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:

```
HWND hWnd = GetWindowHandle(hwnd);
UINT uID = SetTimer(hWnd, 1, 1000, NULL);
```

The following code shows how to stop the timer:

```
KillTimer(hWnd, uID);
```

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.

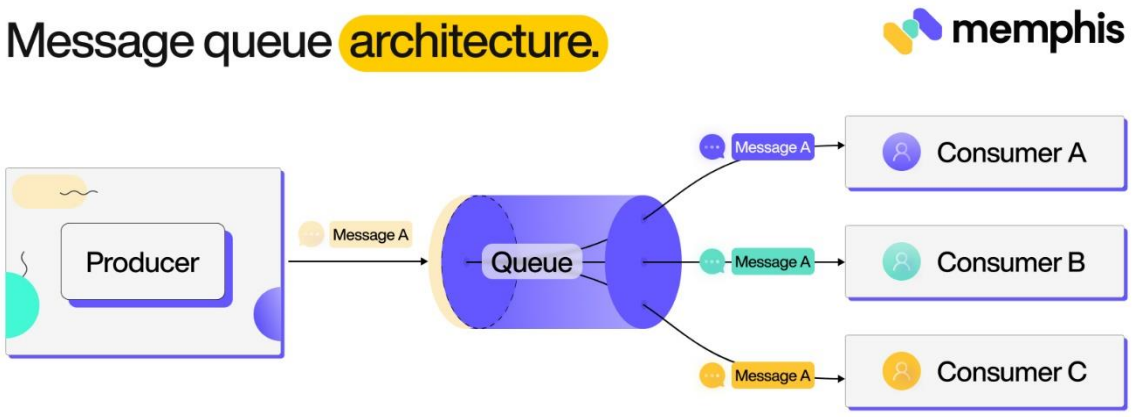
If you need to use a timer in a multithreaded application, you must use a synchronization mechanism, such as a mutex, to prevent race conditions.

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.

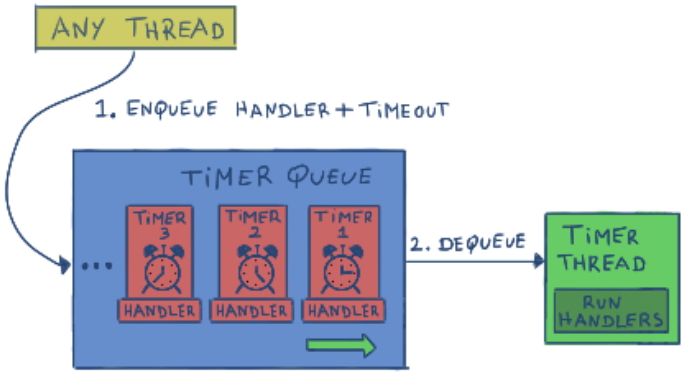
Strategies for Handling Timer Messages

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: FOUR 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:

```
SetTimer(hWnd, 1, 1000, NULL);
```

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:

```
KillTimer(hWnd, 1);
```

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:

```
switch (msg.message) {
    case WM_TIMER:
        switch (wParam) {
            case TIMER_SEC:
                // Update clock every second
                // Add your code here to update the clock for each second
                break;
            case TIMER_MIN:
                // Trigger alarm every minute
                // Add your code here to handle the alarm triggered every minute
                break;
        }
        break;
    default:
        return DefWindowProc(hwnd, msg.message, msg.wParam, msg.lParam);
}
```

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.

Beeper 1.mp4

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:

```
SetTimer(hDlg, 1, 1000, NULL);
```

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:

```
HANDLE hTimerThread = CreateThread(
    NULL,
    0,
    TimerThreadProc,
    NULL,
    0,
    NULL
);
```

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:

```
DWORD WINAPI TimerThreadProc(LPVOID lpParam)
{
    while (TRUE)
    {
        // Wait for the timer to elapse
        WaitForSingleObject(hTimerEvent, INFINITE);

        // Send a notification to the main thread
        PostMessage(hDlg, WM_TIMER, 0, 0);
    }

    return 0;
}
```

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:

```
SetTimer(hwnd, iTimerID, iMsecInterval, TimerProc);
```

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.

BEEPER1

The BEEPER1 program uses Method One, which involves setting a timer in the WM_CREATE message handler and processing timer messages in the WM_TIMER message handler. This method sends timer messages directly to the window procedure.

BEEPER2

The BEEPER2 program uses Method Two, which involves setting a timer in the WM_CREATE message handler and processing timer messages in a separate call-back function named TimerProc. This method directs Windows to send timer messages to the TimerProc function instead of the window procedure.

Comparison

Method Two offers more flexibility and separation of concerns compared to Method One. It allows you to encapsulate the timer handling logic in a separate function, making the code cleaner and more modular. Additionally, Method Two enables you to handle timer events without modifying the window procedure, which can be particularly useful when dealing with multiple timers or more complex timer-based functionalities.

```
72  VOID CALLBACK TimerProc (HWND hwnd, UINT message, UINT iTimerID, DWORD dwTime)
73  {
74      static BOOL fFlipFlop = FALSE ;
75      HBRUSH      hBrush ;
76      HDC         hdc ;
77      RECT        rc ;
78
79      MessageBeep (-1) ;
80      fFlipFlop = !fFlipFlop ;
81
82      GetClientRect (hwnd, &rc) ;
83
84      hdc = GetDC (hwnd) ;
85      hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255)) ;
86
87      FillRect (hdc, &rc, hBrush) ;
88      ReleaseDC (hwnd, hdc) ;
89      DeleteObject (hBrush) ;
90  }
```

METHOD 4: USING A TIMER WITH NULL WINDOW HANDLE

I don’t know if I should call this method three or four or is this the second method??

Method Three offers a unique approach to setting timers by utilizing a NULL window handle. It's particularly useful when you want to manage multiple timers without assigning specific IDs to each one.

Setting the Timer

To set a timer using Method Three, you'll need to modify the SetTimer call:

```
iTimerID = SetTimer(NULL, 0, wMsecInterval, TimerProc);
```

In this call:

- hwnd: Set to NULL, indicating that the timer is not associated with a specific window.
- iTimerID: Set to 0, which instructs Windows to assign an available timer ID.
- wMsecInterval: Specifies the timer interval in milliseconds.
- TimerProc: The address of the timer callback function.

The SetTimer function will return a unique timer ID (iTimerID) that identifies the newly created timer. This ID is crucial for identifying and managing the timer later.

Killing the Timer

To terminate a timer set using Method Three, you'll use the KillTimer function with the NULL window handle and the timer ID:

```
KillTimer(NULL, iTimerID);
```

In this call:

- hwnd: Set to NULL, indicating that the timer is not associated with a specific window.
- iTimerID: The timer ID returned from the corresponding SetTimer call.

This will effectively stop the timer associated with the specified ID.

Benefits of Method Three

Method Three offers several advantages:

Dynamic Timer Assignment: Windows automatically assigns timer IDs, eliminating the need for you to manage them manually.

Reduced Memory Overhead: Using a single NULL window handle for multiple timers reduces memory usage compared to assigning separate window handles for each timer.

Simplified Timer Management: The absence of window handle association makes it simpler to track and manage multiple timers.

Applications of Method Three

Method Three is particularly useful when you need to set and manage multiple timers throughout your program without the complexity of assigning and tracking specific IDs. It's also beneficial when the timer functionality is not directly tied to a specific window.

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.

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.

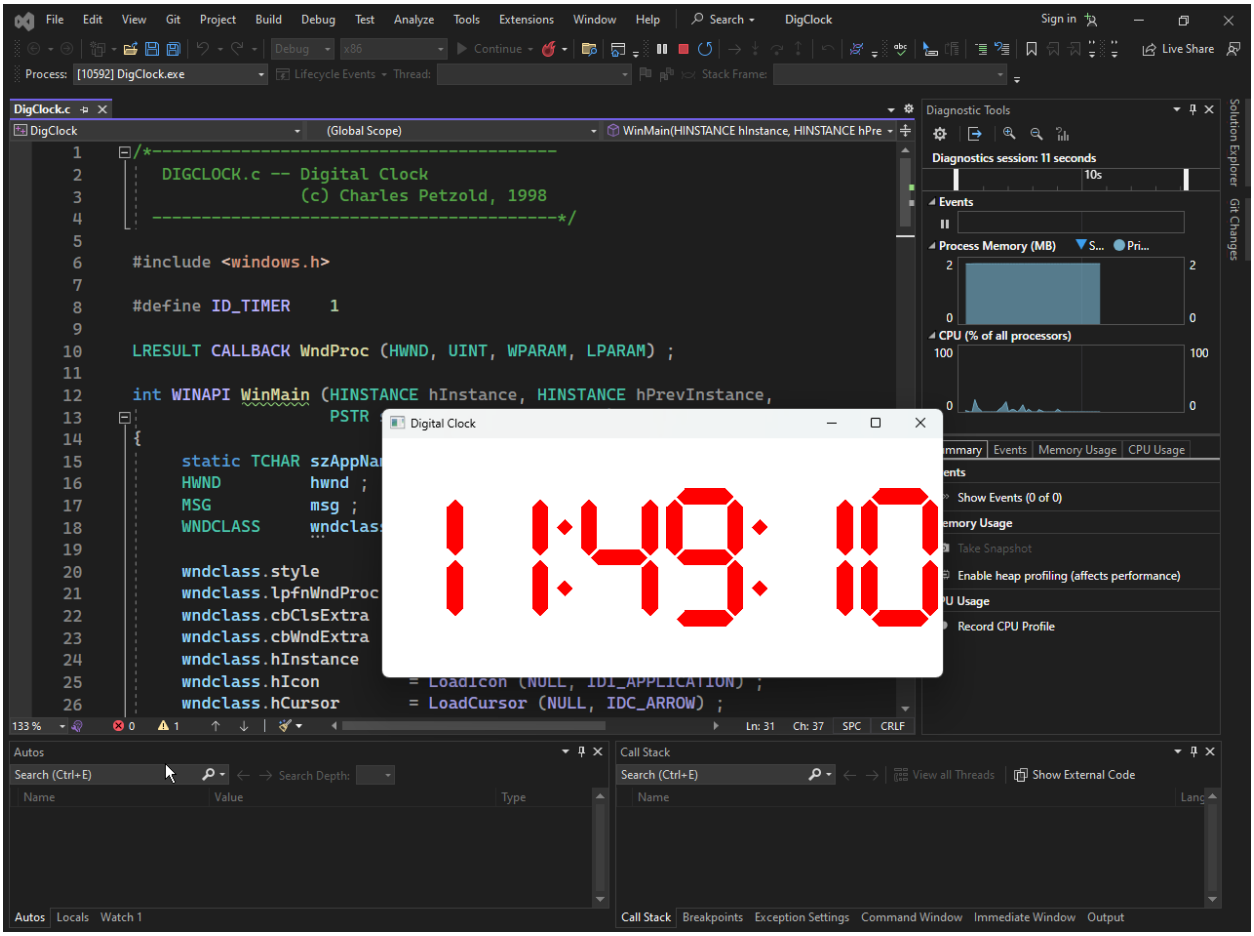
Method 3: Using a Call-Back Function

This method offers more flexibility than Method 1 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.

Method 4: 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.

Program code Chapter 8 DigClock…



The WndProc Function

The WndProc function is the main event handler for the DIGCLOCK window. It handles all messages sent to the window, including window creation, painting, and timer events.

Window Creation

When the window is first created, the WndProc function performs the following tasks:

- Creates a solid red brush using the CreateSolidBrush function.
- Sets a timer for 1000 milliseconds (1 second) using the SetTimer function.
- Calls the InvalidateRect function to redraw the window.

Setting Changes

When the system's locale settings change, the WndProc function is notified by the WM_SETTINGCHANGE message. It responds by:

- Calling the GetLocaleInfo function to retrieve the locale's time format.
- Setting the f24Hour flag to TRUE if the time format is 24-hour, or FALSE if it is 12-hour.
- Setting the fSuppress flag to TRUE if the leading zero on hours is suppressed, or FALSE if it is displayed.
- Calling the InvalidateRect function to redraw the window.

Window Sizing

When the window size changes, the WndProc function is notified by the WM_SIZE message. It responds by:

- Storing the new client width in the cxClient variable.
- Storing the new client height in the cyClient variable.

Timer Events

When the timer fires, the WndProc function is notified by the WM_TIMER message. It responds by:

- Calling the InvalidateRect function to redraw the window.

Painting

When the window needs to be repainted, the WndProc function is notified by the WM_PAINT message. It responds by:

- Obtaining a device context (HDC) for the window using the BeginPaint function.
- Setting the mapping mode to MM_ISOTROPIC using the SetMapMode function.
- Setting the window extent and viewport extent using the SetWindowExtEx and SetViewportExtEx functions.
- Setting the window and viewport origins using the SetWindowOrgEx and SetViewportOrgEx functions.
- Selecting the null pen and the red brush using the SelectObject function.
- Calling the DisplayTime function to display the current time.
- Calling the EndPaint function to release the device context.

Window Destruction

When the window is destroyed, the WndProc function performs the following tasks:

- Kills the timer using the KillTimer function.
- Deletes the red brush using the DeleteObject function.
- Posts a quit message to the message queue using the PostQuitMessage function.

This code is well-written and easy to understand. It uses a modular approach, with each function performing a specific task. The code is also commented extensively, which makes it easy to follow.

Main Points:

Red Brush Creation and Destruction

During the WM_CREATE message, DIGCLOCK's window procedure creates a red brush using the CreateSolidBrush function to paint the clock digits in red. This red brush is held until the window is destroyed, at which point it is deleted using the DeleteObject function during the WM_DESTROY message.

Timer Setup and Termination

The WM_CREATE message also provides an opportunity for DIGCLOCK to set a timer for 1-second intervals using the SetTimer function. This timer triggers the clock update at regular intervals. The timer is stopped during the WM_DESTROY message to prevent unnecessary updates after the window is closed.

Calls to GetLocaleInfo

The code mentions calls to the GetLocaleInfo function, but doesn't elaborate on their purpose. These calls are used to retrieve the locale's time format settings. Based on these settings, DIGCLOCK sets flags to determine whether to display the time in 24-hour format and whether to suppress leading zeros in hours.

Invalidation and Repainting

Upon receiving a WM_TIMER message, DIGCLOCK's window procedure invalidates the entire window using the InvalidateRect function. This forces the window to be repainted, but it's not the most efficient approach as it redraws the entire window every second, potentially causing flickering. A more optimized approach would involve invalidating only the specific areas of the window that need updating based on the time changes.

Mapping Mode and Axis Scaling

In the WM_PAINT message, DIGCLOCK sets the mapping mode to MM_ISOTROPIC using the SetMapMode function. This indicates that DIGCLOCK will use arbitrarily scaled axes that are equal in horizontal and vertical directions. The axes are set using a call to SetWindowExtEx to a size of 276 units horizontally and 72 units vertically. These dimensions are chosen based on the size and spacing of the clock digits.

Window Origin and Viewport Origin

The window origin is set to the point (138, 36) using the SetWindowOrgEx function. This represents the center of the window extents. The viewport origin is set to the center of the client area using SetViewportOrgEx. This ensures that the clock display is centered within the client area.

Red Brush Selection and NULL_PEN

The WM_PAINT processing sets the current brush to the red brush created earlier using the SelectObject function. This ensures that the clock digits are drawn in red. Additionally, the current pen is set to the NULL_PEN using SelectObject, indicating that no lines should be drawn, only filled shapes.

DisplayTime Function Call

The DisplayTime function is called to display the current time. This function takes the current time and the locale settings into account to format and display the time appropriately. It utilizes the red brush and NULL_PEN settings to draw the clock digits.

RETRIEVING THE CURRENT TIME WITH GETLOCALTIME

The DisplayTime function begins by calling the Windows function GetLocalTime, which retrieves the current system time and date. It takes a single argument, a pointer to a SYSTEMTIME structure, defined in WINBASE.H as follows:

```
typedef

struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

This SYSTEMTIME structure encapsulates both the date and time components. The month is represented in a 1-based format, with January as 1.

The day of the week, however, follows a 0-based indexing, where Sunday is represented by 0. The wDay field indicates the current day of the month, also adhering to the 1-based indexing scheme.

The SYSTEMTIME structure primarily serves as an input or output parameter for the GetLocalTime and GetSystemTime functions.

The GetSystemTime function provides the current Coordinated Universal Time (UTC), which closely approximates Greenwich Mean Time (GMT) – the time observed at Greenwich, England.

In contrast, the GetLocalTime function delivers the local time, adjusted according to the computer's configured time zone. The accuracy of these retrieved values hinges on the user's attentiveness in maintaining accurate timekeeping and establishing the correct time zone.

You can verify the time zone set on your machine by double-clicking the time display in the taskbar. A program tailored to synchronize your PC's clock with a precise, reliable time source on the Internet is presented in Chapter 23.

Windows additionally provides functions like SetLocalTime and SetSystemTime, along with a collection of other time-related functions, comprehensively documented in the Platform SDK/Windows Base Services/General Library/Time section.

Drawing Seven-Segment Digits without a Specialized Font

DIGCLOCK could have potentially simplified its implementation by utilizing a font that mimics a seven-segment display. However, it opts to handle this task itself using the Polygon function.

The DisplayDigit function within DIGCLOCK defines two arrays. The fSevenSegment array contains seven BOOL values for each of the ten decimal digits (0 through 9).

These values indicate which segments of the seven-segment display are illuminated (1) and which are not (0).

The ordering of these segments starts from top to bottom and from left to right. Each of the seven segments is represented by a six-sided polygon.

The ptSegment array is an array of POINT structures that specify the graphical coordinates of each vertex for each of the seven segments.

The following code snippet illustrates how each digit is drawn:

```
for (iSeg = 0; iSeg < 7; iSeg++) {
    if (fSevenSegment[iNumber][iSeg]) {
        Polygon(hdc, ptSegment[iSeg], 6);
    }
}
```

This code iterates through the seven segments for the given digit. For each segment that is illuminated (indicated by a 1 value in the fSevenSegment array), the Polygon function is called to draw the corresponding polygon using the coordinates specified in the ptSegment array. The polygon has six vertices, corresponding to the six sides of the seven-segment display.

Similarly, but with a simpler approach, the DisplayColon function draws the colons that separate the hour and minutes, and the minutes and seconds.

Positioning Digits and Colons

The digits are 42 units wide and the colons are 12 units wide. With six digits and two colons, the total width is 276 units, which corresponds to the value used in the SetWindowExtEx call.

Upon entering the DisplayTime function, the origin is positioned at the upper-left corner of the leftmost digit. The DisplayTime function calls DisplayTwoDigits, which in turn calls DisplayDigit twice.

After each call to DisplayDigit, the OffsetWindowOrgEx function is invoked to shift the window origin 42 units to the right. Likewise, the DisplayColon function moves the window origin 12 units to the right after drawing the colon.

This mechanism allows the functions to utilize the same coordinates for the digits and colons, regardless of their position within the window.

Handling 12-Hour and 24-Hour Formats

The only other intricate aspects of this code involve displaying the time in either 12-hour or 24-hour format and suppressing the leading zero for the hours if it's 0.

Leveraging Windows' International Support for Enhanced Date and Time Displays

While DIGCLOCK's method of displaying time is straightforward, for more complex date or time presentations, it's advisable to utilize Windows' built-in international support.

The most convenient approach for formatting dates and times is to employ the GetDateFormat and GetTimeFormat functions. These functions are extensively documented in the Platform SDK, but their purpose is also elaborated upon in the International Features section.

They accept SYSTEMTIME structures as input and format the date and time based on the user's preferences set in the Regional Settings control panel.

DIGCLOCK is unable to utilize the GetDateFormat function as it's designed to display only digits and colons. However, it should adhere to the user's preferences for 12-hour or 24-hour time formats and suppressing or retaining the leading zero in the hour display.

This information can be retrieved using the GetLocaleInfo function. Although GetLocaleInfo falls under the String Manipulation section in the Platform SDK, the identifiers used with this function are documented under National Language Support Constants.

DIGCLOCK initially invokes GetLocaleInfo twice during the WM_CREATE message processing. The first call employs the LOCALE_ITIME identifier to determine whether the 12-hour or 24-hour format should be used.

The second call utilizes the LOCALE_ITLZERO identifier to ascertain whether a leading zero should be suppressed in the hour display.

The GetLocaleInfo function returns all information as strings, but in most cases, it's straightforward to convert these strings to integer data if necessary. DIGCLOCK stores these settings in two static variables and passes them to the DisplayTime function.

If a user modifies any system settings, the WM_SETTINGCHANGE message is broadcast to all applications. DIGCLOCK handles this message by calling GetLocaleInfo again. This allows you to experiment with different settings using the Regional Settings control panel.

Theoretically, DIGCLOCK should also invoke GetLocaleInfo with the LOCALE_STIME identifier. This identifier returns the character that the user has selected to separate the hours, minutes, and seconds parts of the time.

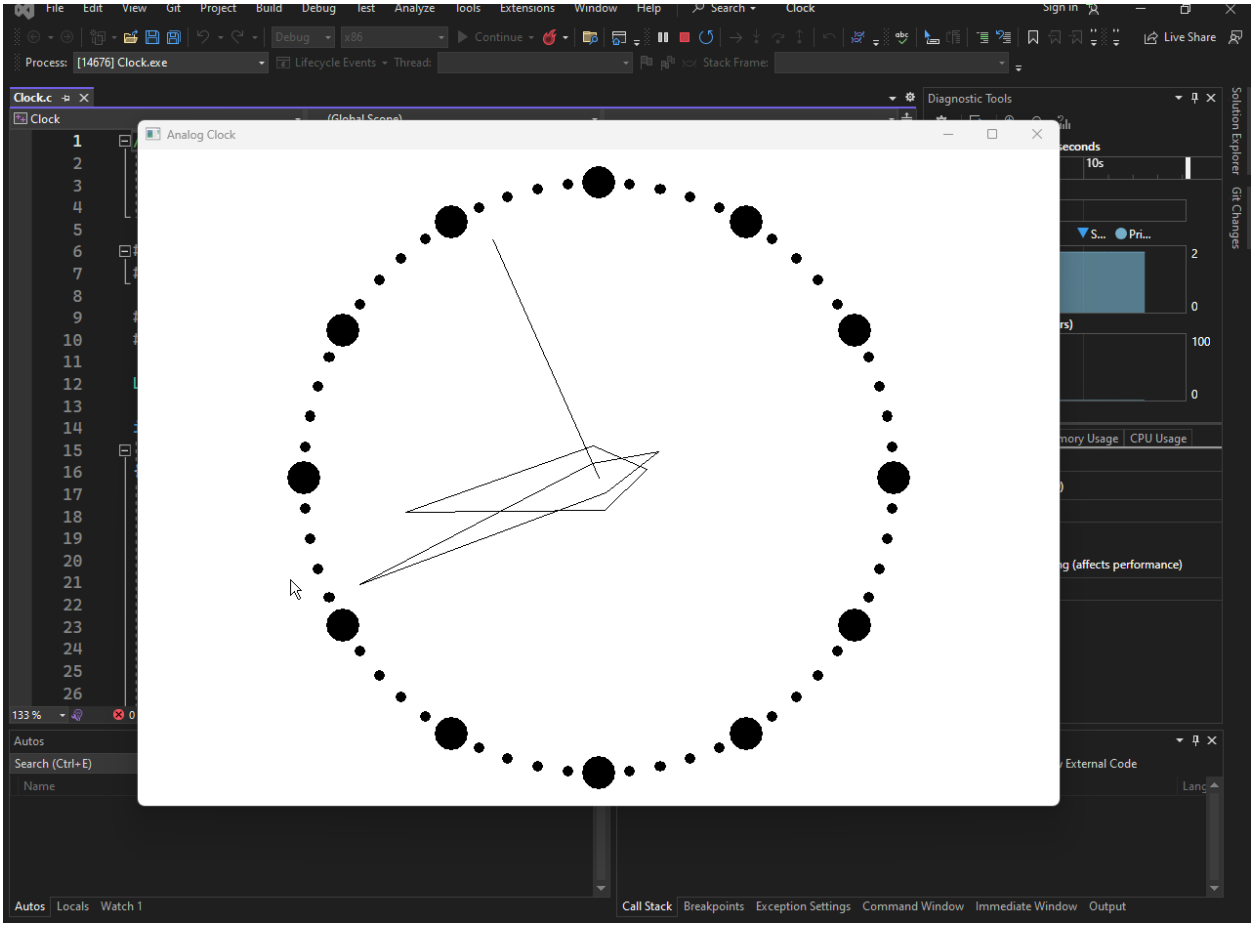
Since DIGCLOCK is configured to display only colons, this is what the user will see even if a different separator is preferred. To indicate whether the time is A.M. or P.M., an application can utilize GetLocaleInfo with the LOCALE_S1159 and LOCALE_S2359 identifiers.

These identifiers enable the program to obtain strings that are appropriate for the user's country and language.

DIGCLOCK could also be modified to process WM_TIMECHANGE messages, which notify applications of changes to the system date or time.

However, as DIGCLOCK is updated every second by WM_TIMER messages, this is unnecessary. Processing WM_TIMECHANGE messages would be more relevant for a clock that is updated every minute.

Analogue clock code next program….



Isotropic Mapping Mode and Setting Window and Viewport Extents

The isotropic mapping mode is once again an ideal choice for this application, as it allows for uniform scaling along both the x and y axes. The SetIsotropic function in CLOCK.C takes care of setting this mapping mode.

After establishing the isotropic mapping mode, the function sets the window extents to 1000 units, indicating the width and height of the drawing area. The viewport extents are set to half the width of the client area and the negative of half the height of the client area.

This positions the viewport origin at the center of the client area, creating a Cartesian coordinate system that extends 1000 units in all directions.

Clock.mp4

Rotating Points with RotatePoint Function

The RotatePoint function introduces trigonometry into the program. It takes three parameters: an array of points, the number of points in the array, and the angle of rotation in degrees.

The function rotates the provided points clockwise around the origin. For instance, rotating a point at (0, 100) (representing the 12:00 position) by 90 degrees would result in the point (100, 0) (representing the 3:00 position). The trigonometric formulas used for this rotation are as follows:

```
x' = x * cos (a) + y * sin (a)
y' = y * cos (a) - x * sin (a)
```

These formulas effectively shift the point along the circumference of a circle with a radius determined by the magnitude of the original point and an angle determined by the rotation angle.

Drawing Clock Face Dots with DrawClock Function

The DrawClock function is responsible for drawing the 60 dots on the clock face. Starting with the top dot (representing 12:00), each dot is positioned 900 units from the origin.

The first dot is placed at (0, 900), and each subsequent dot is rotated 6 degrees clockwise from the vertical. Twelve of these dots are 100 units in diameter, while the rest are 33 units in diameter. The Ellipse function is used to draw these dots.

Drawing Clock Hands with DrawHands Function

The DrawHands function handles the task of drawing the hour, minute, and second hands of the clock. The coordinates defining the shapes of the hands (when pointing straight up) are stored in an array of POINT structures.

Based on the current time, these coordinates are rotated using the RotatePoint function and then displayed using the Windows Polyline function.

It's important to note that the hour and minute hands are only drawn if the bChange parameter to DrawHands is set to TRUE.

When the clock hands are updated, the hour and minute hands typically don't need to be redrawn unless the time has changed significantly.

Window Procedure Handling

The window procedure manages the various messages received by the window.

During the WM_CREATE message, the window procedure retrieves the current time and stores it in the dtPrevious variable. This variable will be used later to determine if the hour or minute has changed since the previous update.

The first time the clock is drawn is during the initial WM_PAINT message. This involves calling the SetIsotropic, DrawClock, and DrawHands functions, with the latter setting the bChange parameter to TRUE to ensure all hands are drawn.

When a WM_TIMER message is received, the window procedure first obtains the new time and checks if the hour or minute hands need to be redrawn.

If so, all the hands are drawn with a white pen using the previous time, effectively erasing them. Otherwise, only the second hand is erased using the white pen. Finally, all the hands are drawn with a black pen, reflecting the updated time.

WhatClr program next, code inside the chapter 8 folder…

What color-displays
color under the curs

Understanding WHATCLR Program

The WHATCLR program is a simple graphical application that displays the RGB color value of the pixel currently under the mouse cursor in hexadecimal format. It utilizes the GetPixel function to retrieve the color information and displays it in a formatted manner.

Program Structure

The WHATCLR program is written in C and consists of five main functions:

FindWindowSize: This function determines the appropriate window size based on the system metrics and text metrics.

WndProc: This is the window procedure for the main window of the program. It handles all of the window messages that are sent to the window.

WinMain: This is the entry point for the program. It creates the main window and initializes the program.

CreateIC: This function creates a device context for the display.

GetPixel: This function retrieves the color of the specified pixel on the display.

Program Operation

The WinMain function creates the main window and sets up the timer to update the color display every 100 milliseconds.

The WM_TIMER message handler retrieves the current position of the mouse cursor and uses the GetPixel function to get the color of the pixel under the cursor.

If the color has changed since the last update, the window is invalidated to trigger a repaint.

The WM_PAINT message handler retrieves the new color value and formats it into a string.

Finally, the formatted color value is displayed in the center of the client area using the DrawText function.

Window Creation and Sizing

WHATCLR creates a non-sizeable window using the WS_BORDER window style in the CreateWindow function. This ensures that the window cannot be resized by the user and is only large enough to display the hexadecimal RGB value.

Device Context Handling

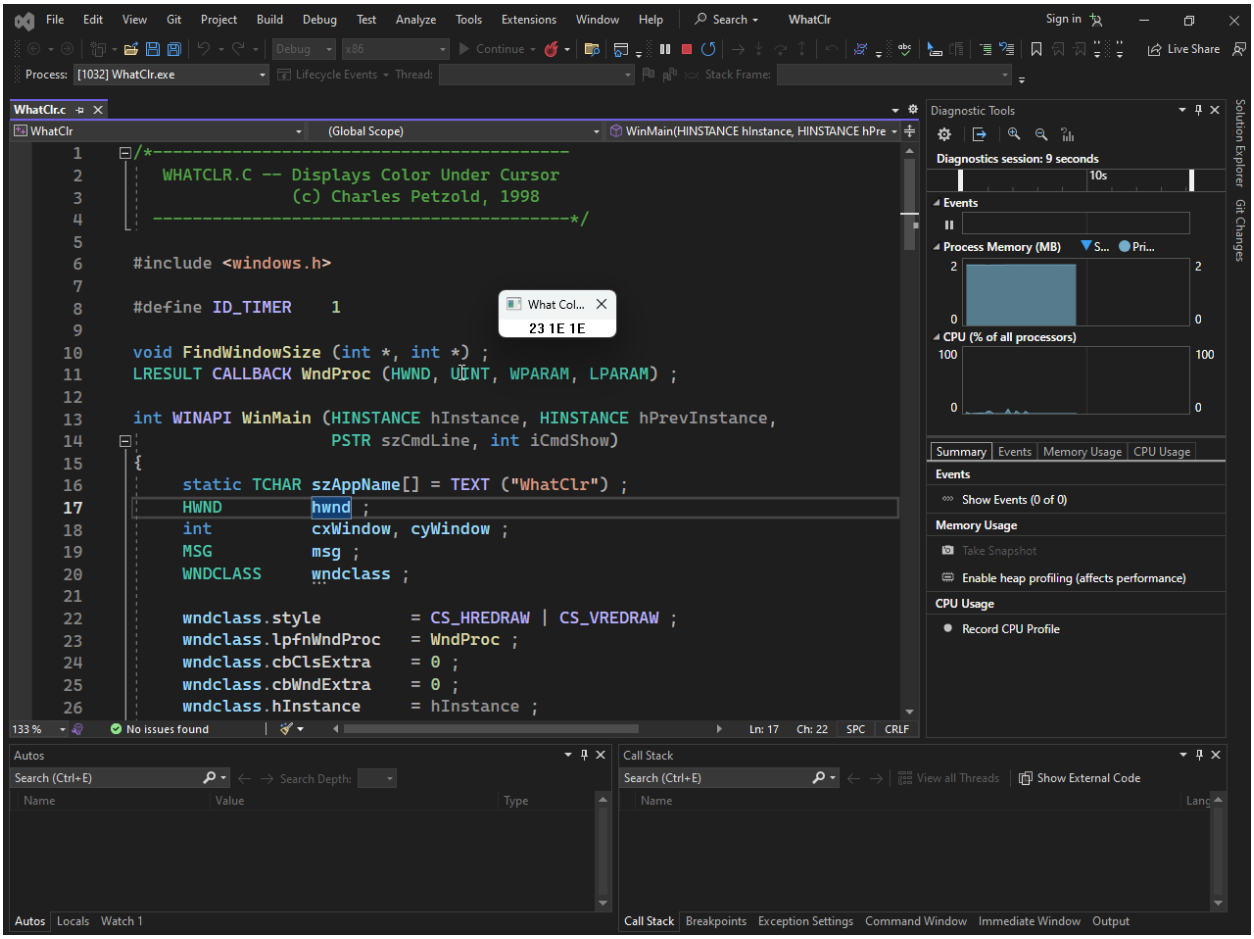
WHATCLR creates a device context for the entire video display during the WM_CREATE message. This device context is maintained throughout the program's lifetime and is used to retrieve the pixel color at the current mouse cursor position.

RGB Color Display

The RGB color value is displayed during the WM_PAINT message. The device context obtained during the WM_CREATE message is used to draw the color value in the center of the client area.

Drawing on the Screen with CreateDC

The CreateDC function can be used to obtain a device context for any part of the screen, not just the current application's window. However, it is generally considered impolite to draw on another application's window without permission.



Comparison with Previous Program

The WHATCLR program differs from the previous programs in the chapter in several ways:

It uses the GetPixel function to retrieve color information from the display, whereas previous programs used the GetText function to retrieve text information.

It updates the displayed color value continuously using a timer, whereas previous programs only updated the display when the window was resized or redrawn.

It displays the color value in a formatted manner, including the RGB components, whereas previous programs simply displayed the text information.

Overall, the WHATCLR program demonstrates the use of the GetPixel function and the WM_TIMER message to create a simple but useful graphical application.