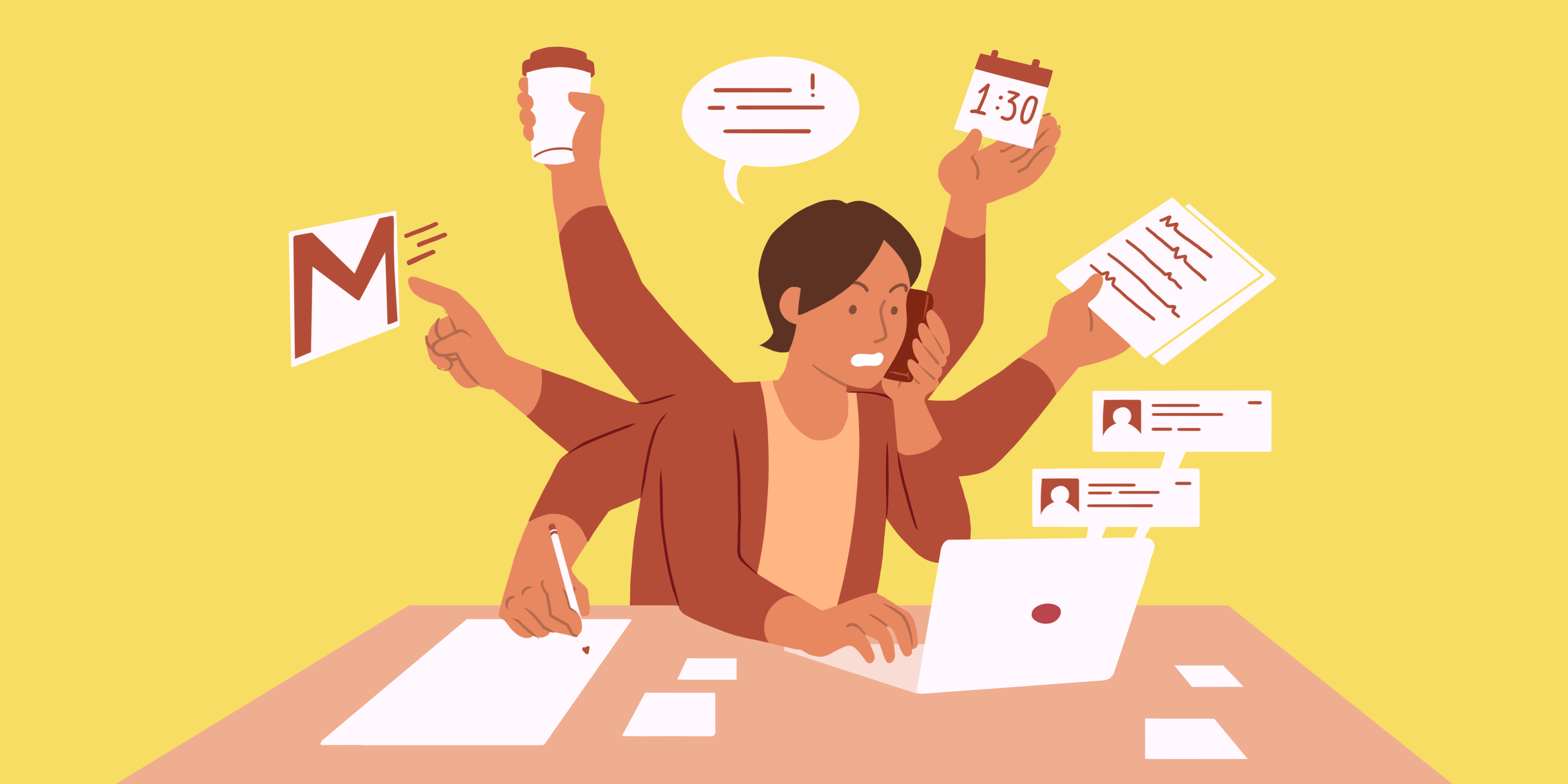
THE TIMER

The Windows timer is a versatile tool that can be used for a variety of purposes in Windows applications. It allows a program to be notified periodically at a specified interval, which can be used for tasks such as:

Multitasking: In a multitasking environment, it is sometimes more efficient for a program to yield control to Windows frequently rather than processing large amounts of data at once. The timer can be used to divide a large task into smaller pieces and process each piece upon receipt of a WM\_TIMER message.



Real-time updates: The timer can be used to display "real-time" updates of continuously changing information, such as system resources or the progress of a task. This is useful for applications that need to provide up-to-date information to the user.



Autosave: The timer can be used to prompt a program to save a user's work to disk at regular intervals. This can help to prevent data loss in the event of a power outage or other unexpected interruption.



Terminating demo versions: Demo versions of software are often limited in time. The timer can be used to terminate such applications when the allotted time has elapsed.



Pacing movement: The timer can be used to control the movement of objects in a game or animation at a consistent rate. This helps to create a smooth and realistic experience for the user.



Multimedia synchronization: Multimedia applications that play audio or video often need to synchronize the media with other events, such as on-screen animations. The timer can be used to accurately determine the playback position of the media and coordinate it with these other events.



In addition to these specific uses, the timer can be used for any general purpose that requires a program to be notified at regular intervals. The timer is a powerful tool that can be used to improve the performance, responsiveness, and usability of Windows applications.

Key Points

* *The Windows timer is an input device that periodically notifies an application when a specified interval of time has elapsed.*
* *The timer is a versatile tool that can be used for a variety of purposes, including multitasking, real-time updates, autosave, terminating demo versions, pacing movement, and multimedia synchronization.*
* *The timer can be used to improve the performance, responsiveness, and usability of Windows applications.*

A timer is a software object that allows a program to be notified at regular intervals. In Windows, timers are implemented using the SetTimer() and KillTimer() functions.

SetTimer()`

The SetTimer() function takes three arguments:

* hWnd: The window handle of the window that will receive the timer messages.
* uID: A timer identifier. This identifier is used to distinguish between multiple timers that the same window may have.
* uElapse: The time-out interval in milliseconds. This is the amount of time that must elapse before the window receives a WM\_TIMER message.

The SetTimer() function returns a non-zero value if the timer was successfully created, and zero if an error occurred.

KillTimer()`

The KillTimer() function takes two arguments:

* hWnd: The window handle of the window that is receiving the timer messages.
* uID: The timer identifier that was returned by the SetTimer() function.

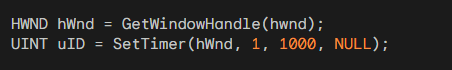
The KillTimer() function stops the timer and removes it from the system.

WM\_TIMER Message

The WM\_TIMER message is a Windows message that is sent to a window when a timer elapses. The wParam parameter of the WM\_TIMER message contains the timer identifier, and the lParam parameter is unused.

Example

The following code shows how to create a timer that will send a WM\_TIMER message to the window every second:



The following code shows how to stop the timer:



Timer Resolution

The resolution of the Windows timer is the minimum amount of time that can elapse between timer notifications.

The resolution of the timer is typically 55 milliseconds. This means that the SetTimer() function will round down the time-out interval to an integral multiple of 55 milliseconds.

For example, a time-out interval of 1000 milliseconds will be rounded down to 989 milliseconds.

Performance Considerations

Timers can be used to implement a variety of features in Windows applications, such as animation, real-time updates, and autosave.

However, it is important to use timers carefully, as they can consume CPU resources and affect the performance of an application.

For example, a timer that is set to a very short time-out interval can cause the application to become unresponsive.

The Windows timer is a single-threaded object. This means that it can only be used by one thread at a time.

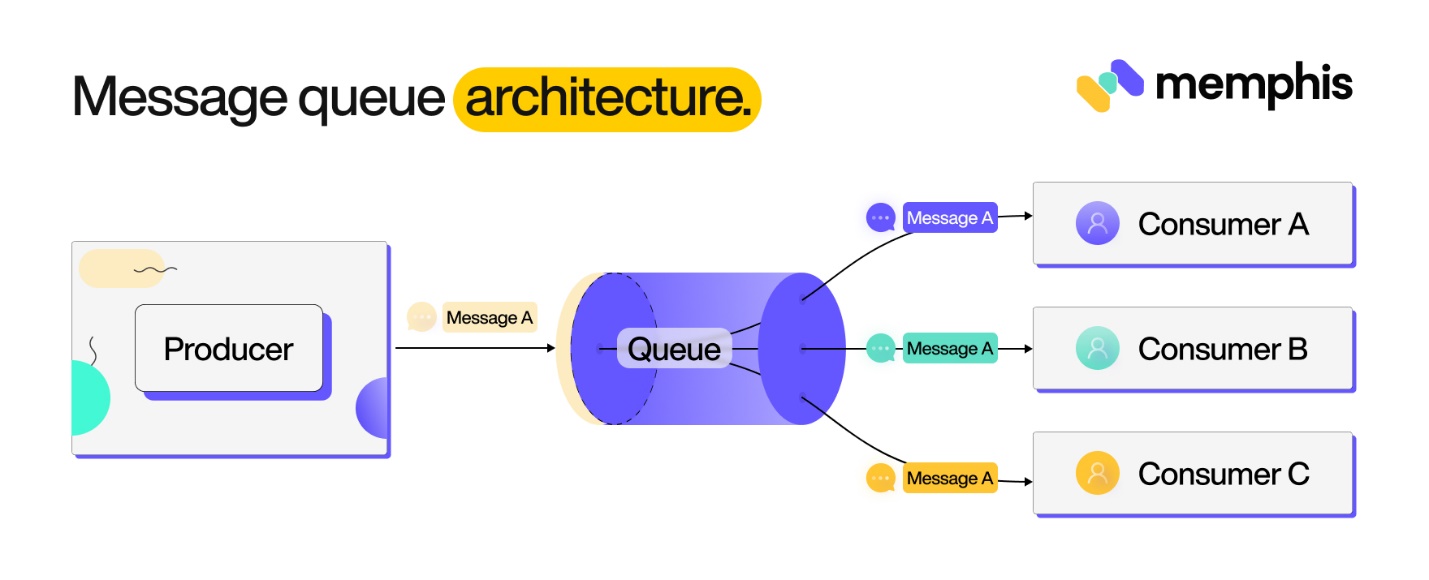
If you need to use a timer in a multithreaded application, you must use a synchronization mechanism, such as a mutex, to prevent race conditions.

The Windows timer is not guaranteed to be accurate. The accuracy of the timer can be affected by factors such as system load and hardware interrupts.

Timer Messages and Asynchronous Processing

Despite their name, timer messages in Windows programming are not asynchronous.

While they are generated by a hardware timer interrupt, the way they are handled by the operating system and delivered to applications means that they are not guaranteed to interrupt the current processing of the application. Instead, they are placed in the message queue along with other messages, such as mouse clicks and keyboard events.



The message queue is a FIFO (First-In, First-Out) data structure, meaning that messages are processed in the order they are received. This means that if an application is busy processing other messages, it may not receive a timer message even if the timer has elapsed.

Implications of Non-Asynchronous Timer Messages

The non-asynchronous nature of timer messages has several implications for Windows programmers:

Programs cannot rely on timer messages to provide precise timing. The actual timing of timer messages can be affected by the load on the system and the order in which other messages are processed.

Programs should not use timer messages to interrupt the current processing of other messages. If a program needs to perform an action at a specific time, it should use a different mechanism, such as a custom timer thread or a system timer API.

Programs should be able to handle the possibility of missing timer messages. If an application relies on timer messages to update its state, it should be able to handle the situation where a timer message is not received as expected.

Strategies for Handling Timer Messages

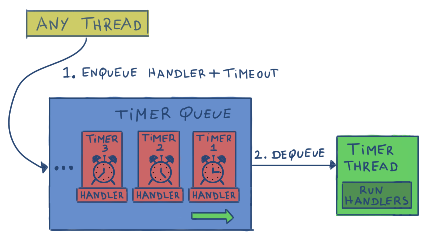
Despite their limitations, timer messages can still be a useful tool for implementing certain features in Windows applications. However, programmers should be aware of the non-asynchronous nature of timer messages and take steps to mitigate the potential problems.

Here are some strategies for handling timer messages:

Use timer messages to trigger updates, not to provide precise timing. The timer can be used to signal to the application that it is time to update its state, but the actual timing of the update should be handled by the application itself.



Use a timer thread if precise timing is required. If an application needs to perform an action at a specific time, it can use a custom timer thread to generate its own timer events. This thread can then interrupt the main thread to perform the action.



Use a system timer API if available. Some system timer APIs, such as the Multimedia Timer API, offer more precise timing than the standard Windows timer.



USING THE TIMER: THREE METHODS

There are three main methods for using the timer in Windows programming:

Method 1: Simple Timer

This method is the simplest and most common way to use the timer. It involves calling the SetTimer function to create a timer that sends WM\_TIMER messages to the window procedure of the application. The SetTimer function takes three arguments:

* hwnd: The window handle of the window whose window procedure will receive the WM\_TIMER messages.
* uID: A timer ID, which should be a nonzero number. This ID is used to distinguish between multiple timers that the same window may have.
* uElapse: The time-out interval in milliseconds. This is the amount of time that must elapse before the window receives a WM\_TIMER message.

The following code shows how to create a timer that sends a WM\_TIMER message to the window every second:



The KillTimer function can be used to stop the timer. The KillTimer function takes two arguments:

* hwnd: The window handle of the window that is receiving the timer messages.
* uID: The timer ID that was returned by the SetTimer function.

The following code shows how to stop the timer:



The WM\_TIMER message is a Windows message that is sent to a window when a timer elapses. The wParam parameter of the WM\_TIMER message contains the timer ID, and the lParam parameter is unused.

Handling WM\_TIMER Messages

When using timers in Windows programming, it's crucial to handle WM\_TIMER messages appropriately within the window procedure. These messages are sent to the window procedure associated with the timer, and they carry information about the timer ID and other relevant details.

Understanding wParam and lParam

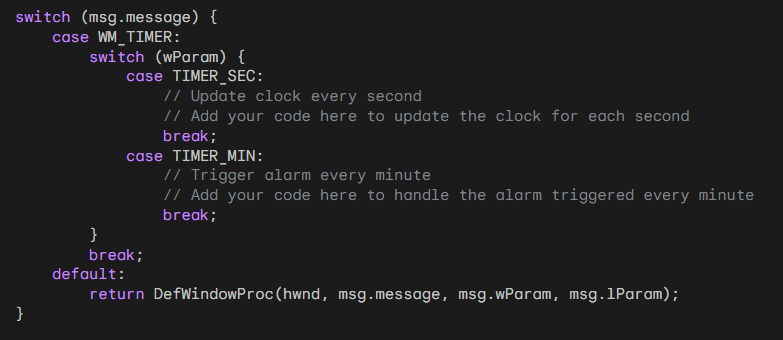
Within the WM\_TIMER message handling code, the wParam parameter holds the timer ID, which is a unique identifier assigned to each timer when it's created using the SetTimer function. The lParam parameter is typically unused and can be disregarded.

Using Timer IDs for Multiple Timers

If an application requires multiple timers, each timer should be assigned a unique ID. This allows the window procedure to differentiate between the timers and handle their respective actions accordingly.

Example: Handling Two Timers

To illustrate the concept of handling multiple timers, consider a scenario where one timer updates the clock every second and another timer triggers an alarm every minute. The window procedure can handle these timers using a switch statement based on the wParam value:



Here is a breakdown of the code:

* The outer switch statement handles all messages sent to the window procedure.
* The inner switch statement handles WM\_TIMER messages.
* The case statements within the inner switch statement handle different timer IDs.
* The break statements at the end of each case statement prevent the code from falling through to the next case.
* The default statement handles all other messages.
* The DefWindowProc function is called to handle all messages that are not handled by the window procedure.

Modifying Timer Intervals

To modify the interval of an existing timer, simply call the SetTimer function again with the desired interval. This is useful for adjusting the timer's behavior dynamically. Example:

A clock program might have an option to toggle the display of seconds. To achieve this, the timer interval can be modified by calling SetTimer with an interval between 1000 milliseconds (1 second) and 60,000 milliseconds (1 minute).

The BEEPER1 program demonstrates the use of timers to create simple interactive applications. It sets a timer for 1-second intervals and alternates coloring the window's client area blue and red upon receiving each WM\_TIMER message. Additionally, it emits a beep using the MessageBeep function.

The BEEPER1 program sets the timer during the WM\_CREATE message handling in the window procedure. This ensures that the timer is started when the window is created.

Within the WM\_TIMER message handling code, BEEPER1 calls MessageBeep to produce a beep, inverts the value of a flag to switch the color, and invalidates the window to generate a WM\_PAINT message.

During the WM\_PAINT message processing, BEEPER1 retrieves the window's dimensions and fills the client area with the appropriate color based on the flag's value.



Main Points in the Code

* The program sets a timer for 1 second intervals. This is done in the WM\_CREATE message handler by calling the SetTimer function.
* When the timer elapses, the program beeps and alternates the color of the client area. This is done in the WM\_TIMER message handler by calling the MessageBeep function and inverting the value of the fFlipFlop flag.
* The program invalidates the window to generate a WM\_PAINT message. This is done in the WM\_TIMER message handler after calling MessageBeep and inverting the value of the fFlipFlop flag.
* The program paints the client area with the appropriate color based on the value of the fFlipFlop flag. This is done in the WM\_PAINT message handler by calling the GetClientRect function, creating a solid brush, and filling the client area with the brush.

Main Points in the Notes

* Timer messages are not precise. The accuracy of timer messages can be affected by factors such as system load and hardware interrupts.
* Timer messages are not asynchronous. They are placed in the message queue and delivered to the application in the order they are received.
* It is considered good form to kill any active timers before your program terminates. This can be done by calling the KillTimer function in the WM\_DESTROY message handler.
* The program uses a flag (fFlipFlop) to alternate the color of the client area. This is a simple way to achieve the desired effect.
* The program uses the MessageBeep function to beep. This function is a convenient way to generate a beep without having to create a sound object.
* The program uses the GetClientRect function to get the dimensions of the client area. This is necessary for painting the entire client area with the desired color.
* The program uses the CreateSolidBrush function to create a solid brush. This is a simple way to create a brush with a solid color.
* The program uses the FillRect function to fill the client area with the brush. This is the easiest way to paint the entire client area with a solid color.

Method 2: Using a Dialog Box Procedure

This method is similar to Method 1, but it uses a dialog box procedure to handle the WM\_TIMER messages. This can be useful if you want to use the timer to perform actions that affect multiple dialog boxes in your application.

The following code shows how to create a timer that sends a WM\_TIMER message to the dialog box procedure every second:

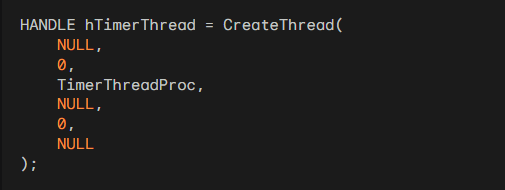


The dialog box procedure can then handle the WM\_TIMER message by performing the desired actions.

Method 3: Using a Timer Thread

This method is the most complex, but it also offers the most flexibility. It involves creating a worker thread that is responsible for managing the timer and sending notifications to the main thread. This can be useful if you need to perform precise timing or if you want to avoid blocking the main thread.

The following code shows how to create a timer thread:

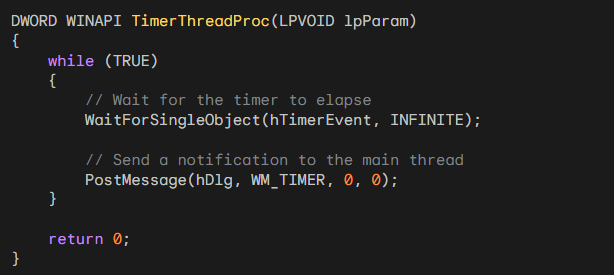


The code:

* NULL for Security Attributes: In Windows, you can set security attributes for the thread. Here, we've used NULL, which means the thread gets default security attributes.
* Stack Size: The stack size for the new thread is set to 0, which means it uses the default stack size.
* Thread Function: TimerThreadProc is the function that the thread will execute. Make sure you've defined TimerThreadProc somewhere in your code.
* Thread Parameters: The fourth parameter is a pointer to a variable to be passed to the thread function. In this case, it's NULL, indicating no specific data is being passed.
* Creation Flags: The thread runs immediately after creation. The creation flags are set to 0.
* Thread ID: The last parameter is a pointer to a variable that receives the thread identifier. In this case, it's set to NULL, meaning the thread identifier is not needed.

This code uses the CreateThread function to create a new thread and assigns the handle of the created thread to hTimerThread. Make sure to handle errors and manage the thread's lifecycle appropriately in your application. And hey, threading can be a bit tricky, but it's a powerful tool for parallel execution! What do you think about threading in C++?

The TimerThreadProc function is responsible for managing the timer and sending notifications to the main thread. The following code shows an example of how to implement the TimerThreadProc function:



The main thread can then handle the WM\_TIMER message by performing the desired actions.

Method Two: Using a Call-Back Function

Method Two offers more flexibility than Method One by allowing you to direct Windows to send timer messages to a specific function within your program, rather than the default window procedure.

This approach is particularly useful when you want to separate the timer handling logic from the window procedure and handle timer events in a more organized and structured manner.

The Call-Back Function: TimerProc

The call-back function, in this case named TimerProc, is responsible for processing WM\_TIMER messages. It receives the following parameters:

* hwnd: The handle to the window specified when you call SetTimer.
* message: The message type, which is always WM\_TIMER for this function.
* iTimerID: The timer ID, which is a unique identifier assigned to each timer when it's created using SetTimer.
* dwTime: A value compatible with the return value from the GetTickCount function, indicating the number of milliseconds elapsed since Windows started.

Within the TimerProc function, you can implement the desired actions to be performed in response to the timer events. For instance, you can update UI elements, perform calculations, or trigger other events based on the elapsed time.

Setting the Timer with Call-Back Function

To set a timer using Method Two, you'll need to modify the SetTimer call. Instead of passing NULL as the fourth argument, you'll pass the address of the TimerProc function:



This informs Windows to send WM\_TIMER messages to the TimerProc function rather than the window procedure.

Example: BEEPER2 Program

The BEEPER2 program demonstrates the use of a call-back function for handling timer events. It's functionally similar to BEEPER1, but it sends the timer messages to TimerProc instead of WndProc. The TimerProc function simply alternates the color of the client area and beeps using MessageBeep.