CHAPTER 11: DIALOG BOXES - A DEEP DIVE

Introduction:

Dialog boxes play a crucial role in user interaction, allowing programs to gather additional input beyond simple menus. They typically appear as popup windows containing various child window controls like text boxes, buttons, and radio buttons.

Dialog Box Creation:

Templates: Developers define the layout and appearance of dialog boxes through templates embedded within the program's resource script file. These templates specify the size, position, and type of each control within the dialog box.

Visual Studio: Modern development environments like Visual Studio offer interactive tools for designing dialog boxes. This simplifies the process and generates the corresponding resource script code automatically.

Dialog Box Management:

Windows Responsibility: Upon invocation, Windows 98 takes over the responsibility of creating the dialog box window, its child controls, and a dedicated window procedure to handle messages.

Dialog Box Manager: This internal Windows code manages various aspects of the dialog box, including keyboard and mouse input, and provides the framework for interaction.

Dialog Procedure:

Program-Defined Function: While Windows handles core functionality, developers can implement a custom "dialog box procedure" to perform specific tasks.

Purpose: This procedure typically initializes child controls, processes messages from them (e.g., button clicks), and handles the dialog box's closing.

Focus and Input: Unlike standalone windows, dialog box procedures don't handle WM_PAINT messages directly or directly process keyboard/mouse input.

Child Window Controls in Dialog Boxes:

Simplified Management: Compared to managing child windows in standalone programs, dialog boxes offer a simpler approach.

Windows Assistance: The built-in dialog box manager takes care of many tasks, including handling focus transition between controls, which was a challenge in Chapter 9.

Building a Simple Dialog Box:

This chapter explores the process of creating and implementing a simple dialog box, showcasing the interplay between the various components involved.

Additional Considerations:

Complexity: While the focus is on a basic example, creating complex dialog boxes with rich features requires more advanced techniques covered later.

Learning Curve: While leveraging child controls within dialog boxes simplifies certain aspects, it introduces new concepts and procedures specific to dialog box interaction.

MODELESS DIALOG BOXES: BEYOND MODALITY

This section delves deeper into the concept of modeless dialog boxes, exploring their characteristics and contrasting them with modal dialog boxes.

Recap: Modal vs. Modeless Dialog Boxes:

Modal: These dialog boxes restrict user interaction to only the dialog box and the program that initiated it. They block access to other windows within the program until closed.

Modeless: These dialog boxes offer greater flexibility by allowing users to switch between the dialog box, the program, and even other applications concurrently.

Benefits of Modeless Dialog Boxes:

Enhanced User Experience: Users can keep the dialog box open for reference while working within the main program, avoiding the need to repeatedly open and close it.

Improved Efficiency: Tasks requiring frequent interaction with both the dialog box and the program are streamlined, minimizing context switching and saving time.

Greater Flexibility: Users can access information displayed in the dialog box while working on other tasks, promoting multitasking and efficient workflow.

Function Comparison: DialogBox vs. CreateDialog

DialogBox: This function is specifically designed for modal dialog boxes. It creates the dialog box, handles user interaction, and only returns after the dialog box is closed.

CreateDialog: This function creates modeless dialog boxes. It returns immediately after creation, handing over the responsibility of managing the dialog box to the program.

Code Comparison:

```
92  // Creating a Modal Dialog Box
93  hDlgModal = DialogBox(hInstance, szTemplate, hwndParent, DialogProc);
94
95  // Creating a Modeless Dialog Box
96  hDlgModeless = CreateDialog(hInstance, szTemplate, hwndParent, DialogProc);
```

Remembering the Difference:

The function names provide a clue to their purpose. "DialogBox" emphasizes the box-like nature of modal dialogs, while "CreateDialog" aligns with the creation of regular windows, similar to "CreateWindow".

Additional Considerations:

Modeless dialog boxes require more careful management than modal ones. Developers need to handle closing, responding to user actions, and ensuring the dialog box remains accessible while not interfering with the main window.

The choice between modal and modeless depends on the specific needs of the application and the intended user interaction.

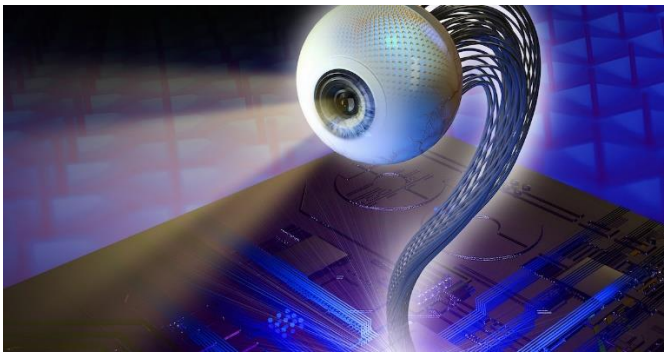
Deep Dive: Modeless Dialog Boxes and Their Differences

This section delves deeper into the differences between modal and modeless dialog boxes, highlighting key aspects and implementation considerations.

Visual Differences:

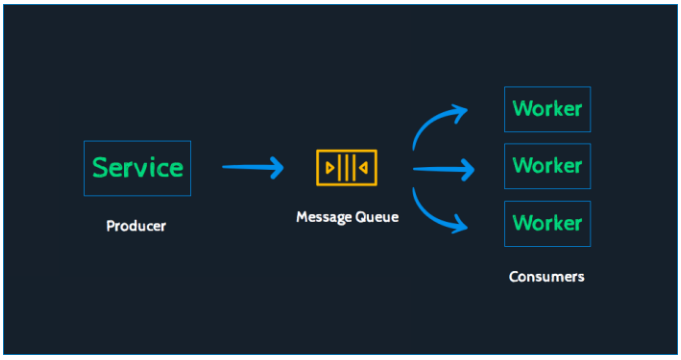
Caption Bar and System Menu: Unlike modal dialogs, modeless ones typically include a caption bar for moving the window and a system menu for additional options. This is reflected in the dialog template's STYLE statement, which usually includes WS_CAPTION and WS_SYSMENU styles.

Visibility: Defaulting to hidden, modeless dialog boxes require either the WS_VISIBLE style in the template or an explicit ShowWindow call with SW_SHOW to become visible. This differs from modal dialogs which are displayed automatically.



Message Handling:

Message Queue: Messages intended for modeless dialog boxes are delivered through the program's message queue, requiring special handling.



IsDialogMessage: This function determines if a message is intended for the modeless dialog box. If so, it sends the message to the appropriate window procedure and returns TRUE. Otherwise, it returns FALSE.

Modified Message Loop: The program's main message loop needs to incorporate IsDialogMessage to filter and dispatch messages accordingly. The basic structure involves checking the dialog box handle (hDlgModeless) and using IsDialogMessage before calling TranslateMessage and DispatchMessage.

Keyboard Accelerators: Programs using keyboard accelerators need to further refine their message loop to ensure proper handling of accelerator messages alongside dialog box messages.

Code Comparison:

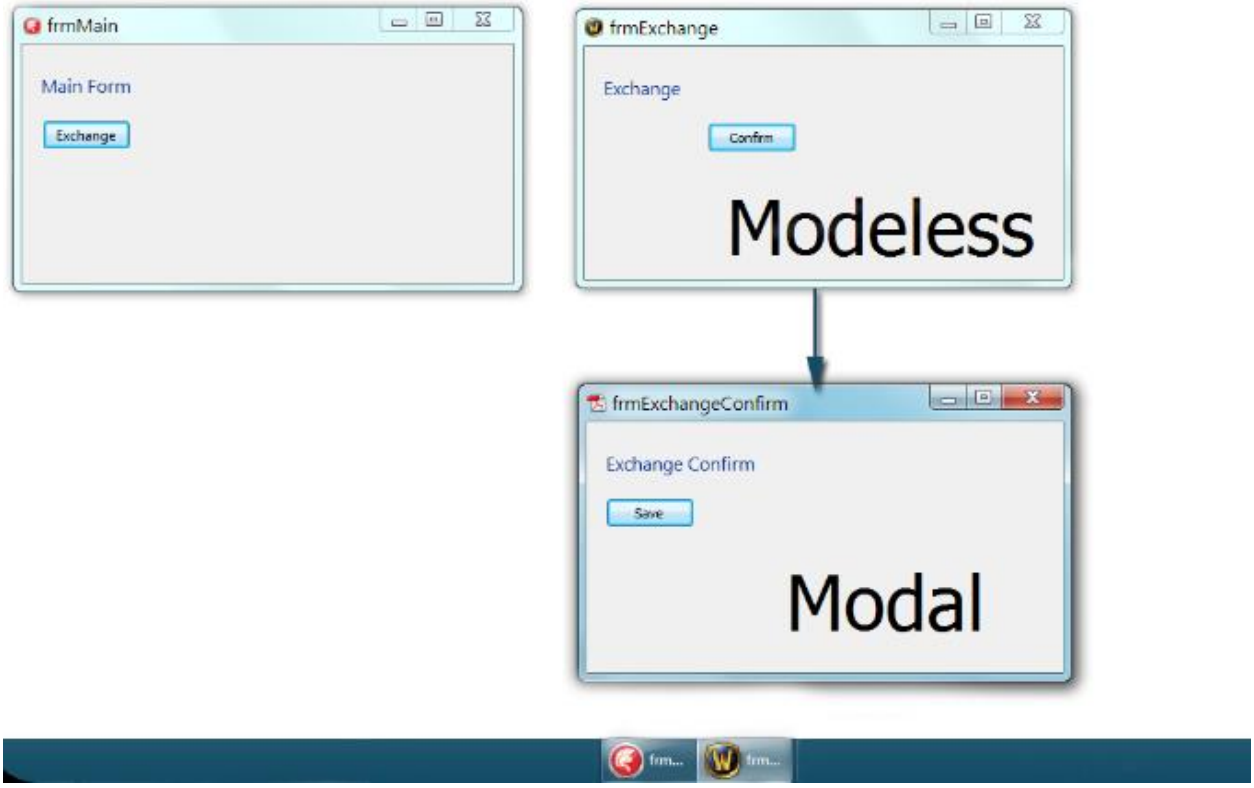
```
100  // Modal Dialog Box
101  hDlgModal = DialogBox(hInstance, szTemplate, hwndParent, DialogProc);
102
103  // Modeless Dialog Box
104  hDlgModeless = CreateDialog(hInstance, szTemplate, hwndParent, DialogProc);
105
106  // Message Loop with IsDialogMessage
107  while (GetMessage(&msg, NULL, 0, 0)) {
108    if (hDlgModeless == 0 || !IsDialogMessage(hDlgModeless, &msg)) {
109      TranslateMessage(&msg);
110      DispatchMessage(&msg);
111    }
112  }
113
114  // Message Loop with Accelerators (additional check)
115  while (GetMessage(&msg, NULL, 0, 0)) {
116    if (hDlgModeless == 0 || !IsDialogMessage(hDlgModeless, &msg)) {
117      if (!TranslateAccelerator(hwnd, hAccel, &msg)) {
118        TranslateMessage(&msg);
119        DispatchMessage(&msg);
120      }
121    }
122  }
```

Additional Considerations:

Managing focus: Modeless dialog boxes need careful attention to focus management, ensuring proper focus transitions between the dialog box and other windows.

Memory management: Since the dialog box remains open, proper memory management of the associated window handle is crucial.

User experience: Modeless dialog boxes offer greater flexibility but require careful design to avoid cluttering the desktop and interfering with the main program's functionality.



In this example:

Modal Dialog Box:

- The dialog box has no caption bar or system menu.
- The dialog box is fully opaque and blocks the underlying window.
- The dialog box must be closed before the user can interact with the underlying window.

Modeless Dialog Box:

- The dialog box has a caption bar and system menu.
- The dialog box is semi-transparent, allowing the underlying window to be partially visible.
- The user can interact with the underlying window while the dialog box is open.

In the image, the "Exchange" dialog box is a modal dialog box, while the "Confirm" dialog box is a modeless dialog box.

Here is a table that summarizes the key differences between modal and modeless dialog boxes:

Characteristic	Modal Dialog Box	Modeless Dialog Box
Caption bar and system menu	No	Yes
Opacity	Opaque	Semi-transparent
User interaction	Blocks underlying window	Allows interaction with underlying window

MASTERING MODELESS DIALOG BOXES: A COMPREHENSIVE GUIDE

This in-depth exploration delves beyond the basics of creating modeless dialog boxes, equipping developers with the knowledge to manage them effectively.

1. The Power of hDlgModeless:

This global variable serves as the central hub for managing the modeless dialog box.

Initialized to 0 by default, it safeguards against invalid handle usage with IsDialogMessage.

Its versatile nature allows for:

- Existence Check: Verifying the dialog box's presence in other program parts.
- Inter-window Communication: Facilitating message exchange between the dialog box and other windows.
- Destruction Control: Identifying the correct handle for proper destruction using DestroyWindow.

2. Ending a Modeless Dialog Box:

Unlike modal dialogs, DestroyWindow replaces EndDialog for closing the dialog box.

Setting hDlgModeless to NULL after DestroyWindow ensures proper memory management.

Users often close the dialog box via the system menu's "Close" option.

The dialog box procedure must capture the WM_CLOSE message:

- It triggers DestroyWindow to close the dialog box.
- Setting hDlgModeless to NULL completes the destruction process.

3. Push Button Closure:

Similar to handling WM_CLOSE, push buttons can also initiate closure.

Upon button click, DestroyWindow is called, followed by setting hDlgModeless to NULL.

4. Data Exchange with Parent Window:

Two primary approaches exist for information exchange between the dialog box and the parent window.

- Global Variables: A convenient method for storing data that needs to be "returned" by the dialog box.
- CreateDialogParam: This advanced technique allows passing a structured data pointer for more complex data exchange.

5. Message Loop Orchestration:

IsDialogMessage plays a crucial role in filtering messages intended for the modeless dialog box.

Its integration within the message loop ensures proper message routing.

Handling keyboard accelerators requires combining IsDialogMessage with TranslateAccelerator for seamless interaction.

hDlgModeless Global Variable:

```
HWND hDlgModeless;

// CreateDialog call
hDlgModeless = CreateDialog(hInstance, szTemplate, hwndParent, DialogProc);

// Message Loop with IsDialogMessage
while (GetMessage(&msg, NULL, 0, 0)) {
  if (hDlgModeless == 0 || !IsDialogMessage(hDlgModeless, &msg)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

Ending a Modeless Dialog Box:

```
case WM_CLOSE:
  DestroyWindow(hDlg);
  hDlgModeless = NULL;
  break;

// Push Button Click Handler
void OnButtonClick(HWND hDlg, WPARAM wParam) {
  if (LOWORD(wParam) == IDCLOSE) {
    DestroyWindow(hDlg);
    hDlgModeless = NULL;
  }
}
```

Push Button Closure:

```
// Push Button Click Handler
void OnButtonClick(HWND hDlg, WPARAM wParam) {
  if (LOWORD(wParam) == IDCLOSE) {
    DestroyWindow(hDlg);
    hDlgModeless = NULL;
  }
}
```

Data Exchange with Parent Window:

a) Global Variables:

```
int dialogReturnValue = 0; // Store data in this variable
...

// In Dialog Procedure
dialogReturnValue = ... // Calculate or retrieve data
...
case WM_DESTROY:
  PostMessage(hwndParent, WM_DIALOGBOX_RETURN, IDOK, (LPARAM)dialogReturnValue);
  break;
```

b) CreateDialogParam:

```
142  struct MyData {
143      int value1;
144      char text[128];
145  };
146
147  ...
148
149  MyData data;
150  data.value1 = 10;
151  strcpy(data.text, "Example text");
152
153  hDlgModeless = CreateDialogParam(hInstance, szTemplate, hwndParent, DialogProc, (LPARAM)&data);
```

5. Message Loop Orchestration:

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

Colors2 program in chapter 11 folder….

Colors2 Program.mp4

COLORS2: A Modeless Dialog Box Approach

COLORS2 represents a significant simplification of the COLORS1 program, leveraging the power of modeless dialog boxes. This document dives deep into its functionality and compares it to its predecessor.

COLORS2 utilizes a modeless dialog box to manage the scroll bars and text items previously implemented using child windows in COLORS1.

This approach significantly reduces complexity, particularly within the WndProc function.

The program still allows users to adjust RGB values via scroll bars and displays the corresponding color.

Key Features:

Modeless Dialog Box: The central element of COLORS2 is the "Color Scroll Scrollbars" dialog box. Unlike COLORS1, this dialog box is not modal, allowing users to interact with both the dialog and the main window simultaneously.

Scroll Bar Functionality: Three scroll bars remain, each controlling one RGB color channel (Red, Green, Blue). Users can adjust the values by clicking the arrows, dragging the scroll bar thumb, or entering a value directly in the associated text box.

Dynamic Color Display: As users adjust the scroll bars, the background color of the main window dynamically updates to reflect the chosen RGB combination.

Implementation Compared to COLORS1:

WndProc Simplification: By utilizing a modeless dialog box, the WndProc function in COLORS2 becomes considerably simpler. It now only handles the WM_DESTROY message for the main window and relies on the dialog box for all scroll bar interactions.

Reduced Complexity: COLORS2 eliminates the need to manage nine child windows individually, reducing the code footprint and complexity.

Message Handling: The program uses IsDialogMessage within the message loop to filter and dispatch messages intended for the dialog box. This ensures efficient message routing.

Code Breakdown:

Main Program: Creates the main window, shows it, and then creates the modeless dialog box using CreateDialog.

Dialog Box: Handles all scroll bar messages (WM_VSCROLL) and updates the corresponding color values, scroll bar positions, text box values, and background color of the main window.

Resource Script: Defines the layout and appearance of the dialog box, including scroll bars and text boxes.

Benefits of Modeless Dialog Box:

Improved User Experience: Users can adjust color values while still having access to the main window, enhancing workflow and interaction.

Reduced Code Complexity: Managing a single dialog box is simpler and more efficient than handling numerous child windows.

Focus Management: COLORS2 automatically handles focus transitions between the dialog box and other elements, ensuring a smooth user experience.

Main Program:

- Creates the main window with the WS_CLIPCHILDREN style, allowing it to repaint without erasing the dialog box.
- Shows the main window.
- Creates the modeless dialog box using CreateDialog and stores the handle in the global variable hDlgModeless.
- Enters the message loop and filters messages with IsDialogMessage to ensure proper message routing.

```
HWND hwnd, hDlgModeless;

...

hwnd = CreateWindow(...);
ShowWindow(hwnd, ...);
hDlgModeless = CreateDialog(...);

while (GetMessage(&msg, NULL, 0, 0)) {
  if (!IsDialogMessage(hDlgModeless, &msg)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

Dialog Box:

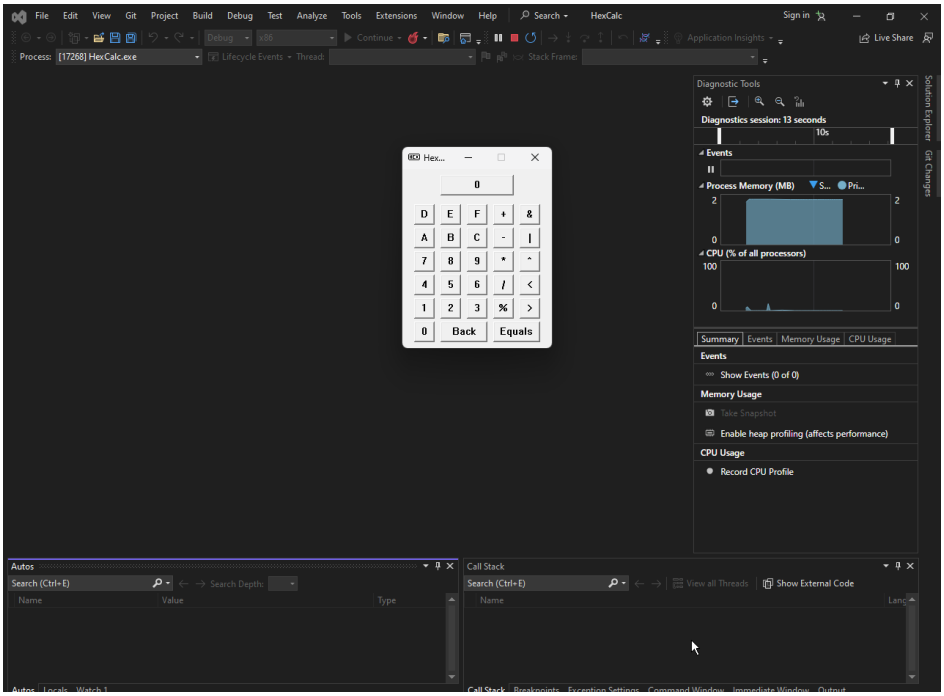
- Handles all scroll bar messages (WM_VSCROLL) based on their ID numbers (10, 11, and 12).
- Updates the corresponding color values (iColor[i]) based on user interactions.
- Sets the scroll bar positions, text box values, and background color of the main window accordingly.

```
case WM_VSCROLL:
  iCtrlID = GetWindowLong((HWND) lParam, GWL_ID);
  ...
  switch (LOWORD(wParam)) {
    // Handle scroll bar actions
  }
  SetScrollPos((HWND) lParam, SB_CTL, iColor[iCtrlID], TRUE);
  SetDlgItemInt(hDlg, iCtrlID + 3, iColor[iCtrlID], FALSE);
  ...
  break;
```

Resource Script:

- Defines the layout and appearance of the dialog box using controls like scroll bars, static text fields, and labels.

HEXCALC PROGRAM



HEXCALC: A Lazy Programming Marvel

HEXCALC, a program written by Charles Petzold, demonstrates the pinnacle of "lazy programming" by achieving a functional hexadecimal calculator with minimal code.

Instead of employing traditional methods like window creation and message processing, it leverages the power of dialog boxes to simplify its implementation.

Key Features:

- 10-Function Hexadecimal Calculator: Performs basic arithmetic and logical operations on hexadecimal numbers.
- Full Keyboard and Mouse Interface: Supports key presses and button clicks for input and operation selection.
- Minimal Code: Achieved through clever use of dialog boxes and resource files, reducing code complexity and size.
- Dynamic Display: Current number and operation are displayed in real-time.

Technical Highlights:

Dialog Box as the Main Window: Instead of creating a custom window, HEXCALC utilizes a dialog box defined in a resource file. This simplifies window management and reduces code overhead.

Resource Script: Defines the layout and functionality of the calculator interface, including buttons, labels, and text fields.

WndProc Handles Key and Button Events: The WndProc function processes key presses and button clicks, performing calculations and updating the display accordingly.

Message Loop: The program runs in a message loop, receiving and responding to user input and system messages.

Lazy Programming Techniques:

Minimal Window Management: Eliminates the need to create and destroy windows, simplifying the program's logic.

Resource-Driven Interface: Relies on pre-defined resources for the interface, reducing code duplication and maintenance.

Message Loop Efficiency: Processes only relevant messages, minimizing unnecessary processing.

Leveraging Sleep Function: Briefly pauses after button presses to simulate a physical button click, enhancing user experience.

Benefits of This Approach:

Reduced Code Complexity and Size: HEXCALC achieves its functionality in under 150 lines of code, showcasing the power of resource files and message-driven programming.

Faster Development: By relying on pre-built components and efficient coding practices, the program can be developed and implemented quickly.

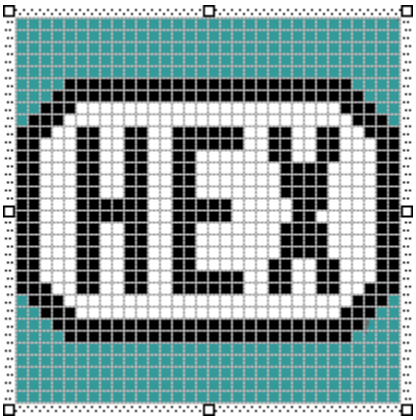
Simplified User Interface: The dialog box interface offers a familiar and user-friendly experience.

Limitations:

Limited Customization: Constrained by the capabilities of dialog boxes, customizability of the interface is limited.

Resource Dependence: Relies heavily on resource files for functionality, requiring additional maintenance alongside the code.

Potentially Less Efficient: May not be as efficient or performant as programs built with more traditional methods.



HEXCALC is a unique program that blurs the lines between traditional windows and modeless dialog boxes. It functions as a standard infix notation calculator for unsigned 32-bit integers, offering basic arithmetic, logical operations, and bit shifts. Key Features:

- Hybrid Window/Dialog Box: HEXCALC utilizes a dialog box template (HEXCALC.DLG) for its interface, yet its functionality is handled by a custom window procedure (WndProc) similar to traditional windows.
- Infix Notation: Users can enter operations and operands in familiar infix notation, similar to a standard calculator.
- 32-Bit Integer Support: Performs calculations on unsigned 32-bit integers, covering a wide range of values.
- Basic Operations: Supports addition, subtraction, multiplication, division, and remainder calculations.
- Bitwise Operations: Offers bitwise AND, OR, and exclusive OR operations for manipulating individual bits.
- Bit Shifts: Allows for left and right shifting of bits within the 32-bit integer.

User Interface and Interaction:

- Mouse and Keyboard Input: Users can interact with the calculator using both mouse clicks on buttons and keyboard input for numbers and operations.
- Display Box: A designated "display" box shows the current entry and the calculated result.
- Clear and Backspace: The "Back" button and keyboard keys like Backspace and Left Arrow allow for correcting mistakes in the entry.
- Result Display: Clicking the "Equals" button or pressing Enter key shows the final result of the calculation.

Behind the Scenes:

- Dialog Box Template: The HEXCALC.DLG file defines the layout and functionality of the calculator interface, including buttons, labels, and text fields.
- Custom Window Procedure: The WndProc function handles all messages sent to the window, including key presses, button clicks, and calculations.
- Message Loop: The program runs in a message loop, receiving and responding to user input and system messages.

Inner working of HexCalc:

HEXCALC is a unique program that combines the characteristics of both traditional windows and modeless dialog boxes. This section delves deeper into its inner workings, highlighting key aspects of its design and implementation.

Dialog Box Template and Window Procedure:

Crucial Difference: While HEXCALC utilizes a dialog box template (HEXCALC.DLG) for its interface, its functionality is managed by a custom window procedure (WndProc) similar to a traditional window.

Overlapping Dialog: The presence of the CLASS statement in the HEXCALC.DLG template distinguishes it from standard dialog boxes. This statement instructs Windows to send messages to the HexCalc window class instead of using its internal window procedure.

Window Class Registration: Similar to a normal window, HEXCALC registers the HexCalc window class in its WinMain function. However, it sets the cbWndExtra field of the WNDCLASS structure to DLGWINDOWEXTRA, which is crucial for handling messages through a custom window procedure.

Dialog Box Creation and Message Handling:

CreateDialog Call: WinMain calls CreateDialog to create the window. This function effectively translates into a CreateWindow call, creating the main window and its 29 child button controls.

Child Window Management: Windows automatically handles creating these child windows and sending WM_COMMAND messages to the WndProc. This simplifies the process of managing a window with multiple child controls.

Minimal Code Size: HEXCALC leverages the ASCII codes of button text as their IDs, eliminating the need for a separate header file with control identifiers. This allows WndProc to handle both WM_COMMAND and WM_CHAR messages efficiently.

Keyboard Interaction and Message Processing:

Left Arrow and Backspace: WndProc intercepts WM_KEYDOWN messages and converts the Left Arrow key to a Backspace key for consistent behavior.

Keyboard Character Conversion: During WM_CHAR message processing, WndProc converts the character code to uppercase and translates the Enter key to the Equals key for intuitive user experience.

Validating Keyboard Input: GetDlgItem ensures the received character corresponds to a button ID defined in the dialog box template. This avoids processing invalid keyboard input.

Button Flashing and User Feedback: WndProc utilizes BM_SETSTATE messages to visually indicate button clicks by flashing them for 100 milliseconds, enhancing user feedback.

Main Window Focus:

Maintaining Focus: Notably, WndProc explicitly sets the input focus back to the main window after handling WM_COMMAND messages. This ensures keyboard input is directed to the main window, preventing accidental focus shifts to buttons.

Conclusion:

HEXCALC serves as a prime example of "lazy programming" done right. By cleverly utilizing dialog boxes and resource files, it achieves a functional and user-friendly application with minimal code, demonstrating the power of efficient coding practices.

While this approach may not be suitable for all applications, it offers a valuable alternative for quick and simple development of small, user-driven programs.

The HEXCALC.DLG file cannot be directly edited through the Visual Studio Dialog Editor. It requires manual editing to include specific functionalities.

The resource script needs to be modified to include the HEXCALC.DLG file using the #include directive.

Understanding the message loop and WndProc function is crucial for modifying or extending the program's functionality.

THE COMMON DIALOG BOXES: A REVOLUTION IN STANDARDIZATION

The standardized user interface was a key objective of Windows from its inception. While achieving uniformity across various software for common menu items like "Alt-File-Open" was rapid, the actual file open dialog boxes remained diverse.

Common Dialog Box Library: A Standardized Solution

Windows 3.1 introduced the "common dialog box library," a revolutionary solution to the problem of inconsistent dialog boxes. This library provides functions that invoke standard dialog boxes for various tasks, including:

- Opening and saving files
- Searching and replacing text
- Choosing colors
- Selecting fonts (demonstrated in this chapter)
- Printing (demonstrated in Chapter 13)

Functionalities and Usage:

- Structure Initialization: Before invoking the dialog box, you initialize the fields of a specific structure relevant to the desired functionality.
- Function Call: Pass a pointer to this structure to a function in the common dialog box library.
- Dialog Box Display: The function creates and displays the standard dialog box for the specified task.
- User Interaction: The user interacts with the dialog box and makes their selections.
- Function Return: When the dialog box closes, the function returns control to your program.
- Information Retrieval: You retrieve information about the user's choices from the structure you passed to the function.

Header File and Reference Documentation:

- Include the COMMDLG.H header file in your C source code to use the common dialog box library.
- Refer to the documentation provided at /Platform SDK/User Interface Services/User Input/Common Dialog Box Library for detailed information on each function and its associated structure.

POPPAD Revisited: Implementing Common Dialog Boxes

Previously in Chapter 10, we added a menu to POPPAD but left several standard menu options unimplemented. Now, we'll enhance POPPAD3 to incorporate functionality for:

- Opening files: Utilize the common dialog box library to open existing text files.
- Saving edited files: Allow users to save their edits back to disk using the common dialog box.
- Selecting fonts: Provide the ability to change the font used for displaying text within POPPAD.
- Searching and replacing text: Implement functionality to search and replace text within the document.

POPPAD3: A FEATURE-RICH TEXT EDITOR

Popadd3 Program
fully implemented n

POPPAD3 is a significantly enhanced version of the text editor explored earlier. It utilizes the common dialog box library and other advanced features to offer a more comprehensive and user-friendly experience. Let's delve deeper into its functionalities:

Enhanced File Management:

Opening Files: POPPAD3 allows users to open existing text files using the common dialog box for choosing files. This provides a familiar and standardized interface for file selection.

Saving Files: Users can save their edited text files using a similar common dialog box, ensuring they have control over the file name and location.

Read and Write Operations: The program utilizes functions like PopFileRead and PopFileWrite to read the contents of opened files and write updated content to saved files, managing data persistence seamlessly.

Search and Replace Functionality:

Find and Find Next: Users can search for specific text within the document using the PopFindFindDlg function, which opens a dedicated Find dialog box. The PopFindNextText function allows for finding the next occurrence of the searched text.

Replace: POPPAD3 offers the ability to replace found text with another string. The PopFindReplaceDlg function opens a Replace dialog box, allowing users to specify the replacement text. The PopFindReplaceText function performs the actual replacement operation.

Font Selection and Customization:

Font Choosing: Users can utilize the PopFontChooseFont function to open the common font dialog box, allowing them to select a different font for displaying the text within the editor.

Font Applying: Once a font is chosen, the PopFontSetFont function applies it to the edit control, instantly reflecting the change in the user interface.

Additional Features:

Improved Edit Control: Compared to the previous version, the edit control now offers functionalities like undo, cut, copy, paste, and select all.

Printing: POPPAD3 integrates with the Windows printing system, allowing users to print the current document using the PopPrntPrintFile function.

Help and About Box: Basic help and about information are accessible through the menu, providing users with additional context and support.

Overall Design Advantages:

Modular Design: POPPAD3 utilizes separate files for specific functionalities like file management, search and replace, and font handling. This promotes modularity and easier maintenance.

Common Dialog Box Integration: Leveraging the common dialog boxes for file selection and font choosing provides a consistent user experience across different applications.

Enhanced User Interface: With more features and improved edit control functionalities, POPPAD3 delivers a more powerful and user-friendly text editing experience.

IN-DEPTH EXPLANATION OF POPFILE.C

This file contains code for managing file operations in POPPAD3, a text editor application. Let's delve deeper into each function:

PopFileInitialize:

This function initializes the OPENFILENAME structure used for common dialog boxes.

It defines various parameters like the owner window (hwnd), instance handle, filter for acceptable file types, maximum file name and title length, default extension, etc.

This initialization ensures consistent behavior across Open and Save dialogs.

PopFileOpenDlg:

This function opens the common Open File dialog box.

It sets the owner window, buffer pointers for file name and title, and flags for the dialog box behavior.

The GetOpenFileName function displays the dialog and retrieves user input for the chosen file.

The function returns TRUE if the user selects a file and FALSE otherwise.

PopFileSaveDlg:

This function opens the common Save File dialog box.

It sets similar parameters to PopFileOpenDlg but with a flag for allowing overwrite prompts.

The GetSaveFileName function displays the dialog and retrieves user input for the desired file name.

The function returns TRUE if the user chooses a name and FALSE otherwise.

PopFileRead:

This function reads the contents of a chosen file and sets the text in the edit control.

It opens the file in read mode, retrieves its size, and allocates memory for the file contents.

The function checks if the file is Unicode based on the signature and performs necessary conversions.

Depending on the edit control type (Unicode or non-Unicode), it converts the text accordingly using WideCharToMultiByte or MultiByteToWideChar functions.

Finally, it sets the text in the edit control using SetWindowText and frees allocated memory.

PopFileWrite:

This function writes the edit control text to a chosen file.

It opens the file in write mode and allocates memory for the text content.

If the edit control is Unicode, it writes a byte order mark (BOM) to the file.

The GetWindowText function retrieves the edit control text, and it is written to the file using WriteFile.

The function checks for successful write operation and frees allocated memory.

Overall Functionality:

These functions provide a comprehensive set of operations for opening, reading, saving, and writing text files in POPPAD3.

They leverage the common dialog box library for user-friendly file selection and offer consistent behavior across different file formats.

The code demonstrates efficient memory management and appropriate conversion between Unicode and non-Unicode formats.

Additional Notes:

The function names and variable names are clear and descriptive, making the code easy to understand.

Error handling is implemented in the functions, ensuring graceful handling of potential issues like file access failures.

By utilizing separate functions for specific tasks, the code maintains modularity and promotes code reuse.

Understanding the code within POPFILE.C provides a deeper understanding of how POPPAD3 manages files and how the common dialog box library is integrated into the application. This knowledge can be valuable for further development and customization of the text editor.

IN-DEPTH EXPLANATION OF POPFIND.C

This file contains code for managing search and replace functionalities in POPPAD3, a text editor application. Let's break down each function:

PopFindFindDlg and PopFindReplaceDlg:

These functions open the common Find and Replace dialog boxes respectively.

They initialize the FINDREPLACE structure with relevant parameters like owner window, flags for behavior, search and replace strings, and length information.

These dialogs allow users to specify search and/or replace text and control options like case sensitivity and whole word matching.

The FindText and ReplaceText functions display the dialogs and handle user interaction.

PopFindFindText:

This function searches for a specific string within the edit control document.

It retrieves the edit control text length and allocates memory to store the text.

The GetWindowText function retrieves the edit control text into the allocated buffer.

The function uses _tcsstr, which works for both Unicode and non-Unicode text, to search for the specified text within the document starting from the provided search offset.

If the text is found, its position and updated search offset are calculated.

The found text is selected within the edit control using SendMessage with EM_SETSEL and the scroll position is adjusted with EM_SCROLLCARET.

The function returns TRUE if the text is found and FALSE otherwise.

PopFindNextText:

This function simplifies the search process by re-using the previously entered search string.

It calls PopFindFindText with the current search offset, allowing for iterative searches through the document.

PopFindReplaceText:

This function replaces the found text with a specified replacement string.

It first calls PopFindFindText to ensure the search string is still valid and its location is determined.

If the text is found, the EM_REPLACESEL message is sent to the edit control, replacing the selected found text with the provided replacement string.

This function returns TRUE if the replacement is successful and FALSE otherwise.

PopFindValidFind:

This function checks if a valid search string has been entered by verifying if the szFindText variable is not empty.

Overall Functionality:

These functions provide a user-friendly and efficient way to search for and replace text within the POPPAD3 text editor.

They leverage the common Find and Replace dialogs for user input and utilize appropriate functions based on the desired action (find or replace).

The code efficiently manages memory allocation and ensures proper text selection and manipulation within the edit control.

Additional Notes:

The use of the static FINDREPLACE structure ensures consistent data across calls to the Find and Replace dialogs.

The functions handle basic error checking and provide appropriate feedback for unsuccessful search or replace operations.

The code demonstrates modularity by separating functionalities into dedicated functions, promoting code reuse and maintainability.

IN-DEPTH EXPLANATION OF POPFONT.C

This file contains code for managing font-related functionalities in POPPAD3, a text editor application. Let's delve deeper into each function:

PopFontChooseFont:

This function opens the common Font dialog box, allowing users to choose a new font for displaying the text in the editor.

It initializes the CHOOSEFONT structure with relevant parameters like owner window, logfont structure for storing font information, and flags for behavior (initial font, screen fonts, and special effects).

The ChooseFont function displays the dialog box and allows user interaction for selecting a font.

The function returns TRUE if a font is chosen and FALSE otherwise.

PopFontInitialize:

This function initializes the font used by the edit control.

It retrieves the system font information using GetStockObject and GetObject functions.

A new font is created based on the retrieved information using CreateFontIndirect.

The SendMessage function sets the new font for the edit control using WM_SETFONT message.

PopFontSetFont:

This function sets the chosen font for the edit control.

It creates a new font object based on the updated logfont structure.

The newly created font is set for the edit control using WM_SETFONT, replacing the previously used font.

The old font is deleted using DeleteObject to release resources.

Finally, the edit control's client area is invalidated and redrawn using InvalidateRect to reflect the font change.

PopFontDeinitialize:

This function releases resources associated with the font used by the edit control.

It calls DeleteObject on the font handle to free memory allocated for the font object.

Overall Functionality:

These functions provide a user-friendly way for users to select and apply desired fonts for displaying text in POPPAD3.

They leverage the common Font dialog box for intuitive interaction and manage font creation and deletion efficiently.

The code ensures the edit control updates its visual appearance based on the chosen font.

Additional Notes:

The use of the static logfont structure allows for storing and manipulating font information throughout the functions.

The code handles memory allocation and release properly, preventing resource leaks.

The separation of responsibilities into distinct functions promotes code clarity and maintainability.

IN-DEPTH ANALYSIS OF POPPAD.RC AND RESOURCE.H (EXCERPTS)

These two files define the resources used by the POPPAD application, including the dialog boxes, menus, and icons. Let's break them down:

POPPAD.RC:

About Box: This dialog displays basic information about the application, including its name, author, and copyright information.

Print Dialog: This dialog allows users to configure and initiate the printing process for the current document.

Menus: These define the different menu options available to users, categorized by functionality (File, Edit, Search, Format, Help). Each menu item has a unique identifier and associated keyboard shortcut.

Accelerators: These specify keyboard shortcuts for performing common actions, such as undo, copy, paste, and opening the help menu.

Icon: This defines the application's icon, displayed in the title bar and taskbar.

RESOURCE.H:

Resource Identifiers: This file defines symbolic names for all resources used in the application. This improves code readability and maintainability by using descriptive names instead of hard-coded numeric values.

Menu Item IDs: These identifiers correspond to each menu item in the application. They are used by the code to handle user interaction with the menus.

Accelerator IDs: These identifiers correspond to each keyboard shortcut defined in the accelerator table. They are used by the code to handle keyboard input and trigger specific actions.

Overall Functionality:

These resources provide a user-friendly interface for interacting with POPPAD and performing various operations.

They offer clear menus and keyboard shortcuts for common tasks, enhancing user experience and efficiency.

The separation of resources in dedicated files facilitates code organization and simplifies maintenance.

Additional Notes:

The use of resource identifiers promotes modularity and allows for easier customization of the application's interface.

The inclusion of an about box and help menu provides users with valuable information and support.

The provided excerpts only offer a glimpse into the complete resource files. Understanding the complete set of resources is crucial for fully comprehending the application's functionality and user interface.

IN-DEPTH EXPLANATION OF POPFILE.C

This section delves deeper into the functionalities and implementation details of POPFILE.C, focusing on file open and save dialogs.

File Open and Save Dialogs:

POPFILE.C utilizes the common dialog boxes provided by Windows API for file open and save operations.

- It uses the OPENFILENAME structure to define and manage parameters for these dialog boxes.
- The structure is initialized in PopFileInitialize and reused across subsequent calls.
- PopFileOpenDlg opens the File Open dialog box, allowing users to select a file to open.
- It sets relevant OPENFILENAME parameters like owner window, buffer pointers for file and title, filter for acceptable file types, and flags for behavior.
- The GetOpenFileName function displays the dialog box and retrieves user input for the chosen file.
- The function returns TRUE if the user selects a file and FALSE otherwise.
- PopFileSaveDlg opens the File Save dialog box, allowing users to save the current document.
- It sets similar parameters to PopFileOpenDlg but with an additional flag for prompting overwrite confirmation.
- The GetSaveFileName function displays the dialog box and retrieves user input for the desired file name.
- The function returns TRUE if the user chooses a name and FALSE otherwise.

OPENFILENAME Structure:

This structure is defined in COMMDLG.H and holds various parameters for configuring the common dialog boxes.

- Some key fields used in POPFILE.C include:
- lStructSize: Size of the structure.
- hwndOwner: Owner window handle of the dialog box.
- lpstrFilter: Pointer to a filter string specifying acceptable file types.
- lpstrFile: Pointer to a buffer to receive the chosen file path.
- nMaxFile: Size of the buffer for the file path.
- lpstrFileTitle: Pointer to a buffer to receive the file name without path.
- nMaxFileTitle: Size of the buffer for the file title.
- Flags: Flags for controlling dialog box behavior (e.g., create file prompt, overwrite prompt).
- lpstrDefExt: Default file extension to be used if not specified by the user.

Filter and Combo Box:

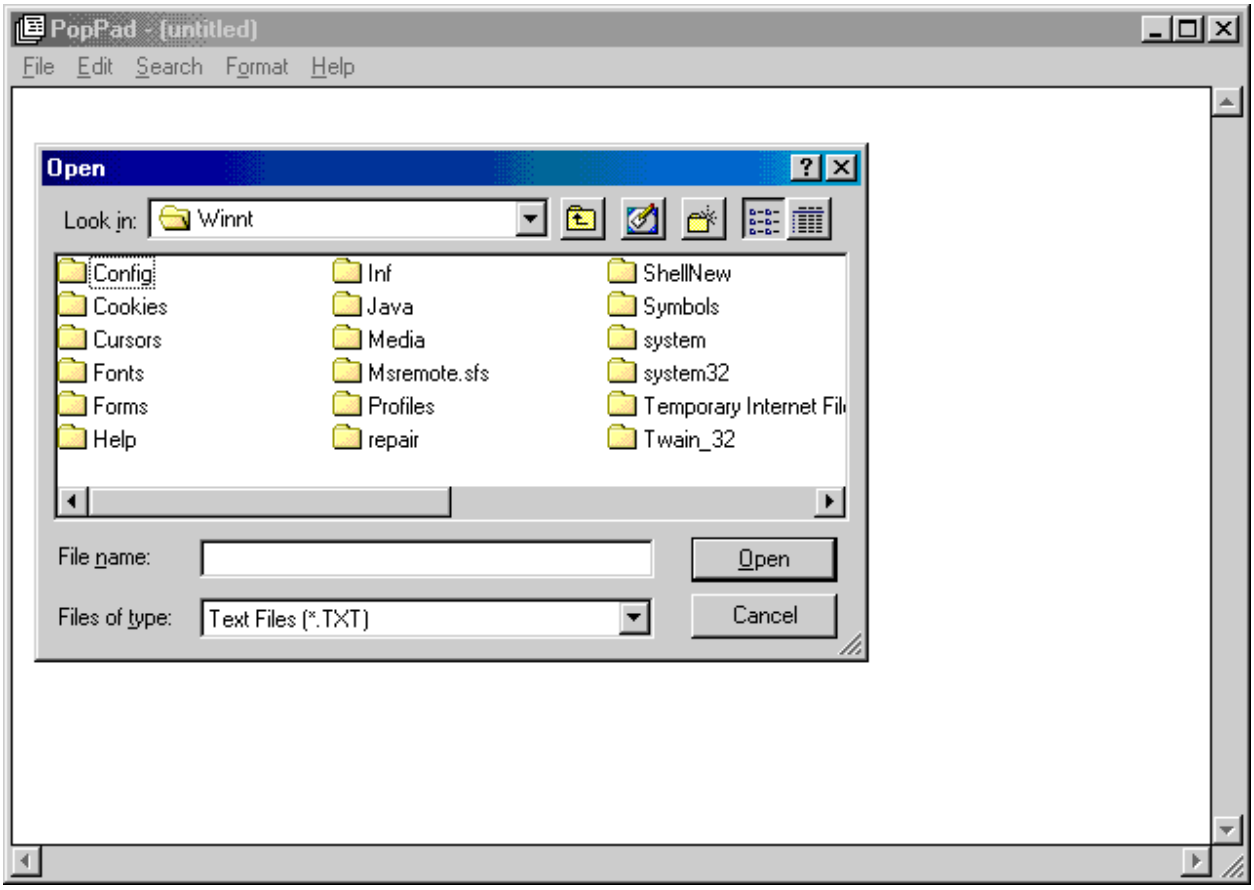
- POPFILE.C defines a filter for text and ASCII files using the szFilter variable.
- This filter is displayed in the combo box of the dialog boxes, allowing users to choose specific file types.
- The nFilterIndex field of the OPENFILENAME structure reflects the user's selected filter.
- This enables persistent filter selection across multiple calls to the dialog boxes.

Key Points:

- POPFILE.C leverages the common dialog box library for a user-friendly file open and save experience.
- The code utilizes the OPENFILENAME structure to configure and manage dialog box parameters.
- The filter allows users to choose specific file types based on their needs.
- The code efficiently handles buffer allocation and retrieval of user-selected information.

Additional Notes:

- The use of a static OPENFILENAME structure enables efficient reuse of information across dialog calls.
- The code utilizes appropriate flags to handle potential file creation and overwrite situations.
- Understanding the details of OPENFILENAME and its fields provides valuable insights into customizing the file selection experience.



IN-DEPTH EXPLANATION OF UNICODE FILE I/O IN POPPAD3

This section focuses on how the Unicode version of POPPAD3 manages file I/O to ensure compatibility with both Unicode and non-Unicode environments.

Problem:

The Unicode version of POPPAD3 works internally with Unicode strings.

Saving files in Unicode format can lead to compatibility issues if opened by the non-Unicode version.

Reading non-Unicode files requires conversion for proper display in the Unicode version.

Solutions:

Byte Order Mark:

- The PopFileWrite function in POPFILE.C writes the code 0xFEFF at the beginning of saved files.
- This serves as a "Byte Order Mark" identifying the file as containing Unicode text.

Detection and Conversion:

- The PopFileRead function uses the IsTextUnicode function to check for the byte order mark in the file.
- It also handles reversed byte order scenarios encountered with files created on non-Intel platforms.

If the file is Unicode:

- For the non-Unicode version, the WideCharToMultiChar function converts the text to ANSI before displaying.

If the file is non-Unicode:

- For the Unicode version, the MultiCharToWideChar function converts the text to Unicode for internal processing.

Impact and Benefits:

- This approach ensures compatibility between the Unicode and non-Unicode versions of POPPAD3 when dealing with files.
- The byte order mark provides a reliable indicator of the file's encoding.
- Conversion functions handle displaying text correctly based on the program's version and file encoding.
- This promotes user experience by preventing unexpected results when working with diverse file origins.

Additional Notes:

- Understanding the concept of byte order marks and conversion functions is crucial for dealing with Unicode data across different platforms and applications.

IN-DEPTH EXPLANATION OF FONT MANAGEMENT IN POPPAD3

This section explores how POPPAD3 implements font selection and application within the program.

Font Initialization:

During WM_CREATE message processing, POPPAD3 calls PopFontInitialize in POPFONT.C.

This function:

- Obtains the system font information using GetStockObject and GetObject functions.
- Creates a new font object based on the retrieved information using CreateFontIndirect.
- Sets the newly created font as the edit control's font using SendMessage with WM_SETFONT.
- Creating a dedicated font object for the edit control ensures proper resource management and avoids modifying the system font directly.

Font Selection:

When the user triggers the "Font..." option, POPPAD3 calls PopFontChooseFont.

This function:

- Initializes a CHOOSEFONT structure with relevant parameters like owner window, logfont for storing font information, and flags for behavior (initial font, screen fonts, and special effects).
- Displays the common Font dialog box using ChooseFont, allowing users to select a new font.
- If the user confirms the selection, ChooseFont returns TRUE.

Font Application:

If a font is chosen, POPPAD3 calls PopFontSetFont. This function:

- Creates a new font object based on the updated logfont structure reflecting the user's selection.
- Sets the newly created font for the edit control using WM_SETFONT, replacing the previously used font.
- Deletes the old font object using DeleteObject to release resources.

This ensures the edit control updates its visual appearance with the selected font.

Font Deinitialization:

- During WM_DESTROY message processing, POPPAD3 calls PopFontDeinitialize to release resources associated with the edit control's font.
- This function calls DeleteObject on the font handle to free memory allocated for the font object.

Overall Functionality:

- These functions provide a user-friendly way for users to choose and apply desired fonts for displaying text in POPPAD3.
- They utilize the common Font dialog box for intuitive interaction and manage font creation and deletion efficiently.
- The code ensures the edit control updates its visual appearance based on the chosen font.

IN-DEPTH EXPLANATION OF SEARCH AND REPLACE FUNCTIONALITY IN POPPAD3

This section dives deeper into the implementation of search and replace features in POPPAD3, focusing on the functionalities and considerations involved.

Dialog Boxes and FINDREPLACE Structure:

POPPAD3 uses the common dialog box library to display search and replace dialog boxes.

These dialogs are modeless, requiring special handling in the message loop.

The FINDREPLACE structure holds information related to the search/replace operation, including:

- Search string
- Replacement string
- Search options (e.g., case sensitivity, whole word)

POPFIND.C contains functions (PopFindFindDlg and PopFindReplaceDlg) to interact with these dialog boxes and manage the FINDREPLACE structure.

Message Loop and Communication:

While the search/replace dialogs are active, POPPAD3's message loop needs to call IsDialogMessage to process messages relevant to the dialogs.

The FINDREPLACE structure is declared as a static variable to ensure its accessibility throughout the dialog box procedure and search/replace operations.

A special message is used for communication between the dialog box and the main window. This message number is obtained using RegisterWindowMessage with the FINDMSGSTRING parameter.

WndProc checks for this message and interprets the Flags field of the FINDREPLACE structure to determine the user's action (find, replace, or cancel).

Search and Replace Functions:

POPPAD3 calls PopFindFindText and PopFindReplaceText from POPFIND.C to perform actual search and replace operations within the edit control.

These functions receive the FINDREPLACE structure populated by the dialog box and use its information to locate and modify text in the edit control.

They consider the specified search options (e.g., case sensitivity, whole word) when performing the search.

Considerations:

Modeless dialog boxes necessitate special handling in the message loop.

The static FINDREPLACE structure ensures consistent data access throughout the process.

Understanding the communication message and its interpretation is crucial for coordinating actions between the dialog box and the main window.

IN-DEPTH EXPLANATION OF COLORS3 PROGRAM WITH ADDITIONAL INSIGHTS

This section expands upon the previous explanation of COLORS3, delving deeper into technical details and providing additional insights into the code and its implications.

```
#include <windows.h>
#include <commdlg.h>

int WINAPI WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    PSTR szCmdLine,
    int iCmdShow)
{
    // Initialize CHOOSECOLOR structure
    CHOOSECOLOR cc = {};
    COLORREF crCustColors[16] = {};

    cc.lStructSize = sizeof(CHOOSECOLOR);
    cc.hwndOwner = NULL;
    cc.hInstance = NULL;
    cc.rgbResult = RGB(0x80, 0x80, 0x80); // Initial color
    cc.lpCustColors = crCustColors;        // Array for custom colors
    cc.Flags = CC_RGBINIT | CC_FULLOPEN;    // Initialize color & full functionality
    cc.lCustData = 0;                       // Custom data (optional)
    cc.lpfnHook = NULL;                     // Hook procedure (optional)
    cc.lpTemplateName = NULL;               // Custom template name (optional)

    // Display color dialog box and retrieve chosen color
    if (ChooseColor(&cc)) {
        // Color selection successful - use cc.rgbResult
    } else {
        // Color selection failed - handle error
    }

    return 0;
}
```

CHOOSECOLOR Structure Breakdown:

lStructSize: Ensuring proper structure size is crucial for memory allocation and function compatibility.

hwndOwner: Setting it to NULL allows the Color dialog box to function independently, similar to a standalone window. This avoids dependence on another window for its existence and behavior.

hInstance: Passing the program's instance handle enables the dialog box to access resources specific to the program.

rgbResult: This field holds both the initial and final color values. Setting it to a default color provides a starting point for the user's selection. After selection, the field stores the chosen color for retrieval.

lpCustColors: This pointer allows the program to provide an array of 16 custom colors for user selection, further enriching the color palette beyond the default options offered by the dialog box itself.

Flags: Setting the appropriate flags controls the behavior of the dialog box. CC_RGBINIT ensures the initial color is displayed, and CC_FULLOPEN enables full functionality, including custom color selection and editing.

lCustData: This field allows storing custom data associated with the dialog box, enabling flexibility for specific program needs.

lpfnHook: Setting a hook procedure allows for intercepting and handling specific events within the dialog box, enabling advanced interaction and customization.

lpTemplateName: Providing a resource template name enables customizing the appearance and layout of the dialog box beyond the standard Windows default.

Program Execution Flow:

Structure Initialization: The program meticulously initializes the CHOOSECOLOR structure with relevant information, ensuring proper communication and functionality.

ChooseColor Function Call: The program relies solely on this single function call to interact with the user and retrieve the selected color. This demonstrates the power and efficiency of the common dialog box library.

Dialog Box Interaction: The ChooseColor function displays the Color selection dialog box, allowing the user to interact with it and choose their desired color.

Color Selection and Retrieval: If the user selects a color, the rgbResult field of the structure is updated to reflect the chosen value. This information is then accessible to the program after the function call.

Program Execution: The program retrieves the return value of the ChooseColor function and can use it to determine the success or failure of the user interaction.

Benefits and Implications:

- COLORS3 showcases a minimalistic approach to program design, achieving color selection with just one function call.
- This approach promotes code conciseness and reduces development complexity.
- Understanding the CHOOSECOLOR structure and its functionality enables developers to leverage the common dialog box library effectively for various user interaction tasks.
- The ability to set the owner window to NULL opens up possibilities for creating independent dialog boxes that function autonomously and can even be utilized across different programs.
- Exploring the available flags and customization options for the ChooseColor function provides opportunities for tailoring the dialog box to specific user needs and program requirements.

End of chapter 11…