Lab Assignment 12: Event-driven Programming for Windows Forms Apps. in C#

John Twipraham Debbarma
Roll Number: 22110108
CS202 Software Tools and Techniques for CSE

April 25, 2025

Abstract

This report presents a complete solution for Lab Assignment 12 focusing on event-driven programming in C# Windows Forms applications. Through this lab, I have developed a practical understanding of the event-driven paradigm and analyzed how control flows in response to user interactions and application states. The implementation of a time-based alarm system demonstrates core event-driven programming concepts in both console and Windows Forms environments.

Environment: Windows
IDE: Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
Framework: .NET 8.0

Contents

1 Introduction, Setup, and Tools

1.1 Background and Motivation

Event-driven programming represents a paradigm where the flow of a program is determined by events such as user actions, sensor outputs, or messages from other programs. This approach is fundamental to modern graphical user interfaces and interactive applications, making it an essential concept in software development.

C# and the .NET platform provide robust support for event-driven programming through the Windows Forms framework, offering a comprehensive set of tools for building responsive, user-friendly applications. As part of the CS202 course, this lab assignment aimed to provide hands-on experience with event-driven programming concepts, focusing on implementing a practical application that responds to both time-based events and user interactions.

The lab assignment focuses on developing an alarm clock application in two forms: a console-based implementation and a Windows Forms-based graphical application. This progression allowed me to understand the core concepts of event-driven programming first in a simpler console environment, before applying them in a more complex graphical interface.

1.2 Development Environment

For this lab assignment, I set up a development environment consisting of:

- **Operating System:** Windows 11
- **IDE:** Visual Studio 2022 Community Edition
- **Framework:** .NET 8.0
- **Programming Language:** C# (latest stable version)

Visual Studio 2022 provides comprehensive tools for C# development, including an integrated debugger, design-time support for Windows Forms, and IntelliSense code completion, making it an ideal environment for this lab assignment.

1.3 Project Structure

I structured my solution to contain two separate projects:

- **AlarmConsoleApp:** A console application implementing a basic alarm clock using the publisher/subscriber model for event handling
- **AlarmWindowsFormsApp:** A Windows Forms application extending the console concept with a graphical interface and visual feedback

This approach allowed me to focus on the core event-driven programming concepts first, before adding the complexity of a graphical user interface. The separation also made it easier to compare the implementation differences between console and Windows Forms environments.

2 Methodology and Execution

2.1 Understanding Event-Driven Programming

Before implementing the alarm applications, I studied the fundamentals of event-driven programming:

- **Events:** Signals that indicate something has happened, such as user input or a timer tick
- **Event Sources (Publishers):** Objects that generate events when certain conditions are met
- **Event Handlers (Subscribers):** Methods that respond to events when they occur
- **Delegates:** Type-safe function pointers that connect events to event handlers

The publisher/subscriber model is central to event-driven programming in C#, providing a clean separation between the code that generates events and the code that responds to them. This separation enhances modularity and makes the code more maintainable.

2.2 Activity 1: Console Application with Event-Driven Alarm

For the first task, I developed a console application that accepts a time input from the user and triggers an alarm when the current system time matches the target time.

2.2.1 Setting Up the Console Project

I followed these steps to create the console application:

1. Launched Visual Studio 2022
2. Selected "Create a new project"
3. Chose "Console App (.NET)" with C# as the language
4. Named the project "AlarmConsoleApp"
5. Selected .NET 8.0 (Long Term Support) as the target framework
6. Clicked "Create" to generate the project

2.2.2 Implementing the Publisher/Subscriber Model

I implemented the publisher/subscriber pattern using the following components:

```
// I'm creating a delegate for the alarm event
public delegate void AlarmEventHandler(object source,
    EventArgs args);

// I'm defining the publisher class that will raise the alarm
     event
public class AlarmClock
{
    // I'm declaring the event using the delegate
    public event AlarmEventHandler RaiseAlarm;

    private DateTime targetTime;
    private bool isRunning = false;

    // I'm creating a method to set the alarm time
    public void SetAlarm(DateTime time)
    {
        targetTime = time;
        Console.WriteLine($"Alarm set for: {targetTime.
    ToString("HH:mm:ss")}");
    }

    // I'm implementing a method to start checking the time
    public void StartMonitoring()
    {
        isRunning = true;
        Console.WriteLine("Alarm monitoring started...");

```

```
        // I'm using a while loop to continuously check the current time
        while (isRunning)
        {
            DateTime currentTime = DateTime.Now;

            // I'm displaying the current time to show the progress
            Console.Write($"\rCurrent time: {currentTime.ToString("HH:mm:ss")}");

            // I'm checking if current time matches the target time
            if (currentTime.Hour == targetTime.Hour &&
                currentTime.Minute == targetTime.Minute &&
                currentTime.Second == targetTime.Second)
            {
                // I'm raising the event when times match
                OnAlarmTime();
                isRunning = false;
            }

            // I'm adding a small delay to prevent excessive CPU usage
            Thread.Sleep(100);
        }
    }

    // I'm creating a protected method to raise the event
    protected virtual void OnAlarmTime()
    {
        // I'm checking if there are any subscribers before raising the event
        if (RaiseAlarm != null)
        {
            RaiseAlarm(this, EventArgs.Empty);
        }
    }
}
```

Listing 1: AlarmClock Publisher and Event Definition

```
// I'm defining the subscriber class that will handle the alarm event
public class AlarmHandler
{
    // I'm creating the event handler method
    public void Ring_alarm(object source, EventArgs e)
    {
```

```
7           Console.WriteLine();
8           Console.WriteLine("
    ==============================================");
9           Console.WriteLine("ALARM! ALARM! It's time to wake up
    !");
10          Console.WriteLine("
    ==============================================");
11      }
12  }
```

Listing 2: AlarmHandler Subscriber Implementation

2.2.3 Main Program Implementation

The main program brings everything together, creating instances of the publisher and subscriber, connecting them through the event, and handling user input:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           Console.WriteLine("Alarm Clock Application");
6           Console.WriteLine("======================");
7
8           // I'm creating instances of the publisher and
    subscriber
9           AlarmClock alarmClock = new AlarmClock();
10          AlarmHandler handler = new AlarmHandler();
11
12          // I'm subscribing to the event
13          alarmClock.RaiseAlarm += handler.Ring_alarm;
14
15          // Getting the time input from the user
16          DateTime targetTime;
17          bool validInput = false;
18
19          while (!validInput)
20          {
21              Console.Write("\nEnter alarm time (HH:MM:SS): ");
22              string timeInput = Console.ReadLine();
23
24              // Validating the user input
25              if (DateTime.TryParseExact(timeInput, "HH:mm:ss",
    null,
26                  System.Globalization.DateTimeStyles.None, out
    targetTime))
27              {
```

```
28                  // Creating a new DateTime with today's date
    and user's time
29                  DateTime today = DateTime.Today;
30                  targetTime = new DateTime(
31                      today.Year, today.Month, today.Day,
32                      targetTime.Hour, targetTime.Minute,
    targetTime.Second
33                  );
34
35                  // Checking if the time is in the past
36                  if (targetTime < DateTime.Now)
37                  {
38                      Console.WriteLine("The time you entered
    is in the past. Please enter a future time.");
39                  }
40                  else
41                  {
42                      validInput = true;
43                      alarmClock.SetAlarm(targetTime);
44                  }
45              }
46              else
47              {
48                  Console.WriteLine("Invalid time format.
    Please use HH:MM:SS format.");
49              }
50          }
51
52          // Starting the alarm monitoring
53          alarmClock.StartMonitoring();
54
55          Console.WriteLine("\nPress any key to exit...");
56          Console.ReadKey();
57      }
58  }
```

Listing 3: Main Program for Console Application

2.2.4 Console Application Output

When running the console application, the user is prompted to enter a time in HH:MM:SS format. After validating the input, the application continuously displays the current system time and checks if it matches the target time. When the times match, the alarm event is raised, triggering the alarm message.

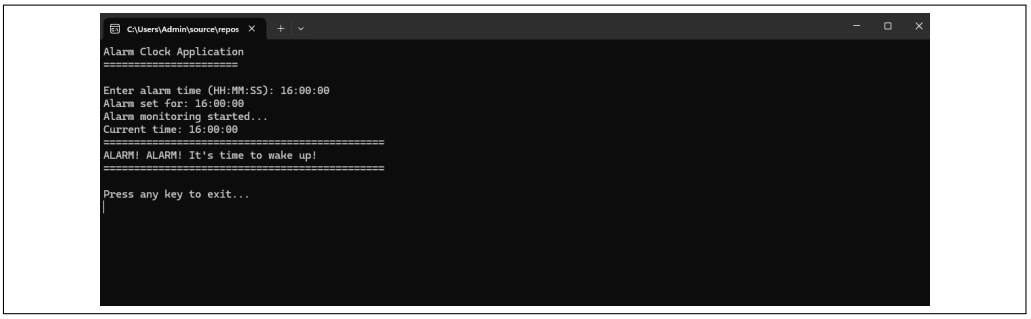

Figure 1: Console Application Output

2.3 Activity 2: Windows Forms Application with Visual Feedback

For the second task, I converted the console application to a Windows Forms application with a graphical user interface, adding visual feedback through background color changes.

2.3.1 Setting Up the Windows Forms Project

I created a new Windows Forms project with the following steps:

1. Launched Visual Studio 2022
2. Selected "Create a new project"
3. Chose "Windows Forms App (.NET)" with C# as the language
4. Named the project "AlarmWindowsFormsApp"
5. Selected .NET 8.0 (Long Term Support) as the target framework
6. Clicked "Create" to generate the project

2.3.2 Designing the Form Interface

I designed the form interface using the Visual Studio Designer, adding the following controls:

- A title label displaying "Alarm Clock Application"
- An instruction label prompting the user to enter time in HH:MM:SS format

- A text box for time input
- A button to start the alarm
- A label to display the current system time
- A label to display the alarm status

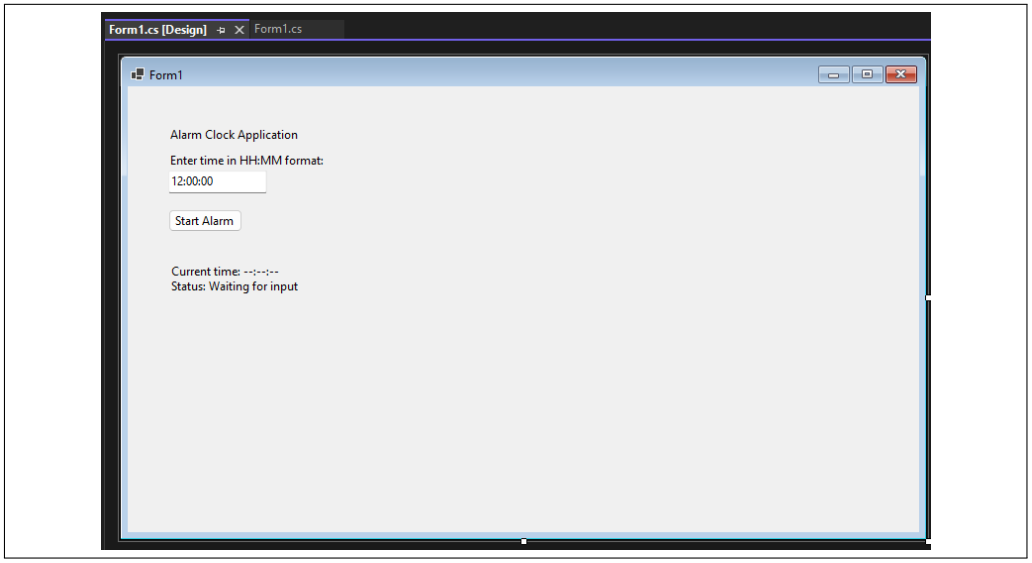

Figure 2: Windows Forms Design

2.3.3 Implementing the Windows Forms Application

I implemented the Windows Forms version of the alarm application, adapting the publisher/subscriber model to work with the Windows Forms environment:

```
public partial class Form1 : Form
{
    // I'm declaring delegate and event for the alarm
    public delegate void AlarmEventHandler(object source,
   EventArgs args);
    public event AlarmEventHandler RaiseAlarm;

    // I'm creating variables to track the alarm state
    private DateTime targetTime;
    private bool isRunning = false;
    private System.Windows.Forms.Timer timer;
    private Random random;

```

```
13      public Form1()
14      {
15          InitializeComponent();
16
17          // I'm initializing the timer for checking time and
    changing colors
18          timer = new System.Windows.Forms.Timer();
19          timer.Interval = 1000; // 1 second
20          timer.Tick += Timer_Tick;
21
22          // I'm initializing the random number generator for
    colors
23          random = new Random();
24
25          // I'm subscribing to my own event (self-subscription
     pattern)
26          RaiseAlarm += Ring_alarm;
27      }
28
29      private void btnStart_Click(object sender, EventArgs e)
30      {
31          // I'm validating the user input
32          if (ValidateTimeInput(txtTimeInput.Text, out
    targetTime))
33          {
34              // I'm checking if the time is in the past
35              if (targetTime < DateTime.Now)
36              {
37                  MessageBox.Show("The time you entered is in
    the past. Please enter a future time.",
38                      "Invalid Time", MessageBoxButtons.OK,
    MessageBoxIcon.Warning);
39                  return;
40              }
41
42              // I'm updating the UI to show the alarm is
    running
43              lblStatus.Text = $"Status: Alarm set for {
    targetTime.ToString("HH:mm:ss")}";
44              btnStart.Enabled = false;
45              txtTimeInput.Enabled = false;
46              isRunning = true;
47
48              // I'm starting the timer
49              timer.Start();
50          }
51          else
52          {
```

```
            MessageBox.Show("Invalid time format. Please use HH:MM:SS format.",
                "Invalid Format", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }

    private void Timer_Tick(object sender, EventArgs e)
    {
        // I'm updating the current time display
        DateTime currentTime = DateTime.Now;
        lblCurrentTime.Text = $"Current time: {currentTime.ToString("HH:mm:ss")}";

        // I'm checking if the alarm should go off
        if (isRunning)
        {
            // I'm changing the background color every second
            ChangeBackgroundColor();

            // I'm checking if current time matches the target time
            if (currentTime.Hour == targetTime.Hour &&
                currentTime.Minute == targetTime.Minute &&
                currentTime.Second == targetTime.Second)
            {
                // I'm raising the event when times match
                if (RaiseAlarm != null)
                {
                    RaiseAlarm(this, EventArgs.Empty);
                }
            }
        }
    }

    private void ChangeBackgroundColor()
    {
        // I'm generating a random light color for better text visibility
        int red = random.Next(100, 256);
        int green = random.Next(100, 256);
        int blue = random.Next(100, 256);

        this.BackColor = Color.FromArgb(red, green, blue);
    }

    private void Ring_alarm(object source, EventArgs e)
    {
        // I'm stopping the timer and color changes
```

```
97          timer.Stop();
98          isRunning = false;
99
100         // I'm resetting the form background to default
101         this.BackColor = SystemColors.Control;
102
103         // I'm updating the UI
104         lblStatus.Text = "Status: Alarm triggered!";
105         btnStart.Enabled = true;
106         txtTimeInput.Enabled = true;
107
108         // I'm displaying the alarm message with an option to
     exit
109         DialogResult result = MessageBox.Show("ALARM! ALARM!
    It's time to wake up!\n\nDo you want to exit the
    application?",
110             "Alarm", MessageBoxButtons.YesNo, MessageBoxIcon.
    Information);
111
112         // I'm checking if the user clicked "Yes" to exit
113         if (result == DialogResult.Yes)
114         {
115             this.Close();
116         }
117     }
118
119     private bool ValidateTimeInput(string timeInput, out
    DateTime result)
120     {
121         // I'm validating the time format
122         bool isValid = DateTime.TryParseExact(
123             timeInput,
124             "HH:mm:ss",
125             null,
126             System.Globalization.DateTimeStyles.None,
127             out DateTime parsedTime);
128
129         if (isValid)
130         {
131             // I'm creating a new DateTime with today's date
    and user's time
132             DateTime today = DateTime.Today;
133             result = new DateTime(
134                 today.Year, today.Month, today.Day,
135                 parsedTime.Hour, parsedTime.Minute,
    parsedTime.Second
136             );
137             return true;
138         }
```

```
139
140             result = DateTime.Now;
141             return false;
142         }
143 }
```

Listing 4: Windows Forms Implementation

2.3.4 Windows Forms Application in Action

The Windows Forms application provides a more interactive user experience compared to the console application. The user enters the target time in the text box and clicks the "Start Alarm" button. The form's background color changes every second, providing visual feedback that the alarm is running. When the target time is reached, the color changes stop, and a message box appears with the alarm message and an option to exit.

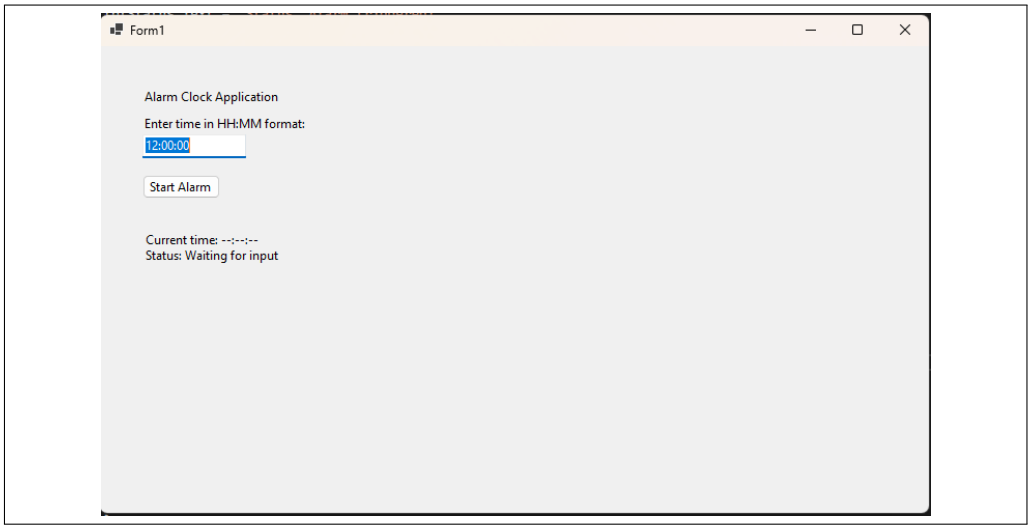

Figure 3: Windows Forms Application Initial State

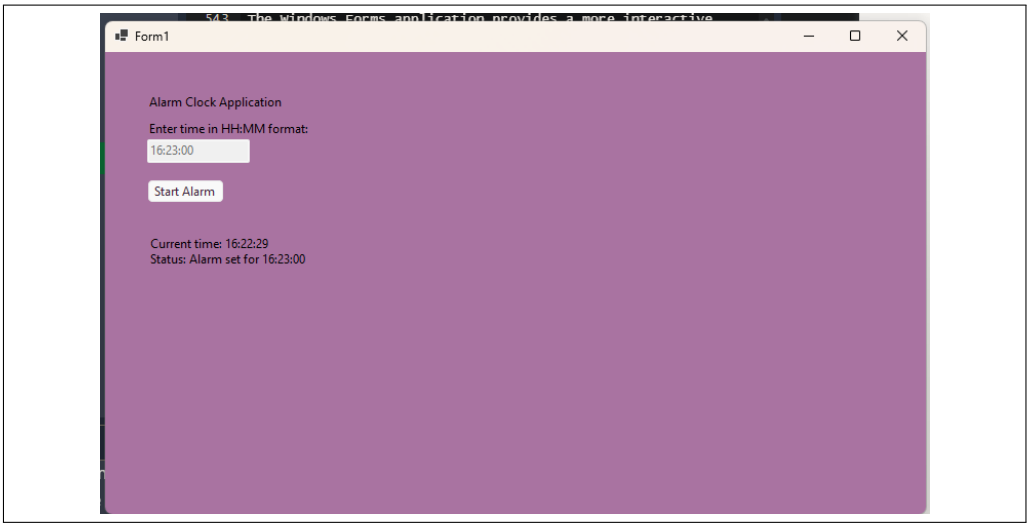

Figure 4: Windows Forms Application Running with Color Change

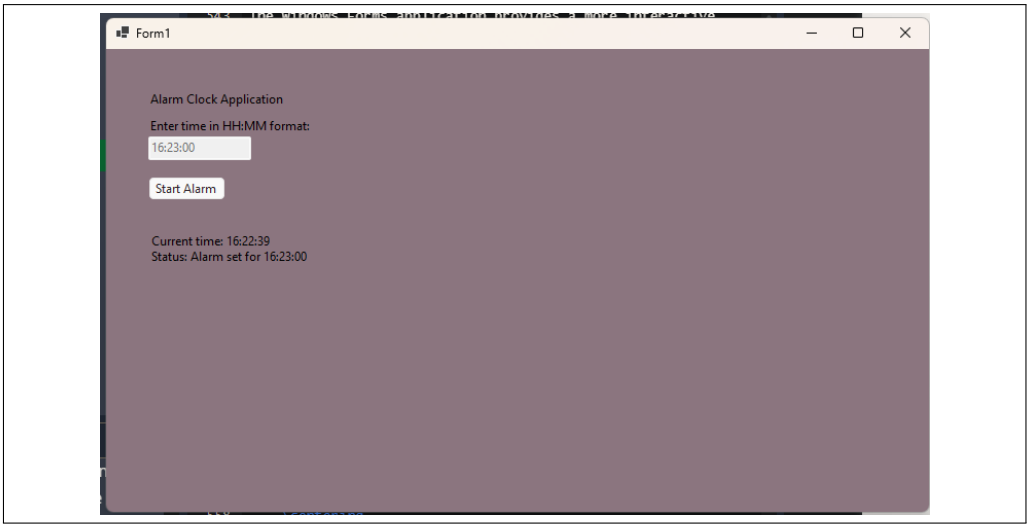

Figure 5: Windows Forms Application Running with Color Change

Figure 6: Windows Forms Application Running with Color Change

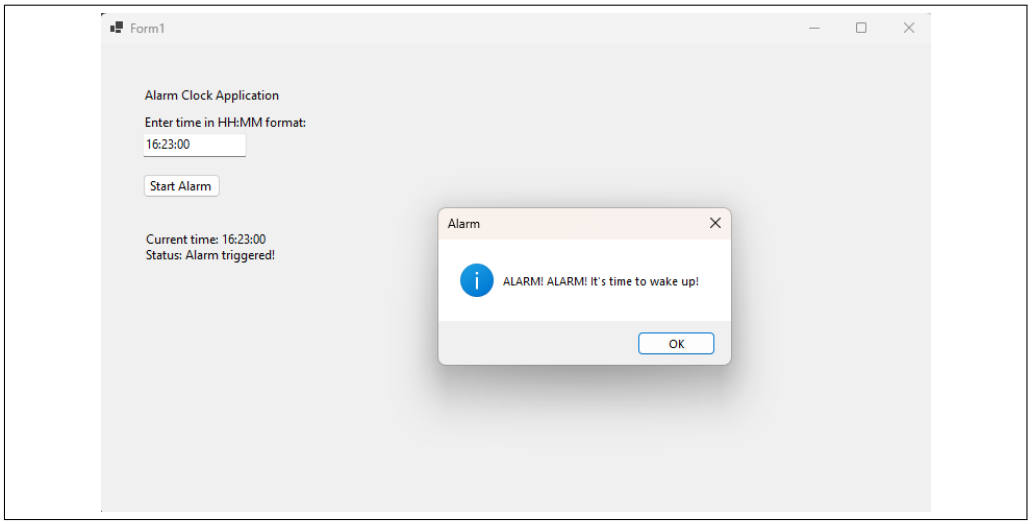

Figure 7: Windows Forms Application Alarm Triggered

I also incorporated in the code (as shown above) to warn for a past time input and an invalid time input as:

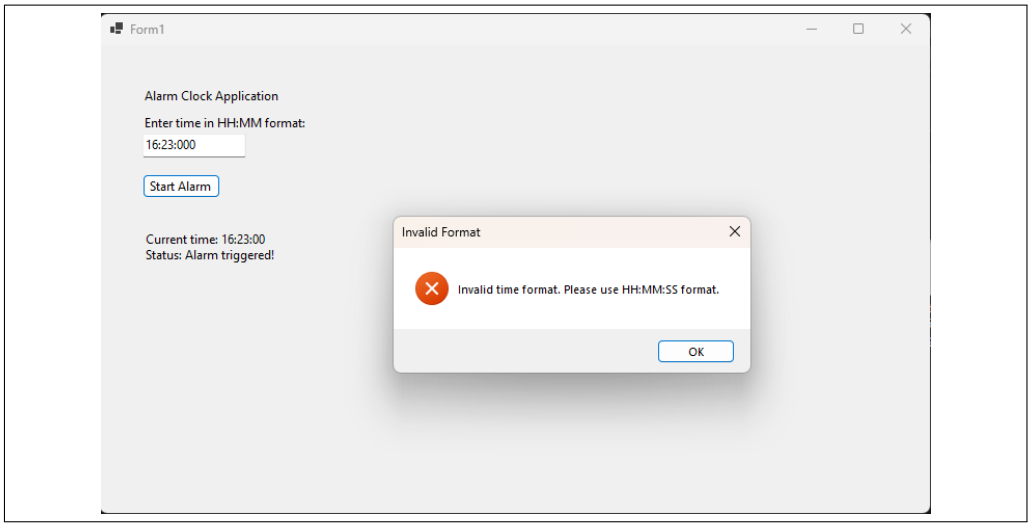

Figure 8: Windows Forms Application Alarm Warning Invalid Time

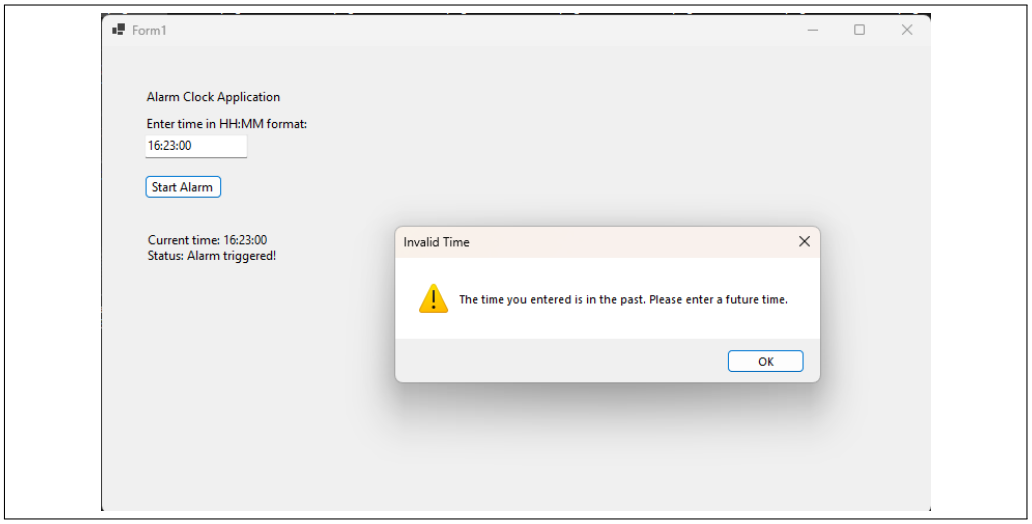

Figure 9: Windows Forms Application Alarm Warning Past Time

3 Results and Analysis

3.1 Implementation Results

I successfully implemented both the console application and Windows Forms application as specified in the lab assignment. Both applications demonstrate

the publisher/subscriber model of event-driven programming, with the Windows Forms application adding visual elements and UI interactivity.

3.1.1 Console Application Results

The console application effectively demonstrates:

- Event-driven programming using custom events and delegates
- The publisher/subscriber pattern with clear separation of concerns
- Time-based event triggering using system time comparison
- User input validation and error handling

The application accepts user input for the target time, continuously checks the current system time, and triggers the alarm event when the times match. The simplicity of the console interface allows the focus to remain on the event-driven programming concepts rather than UI details.

3.1.2 Windows Forms Application Results

The Windows Forms application builds upon the console application concepts and adds:

- Graphical user interface with interactive controls
- Visual feedback through background color changes
- Message box dialogs for user interaction
- UI state management based on the application's state

The Windows Forms implementation demonstrates how event-driven programming is central to modern graphical applications, with events triggered by both user actions (button clicks) and system states (timer ticks).

3.2 Debugging and Troubleshooting

During the development process, I encountered and resolved several issues:

1. **Timer Ambiguity**: Initially, I encountered a compilation error due to an ambiguous reference between System.Windows.Forms.Timer and System.Threading.Timer. I resolved this by explicitly specifying System.Windows.Forms.Timer, which is designed to work with UI applications.

2. **UI Threading Issues**: In the Windows Forms application, I initially tried to use Thread.Sleep() for timing, similar to the console application. However, this caused the UI to freeze. I resolved this by switching to a Timer control, which properly respects the UI thread's event loop.

3.3 Key Insights from Implementation

Implementing these applications provided several key insights into event-driven programming:

- **Separation of Concerns**: The publisher/subscriber model enforces a clean separation between event generation and event handling, making the code more modular and maintainable.
- **Different Event Sources**: Events can be triggered by various sources, including user actions (button clicks), system states (time matches), and timer intervals.
- **UI vs. Console Differences**: Event-driven programming in Windows Forms requires careful consideration of the UI thread, whereas console applications can use simpler synchronous approaches.
- **State Management**: Event-driven applications need clear state management to track whether the alarm is running, what the target time is, and how the UI should reflect the current state.

4 Discussion and Conclusion

4.1 Challenges

Throughout the lab, I encountered several technical and conceptual challenges:

1. **Understanding Event Flow**: Initially, it was challenging to understand how events flow from publishers to subscribers, especially when the same class both publishes and subscribes to its own events (as in the Windows Forms application).
2. **Timer vs. Loop**: Deciding between a continuous loop (as used in the console application) and a timer control (as used in the Windows Forms application) required understanding how each approach affects the application's responsiveness.

3. **UI Thread Management**: The Windows Forms application required careful consideration of the UI thread to ensure the interface remained responsive while performing time-checking operations.

4. **Exit Strategy**: Implementing a clean exit strategy for the Windows Forms application after the alarm triggers required understanding how to properly terminate the application while giving the user control.

4.2 Reflections

Working through this lab assignment has led to several important reflections:

1. **Value of Event-Driven Design**: The event-driven paradigm provides a natural way to structure applications that respond to user actions and system states. This approach is particularly valuable for interactive applications where the order of events cannot be predicted in advance.

2. **UI Design Considerations**: Designing the Windows Forms interface required thinking not just about functionality but also about user experience. The changing background color provides immediate visual feedback that the alarm is running, enhancing the user's understanding of the application state.

3. **Importance of Validation**: Both applications implement input validation to ensure the user enters a valid time format and a future time. This defensive programming approach prevents potential runtime errors and improves the user experience.

4. **Platform-Specific Considerations**: Moving from a console application to a Windows Forms application required adapting the event-driven approach to respect the UI thread's event loop, demonstrating how the underlying platform influences implementation details.

4.3 Lessons Learned

This lab assignment provided several valuable lessons that will benefit my future software development work:

1. **Event-Driven Paradigm**: I gained a practical understanding of how events drive application flow, which is crucial for developing modern interactive applications.

2. **UI Thread Management**: I learned the importance of respecting the UI thread in Windows Forms applications, using appropriate mechanisms (like Timer) for background operations.

3. **Defensive Programming**: Implementing validation and error handling reinforced the importance of anticipating user input errors and handling them gracefully.

4. **Visual Feedback**: The background color changes in the Windows Forms application demonstrated how visual feedback enhances the user experience by communicating the application's state.

4.4 Value of Practical Implementation

The practical implementation of both a console application and a Windows Forms application provided a comprehensive understanding of event-driven programming that would be difficult to achieve through theoretical study alone. By working through the entire development process, from concept to functional application, I gained insights into:

- How events flow through an application
- The relationship between publishers and subscribers
- The role of delegates in type-safe event handling
- Platform-specific considerations for different application types

These insights will be valuable not only for future C# development but also for working with event-driven systems in other programming languages and environments.

4.5 Conclusion

This lab assignment provided a comprehensive introduction to event-driven programming in C# through the implementation of both console and Windows Forms applications. The alarm clock application served as a practical example of how events drive application flow, with events triggered by both time conditions and user interactions.

Overall, this lab assignment has provided valuable practical experience with event-driven programming concepts that will be applicable across a wide range of application types and programming environments in my future software development work.

References

[1] Lab Manual