the "Exit" option sends a WM_CLOSE message to the window procedure, initiating the termination process.

The AskConfirmation function is used to display a message box, prompting the user for confirmation before closing the program.

If the user selects "Yes," the DestroyWindow function is called to close the program.

```
case IDM_EXIT:
    SendMessage(hwnd, WM_CLOSE, 0, 0);
    return 0;
```

Handling WM_CLOSE:

The WM_CLOSE message is processed in the window procedure, and user confirmation is sought through the AskConfirmation function.

If the user confirms by selecting "Yes," the program is terminated using DestroyWindow.

```
case WM_CLOSE:
    if (IDYES == AskConfirmation(hwnd))
        DestroyWindow(hwnd);
    return 0;
```

Handling WM_QUERYENDSESSION:

To provide confirmation before ending a program during a system shutdown, the window procedure processes WM_QUERYENDSESSION.

The AskConfirmation function is again used for user confirmation.

If the user confirms, a value of 1 is returned, indicating approval for the session to end.

```
case WM_QUERYENDSESSION:
    if (IDYES == AskConfirmation(hwnd))
        return 1;
    else
        return 0;
```

WM_ENDSESSION:

This message is mentioned for completeness, indicating that it follows WM_QUERYENDSESSION and informs whether the program was successfully terminated.

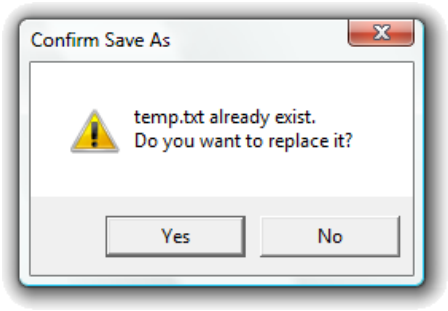
The AskConfirmation function displays a message box asking for confirmation to close POPPAD2.

```
100 {
101     AskConfirmation(HWND hwnd)
102     return MessageBox(hwnd, TEXT("Really want to close Poppad2?"), szAppName, MB_YESNO | MB_ICONQUESTION);
103 }
```

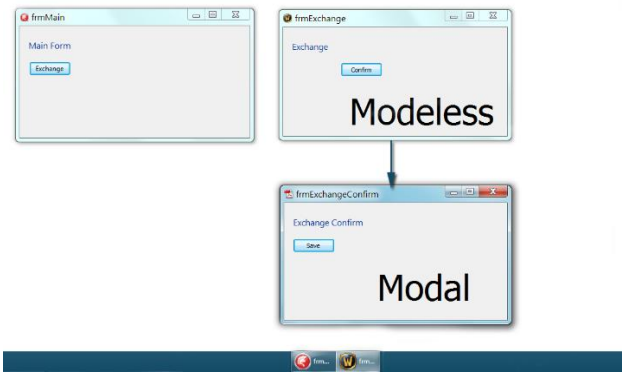
MODAL DIALOG BOXES: IN-DEPTH EXPLANATION

In Windows programming, dialog boxes are windows that appear on top of the main application window and require user interaction before the user can continue using the main application. They are categorized into two types: modal and modeless.

Modal dialog boxes: These are the most common type. When displayed, they capture the user's focus and prevent them from interacting with the main application window or any other window in the program. The user must explicitly close the dialog box, usually by clicking a button like "OK" or "Cancel," before they can continue using the program.



Modeless dialog boxes: These allow the user to interact with both the dialog box and the main application window simultaneously. The user can switch back and forth between them without closing the dialog box.



Creating an "About" Dialog Box:

The ABOUT1 program, shown in Figure 11-1, demonstrates a simple example of creating a modal dialog box. This program displays an "About" dialog box containing the program's name, icon, copyright notice, and an "OK" button.

Understanding the Code:

The code for the ABOUT1 program includes three main components:

WinMain function: This is the main entry point of the program. It performs the following tasks:

- Registers the window class with Windows.
- Creates the main application window.
- Displays and updates the window.
- Processes messages until the user closes the window.

WndProc function: This function takes messages from Windows and processes them accordingly. It handles the following messages:

- WM_CREATE: Initializes the program's instance handle.
- WM_COMMAND: Handles the "About" menu item selection by displaying the "About" dialog box.
- WM_DESTROY: Posts a quit message to terminate the application.
- AboutDlgProc function: This function is the callback function for the "About" dialog box. It handles the following messages:
- WM_INITDIALOG: Initializes the dialog box controls.
- WM_COMMAND: Handles clicking the "OK" or "Cancel" button by closing the dialog box.

Code Breakdown and Explanation:

Registering the window class: The RegisterClass function registers the window class with Windows. The window class defines the style and behavior of the application's windows.

Creating the main application window: The CreateWindow function creates the main application window. The function takes various parameters to specify the window's title, style, position, size, and parent window.

Displaying and updating the window: The ShowWindow and UpdateWindow functions display and update the window on the screen.

Processing messages: The GetMessage and TranslateMessage/DispatchMessage functions retrieve messages from the Windows message queue and process them accordingly.

Handling window messages: The WndProc function handles different window messages, including WM_CREATE, WM_COMMAND, and WM_DESTROY.

Creating the "About" dialog box: The DialogBox function creates and displays the "About" dialog box. It takes the program's instance handle, the dialog box resource identifier, the parent window handle, and the callback function for the dialog box.

Handling dialog box messages: The AboutDlgProc function handles different dialog box messages, including WM_INITDIALOG and WM_COMMAND. It initializes the dialog box controls and closes the dialog box when the user clicks the "OK" or "Cancel" button.

ABOUT1 Resources Explained

The provided excerpts describe the resources used by the ABOUT1 program, focusing on the dialog box and the menu. Here's a breakdown:

Dialog Box:

Style:

- DS_MODALFRAME: This style makes the dialog box modal, requiring user interaction before returning to the main application.
- WS_POPUP: This style removes the window title bar and borders, giving the dialog box a pop-up appearance.
- Font: The dialog box uses the "MS Sans Serif" font with a size of 8 points.

Controls:

- Push Button: This button displays the text "OK" and has the ID IDOK. When clicked, it closes the dialog box.
- Static Text: There are four static text controls:
- ID_STATIC: This control displays the text "About1".
- IDC_STATIC: This control displays the text "About Box Demo Program".
- IDC_STATIC: This control displays the text "(c) Charles Petzold, 1998".
- Icon: The dialog box displays an icon with the ID IDC_STATIC located at coordinates (7, 7).

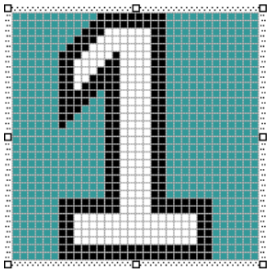
Menu:

- The menu has one popup item named "&Help".
- The "Help" menu has a single menu item named "&About About1...", which has the ID IDM_APP_ABOUT. Clicking this item displays the "About" dialog box.

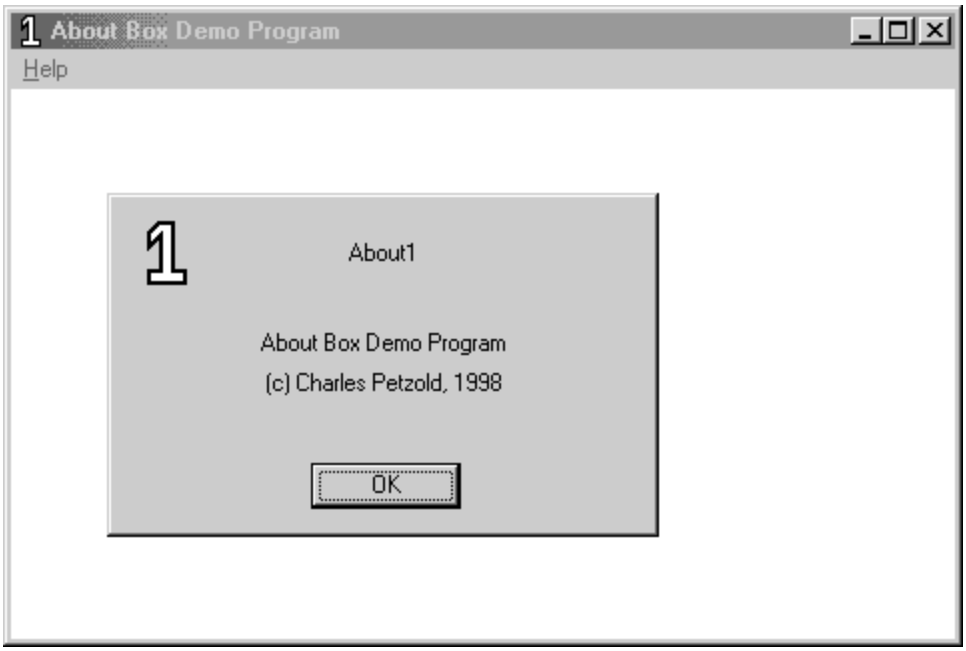
Icon:

- The program uses an icon named "About1.ico" for the window and the dialog box icon control.

ABOUT1.ICO



The source code for about1 program is in the Chapter10 folder including its resource files (I couldn't locate the official program).



Summary:

These resources define the visual appearance and functionality of the "About" dialog box and the application's menu. The dialog box displays information about the program and allows the user to close it with an "OK" button. The menu provides access to the "About" dialog box through a menu item.

Adding and Configuring the "About" Dialog Box in Visual C++

This section dives deeper into the process of adding and customizing the "About" dialog box in Visual C++ Developer Studio for the ABOUT1 program.

Adding the Dialog Box:

Insert Menu: Choose "Resource" from the Insert menu and then select "Dialog Box". This creates a basic dialog box with a title bar and caption ("Dialog") as well as "OK" and "Cancel" buttons.

Control Toolbar: You can utilize the Controls toolbar to add various controls like text fields, buttons, and icons to the dialog box.

Dialog Box ID: By right-clicking the dialog box and selecting "Properties", you can change its ID from the default "IDD_DIALOG1" to "AboutBox". This ID will be used to identify the dialog box in the program code.

Dialog Box Position: Update the "X Pos" and "Y Pos" fields to "32" to position the dialog box 32 pixels from the left and top of the main application window.

Dialog Box Style: Uncheck the "Title Bar" checkbox in the Styles tab of the Properties dialog to remove the title bar from the dialog box.

Customizing the Dialog Box:

Removing the Cancel Button: Click the "Cancel" button and press "Delete" to remove it from the dialog box as we only need the "OK" button.

Positioning the OK Button: Click the "OK" button and move it to the bottom of the dialog box. Center it horizontally using the "Center" button on the toolbar.

Adding the Icon: Click the "Pictures" button on the Controls toolbar and drag a square on the dialog box where the icon should appear.

Icon Properties: Right-click the square and select "Properties". Set the ID to "IDC_STATIC" and change the Type to "Icon". You can then type the name of the program's icon ("About1") or select it from the combo box.

Adding Static Text: Select "Static Text" from the Controls toolbar and position three text boxes in the dialog box. Right-click each text box, select "Properties", and enter the desired text in the "Caption" field. Choose "Center" for the alignment in the Styles tab.

Resizing the Dialog Box: Drag the dialog box outline or use the keyboard arrow keys to adjust its size and position. You can also use the arrow keys with Shift to resize individual controls.

The current coordinates and sizes are displayed in the bottom right corner of the Developer Studio window.

```
//It's resource files: ABOUT1.RC

#include "resource.h"
#include "afxres.h"

// Dialog
ABOUTBOX DIALOG DISCARDABLE 32, 32, 180, 100
STYLE DS_MODALFRAME | WS_POPUP
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON "OK", IDOK, 66, 80, 50, 14
    ICON "ABOUT1", IDC_STATIC, 7, 7, 21, 20
    CTEXT "About1", IDC_STATIC, 40, 12, 100, 8
    CTEXT "About Box Demo Program", IDC_STATIC, 7, 40, 166, 8
    CTEXT "(c) Charles Petzold, 1998", IDC_STATIC, 7, 52, 166, 8
END

// Menu
ABOUT1 MENU DISCARDABLE
BEGIN
    POPUP "&Help"
    BEGIN
        MENUITEM "&About About1...", IDM_APP_ABOUT
    END
END

// Icon
ABOUT1 ICON DISCARDABLE "About1.ico"


//====================================================================================//

// Microsoft Developer Studio generated include file.
// Used by About1.rc
#define IDM_APP_ABOUT 40001
#define IDC_STATIC -1
```

Name: ABOUTBOX is the name assigned to the dialog box.

Keywords: DIALOG indicates the resource type, and DISCARDABLE defines it as discardable, meaning it can be discarded from memory when not in use.

Coordinates: 32, 32 specifies the x and y coordinates of the dialog box's upper left corner relative to the parent window's client area when invoked.

Dimensions: 180, 100 defines the width and height of the dialog box.

These coordinates and dimensions are not measured in pixels but in units based on the dialog box font size. In this case, the font is "8-point MS Sans Serif."

X-coordinates and width: are measured in units of 1/4 of the average character width. So, the dialog box is 5 characters from the left edge and 40 characters wide.

Y-coordinates and height: are measured in units of 1/8 of the character height. This places the dialog box 2-1/2 characters from the top and makes it 10 characters high.

This coordinate system ensures the dialog box maintains its relative size and appearance regardless of display resolution or font selection.

```
STYLE DS_MODALFRAME | WS_POPUP
```

This line defines the dialog box style, similar to the style field of a CreateWindow call.

- WS_POPUP: Removes the title bar and borders, giving a pop-up appearance.
- DS_MODALFRAME: Makes the dialog box modal, requiring user interaction before returning to the main application.

These lines define the child window controls within the dialog box using BEGIN and END statements.

- DEFPUSHBUTTON: Creates a default push button with the text "OK", ID IDOK, located at coordinates (66, 80) with dimensions (50, 14).
- ICON: Adds the program's icon with the name "ABOUT1" and ID IDC_STATIC positioned at (7, 7) with dimensions (21, 20).
- CTEXT: Defines three centered text controls with ID IDC_STATIC:
- The first displays "About1" at (40, 12) with dimensions (100, 8).
- The second displays "About Box Demo Program" at (7, 40) with dimensions (166, 8).
- The third displays "(c) Charles Petzold, 1998" at (7, 52) with dimensions (166, 8).

This explanation dives into the ABOUT1.RC dialog box template, detailing the meaning of each line and the various parameters used to define its style, size, and child controls. This provides a clear understanding of how the dialog box is created and configured using the resource script file.

Deep Dive into Child Window Controls of ABOUT1 Dialog Box

This section delves deeper into the child window controls defined in the ABOUT1 dialog box template:

Control Identifiers:

- DEFPUSHBUTTON, ICON, and CTEXT: These identifiers are specific to dialog boxes and represent shorthand versions of a window class and specific window styles.
- CTEXT: This identifier corresponds to the "static" class and the following combined styles:
- WS_CHILD: Indicates the control is a child of the dialog box.
- SS_CENTER: Centers the text horizontally.
- WS_VISIBLE: Makes the control visible.
- WS_GROUP: This style will be discussed later.
- ICON: This identifier uses the program's icon resource name defined in ABOUT1.RC.
- DEFPUSHBUTTON: This identifier uses the text displayed on the button and a pre-defined ID IDOK (equal to 1).

Control Properties:

- ID: Each control has an ID used for message communication.
- IDOK: Used for the push button.
- IDC_STATIC: Used for both the icon and text controls (predefined as -1 in RESOURCE.H).
- Text: For CTEXT and DEFPUSHBUTTON, this defines the displayed text.
- CTEXT: Displays "About1", "About Box Demo Program", and "(c) Charles Petzold, 1998".
- DEFPUSHBUTTON: Displays "OK".
- Position and Size: These are specified relative to the dialog box's client area, expressed in units of 1/4 average character width and 1/8 character height.
- CTEXT: Position and size for each text control are provided in the code snippet.
- ICON: Only position is specified, width and height are ignored.
- DEFPUSHBUTTON: Position and size are defined in the code snippet.

Additional Styles:

- WS_GROUP: This style is present in the DEFPUSHBUTTON statement. Its purpose will be discussed further in the ABOUT2 program.
- WS_TABSTOP: This related style is also relevant to ABOUT2.

Comparison with Chapter 9:

While similar window styles like WS_CHILD and SS_CENTER were encountered in Chapter 9 for creating static text controls, the context of dialog boxes introduces new aspects like specific identifiers and the WS_GROUP style.

Understanding the ABOUT1 Dialog Box Procedure

This section delves into the code and functionality of the AboutDlgProc function, which handles messages for the ABOUT1 dialog box:

Function Definition:

```
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
```

Parameters:

- hDlg: Handle to the dialog box window.
- message: Message sent to the dialog box.
- wParam: Additional message-specific information.
- lParam: Additional message-specific information.

Return Type:

- BOOL: TRUE if the message is processed, FALSE otherwise.

Differences from a Window Procedure:

- Return Type: Dialog box procedures return BOOL (treated as int), while window procedures return LRESULT.
- Message Processing: Dialog box procedures return TRUE when handling a message and FALSE otherwise, unlike window procedures that require calling DefWindowProc if they don't handle the message.
- Messages Handled: Dialog box procedures don't handle WM_PAINT, WM_DESTROY, or receive WM_CREATE. They specifically handle WM_INITDIALOG for initialization and WM_COMMAND for user interactions.

Message Processing Details:

WM_INITDIALOG:

- This is the first message received by the dialog box procedure.
- If the procedure returns TRUE, Windows sets focus to the first child window with WS_TABSTOP style (the push button in this case).
- Alternatively, the procedure can use SetFocus to set focus to a specific child window and return FALSE.

WM_COMMAND:

- This message is sent from the push button when clicked or when the Spacebar is pressed with focus on the button.
- The low word of wParam contains the ID of the control (IDOK).
- The procedure calls EndDialog to close the dialog box.

Other Messages:

- For messages not handled explicitly, the procedure returns FALSE to let Windows handle them.

Keyboard Accelerators:

- Messages for a modal dialog box don't go through the program's message queue, so keyboard accelerators within the dialog box don't affect the main program.

Overall Functionality:

- The AboutDlgProc function handles the WM_INITDIALOG message for initialization and the WM_COMMAND message for closing the dialog box.
- It returns TRUE for these messages and FALSE for any others, allowing Windows to handle them. This ensures the dialog box functions as intended, displaying the About information and closing when the user clicks the "OK" button.

Invoking the ABOUT1 Dialog Box: A Detailed Analysis

This section examines how the ABOUT1 program displays its "About" dialog box:

1. Obtaining Instance Handle:

During the processing of the WM_CREATE message in WndProc, the program uses the low word of lParam to retrieve the program's instance handle and store it in the static variable hInstance.

2. Checking for Menu Command:

The program monitors for WM_COMMAND messages where the low word of wParam is equal to IDM_APP_ABOUT. This indicates the user has selected "About About1" from the menu.

3. Invoking DialogBox Function:

When a relevant WM_COMMAND message is received, the program calls the DialogBox function with the following arguments:

- hInstance: The program's instance handle (previously stored).
- TEXT("AboutBox"): The name of the dialog box template defined in the resource script.
- hwnd: The handle of the program's main window, which becomes the parent of the dialog box.
- AboutDlgProc: The address of the dialog box procedure function.

4. Dialog Box Display and Interaction:

If the program uses a numeric identifier instead of a name for the dialog box template, the MAKEINTRESOURCE macro can convert it to a string.

Selecting "About About1" from the menu triggers the dialog box display as shown in Figure 11-2.

The user can close the dialog box by:

- Clicking the "OK" button with the mouse.
- Pressing the Spacebar while the button has focus.
- Pressing Enter.

For any dialog box with a default push button (like "OK"), pressing Enter or Spacebar sends a WM_COMMAND message with the ID of the default button (IDOK) to the dialog box.

Alternatively, pressing Escape sends a WM_COMMAND message with an ID of IDCANCEL.

5. Ending the Dialog Box:

The DialogBox function doesn't return control to WndProc until the dialog box is ended.

The EndDialog function in the dialog box procedure closes the dialog and returns a value.

This returned value is the second parameter to the EndDialog call in WndProc.

Finally, WndProc returns control to Windows.

6. Concurrent Message Processing:

Even when the dialog box is displayed, WndProc can still receive messages.

The program can even send messages to WndProc from within the dialog box procedure.

This is possible because the program's main window is the parent of the dialog box popup window.

To send a message from the dialog box procedure, the code would start with:

```
SendMessage(GetParent(hDlg), ..., ...);
```

Exploring Variations and Customization Options for Dialog Boxes

This section delves deeper into various options for creating and customizing dialog boxes in Visual C++:

Understanding Resource Script Syntax:

While the visual editors in Visual C++ simplify dialog box creation, understanding resource script syntax provides a deeper understanding of their structure and limitations.

Manual creation of dialog box templates can be helpful in specific scenarios (e.g., HEXCALC program later in the chapter).

Resource compiler and script syntax are documented in the Windows Programming Guidelines.

Window Styles and Effects:

ABOUT1 uses the most common style for modal dialog boxes: WS_POPUP | DS_MODALFRAME.

Experimenting with other styles can achieve different effects:

- WS_CAPTION: Adds a caption bar with title and draggable functionality.
- Coordinates specified in the DIALOG statement are relative to the parent window's client area.
- Caption text can be set with the CAPTION statement or SetWindowText function.
- WS_SYSMENU: Adds a system menu box for Move and Close options.
- WS_THICKFRAME: Allows resizing the dialog box.
- Maximize box: Can be added for further customization.

Menus for Dialog Boxes:

- An uncommon feature, menus can be added to dialog boxes using the MENU menu-name statement.
- Menu name or ID should be unique from other menu and control IDs.

Customizing Font and Window Procedure:

- The FONT statement allows using a custom font for dialog box text instead of the default system font.
- Shipping a program-specific font can create a unique look for dialog boxes and other text elements.
- Replacing the default Windows dialog box procedure with a custom one requires specifying a window class name in the CLASS "class-name" statement. This advanced technique will be demonstrated in the HEXCALC program later.

Further Exploration:

- The provided information offers a comprehensive overview of customizing dialog boxes in Visual C++.
- Experimenting with various styles, fonts, and even custom window procedures can help create unique and functional dialog boxes for your applications.

DEEP DIVE INTO DIALOG BOX CREATION AND CUSTOMIZATION

This section delves deeper into the mechanics and customization options for creating dialog boxes:

DialogBox Function and Windows Handling:

When DialogBox is called with a dialog box template name, Windows utilizes the CreateWindow function to build the popup window.

Windows extracts necessary information from the dialog box template: coordinates, size, style, caption, menu, window class (if not specified).

Windows registers a special class for dialog boxes and uses the provided address of the dialog box procedure to communicate with your program.

Manually creating the popup window requires bypassing DialogBox and managing the window creation and message handling yourself.

DialogBoxIndirect and Dynamic Dialog Creation:

For situations where using a resource script isn't ideal, DialogBoxIndirect provides an alternative.

This function utilizes data structures to define the dialog box template dynamically during program execution.

Window Classes and Styles for Child Controls:

CTEXT, ICON, and DEFPUSHBUTTON are shorthand notations for defining child window controls in a dialog box template.

Each notation corresponds to a specific window class and predefined window style.

This table summarizes common control types, their corresponding window classes and styles:

Control Type	Window Class	Window Style
PUSHBUTTON	button	BS_PUSHBUTTON
DEFPUSHBUTTON	button	BS_DEFPUSHBUTTON
CHECKBOX	button	BS_CHECKBOX
RADIOBUTTON	button	BS_RADIOBUTTON
GROUPBOX	button	BS_GROUPBOX
LTEXT	static	SS_LEFT
CTEXT	static	SS_CENTER
RTEXT	static	SS_RIGHT
ICON	static	SS_ICON
EDITTEXT	edit	ES_LEFT
SCROLLBAR	scrollbar	SBS_HORZ
LISTBOX	listbox	LBS_NOTIFY
COMBOBOX	combobox	CBS_SIMPLE

Resource Compiler and Control Statement Format:

Only the resource compiler understands the shorthand notation for control types.

- Each control type except EDITTEXT, SCROLLBAR, LISTBOX, and COMBOBOX uses the format: control-type "text", id, xPos, yPos, xWidth, yHeight, iStyle
- The remaining control types use the format: control-type id, xPos, yPos, xWidth, yHeight, iStyle
- All control types have the styles WS_CHILD and WS_VISIBLE by default.
- The iStyle parameter is optional and allows specifying additional styles.

DEEP DIVE INTO CHILD WINDOW CONTROL CUSTOMIZATION

This section delves deeper into customizing child window controls within dialog boxes:

Size and Positioning:

Refer back to Chapter 9 for rules on determining width and height of pre-defined child windows.

Sizes specified in dialog box templates are in terms of 1/4 average character width and 1/8 character height.

Optional Style Field:

The "style" field in control statements allows specifying additional styles beyond the defaults.

Example: Creating a checkbox with text to the left of the box using BS_LEFTTEXT.

Creating Borderless Edit Control:

By default, EDITTEXT controls have borders.

To remove the border, use NOT WS_BORDER in the style field.

Generalized CONTROL Statement:

This statement allows creating any child window control by specifying its class and full style.

Format: CONTROL "text", id, "class", iStyle, xPos, yPos, xWidth, yHeight.

Example: Creating a custom control with window class defined in your program.

This approach can be used in ABOUT3 to create a control with a custom class.

Dialog Manager Processing:

When using CONTROL statements, the WS_CHILD and WS_VISIBLE styles are automatically included.

Windows creates a popup window first and then child windows for each control within the popup.

The CreateWindow call translates the CONTROL statement into a specific call with parameter values based on character size and provided handles.

```
CONTROL "OK", IDOK, "button", WS_CHILD | WS_VISIBLE |
BS_PUSHBUTTON | WS_TABSTOP, 10, 20, 32, 14
```

This statement defines a child window control in a dialog box template. Let's break it down:

- CONTROL: Keyword indicating a child window definition.
- "OK": Text displayed on the control (button label).
- IDOK: Unique identifier for the control.
- "button": Window class of the control (predefined for PUSHBUTTON).
- WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP: Window styles for the control.
- WS_CHILD: Indicates child of another window (dialog box).
- WS_VISIBLE: Makes the control visible.
- BS_PUSHBUTTON: Defines the control type as a push button.
- WS_TABSTOP: Makes the control accessible using the Tab key.
- 10, 20: X and Y coordinates of the control's upper-left corner relative to the dialog box client area (in units of 1/4 average character width).
- 32, 14: Width and height of the control (also in units of 1/4 average character width and 1/8 character height).

Comparison to PUSHBUTTON statement:

```
PUSHBUTTON "OK", IDOK, 10, 20, 32, 14
```

This simpler statement defines the same control using the predefined shorthand notation. It implicitly defines the same window class (button) and window styles (WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP).

Importance of CONTROL statement:

While PUSHBUTTON is convenient for common controls, CONTROL provides greater flexibility:

- Define any type of child window by specifying its window class.
- Customize the complete window style by including additional style flags.

Additional notes:

- When using CONTROL, the WS_CHILD and WS_VISIBLE styles are automatically added by Windows.
- The CreateWindow call in Windows translates the CONTROL statement into a specific call based on character size and provided handles.

Understanding the role of the code:

- This code snippet demonstrates how to define a custom child window control within a dialog box template using the CONTROL statement. This approach allows for greater flexibility in control type and customization compared to predefined shorthand notations.

ABOUT2 PROGRAM IN CHAPTER 11 FOLDER...

This program demonstrates creating and managing a more complex dialog box compared to the simple ABOUT1 program. It showcases radio buttons within the dialog box and painting on its client area.

Main Function (WinMain):

- Registers the window class for the main window.
- Creates the main window with specific style attributes.
- Enters the message loop to process messages.
- Handles WM_COMMAND message for the IDM_APP_ABOUT menu item to display the about dialog box.

Window Procedure (WndProc):

- Handles various messages sent to the main window:
- WM_CREATE: Saves the instance handle.

- WM_COMMAND: Handles menu item selection, specifically the IDM_APP_ABOUT for displaying the about dialog box.
- WM_PAINT: Paints the client area based on the current color and figure.
- WM_DESTROY: Posts a quit message to terminate the program.

PaintWindow Function:

- Fills the client area with a solid color based on the provided color index.
- Paints a rectangle or ellipse based on the provided figure choice.

PaintTheBlock Function:

- Invalidates the client area of the specified control (hCtrl).
- Updates the control window.
- Calls PaintWindow to actually paint the control's area.

About Dialog Box Procedure (AboutDlgProc):

- Manages the about dialog box interactions.
- Handles various messages:
- WM_INITDIALOG: Initializes the dialog box state based on current color and figure.
- WM_COMMAND: Handles button clicks and radio button selections:
- Updates the current color and figure based on user choices.
- Paints the control area with the selected color and figure.
- Responds to OK and Cancel buttons to close the dialog box.
- WM_PAINT: Paints the control area based on the current color and figure.

Key Features and Concepts:

- Radio buttons enable users to choose between options.
- CheckRadioButton function sets and clears radio buttons based on selection.
- GetDlgItem retrieves the handle of a specific control within the dialog box.
- EndDialog function closes the dialog box and sends a WM_CLOSE message to its parent window.
- Custom painting is achieved by handling the WM_PAINT message and drawing the desired content using GDI functions.

The ABOUT2 program uses several resource files to define the visual elements and functionality of the application. Here's a breakdown of each resource file and its role:

ABOUT2.RC:

This file defines the dialog box layout, menu bar, and icon. Here's a breakdown of its key sections:

- Dialog: This section defines the layout of the "About Box" dialog box. It includes elements like text labels, radio buttons, and buttons.
- CTEXT: Static text labels displaying program title and information.
- LTEXT: Placeholder for dynamic content (painted area).
- GROUPBOX: Groups related radio button controls for color and figure selection.
- RADIOBUTTON: Options for color and figure selection.
- DEFPUSHBUTTON: Sets default button for activating with Enter key (OK).
- PUSHBUTTON: Button for canceling the dialog box (Cancel).
- Icon: This section defines the program icon displayed in the title bar and taskbar.
- Menu: This section defines the application's menu bar. It only includes a single "Help" menu with the "About" option triggering the ABOUTBOX dialog display.

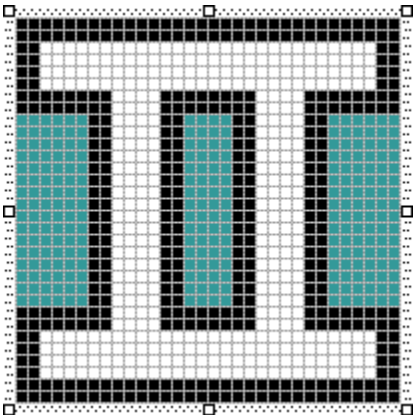
RESOURCE.H:

This file defines numerical identifiers for various resources used in the program code. These identifiers make code references to resources clearer and easier to understand.

- IDC_: Identifiers for dialog box controls like radio buttons and the painted area.
- IDM_: Identifier for the "About" menu item.
- IDC_STATIC: Special identifier used for static text elements.

ABOUT2.ICO:

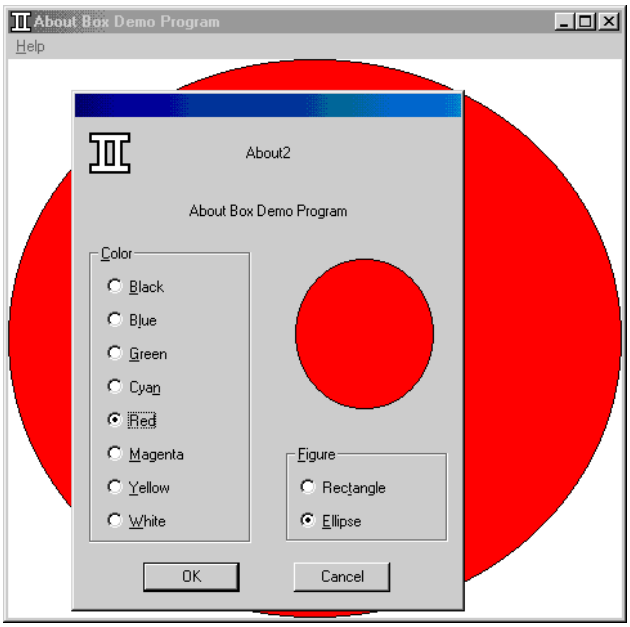
This file contains the actual image data for the program icon. It's used in conjunction with the resource definition in ABOUT2.RC to display the icon.



Overall Importance:

These resource files are crucial for defining the user interface and functionality of the ABOUT2 program. They allow for a clean separation between code and visual elements, making the program easier to maintain and modify.

About2 Program.mp4



ABOUT2 Dialog Box Design and Functionality:

The ABOUT2 program's dialog box uses two groups of radio buttons: one for color selection and another for choosing a rectangle or ellipse. The selected figure is displayed within the dialog box with the currently chosen color. Pressing OK confirms the choice and the main window draws the figure; Cancel leaves the main window unchanged.

Key Points:

Predefined Identifiers: Although the OK and Cancel buttons use standard IDs, radio buttons have specific identifiers starting with "IDC_". Defined in RESOURCE.H, these identifiers ensure proper reference in the code.

Radio Button Creation Order: Creating radio buttons in the order shown in ABOUT2.RC allows Developer Studio to assign sequentially valued identifiers, critical for program functionality.

Uncheck Auto Option: Disable the "Auto" option for each radio button to customize them with specific identifiers.

Group Options: Check the "Group" option for specific controls:

- OK and Cancel buttons ensure only one is active at a time.
- Figure group box and first radio buttons (Black and Rectangle) in each group define selection options.
- Tab Order: Define the order in which users can navigate through controls using the Tab key by clicking them in the order specified in the ABOUT2.RC resource script.

Understanding the Implications:

- Specific identifiers enable the program to handle radio button selections accurately.
- Creating buttons in the correct order ensures proper functionality.
- Group options enhance user experience by defining selection behavior.
- Tab order facilitates efficient keyboard navigation.

Overall:

The ABOUT2 dialog box demonstrates various design considerations, including utilizing radio buttons, assigning unique identifiers, and setting group and tab order for optimal user interaction and program logic.

UNDERSTANDING THE COMMUNICATION FLOW BETWEEN CHILD CONTROLS AND DIALOG BOX PROCEDURES:

Child Window Action: When a user interacts with a child window control (e.g., clicking a radio button), it sends a WM_COMMAND message to its parent window.

Message Components: This message contains information about the event:

- wParam: Low word holds the control's ID. High word carries notification codes (e.g., BN_CLICKED for radio buttons).
- lParam: Holds the window handle of the control.

Dialog Box Procedure:

Windows passes the received WM_COMMAND message to the dialog box procedure within ABOUT2.C.

The procedure analyzes the message:

If it's for a radio button:

- Identifies the specific button by its ID from wParam.
- Enables the clicked button (sets its checkmark).
- Disables all other buttons in the group (removes their checkmarks).

Example: ABOUT2 Radio Button Handling:

User clicks the "Red" radio button.

The button sends a WM_COMMAND message to the dialog box window.

wParam contains:

- Low word: IDC_RED (ID of the clicked button).
- High word: BN_CLICKED (notification code for radio button click).

lParam holds the handle of the "Red" button.

The dialog box procedure receives this message and:

- Identifies the "Red" button based on its ID.
- Sets the checkmark for the "Red" button.

Removes checkmarks from all other color radio buttons in the group.

Key Takeaways:

- WM_COMMAND messages enable communication between child controls and the dialog box procedure.
- Radio button clicks trigger BN_CLICKED notification code.
- The dialog box procedure analyzes these messages and updates the state of controls (checkmarks) accordingly.

Additional Points:

- This interaction applies not only to radio buttons but also to other child controls like check boxes, push buttons, etc.
- Each control might have specific notification codes associated with different actions (e.g., BN_CLICKED vs. BN_HILITE).
- Understanding this communication flow is essential for building responsive and functional dialog boxes.

Problem:

In a dialog box with multiple radio buttons, you need to check and uncheck buttons based on user interaction. However, you don't initially know the window handles of all radio buttons.

Solution:

Getting Window Handle: Use the GetDlgItem function to obtain the window handle of a specific control within the dialog box. This function takes two arguments:

- hDlg: The window handle of the dialog box.

- id: The ID of the control you want to retrieve the handle for.

Checking and Unchecking: Use the SendMessage function to send the BM_SETCHECK message to the target radio button control. This message tells the button to check or uncheck itself. The function takes four arguments:

- hwndCtrl: The window handle of the radio button control.
- BM_SETCHECK: The message identifier for setting the check state.
- wParam: A flag indicating whether to check (1) or uncheck (0) the button.
- lParam: Usually set to 0.

Here's how the code would look like:

```
// Dialog box procedure
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        // Handle messages from radio buttons
        case IDC_BLACK:
        case IDC_RED:
        case IDC_GREEN:
        case IDC_YELLOW:
        case IDC_BLUE:
        case IDC_MAGENTA:
        case IDC_CYAN:
        case IDC_WHITE:
            // Save the selected color ID
            iColor = LOWORD(wParam);

            // Loop through all radio buttons
            for (int i = IDC_BLACK; i <= IDC_WHITE; ++i) {
                // Get the window handle of the current button
                HWND hwndButton = GetDlgItem(hDlg, i);

                // Send BM_SETCHECK message to check or uncheck
                SendMessage(hwndButton, BM_SETCHECK,
                    i == LOWORD(wParam) ? 1 : 0, 0);
            }

            return TRUE;
    }
    // ... other message handling ...
```

Explanation:

- The case statement checks for messages from the radio buttons identified by their ID values.
- Upon receiving a message, the selected color ID is saved in the iColor variable.
- A loop iterates through all radio button IDs (IDC_BLACK to IDC_WHITE).
- For each iteration, the GetDlgItem function retrieves the window handle of the current button.
- The SendMessage function sends the BM_SETCHECK message to the retrieved handle.

- The wParam argument of the message is set to 1 if the current button's ID matches the selected color ID, otherwise it's set to 0. This tells the button to be checked or unchecked accordingly.
- Finally, the function returns TRUE to indicate that the message has been handled.
- This code successfully checks and unchecks radio buttons in the dialog box based on user interaction.

Dialog Box Shortcuts for Radio Buttons and Check Boxes

This section discusses shortcuts available in Windows for handling radio buttons and check boxes in dialog boxes.

1. SendDlgItemMessage:

This function provides a shortcut to send messages directly to child controls within a dialog box. It takes five arguments:

- hDlg: The window handle of the dialog box.
- id: The ID of the child control.
- iMsg: The message identifier to send.
- wParam: Additional message-specific parameter.
- lParam: Additional message-specific parameter.

This function is equivalent to:

```
SendMessage(GetDlgItem(hDlg, id), iMsg, wParam, lParam);
```

2. CheckRadioButton:

This function simplifies checking and unchecking radio buttons within a specific range. It takes four arguments:

- hDlg: The window handle of the dialog box.
- idFirst: The ID of the first radio button in the range.
- idLast: The ID of the last radio button in the range.

- idCheck: The ID of the radio button to check.

This function unchecks all radio buttons in the specified range except for the one with the idCheck value, which will be checked.

```
// In the dialog box procedure:
case WM_COMMAND:
  switch (LOWORD(wParam)) {
    // Handle radio button clicks
    case IDC_RADIO_1:
    case IDC_RADIO_2:
      // Check the selected button and uncheck others
      CheckRadioButton(hDlg, IDC_RADIO_1, IDC_RADIO_2, LOWORD(wParam));
      break;
    // ... other message handling ...
  }
```

3. CheckDlgButton:

This function controls the check mark of a check box within a dialog box. It takes three arguments:

- hDlg: The window handle of the dialog box.
- idCheckbox: The ID of the check box.
- iCheck: Whether to check (1) or uncheck (0) the box.

Setting iCheck to 1 checks the box, while setting it to 0 unchecks it.

```
// In the dialog box procedure:
case WM_COMMAND:
  switch (LOWORD(wParam)) {
    case IDC_CHECKBOX:
      // Toggle the check box state
      CheckDlgButton(hDlg, IDC_CHECKBOX,
        !IsDlgButtonChecked(hDlg, IDC_CHECKBOX));
      break;
    // ... other message handling ...
  }
```

4. IsDlgButtonChecked:

This function retrieves the current check state of a check box within a dialog box. It takes two arguments:

- hDlg: The window handle of the dialog box.
- idCheckbox: The ID of the check box.

The function returns 1 if the checkbox is checked, and 0 if it is unchecked.

```
// In the dialog box procedure:
case WM_CLOSE:
  // Check if the checkbox is checked
  if (IsDlgButtonChecked(hDlg, IDC_CHECKBOX)) {
    // Perform action based on checked state
    ...
  }
  break;
  // ... other message handling ...
```

Using CheckDlgButton with BS_AUTOCHECKBOX:

If you define a checkbox with the BS_AUTOCHECKBOX style, you don't need to process the WM_COMMAND message. You can simply use IsDlgButtonChecked to retrieve the current state of the checkbox before closing the dialog box.

```
// Define checkbox with BS_AUTOCHECKBOX style in dialog resource file
IDC_CHECKBOX, "My Checkbox", ... BS_AUTOCHECKBOX ...

// In the dialog box procedure:
case WM_CLOSE:
  // Retrieve the checkbox state
  int isChecked = IsDlgButtonChecked(hDlg, IDC_CHECKBOX);
  // Perform action based on checkbox state
  if (isChecked) {
    ...
  }
  break;
```

Using BS_AUTORADIOBUTTON:

With the BS_AUTORADIOBUTTON style, using IsDlgButtonChecked is inefficient because you would need to call it for each button until it returns true. Instead, you should still trap WM_COMMAND messages to track the selected button.

```
static int selectedId; // Store selected radio button ID

// In the dialog box procedure:
case WM_COMMAND:
  switch (LOWORD(wParam)) {
    // Handle radio button clicks
    case IDC_RADIO_1:
    case IDC_RADIO_2:
      selectedId = LOWORD(wParam);
      break;
    // ... other message handling ...
  }
```

Remember to replace IDC_ constants with your actual control IDs.

These shortcuts offer several benefits:

- Improved code readability and conciseness.
- Reduced code complexity compared to using SendMessage and loops.
- Increased efficiency, especially for CheckRadioButton.

By using these shortcuts, you can improve the overall quality and maintainability of your code when dealing with radio buttons and check boxes in dialog boxes.

Handling OK and Cancel Buttons in a Dialog Box

This section dives deep into the behavior of OK and Cancel buttons in ABOUT2's dialog box, focusing on keyboard interaction and message handling.

Button IDs and Default Button:

- The OK button has an ID of IDOK (defined as 1).
- The Cancel button has an ID of IDCANCEL (defined as 2).
- The OK button is the default button, denoted by DEFPUSHBUTTON in the dialog resource file.

Keyboard Interaction:

- Pressing Enter sends a WM_COMMAND message with the LOWORD of wParam set to the ID of the default button (IDOK) unless another button has focus (then that button's ID is used).

- Pressing Esc or Ctrl+Break sends a WM_COMMAND message with the LOWORD of wParam set to IDCANCEL.

Message Handling in AboutDlgProc:

The switch statement checks the LOWORD of wParam:

For IDOK:

- Saves the selected color and figure to global variables.
- Calls EndDialog(hDlg, TRUE).
- Returns TRUE.

For IDCANCEL:

- Calls EndDialog(hDlg, FALSE).
- Returns TRUE.

EndDialog and its Significance:

- EndDialog closes the dialog box and returns control to the main window procedure.
- The second parameter of EndDialog is passed back as the return value of DialogBox.
- In ABOUT2, TRUE signifies OK and FALSE signifies Cancel.
- This value helps the main window determine the user's action and update accordingly.

Benefits of TRUE and FALSE:

Using TRUE and FALSE simplifies communication between the dialog box and main window.

It provides a clear and concise way to indicate user intent.

Beyond TRUE and FALSE:

The argument to EndDialog is an int, allowing for more nuanced information.

This could be useful for passing multiple values or specific user choices.

Code Example:

```
// AboutDlgProc function
switch (LOWORD(wParam)) {
  case IDOK:
    // Save selected color and figure
    iCurrentColor = iColor;
    iCurrentFigure = iFigure;
    EndDialog(hDlg, TRUE);
    return TRUE;
  case IDCANCEL:
    EndDialog(hDlg, FALSE);
    return TRUE;
}
```

Additional Notes:

- Windows handles the translation of keyboard shortcuts like Enter, Esc, and Ctrl+Break.
- This simplifies the dialog box procedure by eliminating the need for specific keyboard handling code.
- Understanding how OK and Cancel buttons work with keyboard interaction and message handling is crucial for creating user-friendly dialog boxes.

Using Structures to Avoid Global Variables in Dialog Boxes

While global variables can be convenient, excessive use can lead to code that's difficult to maintain.

In the case of ABOUT2, where the iCurrentColor and iCurrentFigure variables are used in both the window procedure and the dialog procedure, using a structure can offer a more robust and organized approach.

Structure Definition:

```
typedef struct {
  int iColor;
  int iFigure;
} ABOUTBOX_DATA;
```

Window Procedure (WndProc):

Define and initialize a static variable based on the structure:

```
static ABOUTBOX_DATA ad = { IDC_BLACK, IDC_RECT };
```

Replace all occurrences of iCurrentColor and iCurrentFigure with ad.iColor and ad.iFigure respectively.

Use DialogBoxParam with the structure as the last argument:

```
case IDM_ABOUT:
  if (DialogBoxParam(hInstance, TEXT("AboutBox"), hwnd, AboutDlgProc, &ad)) {
    InvalidateRect(hwnd, NULL, TRUE);
  }
  return 0;
```

Dialog Procedure (AboutDlgProc):

Define two static variables:

```
static ABOUTBOX_DATA ad, *pad;
```

In the WM_INITDIALOG message handler:

```
pad = (ABOUTBOX_DATA *)lParam;
ad = *pad;
```

Throughout the dialog procedure, replace iColor and iFigure with ad.iColor and ad.iFigure respectively.

When the user presses OK:

```
case IDOK:
  *pad = ad;
  EndDialog(hDlg, TRUE);
  return TRUE;
```

Benefits:

- Reduced Global Variables: Eliminates the need for separate global variables for the dialog box data.
- Improved Organization: Encapsulates all dialog box-related data within a single structure.
- Enhanced Code Clarity: Makes code easier to understand and maintain by explicitly defining the data structure.
- Increased Flexibility: Allows for easily adding more data to the structure if needed.

Additional Notes:

- This approach can be applied to any dialog box in your program, promoting consistency and organization.
- By using structures and passing them via DialogBoxParam, you can avoid cluttering your code with global variables, leading to a cleaner and more maintainable codebase.

TAB STOPS AND GROUPS IN DIALOG BOXES

This section details how tab stops and groups work in dialog boxes, along with their implementation using window styles and keyboard interaction.

Tab Stops:

- Controls with the WS_TABSTOP window style receive focus when the Tab key is pressed.
- By default, some controls like radio buttons and push buttons include WS_TABSTOP.
- Other controls, like static text, do not have it by default as they don't require input focus.
- Unless explicitly set, the first control with WS_TABSTOP receives focus upon opening the dialog box.

Groups:

- Controls with the WS_GROUP window style mark the beginning of a group for cursor key navigation.
- Cursor keys move focus within a group, cycling back to the first control if necessary.
- Controls like LTEXT, CTEXT, RTEXT, and ICON have WS_GROUP by default, marking group ends.
- Other controls might need explicit addition of WS_GROUP to define group boundaries.

Example: ABOUT2 Dialog Box:

- The first radio button in each group and both push buttons have WS_TABSTOP.
- The first radio button in each group and the default push button have WS_GROUP.
- This allows Tab key navigation between groups and cursor key navigation within groups.
- Pressing the underlined letter in a group label focuses the currently checked radio button.

Windows Magic:

- Windows automatically updates the WS_TABSTOP style for the currently checked radio button.
- This ensures the Tab key always focuses the chosen option within a group.

Additional Functions:

- GetNextDlgTabItem: Retrieves the next or previous tab stop control based on bPrevious parameter.
- GetNextDlgGroupItem: Retrieves the next or previous group item based on bPrevious parameter.

Benefits:

- Simplifies keyboard interaction within dialog boxes.
- Provides intuitive navigation for users.
- Enhances accessibility for users with limited mouse input.

Code Example:

```
// Dialog box template in ABOUT2.RC
// Explicit WS_TABSTOP for first radio buttons
IDC_BLACK, "Black", ... WS_TABSTOP ...

// Default WS_TABSTOP for push buttons
DEFPUSHBUTTON "OK", ... WS_TABSTOP ...

// Explicit WS_GROUP for first radio buttons and default push button
IDC_BLACK, "Black", ... WS_GROUP ...
DEFPUSHBUTTON "OK", ... WS_GROUP ...
```

Tabstop is a feature in some dialog boxes that allows you to move between controls using keyboard shortcuts:

- Tab key: Pressing the Tab key moves focus from one control with the WS_TABSTOP style to the next in the tab order.
- Cursor keys: Within a group of controls marked with the WS_GROUP style, the cursor keys (up, down, left, right) move focus between the controls in that group.
- Underlined letters: Pressing the underlined letter in a group label focuses the currently checked radio button within that group.

This feature makes navigating dialog boxes easier and more efficient, especially for users who may have difficulty using a mouse.

Here are some additional details about tabstops:

- The first control with the WS_TABSTOP style receives focus when the dialog box opens, unless otherwise specified.
- Some controls, like radio buttons and push buttons, have the WS_TABSTOP style by default. Others, like static text, lack it by default and require explicit addition.
- Controls like LTEXT, CTEXT, RTEXT, and ICON have the WS_GROUP style by default, marking group boundaries. Other controls might need explicit addition of WS_GROUP to define group boundaries.
- Windows automatically adds the WS_TABSTOP style to the currently checked radio button within a group. This ensures that the Tab key always focuses the chosen option within a group, enhancing user experience.

- Two functions are available in Windows: GetNextDlgTabItem and GetNextDlgGroupItem. These functions retrieve the next or previous tab stop control or group item based on a parameter.

```
// Dialog box template styles
// Specify WS_TABSTOP for controls you want to access using the Tab key
// Example:
// IDC_BLACK, IDC_RED, IDC_GREEN, IDC_YELLOW are radio buttons with WS_TABSTOP
// IDC_RECTANGLE, IDC_ELLIPSE are radio buttons within a group marked by WS_GROUP
// IDC_OK and IDC_CANCEL are push buttons with WS_TABSTOP

// Set focus to the first control with WS_TABSTOP during WM_INITDIALOG
case WM_INITDIALOG:
    SetFocus(GetDlgItem(hDlg, IDC_BLACK)); // Set initial focus to IDC_BLACK
    return FALSE;

// Handle Tab key navigation between controls with WS_TABSTOP
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case IDC_BLACK:
        case IDC_RED:
        case IDC_GREEN:
        case IDC_YELLOW:
            // Handle radio button selection
            break;
        case IDC_RECTANGLE:
        case IDC_ELLIPSE:
            // Handle radio button selection within a group
            break;
        case IDC_OK:
        case IDC_CANCEL:
            // Handle push button clicks
            break;
    }
```

Overall, tabstop is a valuable feature that improves the usability and accessibility of dialog boxes.

PAINTING ON THE ABOUT2 DIALOG BOX

The ABOUT2 dialog box showcases a unique feature: custom painting within its own client area. This section delves deeper into the mechanics behind this achievement, analyzing the code and explaining the key concepts involved.

The Blank Canvas: An LTEXT Control Takes Center Stage

The foundation of this custom painting lies in ABOUT2.RC, where a static LTEXT control is defined with specific positioning and size. This control, despite having no text assigned, serves as the canvas upon which the desired artwork is drawn. Its dimensions, 18 characters wide and 9 characters high, determine the boundaries of the painting area.

Initiating the Painting Process: From User Interaction to Function Calls

When the user interacts with the dialog box, selecting a different color or figure option, a specific chain of events unfolds. This chain ultimately leads to the execution of the PaintTheBlock function, responsible for orchestrating the painting process.

Step-by-Step Breakdown of PaintTheBlock:

Invalidation: The function begins by invalidating the child window control using InvalidateRect. This triggers the control to send a WM_PAINT message, requesting its own redrawing.

Message Generation: UpdateWindow is called, ensuring that the WM_PAINT message is sent to the child window control, effectively notifying it of the need to repaint its client area.

Painting Orchestration: Finally, PaintWindow, another function within ABOUT2.C, is invoked. This function takes the reins and handles the actual drawing process.

The Inner Workings of PaintWindow: Bringing the Figure to Life

PaintWindow performs several crucial tasks to achieve the desired painting effect:

Device Context Acquisition: It first retrieves a device context handle for the child window control. This context acts as the bridge between the painting code and the actual drawing surface.

Figure Drawing: Based on the provided color and figure parameters, the function draws the selected figure onto the canvas. The choice of color dictates the brush used for filling the figure, creating the visual representation.

Pixel-Perfect Positioning: To ensure accurate drawing, PaintWindow utilizes GetClientRect to retrieve the child window's dimensions in pixels. This ensures that the figure is drawn within the designated bounds of the painting area.

Coloring the Canvas: The chosen color is brought to life by creating a brush based on its value. This brush is then used to fill the drawn figure, breathing life into the artwork.

Beyond the Technicalities: Benefits and Considerations

This approach to custom painting offers several advantages:

Targeted Painting: It enables painting on a specific area within the dialog box, leaving the remaining client area untouched.

Leveraging Standard Mechanisms: By utilizing the existing WM_PAINT message handling for child controls, the code adheres to established Windows procedures, promoting efficiency and maintainability.

Organizational Clarity: Separating the painting logic into a dedicated function (PaintWindow) enhances code organization and simplifies the main dialog procedure.

Alternative Approaches:

While GetClientRect effectively retrieves pixel dimensions, another option exists:

- MapDialogRect. This function offers a different approach, converting character coordinates defined in the dialog box template to pixel coordinates within the client area.

Conclusion: Unveiling the Art of Dialog Box Painting

The ABOUT2 dialog box demonstrates a clever and effective technique for achieving custom painting within a dialog box environment. By leveraging a child window control and its WM_PAINT message handling, the code draws the desired figure within the designated area, offering a glimpse into the creative possibilities within Windows applications. This approach serves as a valuable example for developers seeking to enhance the visual appeal and functionality of their dialog boxes.

EXPLORING ADVANCED TECHNIQUES FOR DIALOG BOXES:

This section delves into additional functions and capabilities associated with dialog boxes, venturing beyond the basics.

Beyond Basic Functionality: Moving Controls and Enabling/Disabling Features

While dialog boxes offer a pre-defined set of functionalities, some occasions call for further customization. Fortunately, developers can leverage existing Windows functions to achieve desired effects:

- MoveWindow: This function, originally meant for child windows, can be employed in dialog boxes as well. It allows developers to dynamically move controls around the dialog box, creating a unique or playful experience for users.
- EnableWindow: This function offers control over the interaction capabilities of individual controls within a dialog box. By setting the 'bEnable' parameter to TRUE or FALSE, developers can enable or disable specific controls based on user interactions or

other conditions. This ensures that only relevant options are available at any given time, enhancing the user experience.

Dynamic Dialog Boxes: Enabling On-the-Fly Control

Dialog boxes often require adjustments based on user choices or other dynamic factors. Here's how developers can achieve this level of flexibility:

- EnableWindow: As mentioned earlier, this function plays a crucial role in adapting the dialog box to user actions. By dynamically enabling and disabling controls based on selections or other inputs, developers can create a more responsive and intuitive experience.
- Custom Control Definition: While Windows provides various standard controls, developers can define their own to fulfill specific needs. For instance, instead of using rectangular push buttons, a developer could create elliptical ones by registering a custom window class and implementing a dedicated window procedure. This level of customization allows for unique and innovative dialog box designs.

A Case Study: ABOUT3 and Custom Elliptical Buttons

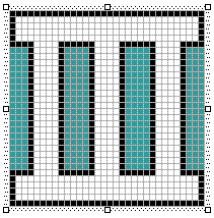
The ABOUT3 program serves as a concrete example of defining custom controls within a dialog box. This program utilizes a custom window class and window procedure to create elliptical push buttons, replacing the standard rectangular buttons. This demonstrates how developers can extend the capabilities of dialog boxes beyond their default functionality.

By venturing beyond basic functionalities and exploring advanced techniques like custom control definition and dynamic manipulation of controls, developers can unlock the full potential of dialog boxes.

These advanced techniques can enhance the user experience, create unique and visually appealing interfaces, and adapt the dialog box behavior to specific needs, ultimately leading to more engaging and interactive applications.

About 3 program in Chapter 11 folder….

About3
Program-Press the o

 The icon for the program.

About3 demonstrates several unique aspects compared to standard dialog boxes:

1. Custom Elliptical Push Buttons:

Instead of utilizing the standard rectangular push buttons, About3 defines its own "EllipPush" control. This custom control uses a dedicated window class and its associated window procedure (EllipPushWndProc) to handle messages and draw elliptical buttons. This showcases the ability to create custom controls beyond the standard offerings.

2. Painting on the Dialog Box:

While other dialog boxes typically handle painting within their child controls, About3 takes a different approach. It leverages the WM_PAINT message and a dedicated function (PaintWindow) to directly paint on the client area of the dialog box itself. This allows for drawing outside the boundaries of child controls, offering greater flexibility.

3. Dynamic Control Behavior:

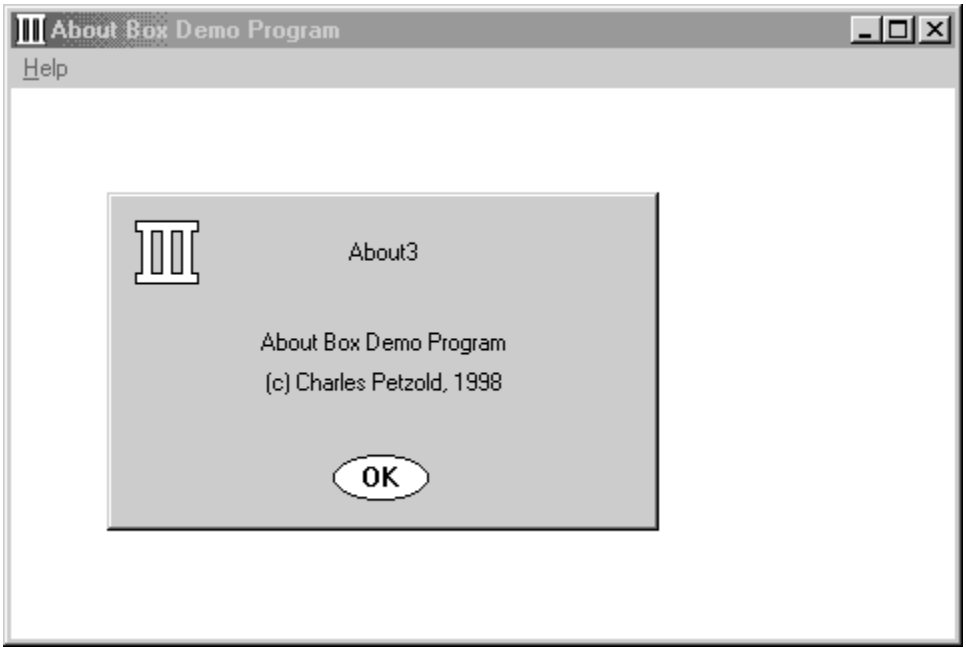
About3 utilizes the EnableWindow function to dynamically enable and disable the "OK" button based on user actions. This allows for a more responsive and interactive experience, ensuring only relevant options are available at any given time.

4. Interfacing with Parent Window:

The EllipPushWndProc function demonstrates how custom controls can communicate with their parent window. When the user interacts with the elliptical button, a message is sent to the parent dialog box, notifying it of the user's action. This enables custom controls to trigger specific behaviors within the containing dialog box.

5. Text and Color Customization:

About3 showcases how custom controls can manipulate their own text and color attributes. The EllipPushWndProc function retrieves the button's text and paints it on the elliptical surface. Additionally, it uses system colors to fill the ellipse and render text, ensuring visual consistency with the overall theme.



Key Takeaways from About3's Custom Control:

1. EllipPush Window Class:

This section dives deeper into the unique aspects of the EllipPush custom control:

- Window Class Registration: Unlike standard controls, EllipPush requires registering its own window class with specific attributes. This class defines the behavior and appearance of the elliptical buttons.
- Dialog Box Integration: Within the dialog editor, the "Custom Control" option allows embedding an EllipPush instance. Setting the "Class" property to "EllipPush" replaces the standard DEFPUSHBUTTON statement in the dialog box template. This binds the button to the custom window class.

2. EllipPushWndProc Function:

This function serves as the heart of the EllipPush control:

- Message Processing: It handles three specific messages: WM_PAINT, WM_KEYUP, and WM_LBUTTONUP.
- WM_PAINT Handling: This message triggers the drawing of the ellipse and its text. GetClientRect retrieves the button's size, while GetWindowText obtains the displayed text. The Windows API functions Ellipse and DrawText are then utilized to draw the desired shape and text within the button's boundaries.
- User Interaction: Both WM_KEYUP (with SPACE key) and WM_LBUTTONUP (mouse click) events trigger the same behavior. The control sends a WM_COMMAND message to its parent window (dialog box) with its ID as wParam. This informs the dialog box of the user's action and allows it to respond accordingly.

3. Benefits and Applications:

Creating custom controls like EllipPush offers several benefits:

- Visual Customization: It enables developers to design buttons and other controls with unique shapes and appearances, exceeding the limitations of standard controls.
- Enhanced Interaction: Custom controls can handle user interaction in specific ways, providing more tailored and engaging experiences.
- Extending Functionality: By defining custom window procedures, developers can implement functionalities not readily available in pre-built controls.

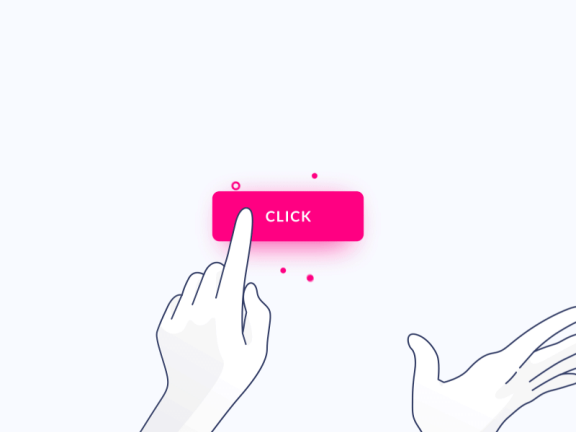
4. Broader Implications:

The EllipPush example serves as a stepping stone for exploring further customization possibilities within dialog boxes. Developers can leverage this approach to create various custom controls, each with its own unique design, interaction patterns, and functionalities. This opens doors to richer and more interactive dialog box experiences.

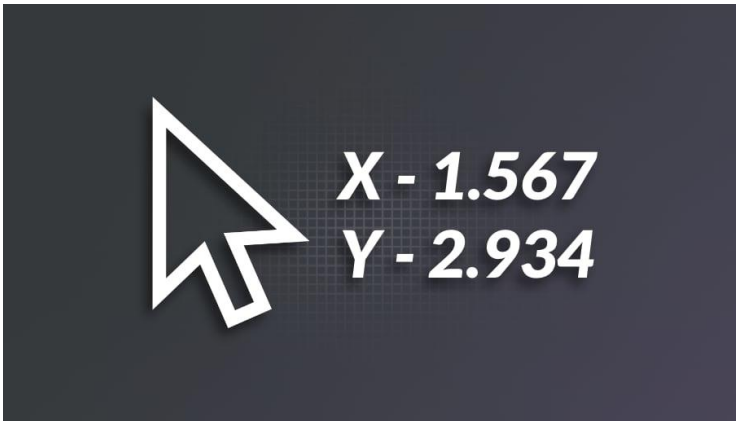
ELLIPPUSHWNDPROC: BEYOND THE BASICS

While EllipPushWndProc demonstrates the core functionality of a custom child window control, it lacks certain features and optimizations found in standard controls. Here's a deeper look at what's missing:

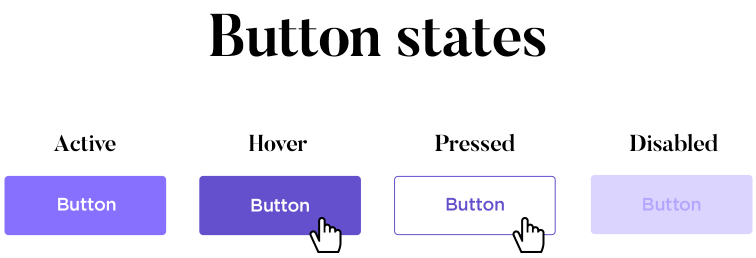
Button Flashing: Unlike standard push buttons, the EllipPush button doesn't visually indicate activation by flashing its colors. To achieve this effect, the window procedure needs to handle WM_KEYDOWN (Spacebar) and WM_LBUTTONDOWN messages. By inverting the interior colors on these events and restoring them on release or loss of focus, the button simulates the visual feedback of standard buttons.



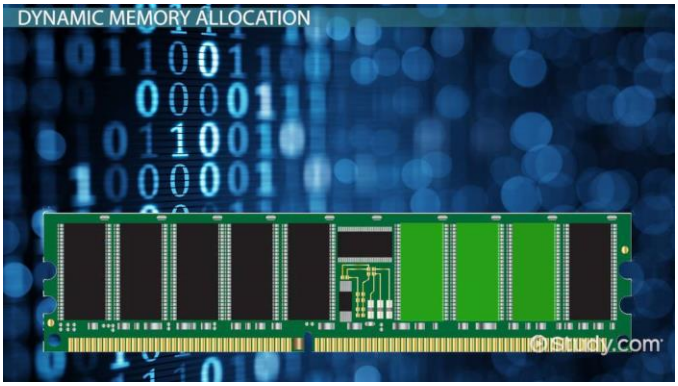
Mouse Capture: To handle clicks outside the button's bounds while it's pressed, the window procedure should capture the mouse on WM_LBUTTONDOWN. This involves setting the capture flag with SetCapture. If the mouse moves outside the button's area while the button is depressed, the window procedure should track the mouse movement and release the capture (and restore normal colors) on WM_MOUSEMOVE. Only if the button is released within its bounds should a WM_COMMAND message be sent to the parent.



Handling WM_ENABLE: Currently, EllipPushWndProc doesn't respond to WM_ENABLE messages. When a dialog box disables a control using EnableWindow, the control's text should turn gray to indicate its inactive state. To achieve this, the window procedure needs to process WM_ENABLE and update the text color based on the message's wParam value.



Control-Specific Data Storage: If a child window requires storing data specific to each instance, it can leverage the cbWndExtra field in the window class structure. By setting this value to a positive value, the control allocates additional memory within its internal structure. This memory can be accessed and manipulated using SetWindowLong and GetWindowLong, allowing for custom data storage and retrieval.



By addressing these aspects, EllipPushWndProc can be enhanced to provide a more complete and user-friendly custom control experience, offering functionality comparable to standard controls while maintaining the flexibility and customization possibilities of custom control creation.