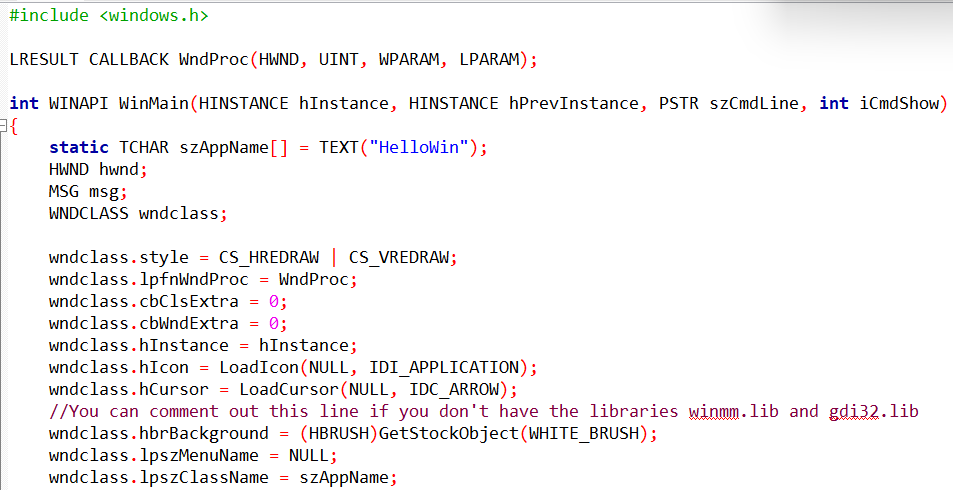
HELLOWIN.C IN DEPTH

**Defining and Initializing the WNDCLASS Structure**

The WNDCLASS structure defines the characteristics of a window class, which is a template for creating windows.

The WNDCLASS structure contains ten fields that specify various aspects of the window's appearance and behavior.

In the provided code, the WNDCLASS structure is declared and initialized as follows:



This initialization sets the following values for the WNDCLASS structure's fields:

**style:** This field specifies the window's style flags. The CS\_HREDRAW and CS\_VREDRAW flags indicate that the window should be completely repainted whenever the horizontal or vertical window size changes, respectively.

**lpfnWndProc:** This field is a pointer to the window procedure function. The window procedure function is responsible for handling all window messages that are sent to windows of this class. In this case, the window procedure function is WndProc.

**cbClsExtra:** This field specifies the size of extra data that is associated with each window of this class. In this case, the value is set to 0, indicating that there is no extra data.

**cbWndExtra:** This field specifies the size of extra data that is associated with each individual window of this class. In this case, the value is set to 0, indicating that there is no extra data.

**hInstance:** This field is the handle of the application instance. The application instance is a unique identifier for the application.

**hIcon:** This field is the handle of the icon that will be used for the application. The icon is a small image that is displayed in the title bar of the window and in the taskbar.

**hCursor:** This field is the handle of the cursor that will be used for the application. The cursor is a small image that follows the mouse pointer.

**hbrBackground:** This field is the handle of the brush that will be used to paint the background of the window. The brush is a tool used to fill in an area with a solid color or pattern.

**lpszMenuName:** This field is the name of the menu resource that will be used for the application. The menu is a list of options that the user can select to control the application. In this case, the value is set to NULL, indicating that no menu will be used.

**lpszClassName:** This field is the name of the window class. The window class is a template for creating windows. The name of the window class is used to identify the class of a window when it is created.

**Registering the Window Class**

After initializing the WNDCLASS structure, it is registered with the Windows operating system using the RegisterClass function. The RegisterClass function takes a pointer to the WNDCLASS structure as its argument. Once the window class is registered, it can be used to create windows.

**Window Procedure**

The window procedure is a function that is responsible for handling all window messages that are sent to windows of a particular class. In this case, the window procedure is WndProc. The WndProc function is responsible for responding to the message in a way that is appropriate for the application. For example, if the message is a WM\_CREATE message, the WndProc function would create any child windows that the application needs.

**Setting the Window Procedure**

The second field of the WNDCLASS structure, lpfnWndProc, is set to the address of the window procedure function, WndProc.

This means that all windows created based on this window class will use the WndProc function to process messages.

The WndProc function is responsible for handling all messages sent to the window, such as a WM\_CREATE message when the window is created, a WM\_PAINT message when the window needs to be repainted, or a WM\_DESTROY message when the window is destroyed.

**Reserving Extra Space**

The next two fields of the WNDCLASS structure, cbClsExtra and cbWndExtra, are used to reserve extra space in the class structure and the window structure, respectively.

This extra space can be used by the application for its own purposes. The cbClsExtra field specifies the size of the extra space in the class structure, and the cbWndExtra field specifies the size of the extra space in the window structure.

In this case, both fields are set to 0, indicating that no extra space is needed.

**Setting the Instance Handle**

The next field of the WNDCLASS structure, hInstance, is set to the instance handle of the program. The instance handle is a unique identifier for the application instance.

The application instance is the running instance of the application, including all of its data and resources. The hInstance value is passed to the application's WinMain function when the application is started.

**Setting the Icon**

The statement wndclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION); sets the icon for all windows created based on this window class.

The icon is a small bitmap image that represents the program to the user.

When the program is running, the icon appears in the Windows taskbar and at the left side of the program window's title bar.

The LoadIcon function is used to load a predefined icon.

The first argument to the LoadIcon function is set to NULL, indicating that a predefined icon is being loaded.

The second argument to the LoadIcon function is IDI\_APPLICATION, which is the identifier for the predefined icon that is a little picture of a window.

**Loading and Setting the Cursor**

The statement wndclass.hCursor = LoadCursor(NULL, IDC\_ARROW); loads a predefined mouse cursor known as IDC\_ARROW and assigns its handle to the hCursor field of the WNDCLASS structure.

This means that when the mouse cursor appears over the client area of a window created based on this class, the cursor will become a small arrow.

**Setting the Background Color**

The statement wndclass.hbrBackground = GetStockObject(WHITE\_BRUSH); sets the background color of the client area of windows created based on this class.

The GetStockObject function returns a handle to a white brush, which means that the background of the client area of the window will be solid white.

**Specifying the Window Class Menu**

The statement wndclass.lpszMenuName = NULL; indicates that the window class has no application menu. This means that the window will not have a menu bar at the top of the window.

**Setting the Window Class Name**

The statement wndclass.lpszClassName = szAppName; sets the name of the window class to the value stored in the szAppName variable.

This string can be either ASCII or Unicode characters depending on whether the UNICODE identifier has been defined.

**Registering the Window Class**

When all the fields of the WNDCLASS structure have been initialized, the RegisterClass function is called to register the window class.

The only argument to the function is a pointer to the WNDCLASS structure.

There are actually three versions of the RegisterClass function: RegisterClassA, RegisterClassW, and RegisterClassEx.

The specific function used determines whether messages sent to the window will contain ASCII text or Unicode text.

**Handling Unicode Compatibility Issues**

If you compile the program with the UNICODE identifier defined, your program will call RegisterClassW instead of RegisterClassA.

This is fine if you're running the program on Microsoft Windows NT, which has full support for Unicode.

However, if you're running the program on Windows 98, the RegisterClassW function is not fully implemented.

While there is an entry point for the function, it simply returns an error code indicating that the function is not available.

To handle this compatibility issue, the provided code snippet checks the return value of the RegisterClass function. If the function fails, it displays a message box informing the user that the program requires Windows NT and terminates the program.

The MessageBoxW function is used for this purpose because it is one of the few Unicode functions implemented in Windows 98.

The code snippet assumes that RegisterClass is not failing for any other reason, such as an invalid lpfnWndProc field in the WNDCLASS structure.

In such cases, you can use the GetLastError function to determine the exact cause of the error. GetLastError is a general-purpose function in Windows that provides extended error information when a function call fails.

The documentation for individual functions will indicate whether they support error retrieval using GetLastError. In the case of calling RegisterClassW in Windows 98, GetLastError returns 120, which corresponds to the ERROR\_CALL\_NOT\_IMPLEMENTED identifier defined in WINERROR.H.

While some Windows programmers advocate for checking the return value of every function call for errors, this can become tedious and unnecessarily complex.

For instance, checking for errors when allocating memory is crucial, as many Windows functions rely on memory allocation.

However, if a function like RegisterClass fails due to memory allocation issues, the system is likely already in a critical state.

For the sample programs in this book, the author minimizes error checking to focus on illustrating the main concepts and avoid distracting from the core learning objectives.

This doesn't imply that error checking is unimportant; it's an essential practice in real-world software development.

**Historical Context of hPrevInstance Check**

In some older Windows sample programs, you might encounter code that checks the value of hPrevInstance before initializing the window class and registering it.

This practice is rooted in the behavior of 16-bit versions of Windows.

In 16-bit Windows, when you launched a new instance of a program that was already running, the hPrevInstance parameter passed to the WinMain function would contain the instance handle of the existing instance.

This allowed multiple instances of the same program to share the same window class, which was a memory-saving technique.

Therefore, the window class was only registered if hPrevInstance was NULL, indicating that no other instances of the program were running.

With the advent of 32-bit Windows, the behavior of hPrevInstance changed.

In 32-bit versions of Windows, hPrevInstance is always NULL, regardless of whether another instance of the program is running or not.

This means that the code snippet mentioned earlier, which checks for hPrevInstance to be NULL before registering the window class, is no longer necessary.

While the code snippet will still work properly in 32-bit Windows, it's an outdated practice that doesn't reflect the current behavior of the hPrevInstance parameter.

It's recommended to avoid this unnecessary check and directly register the window class without checking for hPrevInstance.

**Distinction between Window Class and Window**

A window class defines the general characteristics of a type of window, such as its default behavior, appearance, and functionality.

It serves as a template for creating multiple windows with similar attributes.

When you create a specific window using the CreateWindow function, you provide more detailed information about the window's placement, size, and behavior.

The distinction between the window class and the window lies in the level of abstraction.

The window class represents a general category of windows, while the window represents a specific instance of that class.

This separation allows for efficient memory management and code reuse.

For example, all push buttons in Windows are created based on the same window class, which encapsulates the common behavior and appearance of push buttons.

This window class handles keyboard and mouse input, defines the button's appearance, and ensures that all push buttons behave consistently.

However, individual push buttons can have different sizes, locations, and text labels, which are specified when the button is created.

**CreateWindow Function and Its Arguments**

The CreateWindow function creates a new window based on a specified window class and additional parameters.

It takes several arguments, each providing specific information about the window:

**szAppName:** The name of the window class, which determines the window's general characteristics.

**TEXT("The Hello Program"):** The text that appears in the title bar of the window.

**WS\_OVERLAPPEDWINDOW:** The window style, which defines the window's appearance and behavior, such as its border, title bar, and minimize/maximize buttons.

**CW\_USEDEFAULT:** The initial x-position of the window. CW\_USEDEFAULT indicates that Windows should automatically position the window on the screen.

**CW\_USEDEFAULT:** The initial y-position of the window. CW\_USEDEFAULT indicates that Windows should automatically position the window on the screen.

**CW\_USEDEFAULT:** The initial x-size of the window. CW\_USEDEFAULT indicates that Windows should use the default width for the window class.

**CW\_USEDEFAULT:** The initial y-size of the window. CW\_USEDEFAULT indicates that Windows should use the default height for the window class.

**NULL:** The handle of the parent window. If not specified, the window has no parent window.

**NULL:** The handle of the window menu. If not specified, the window has no menu.

**hInstance:** The instance handle of the program. This is the same handle passed to the WinMain function when the program starts.

**NULL:** Creation parameters. These parameters are typically used for advanced window creation options.

**Code Explanation**

The CreateWindow call in HELLOWIN.C creates a new window with the following characteristics:

* The window class is named szAppName.
* The window caption is "The Hello Program".
* The window style is WS\_OVERLAPPEDWINDOW, which means it is an overlapping window with a title bar, border, minimize/maximize buttons, and a system menu.
* The initial x-position and y-position are set to CW\_USEDEFAULT, indicating that Windows should automatically position the window on the screen.
* The initial x-size and y-size are also set to CW\_USEDEFAULT, indicating that Windows should use the default width and height for the window class.
* The parent window handle is set to NULL, indicating that the window has no parent window.
* The window menu handle is set to NULL, indicating that the window has no menu.
* The program instance handle is set to hInstance, which is the same handle passed to the WinMain function.
* There are no creation parameters specified.

**Creating the Window with CreateWindow**

The CreateWindow function is responsible for creating a new window based on the specified window class and additional parameters.

It takes several arguments, each providing crucial information about the window's characteristics:

**Window Class Name:** The first argument, szAppName, represents the name of the window class, which identifies the type of window being created. In this case, the window class name is "HelloWin", which corresponds to the class registered earlier using the RegisterClassEx function.

**Window Caption:** The second argument, L"Hello, World!", specifies the text that will appear in the window's title bar. This title serves as a label or identifier for the window.

**Window Style:** The third argument, WS\_OVERLAPPEDWINDOW, defines the window's style, determining its overall appearance and behavior. This style encompasses standard elements like a title bar, system menu, border, and minimize/maximize/close buttons.

**Initial X Position:** The fourth argument, CW\_USEDEFAULT, indicates the initial horizontal position of the window's top-left corner relative to the screen's top-left corner. Using CW\_USEDEFAULT instructs Windows to automatically position the window on the screen.

**Initial Y Position:** The fifth argument, CW\_USEDEFAULT, similarly specifies the initial vertical position of the window's top-left corner relative to the screen's top-left corner. Using CW\_USEDEFAULT allows Windows to automatically determine the window's placement.

**Initial X Size:** The sixth argument, 240, defines the initial width of the window in pixels. This value sets the horizontal dimension of the window when it is first displayed.

**Initial Y Size:** The seventh argument, 120, specifies the initial height of the window in pixels. This value determines the vertical dimension of the window when it is first created.

**Parent Window Handle:** The eighth argument, NULL, indicates whether the window has a parent-child relationship with another window. In this case, NULL indicates that the window is a top-level window, meaning it doesn't have a parent window.

**Window Menu Handle:** The ninth argument, NULL, specifies whether the window has a menu. In this case, NULL indicates that the window has no menu.

**Program Instance Handle:** The tenth argument, hInstance, represents the instance handle of the program. This handle is passed to the WinMain function when the program starts and uniquely identifies the running instance of the application.

**Creation Parameters:** The eleventh argument, NULL, is used for passing additional creation parameters that might be specific to certain window types or advanced window creation scenarios. In this case, NULL indicates that there are no additional creation parameters.

**Return Value and Window Handle**

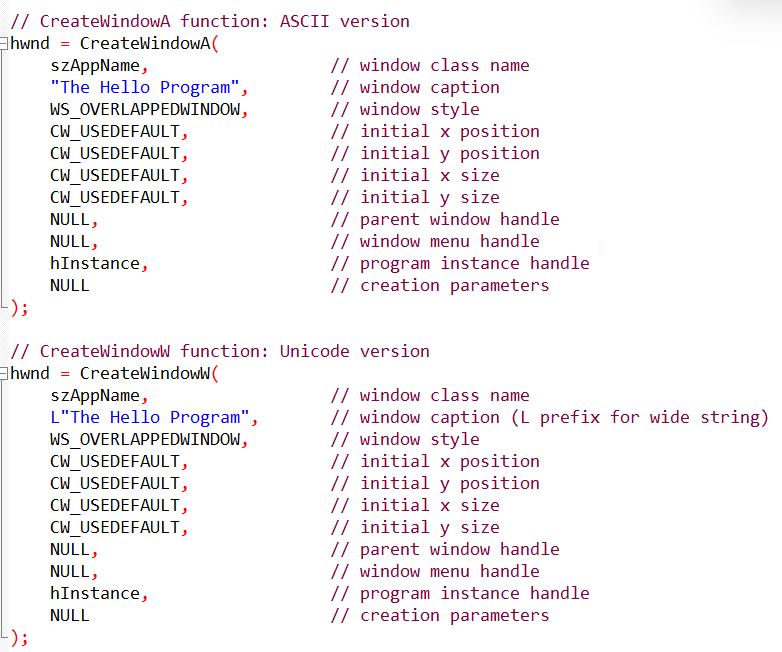
The CreateWindow function returns a handle to the newly created window.

This handle, stored in the variable hwnd, serves as a unique identifier for the window within the Windows operating system.

Every window in Windows has a handle, and programs use this handle to reference and interact with the window.

Many Windows functions require the window handle as an argument to identify the specific window they should operate on.

In summary, the CreateWindow function plays a crucial role in the window creation process, allowing programs to specify the window's class, appearance, behavior, and placement within the Windows environment.



**Explanation:**

The CreateWindowA function is used for the ASCII version, where string literals are represented in the standard character set.

The CreateWindowW function is used for the Unicode version, where string literals are represented in the wide character set. The 'L' prefix before the string indicates that it's a wide string.

The first parameter of both functions is the window class name, which associates the created window with a particular window class. In this case, it's szAppName.

The second parameter is the window caption, providing the text that appears in the title bar of the window.

The third parameter is the window style, specified here as WS\_OVERLAPPEDWINDOW. This style includes various flags (captured in the comment) that define the appearance and behavior of the window, such as having a title bar, system menu, sizing border, and minimize/maximize buttons.

The subsequent parameters define the initial position and size of the window, the parent window handle (NULL for a top-level window), the window menu handle, the program instance handle (hInstance), and additional creation parameters (NULL in this case).

**Window Caption, Initial Position, and Size**

The "window caption" refers to the text that appears in the title bar of the window.

This text serves as a label or identifier for the window, providing context for the user about the purpose or content of the window.

The "initial x position" and "initial y position" arguments specify the initial coordinates of the window's top-left corner relative to the top-left corner of the screen.

These values determine where the window will initially appear on the user's desktop.

By using the identifier CW\_USEDEFAULT for these parameters, the program indicates that it wants Windows to automatically position the window on the screen.

This means that Windows will determine the appropriate placement for the window based on the available space and existing windows.

By default, Windows positions newly created windows with a stepped offset from the upper left corner of the display. This ensures that subsequent windows don't overlap with each other.

Similarly, the **"initial x size"** and **"initial y size"** arguments specify the initial width and height of the window, respectively.

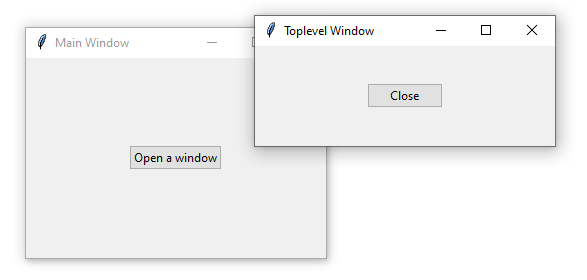
These values determine the initial dimensions of the window when it is first displayed.

Again, using CW\_USEDEFAULT for these arguments instructs Windows to use a default size for the window. This means that the window will initially appear with a size appropriate for the content it will display and the screen resolution.

**Parent Window Handle, Window Menu Handle, and Program Instance Handle**

The "parent window handle" is set to NULL when creating a "top-level" window, such as an application window.

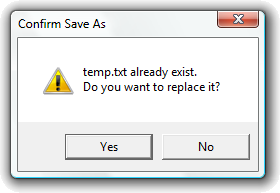
Top-level windows are not children of any other window and exist independently on the desktop.



In contrast, child windows have a parent-child relationship with another window.

When a parent-child relationship exists, the child window is always displayed on the surface of its parent window.

For example, a dialog box might be a child window of an application window.



The "window menu handle" is also set to NULL in this case because the window does not have a menu.

A menu provides options for user interaction, such as file operations, editing tools, or program settings. If a window doesn't require user interaction through a menu, it can be omitted.

The "program instance handle" is set to the instance handle passed to the program as a parameter of WinMain.

This handle is a unique identifier for the running instance of the application. It is used internally by Windows to distinguish between multiple instances of the same program.

**Creation Parameters**

Finally, the "creation parameters" pointer is set to NULL.

Creation parameters provide a way to pass additional data to the CreateWindow function that might be specific to certain window types or advanced window creation scenarios.

In this case, there are no additional parameters needed, so NULL is used.

**Handle to the Created Window**

The CreateWindow call returns a handle to the newly created window.

This handle is a unique identifier that is used by the program to refer to the window.

Many Windows functions require the window handle as an argument to identify the specific window they should operate on.

The handle is stored in the variable hwnd, which is defined to be of type HWND ("handle to a window").

Every window in Windows has a handle, and programs use these handles to interact with and manage the windows they create.

**Making the Window Visible**

After successfully creating the window using the CreateWindow function, the window exists internally within Windows but is not yet visible on the screen.

To make the window visible, two additional function calls are required:

ShowWindow(hwnd, iCmdShow);

This function brings the specified window to the forefront and displays it on the screen according to the provided iCmdShow parameter.

The iCmdShow value determines how the window should initially appear, whether it's in a normal, minimized, or maximized state.

This value is typically passed to WinMain and reflects the user's preference for window display settings.

If iCmdShow is **SW\_SHOWNORMAL**, the window is displayed in its normal state, with the client area erased using the background brush specified in the window class.

If iCmdShow is **SW\_SHOWMAXIMIZED**, the window is maximized, taking up the entire screen.

If iCmdShow is **SW\_SHOWMINNOACTIVE**, the window is minimized and displayed only in the taskbar.

**UpdateWindow(hwnd);**

This function causes the client area of the window to be repainted, ensuring that the window's contents are displayed correctly.

It achieves this by sending a WM\_PAINT message to the window procedure, which is the WndProc function defined in the program's source code.

The WndProc function will handle the WM\_PAINT message and update the window's contents accordingly.

**Summary**

The ShowWindow and UpdateWindow functions are crucial steps in making the newly created window visible to the user. They ensure that the window appears on the screen in the desired state and that its contents are correctly displayed.

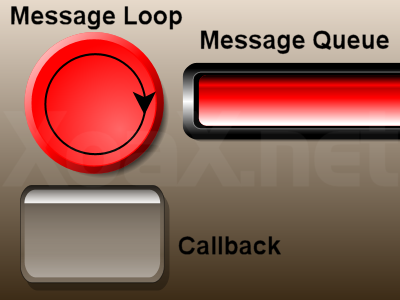
**Processing User Input via the Message Loop**

Once the window is visible using the UpdateWindow function, the program needs to establish a mechanism to handle user input, such as keyboard presses and mouse clicks. Windows provides a message-driven architecture for this purpose.

**Message Queue and Message Structure**

Windows maintains a separate message queue for each running program.

When a user interacts with the program, such as pressing a key or clicking the mouse, Windows generates a corresponding message and places it in the program's message queue.

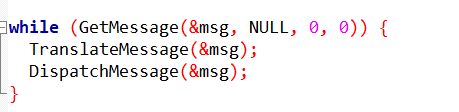


The MSG structure, defined in the WINUSER.H header file, represents a message and holds the following information:

* **hwnd:** Handle of the window associated with the message.
* **message:** Identifier of the message type.
* **wParam:** Additional message-specific information.
* **lParam:** Additional message-specific information.
* **time:** Timestamp of the message generation.
* **pt:** Point coordinates (for mouse-related messages).
* **Message Loop:** Retrieving and Dispatching Messages.

The program enters the message loop, a block of code that continuously retrieves messages from the message queue, translates them into meaningful actions, and dispatches them to the appropriate window procedure for handling.

The message loop typically looks like this:

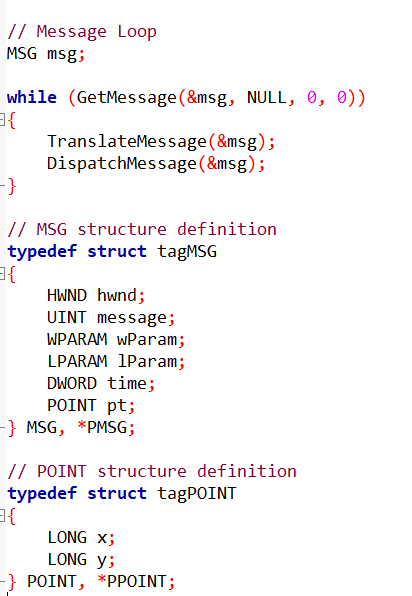


**GetMessage(&msg, NULL, 0, 0):** Retrieves the next message from the program's message queue and stores it in the msg variable.

**TranslateMessage(&msg):** Translates virtual-key messages into character messages, allowing the program to handle character input.

**DispatchMessage(&msg):** Dispatches the retrieved message to the appropriate window procedure for handling. The window procedure is responsible for interpreting the message and taking corresponding actions.

The message loop continues to execute until the GetMessage function returns FALSE, indicating that there are no more messages in the queue and the program should terminate.



**Explanation:**

The code presents a typical message loop structure.

The GetMessage function retrieves messages from the program's message queue.

The loop continues as long as there are messages to process. The TranslateMessage function translates virtual-key messages into character messages, and DispatchMessage dispatches the message to the appropriate window procedure.

The MSG structure represents a message. It contains information about the message, including the window handle (hwnd), the message type (message), and additional parameters (wParam, lParam, time, pt). wParam and lParam carry additional information specific to the message type.

The POINT structure represents a point in a two-dimensional coordinate system. In this case, it's defined with x and y coordinates. The POINT structure is used in the MSG structure to specify the cursor position when the message was generated.

The message loop repeatedly retrieves messages from the message queue, translates them if necessary, and dispatches them to the appropriate window procedure for handling. This cycle continues until there are no more messages in the queue, indicating that the program should terminate.

**Retrieving Messages from the Message Queue**

The GetMessage function, represented as **GetMessage(&msg, NULL, 0, 0)**, initiates the message loop by retrieving a message from the program's message queue. This function takes four arguments:

**&msg:** A pointer to a MSG structure that will receive the retrieved message information.

**NULL:** A window handle placeholder, indicating that the program is interested in messages for all windows it has created.

**0:** A filtering parameter that allows retrieval of all message types.

**0:** Another filtering parameter that allows retrieval of messages from all message sources.

**Message Structure and Message Identifier**

The MSG structure, defined in the WINUSER.H header file, holds the retrieved message's details:

**hwnd:** The handle of the window associated with the message. In this case, it's the same as the hwnd value returned from CreateWindow, as it's the only window created by the program.

**message:** The message identifier, a numerical value that uniquely identifies the type of message. For each message, there's a corresponding identifier defined in Windows header files, usually starting with WM ("window message"). For instance, when the mouse pointer is over the window's client area and the left mouse button is pressed, Windows places a message with message equal to WM\_LBUTTONDOWN (0x0201) in the queue.

**wParam:** A 32-bit message parameter, providing additional message-specific information. Its meaning and value depend on the specific message type.

**lParam:** Another 32-bit message parameter, providing additional message-specific information. Its meaning and value depend on the specific message type.

**time:** The timestamp indicating when the message was placed in the message queue.

**pt:** The mouse coordinates at the time the message was placed in the message queue.

**GetMessage Return Value and WM\_QUIT**

If the message field of the retrieved message is not equal to WM\_QUIT (0x0012), GetMessage returns a non-zero value, indicating that there are more messages to process.

However, if the message is WM\_QUIT, it signals the program's termination, and GetMessage returns 0. The WM\_QUIT message is typically sent when the user closes the window or initiates an action that instructs the program to exit.

**Message Translation and Dispatching**

After retrieving a message from the queue using GetMessage, the program performs two crucial steps:

**TranslateMessage(&msg);**

This function passes the retrieved MSG structure back to Windows for keyboard translation.

This process involves converting virtual-key messages into character messages, allowing the program to handle character input.

For example, when the user presses a key, Windows generates a virtual-key message, but the program typically needs to interpret this as a character input to perform actions like text editing or command execution.

**DispatchMessage(&msg);**

This function sends the translated message, now stored in the MSG structure, back to Windows.

Windows then routes the message to the appropriate window procedure for processing.

The window procedure is the function responsible for handling messages specific to a particular window. In the HELLOWIN program, the window procedure is WndProc.

**Window Procedure Handling and Message Loop Continuation**

When the message reaches the WndProc function, it is interpreted and acted upon accordingly.

The WndProc function is responsible for understanding the message type and taking the appropriate actions, such as updating the window's appearance, responding to user input, or triggering other program logic.

Once the WndProc function has processed the message, it returns control to Windows, indicating that it has handled the message. Windows then resumes its processing of the DispatchMessage call.

After Windows returns to the HELLOWIN program following the DispatchMessage call, the message loop continues.

The program executes the next GetMessage call, retrieving another message from the queue, and the cycle repeats until there are no more messages left, signaling the program's termination.

The TranslateMessage and DispatchMessage functions play essential roles in the message loop, ensuring that user input is translated, dispatched to the appropriate window procedures, and handled effectively. This mechanism is fundamental to the responsiveness and interactivity of Windows programs.

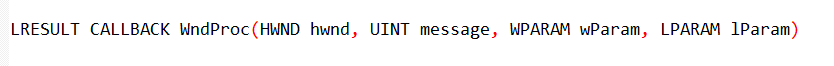
**The Heart of Window Management: The Window Procedure**

After establishing the window class, creating the window, displaying it on the screen, and entering the message loop, the program reaches the core of its functionality: the window procedure.

This function serves as the control center for handling user interactions and defining the window's behavior.

**THE WNDPROC AND ITS FUNCTIONS**

The window procedure, typically named WndProc, is the function responsible for processing messages sent to the window. It has a fixed structure:



This function receives four parameters that mirror the first four fields of the MSG structure:

**hwnd:** The handle of the window receiving the message. This is the same handle returned from the CreateWindow function and uniquely identifies the window.

**message:** An identifier representing the type of message, such as WM\_PAINT for redrawing the window or WM\_KEYDOWN for keyboard input.

**wParam:** A 32-bit message parameter providing additional message-specific information.

**lParam:** Another 32-bit message parameter providing additional message-specific information.

**Message Handling and Program Control**

The window procedure's primary task is to interpret the received message and take appropriate actions.

It may update the window's contents, respond to user input, or trigger other program logic.

The specific actions depend on the message type and the program's functionality.

Once the window procedure has handled the message, it returns a value of type LRESULT. This value indicates the result of processing the message and may be used by Windows for further handling.

**Indirectly Calling Window Procedures**

Programs typically don't directly call their window procedures.

Instead, Windows invokes the window procedure when a message is sent to the corresponding window.

However, programs can indirectly call their own window procedure using the SendMessage function, which will be discussed in later chapters.

**Multiple Windows and Window Procedures**

A window procedure can be associated with multiple windows if they are created based on the same window class.

This allows for consistent behavior across windows of the same type.

Additionally, a program can contain multiple window procedures, each responsible for handling messages for different types of windows or specific functionalities.

**Summary**

The window procedure plays a central role in the message-driven architecture of Windows programs.

It dictates how the window responds to user interactions, updates its appearance, and interacts with other program elements.

By interpreting and handling messages, the window procedure brings the program to life, enabling it to respond to user actions and fulfill its intended purpose.

**Message Handling and Default Processing**

Every message received by a window procedure is identified by a unique numerical value, referred to as the message parameter.

The Windows header file WINUSER.H defines identifiers starting with WM ("window message") for each type of message.

These identifiers provide a standardized way to classify and handle different types of user interactions and window events.

**Processing Messages with a Switch Statement**

Windows programmers typically use a switch and case construction within the window procedure to determine the specific message received and execute the corresponding processing logic.

Each case block handles a particular message type and performs the necessary actions, such as updating the window's contents, responding to user input, or triggering other program code.

**Returning Control and Default Processing**

After processing a message, the window procedure should return a value of type LRESULT.

This value indicates the result of handling the message and may be used by Windows for further processing.

If the window procedure chooses not to handle a particular message, it must pass the message to a Windows function named DefWindowProc.

The value returned from DefWindowProc must then be returned from the window procedure.

Example: Processing WM\_CREATE, WM\_PAINT, and WM\_DESTROY in HELLOWIN

In the HELLOWIN program, the WndProc function specifically handles three messages:

**WM\_CREATE:** This message is received when the window is first created. The window procedure typically performs initialization tasks here, such as setting up graphics objects or allocating resources.

**WM\_PAINT:** This message is received when the window needs to be repainted, such as when it is first displayed or when the user resizes it. The window procedure typically updates the window's contents in response to this message.

**WM\_DESTROY:** This message is received when the window is about to be destroyed, usually when the user closes it. The window procedure can perform cleanup tasks here, such as releasing resources or saving data.

**Importance of Default Processing**

It is crucial to call DefWindowProc for all messages that the window procedure does not specifically handle.

This ensures that default processing occurs for essential functionality, such as responding to system commands (e.g., closing the window) and handling keyboard input.

Neglecting to call DefWindowProc can lead to unexpected behavior and prevent the program from functioning correctly.

Message handling is a fundamental aspect of Windows programming. By processing messages and communicating with the operating system, window procedures empower programs to respond to user actions, manage their appearance, and fulfill their intended purpose. The combination of message-driven architecture and default processing ensures a well-defined and consistent framework for interacting with the user and the Windows environment.

**Processing WM\_CREATE: Initializing with a Sound**

When a **window is created** using the CreateWindow function, Windows sends a WM\_CREATE message to the window procedure.

This message indicates that the window has been successfully created and is ready for initialization. The window procedure can perform any necessary setup tasks in response to this message.

**HELLOWIN's WM\_CREATE Handling**

The HELLOWIN program chooses to handle the WM\_CREATE message by playing a waveform sound file named HELLOWIN.WAV. This adds a simple auditory cue to the program's startup.

**Using the PlaySound Function**

The PlaySound function is used to play waveform audio files. It takes three arguments:

**pszSound:** The name of the waveform file, sound alias, or resource identifier. In this case, it's the filename HELLOWIN.WAV.

**hMod:** A handle to the module containing the sound resource. This argument is only used if the sound file is a resource. In this case, it's NULL since the file is not a resource.

**dwFlags:** Flags that specify playback options. In this case, it's set to SND\_ASYNC to play the sound asynchronously, allowing the program to continue initializing without waiting for the sound to finish playing.

**Returning from WM\_CREATE Processing**

After handling the WM\_CREATE message, the window procedure returns 0, indicating that the message has been processed successfully.

This allows Windows to continue its processing of the CreateWindow call and return control to the WinMain function.

The WM\_CREATE message provides an opportunity for the window procedure to perform initial setup tasks when the window is created. The HELLOWIN program utilizes this message to play a sound file, adding an auditory element to the program's startup. The PlaySound function facilitates efficient playback of waveform audio files, allowing the program to continue with its initialization without blocking until the sound finishes playing.

**The WM\_PAINT Message: Updating the Window's Client Area**

The WM\_PAINT message plays a crucial role in Windows programming, indicating that a window's client area needs to be redrawn or "painted."

This message is triggered by various events, such as window creation, resizing, minimization/restoration, and uncovering previously obscured areas.

**Causes of Client Area Invalidation**

Several factors can cause the client area to become invalid and necessitate repainting:

**Window Creation:** Upon window creation, the entire client area is invalid since it has not yet been drawn upon. The first WM\_PAINT message instructs the window procedure to draw the initial contents.

**Window Resizing:** When the window size is changed, the client area becomes invalid. The CS\_HREDRAW and CS\_VREDRAW flags in the window class instruct Windows to invalidate the entire window when its size is altered, leading to a WM\_PAINT message.

**Window Minimization/Restoration:** Minimizing and then restoring a window causes Windows to discard the client area's contents. Upon restoration, the window receives a WM\_PAINT message to redraw the contents.

**Window Overlapping:** When windows overlap, the obscured portions of the client area become invalid. Uncovering these areas triggers WM\_PAINT messages for the respective windows to repaint their contents.

**BeginPaint and EndPaint: Framing the Painting Process**

The WM\_PAINT message processing typically begins with a call to BeginPaint:



This function provides a device context (DC) handle for drawing in the client area.

The DC handle represents the physical output device (e.g., video display) and its associated driver.

**Role of the PAINTSTRUCT Structure**

The PAINTSTRUCT structure holds information relevant to the painting process:

* rcPaint: A rectangle defining the area of the client area that needs to be repainted.
* fErase: A flag indicating whether the background should be erased before painting.
* dwRop: A raster operation (ROP) code specifying how the painted content should be combined with the existing background.
* hDC: A handle to the DC for the client area.
* EndPaint: Completing the Painting Process

Once the painting is complete, the EndPaint function is called to release the DC handle:



Releasing the DC ensures that the handle is no longer valid and prevents unauthorized access to the device or unintended drawing operations.

The WM\_PAINT message is a fundamental aspect of Windows programming, informing the window procedure when the client area needs to be redrawn.

The BeginPaint and EndPaint functions mark the beginning and end of the painting process, respectively, and the PAINTSTRUCT structure provides relevant information for efficient client area repainting.

**Window Procedure's Role in WM\_PAINT Handling**

The window procedure plays a crucial role in responding to the WM\_PAINT message, which indicates that the client area needs to be redrawn.

In rare cases where the window procedure doesn't handle this message, it must be passed to DefWindowProc to ensure that the client area is at least validated.

**Retrieving Client Area Dimensions**

Upon receiving the WM\_PAINT message, the WndProc function calls GetClientRect to retrieve the dimensions of the client area:



**This function takes two arguments:**

**hwnd:** The handle to the program's window.

**&rect:** A pointer to a RECT structure that will receive the client area dimensions.

The GetClientRect function sets the left and top fields of the RECT structure to 0, indicating the starting coordinates of the client area.

The right and bottom fields represent the width and height of the client area in pixels, respectively.

**Drawing Text with DrawText**

The WndProc function utilizes the DrawText function to display the text "Hello, Windows 98!" in the center of the client area. This function takes five arguments:

**hdc:** The device context handle obtained from BeginPaint, representing the physical output device and its driver.

**lpszText:** The text string to be drawn, in this case, "Hello, Windows 98!".

**cchText:** The number of characters in the text string. Setting this to -1 implies the text string is terminated by a null character.

**lprc:** A pointer to a RECT structure specifying the area where the text should be drawn. In this case, it's the rect obtained from GetClientRect, ensuring the text is centered within the client area.

**wFlags:** A combination of bit flags defining how the text should be displayed. In this case, it's set to DT\_SINGLELINE | DT\_CENTER | DT\_VCENTER, indicating a single line of text centered horizontally and vertically within the specified rectangle.

**Responding to Client Area Invalidation**

Whenever the client area becomes invalid, such as when the window size is changed, the WndProc function receives another WM\_PAINT message.

To handle this, it again calls GetClientRect to obtain the updated window dimensions and then calls DrawText to display the text centered within the new client area boundaries.

The WndProc function plays a central role in handling the WM\_PAINT message, ensuring that the client area is updated and repainted whenever necessary. By utilizing functions like GetClientRect and DrawText, the window procedure effectively positions and displays text within the client area, maintaining the program's visual appearance and conveying information to the user.

**The WM\_DESTROY Message: Window Destruction and Program Termination**

The WM\_DESTROY message is a critical message in Windows programming, indicating that Windows is about to destroy a window based on user action, such as clicking the Close button or selecting Close from the system menu.

This message is triggered when the user initiates a window closure, allowing the program to perform any necessary cleanup tasks before the window is removed.

**HELLOWIN's Response to WM\_DESTROY**

In response to the WM\_DESTROY message, the HELLOWIN program calls the

**PostQuitMessage function:**

****

This function inserts a **WM\_QUIT message** into the program's message queue. The WM\_QUIT message serves as a signal to terminate the program's message loop and exit the program.

**The Role of GetMessage**

The GetMessage function, which retrieves messages from the message queue, returns a non-zero value for all messages except for WM\_QUIT.

When GetMessage encounters a WM\_QUIT message, it returns 0. This triggers the termination of the message loop and the execution of subsequent cleanup code.

**Program Termination and Exit Code**

After the message loop exits, the program executes the following statement:



The wParam field of the MSG structure contains the value passed to the PostQuitMessage function, typically 0. This value is returned from the WinMain function, signaling the termination of the program with the specified exit code.

The **WM\_DESTROY message** plays a vital role in window management, informing the program when a window is about to be destroyed. The HELLOWIN program's response to this message, by calling PostQuitMessage, ensures proper cleanup and termination of the program, allowing Windows to reclaim resources and maintain system stability.

**Conceptual Hurdles in Windows Programming: Shifting from Character-Mode to Message-Driven**

Transitioning from character-mode programming, where the program's logic resides primarily in the main function, to Windows programming introduces a significant conceptual shift.

In Windows programming, the program's behavior is driven by messages sent to window procedures, a fundamental concept that governs how programs interact with the operating system and respond to user actions.

**The Central Role of the Message Loop and Window Procedures**

The core of a Windows program lies in the message loop, a continuous cycle that retrieves messages from the message queue and dispatches them to the appropriate window procedures. Window procedures are functions responsible for handling messages specific to a particular window, such as processing keyboard input, redrawing the window's contents, or responding to system commands.

**HELLOWIN as an Example of Message-Driven Programming**

The HELLOWIN program, despite its simplicity, exemplifies this message-driven paradigm. While the WinMain function handles initial setup tasks like registering the window class and creating the window, the real action happens in the WndProc function. This function processes messages such as WM\_CREATE, WM\_PAINT, and WM\_DESTROY, performing actions like playing a sound file and displaying text.

**Conceptual Leap for Windows Programmers**

The transition from character-mode to message-driven programming requires a conceptual leap.

Programmers must adapt to a reactive programming style, where the program's behavior is dictated by the messages it receives and the corresponding actions taken in the window procedures.

This shift in mindset is essential for developing responsive and interactive Windows applications.

**A Paradigm Shift: Windows Programs Responding to Operating System Calls**

Programmers are accustomed to initiating actions by calling functions within the operating system, such as opening a file using the fopen function in C programming.

However, Windows programming introduces a distinct paradigm shift, where the operating system takes the initiative by sending messages to the program's window procedure, a function specifically designed to handle these messages.

**The Role of the Window Procedure**

The window procedure, associated with a window class registered using the RegisterClass function, acts as the communication bridge between the operating system and the program.

It receives messages from the operating system and performs the corresponding actions, such as updating the window's content, responding to user input, or handling system events.

**Windows Initiating Actions Through Messages**

Windows sends messages to the window procedure in response to various user interactions and system events.

These messages, identified by names starting with WM and defined in the WINUSER.H header file, provide detailed information about the event that triggered the communication.

**Examples of Message-Driven Interactions**

Windows sends messages to the window procedure in various scenarios:

* When a window is first created (WM\_CREATE).
* When a window is resized, moved, or minimized (WM\_SIZE, WM\_MOVE, WM\_MINIMIZE).
* When a user clicks on the window with the mouse (WM\_LBUTTONDOWN, WM\_RBUTTONDOWN).
* When characters are typed from the keyboard (WM\_CHAR).
* When an item is selected from a menu (WM\_COMMAND).
* When a scroll bar is manipulated or clicked (WM\_HSCROLL, WM\_VSCROLL).
* When the client area needs to be repainted (WM\_PAINT).

**Handling Messages and Default Processing**

The window procedure is responsible for processing these messages and taking appropriate actions.

It may perform tasks like updating the window's content, responding to user input, or validating data.

If the window procedure chooses not to handle a specific message, it can pass it to DefWindowProc for default processing.

**wParam and lParam:** Providing Additional Information. The wParam and lParam parameters to the window procedure provide additional context and information about the message. The meaning of these parameters varies depending on the specific message type.

**Comparison to Character-Mode Interrupt Handling**

While the concept of **routines** within a program being called from outside is not entirely new, Windows programming elevates this concept to a central paradigm.

In **character-mode programming**, the signal function in C can handle interrupts like Ctrl+C or hardware interrupts from MS-DOS.

However, Windows extends this concept to encompass the entire program's interaction with the operating system.

**Handling Window Size Changes: An Example of Message Processing**

Windows programming involves responding to various messages sent by the operating system to the program's window procedure.

One such message is WM\_SIZE, which indicates that the size of a window has changed. Let's delve into how the window procedure handles this message:

**Identifying the Window and Message Type**

The hwnd parameter to the window procedure identifies the specific window whose size has changed. This parameter is crucial since the window procedure may be responsible for multiple windows created from the same window class.

**Extracting Size Information from wParam and lParam**

The wParam parameter provides information about the new window size:

* **SIZE\_RESTORED (0):** Indicates the window is being restored to a non-minimized or non-maximized size.
* **SIZE\_MINIMIZED (1):** Indicates the window is being minimized.
* **SIZE\_MAXIMIZED (2):** Indicates the window is being maximized.
* **SIZE\_MAXSHOW (3):** Indicates the window is being shown and maximized.
* **SIZE\_MAXHIDE (4):** Indicates the window is being hidden and maximized.

The lParam parameter contains the new dimensions of the window. The 16-bit width and 16-bit height values are combined into a single 32-bit lParam value. The WINDEF.H header provides macros to extract these values separately.

**DefWindowProc's Role in Message Handling**

DefWindowProcplays a crucial role in message processing. When the window procedure doesn't handle a specific message, it typically passes it to DefWindowProc for default processing.

Message Cascading: WM\_SYSCOMMAND, WM\_CLOSE, and WM\_DESTROY

In some cases, messages trigger a chain of subsequent messages. For instance, closing a window using the Close button or the system menu can trigger a cascade of messages:

* The system sends a WM\_SYSCOMMAND message to the window procedure.
* The window procedure passes the message to DefWindowProc.
* DefWindowProc sends a WM\_CLOSE message to the window procedure.
* The window procedure again passes the message to DefWindowProc.
* DefWindowProc responds to WM\_CLOSE by calling DestroyWindow.
* DestroyWindow prompts Windows to send a WM\_DESTROY message to the window procedure.
* The window procedure finally handles WM\_DESTROY by calling PostQuitMessage, inserting a WM\_QUIT message into the message queue.
* The message loop in WinMain detects the WM\_QUIT message, terminating the program.

The WM\_SIZE message exemplifies the message-driven architecture of Windows programming. The window procedure extracts relevant information from the message parameters, **wParam** and **lParam**, to determine the new window size. In some cases, message handling involves a chain of messages, with DefWindowProc playing a crucial role in routing and processing messages.

**Queued and Nonqueued Messages: A Deeper Dive into Message Handling**

In the realm of Windows programming, messages serve as the primary means of communication between the operating system, the program, and the user.

These messages convey information about user actions, system events, and program-specific requests.

While Windows has the capability to directly call the window procedure for certain messages, a significant portion of message handling involves the use of message queues.

**Queued Messages: The Message Queue as a Communication Hub**

Queued messages are those placed in a program's message queue by Windows.

This queue acts as a temporary storage area, holding messages until the program's message loop retrieves and processes them.

The message loop, a fundamental component of Windows programs, continuously checks the message queue for pending messages.

Upon finding a message, the message loop retrieves it using the GetMessage function and dispatches it to the appropriate window procedure using the DispatchMessage function.

**Nonqueued Messages: Direct Communication with the Window Procedure**

Nonqueued messages, on the other hand, bypass the message queue and are sent directly to the window procedure by Windows.

These messages typically result from calls to specific Windows functions made by the program.

For instance, when the CreateWindow function is called to create a new window, Windows sends a WM\_CREATE message directly to the window procedure associated with that window.

**The Role of the Window Procedure: Message Processing Central**

The window procedure, the heart of a window's message handling, is responsible for processing all messages, whether queued or nonqueued. It receives and interprets messages, taking appropriate actions based on the message type and its accompanying parameters.

**Examples of Queued Messages**

Queued messages arise from various user interactions and system events. Some common examples include:

WM\_KEYDOWN and WM\_KEYUP: Messages indicating keystrokes pressed and released

WM\_CHAR: Messages representing characters generated from keystrokes

WM\_MOUSEMOVE: Messages indicating mouse movement

WM\_LBUTTONDOWN: Messages indicating left mouse button clicks

WM\_TIMER: Messages triggered by timer events

WM\_PAINT: Messages requesting the window's client area to be repainted

WM\_QUIT: Messages signaling the program's termination

**Examples of Nonqueued Messages**

Nonqueued messages often stem from explicit calls to Windows functions. Here are a few examples:

**WM\_CREATE:** Message sent when a window is created using CreateWindow

**WM\_SIZE** and **WM\_SHOWWINDOW** messages sent when a window is displayed using ShowWindow

**WM\_PAINT** message sent when a window's client area needs to be repainted using UpdateWindow

**WM\_COMMAND** message indicating that a menu item has been selected, triggered by keyboard or mouse input

**Maintaining Message Order and Context**

Windows ensures that messages are delivered to the window procedure in an orderly manner, preventing interruptions while processing an existing message.

Additionally, when multiple threads are involved, each thread has its own message queue, ensuring that messages for a specific window are handled by the thread associated with that window's window procedure.

To maintain context across multiple message processing cycles, window procedures often utilize static variables defined within the procedure itself or global variables accessible throughout the program.

These variables enable the window procedure to retain information obtained from one message and use it while processing subsequent messages.

**Summary**

Queued and nonqueued messages play distinct roles in Windows programming. Queued messages provide a structured mechanism for handling user input and system events, while nonqueued messages allow for direct communication between Windows and the program. The window procedure, acting as the central message handler, processes both queued and nonqueued messages, ensuring that the program responds appropriately to user actions and system events.

**Managing Message Delivery: Order, Synchronization, and Reentrancy**

Windows programming involves a complex interplay of messages, message queues, and window procedures.

While the underlying mechanisms may seem intricate, the window procedure experiences a well-ordered and synchronized flow of messages.

**Orderly Message Delivery: No Mid-Processing Interruptions**

Unlike hardware interrupts that can disrupt program execution, messages arrive at the window procedure in an orderly fashion.

This means that while processing one message, the program is not suddenly interrupted by another message.

Windows ensures this orderly delivery, allowing the window procedure to focus on handling each message fully before moving on to the next.

**Thread-Specific Message Queues: Segregating Messages for Each Thread**

In multithreaded Windows programs, each thread maintains its own message queue. This segregation ensures that messages for a particular window are handled by the thread associated with that window's window procedure. As a result, message processing remains synchronized even when multiple threads are running concurrently.

**DispatchMessage's Role: Waiting for Message Processing Completion**

The DispatchMessage function, responsible for sending messages from the message queue to the window procedure, does not return control until the window procedure has finished processing the message.

This synchronization ensures that the window procedure has ample time to handle each message without interruptions from the message queue.

**Reentrant Window Procedures: Handling Nested Messages**

Window procedures must be **reentrant**, meaning they can handle nested messages without data corruption.

This reentrancy is crucial because a window procedure may receive a new message while processing an existing one.

**Static and Global Variables: Preserving Message Context**

To maintain context across multiple message processing cycles, window procedures often utilize static variables defined within the procedure itself or global variables accessible throughout the program.

These variables enable the window procedure to retain information obtained from one message and use it while processing subsequent messages.

**Compiler Optimization Considerations: Avoiding Unexpected Variable Alterations**

Certain forms of compiler optimization may interfere with the reentrancy requirements of window procedures.

Therefore, it is essential to disable such optimizations when compiling Windows programs to ensure data integrity.

**Preemptive Multitasking: A Mixed Blessing**

Windows 10 and 11, like their predecessors Windows 98 and Windows NT, are preemptive multitasking environments.

This means that the operating system can switch control between multiple programs, even if one program is performing a lengthy task.

While this feature allows users to interact with multiple applications simultaneously, it can also lead to situations where a program becomes unresponsive due to being stuck in a lengthy message processing cycle.

**Impact of Lengthy Message Processing on User Interaction**

When a program spends a significant amount of time processing a specific message, its window becomes unresponsive to user input.

Users cannot move, resize, minimize, or close the window, essentially rendering it unusable until the message processing completes.

This unresponsiveness stems from the fact that the window procedure, responsible for handling window-related events, is busy executing the lengthy task.

**DefWindowProc's Role in Window Operations**

The window procedure's responsibilities extend beyond simply processing messages.

It also handles operations like window movement, resizing, and minimization, which are typically delegated to the DefWindowProc function.

When the window procedure is engaged in a lengthy processing task, these operations become unavailable to the user.

**Mitigating the Impact of Lengthy Message Processing**

While preemptive multitasking provides the illusion of multiple programs running concurrently, it's crucial to handle lengthy message processing tasks in a considerate manner.

Techniques like message posting and utilizing idle time can be employed to minimize the impact on user interaction.

**The User Experience: Avoiding Annoyance**

Leaving an unresponsive window on the screen is akin to encountering bugs, nonstandard behavior, or incomplete help files – it creates an unpleasant user experience.

To avoid frustrating users, it's essential to ensure that window procedures handle messages efficiently and return control to the operating system promptly, allowing users to interact with the program as expected.

**Summary**

Windows programs handle messages in an **orderly** and **synchronized manner**, ensuring that the window procedure processes messages sequentially without interruptions.

Message queues, thread-specific message handling, and DispatchMessage synchronization contribute to this smooth message flow.

Window procedures must be **reentrant to handle nested messages effectively**, and static or global variables help maintain context across message processing cycles. By carefully managing message delivery and reentrancy, programmers can create responsive and robust Windows applications.