Event-Driven Programming in C# Windows Forms

Bhoumik Patidar

April 22, 2025

1 Introduction, Setup, and Tools

1.1 Overview

This report documents the implementation of an alarm clock application using event-driven programming in C#, covering both console and Windows Forms versions. The assignment demonstrates the publisher-subscriber pattern and event handling in .NET applications.

1.2 Environment Setup

The development environment consisted of:

- Windows 10 Operating System
- Visual Studio 2022 Community Edition
- .NET 6.0 SDK
- Windows Forms App (.NET Framework) template

1.3 Learning Objectives Achieved

- Implemented publisher-subscriber pattern in console application
- Created event-driven Windows Forms application
- Demonstrated timer-based events and UI updates
- Validated user input in both implementations

2 Methodology and Execution

2.1 Part 1: Console Application

2.1.1 Design Approach

The console application follows the publisher-subscriber model with these components:

- **Publisher**: `AlarmClock` class that monitors system time
- **Event**: `RaiseAlarm` triggered when target time matches system time
- **Subscriber**: `RingAlarm` method that handles the event

2.1.2 Key Code Implementation

```
class AlarmClock
{
    private readonly string _targetTime;
    public event EventHandler? RaiseAlarm;

    protected virtual void OnRaiseAlarm()
    {
        RaiseAlarm?.Invoke(this, EventArgs.Empty);
    }

    public void Start()
    {
        while (true)
        {
            if (DateTime.Now.ToString("HH:mm:ss") == _targetTime)
            {
                OnRaiseAlarm();
                break;
            }
            Thread.Sleep(1000);
        }
    }
}
```

Listing 1: Publisher class implementation

```
static void Main(string[] args)
{
    var alarm = new AlarmClock(input);
    alarm.RaiseAlarm += RingAlarm;
    alarm.Start();
}

private static void RingAlarm(object? sender, EventArgs e)
{
    Console.WriteLine("          Alarm! Time's up!          ");
}
```

Listing 2: Subscriber implementation

2.1.3 Execution Flow

1. User enters time in HH:mm:ss format
2. `AlarmClock` starts monitoring system time
3. When times match, `RaiseAlarm` event is triggered
4. `RingAlarm` handler executes with alarm message

2.2 Part 2: Windows Forms Application

2.2.1 Design Approach

The Windows Forms version enhances the console application with:

- Graphical user interface with input validation
- Timer-based background color changes
- Message box notification on alarm trigger
- Proper event handling for button clicks

2.2.2 Key Components

```
private void InitializeComponents()
{
    // Form settings
    this.Text = "Alarm Clock";
    this.ClientSize = new Size(320, 120);

    // Timer setup
    timer = new System.Windows.Forms.Timer()
    {
        Interval = 1000,
        Enabled = false
    };
    timer.Tick += Timer_Tick;
}
```

Listing 3: Form initialization

```
private void ButtonStart_Click(object? sender, EventArgs e)
{
    if (!TimeSpan.TryParseExact(
            textBoxTime.Text.Trim(),
            "hh\\:mm\\:ss",
            null,
            out targetTime))
    {
        MessageBox.Show("Invalid time format");
        return;
    }
    timer.Start();
}

private void Timer_Tick(object? sender, EventArgs e)
{
    this.BackColor = Color.FromArgb(
        rng.Next(256), rng.Next(256), rng.Next(256));

    if (DateTime.Now.TimeOfDay >= targetTime)
    {
        timer.Stop();
        MessageBox.Show("    Alarm! Time's up!    ");
    }
}
```

Listing 4: Event handlers

2.2.3 Execution Flow

1. User enters time in textbox and clicks Start
2. Input validation checks HH:mm:ss format
3. Timer begins 1-second interval ticks
4. Each tick changes form background color randomly
5. On target time match:
    - Timer stops hanges stop
    - Message box displays alarm

3 Results and Analysis

3.1 Console Application Results

- Successfully implemented event-driven architecture

- Proper time validation using `TimeSpan.TryParse`
- Clean separation of publisher and subscriber
- Thread.Sleep used for periodic checking (1 second intervals)

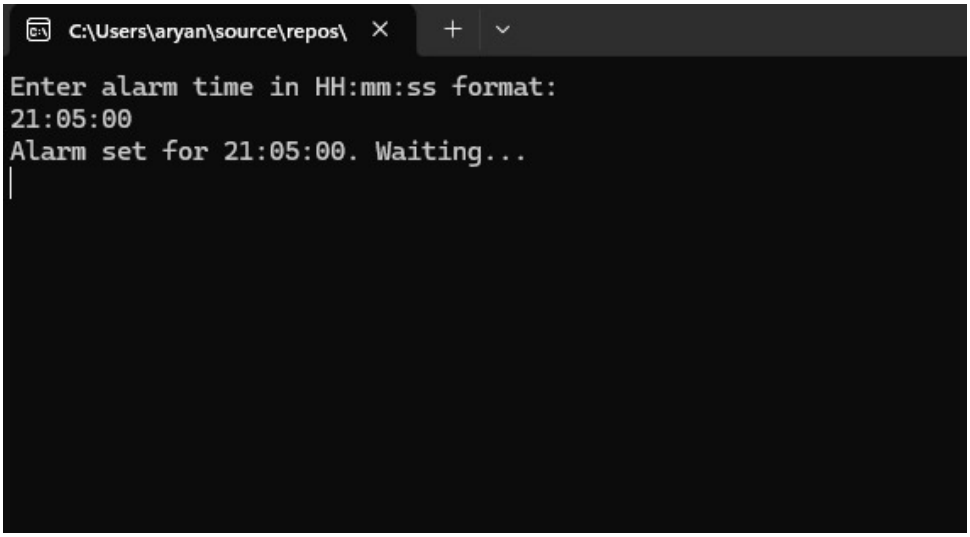

Figure 1: The program displays the time for which the clock is set for and is waiting for that time to come

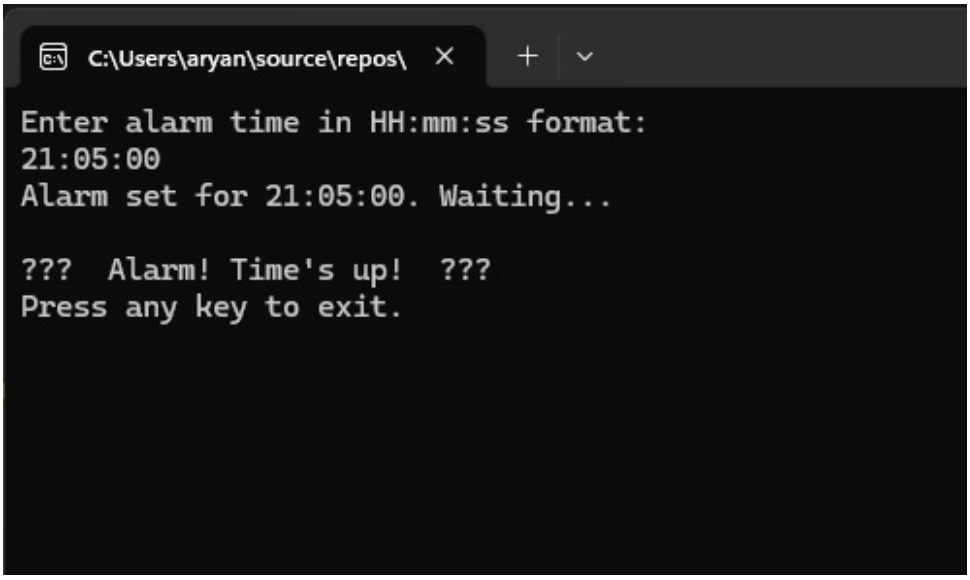

Figure 2: As the time hits you get the message Alarm

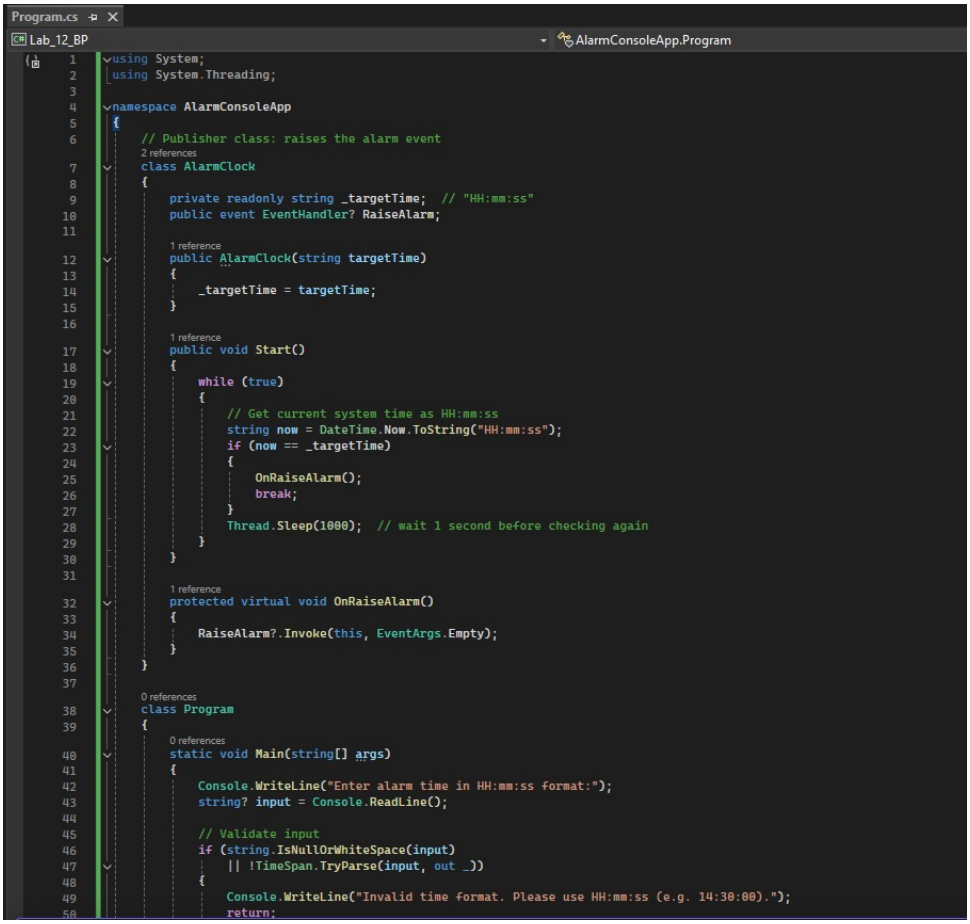

Figure 3: Code screenshot part 1

Figure 4: Code screenshot part 2

3.2 Windows Forms Results

- Fully functional GUI with proper input validation
- Smooth background color transitions (1 second intervals)
- Correct alarm triggering with message box
- Disabled controls during active alarm monitoring

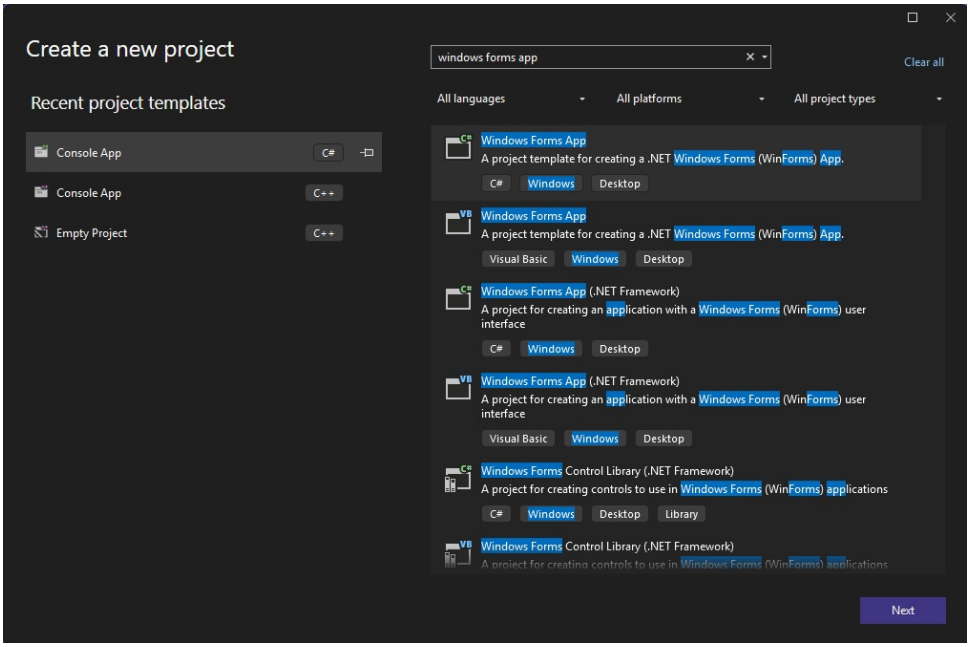

Figure 5: Creating a Windows Form

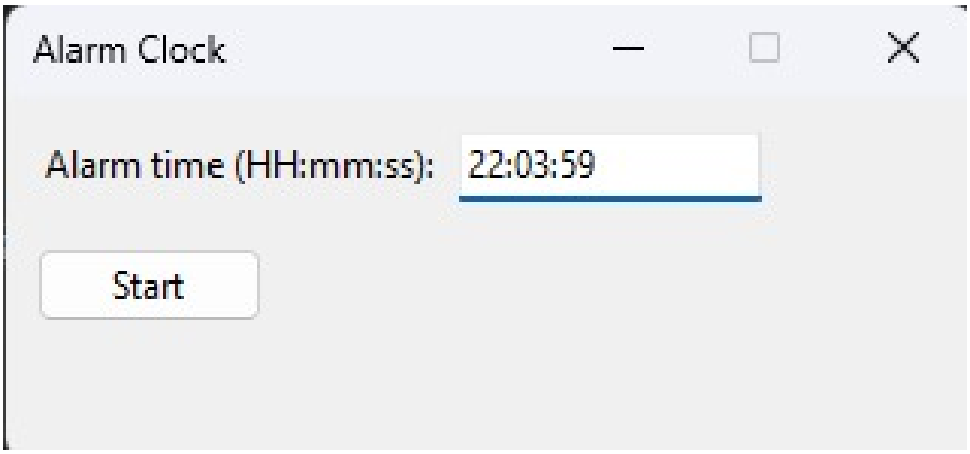

Figure 6: User can enter the target time

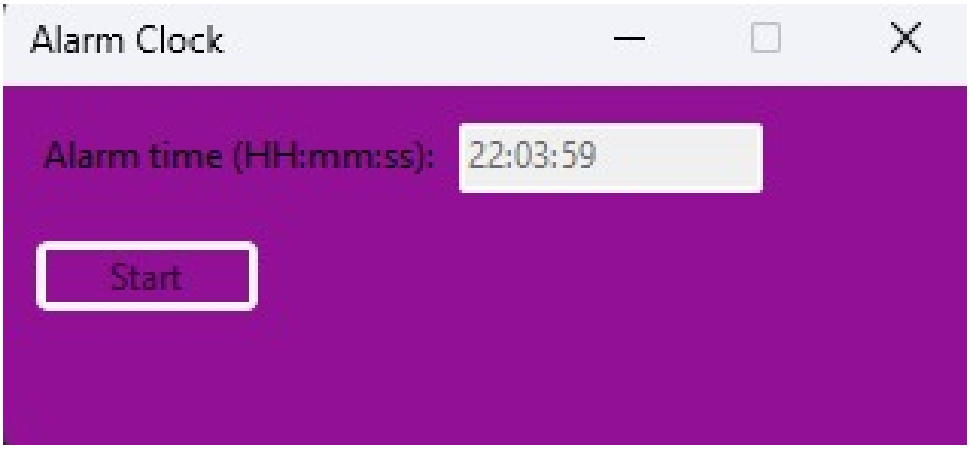

Figure 7: The color keeps changing till the target time is reached

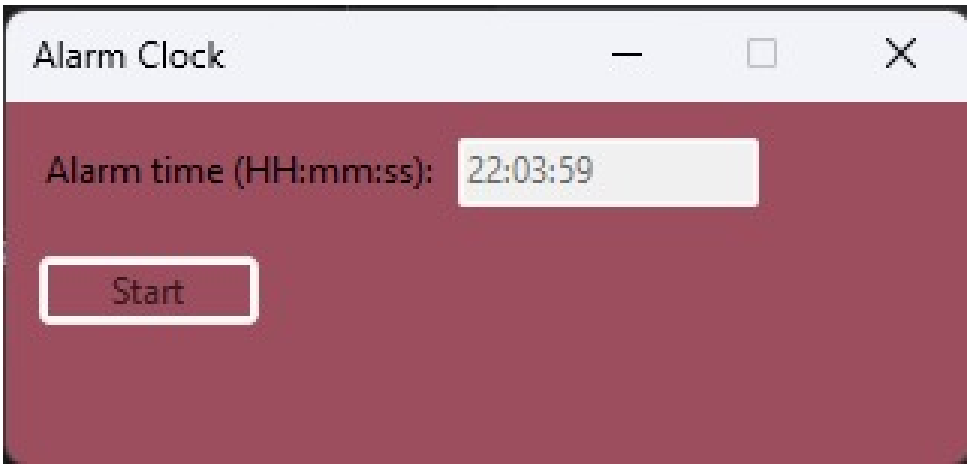

Figure 8: The color keeps changing till the target time is reached

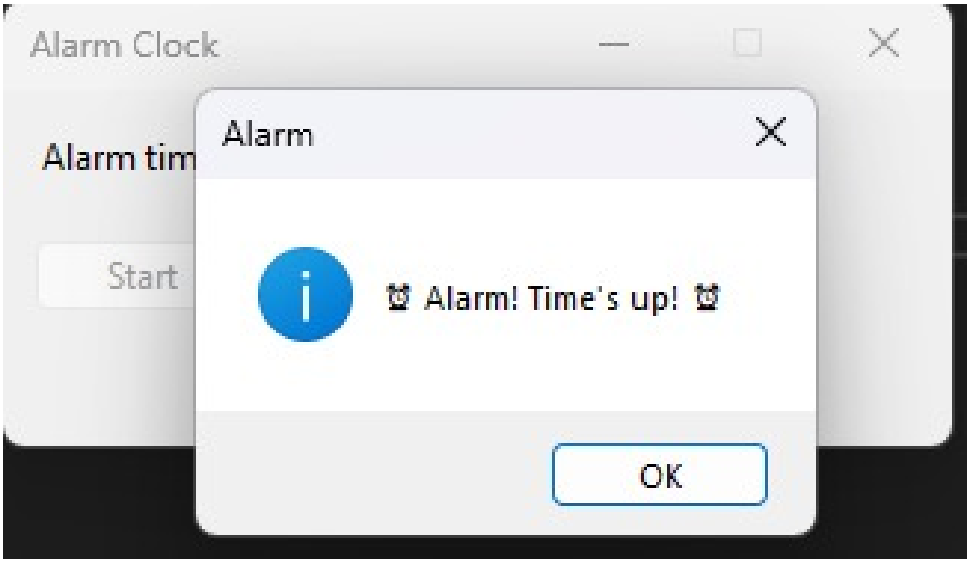

Figure 9: Alarm message pops up when the target time is reached

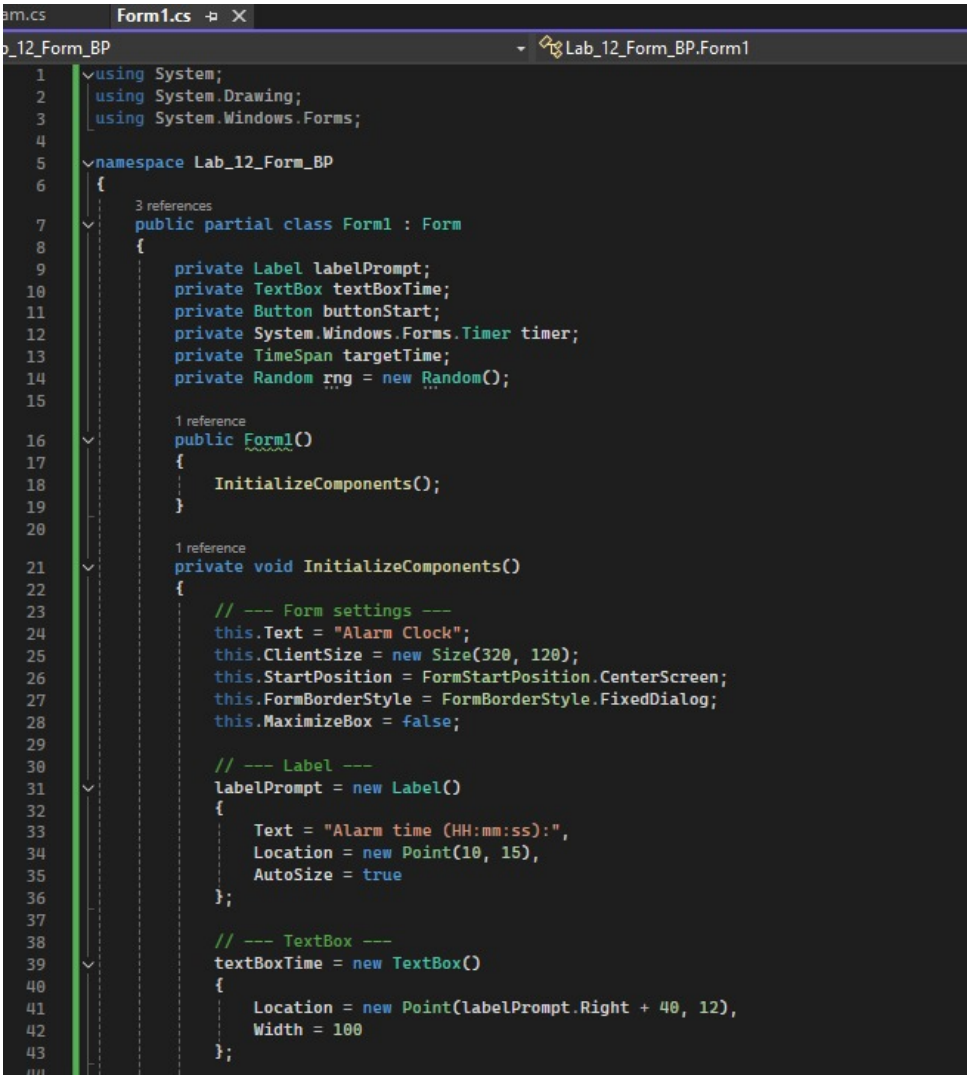

Figure 10: Code screenshot

3.3 Performance Analysis

- Console version uses continuous polling (CPU intensive)
- Forms version uses event-driven timer (more efficient)
- Both versions handle edge cases (midnight rollover)
- Forms version provides better user experience

4 Discussion and Conclusion

4.1 Input Validation Implementation

4.1.1 Console Application Validation

The console version implements robust time format validation using `TimeSpan.TryParse`:

```
if (string.IsNullOrWhiteSpace(input) || !TimeSpan.TryParse(input, out _))
{
    Console.WriteLine("Invalid time format. Please use HH:mm:ss (e.g. 14:30:00).");
    return;
}
```

Listing 5: Console validation code

Key validation features:

- Checks for empty or whitespace input
- Verifies the input matches HH:mm:ss format
- Ensures time values are within valid ranges (hours 0-23, minutes 0-59, seconds 0-59)
- Provides clear error messaging to guide user correction

4.1.2 Windows Forms Validation

The GUI version enhances validation with `TimeSpan.TryParseExact` for stricter control:

```
if (!TimeSpan.TryParseExact(
        textBoxTime.Text.Trim(),
        "hh\\:mm\\:ss",
        null,
        out targetTime))
{
    MessageBox.Show(
        "Please enter a valid time in HH:MM:SS format.",
        "Invalid Input",
        MessageBoxButtons.OK,
        MessageBoxIcon.Warning);
    return;
}
```

Listing 6: Forms validation code

Additional validation improvements:

- Uses exact format matching with `TryParseExact`
- Provides modal dialog feedback for better UX
- Includes warning icon for visual emphasis
- Maintains input focus until valid time is entered

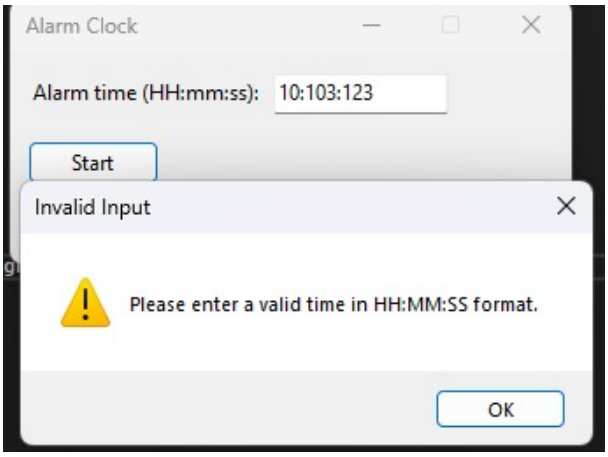

Figure 11: Validation message shown for invalid time format

The validation ensures:

- Only properly formatted times proceed to alarm setting
- Clear user feedback when input is rejected
- Prevention of logical errors from malformed input
- Consistent time handling across both application versions

4.2 Key Challenges

- Time format validation in both implementations
- Threading considerations in console version
- UI thread safety in Windows Forms
- Proper event handler cleanup

4.3 Lessons Learned

- Importance of proper event unsubscribe
- Benefits of `TimeSpan.TryParseExact` for validation
- UI updates must occur on main thread in Windows Forms
- Timer-based approaches are more efficient than polling

4.4 Conclusion

This assignment successfully demonstrated event-driven programming principles in both console and Windows Forms environments. The implementation highlighted the advantages of the publisher-subscriber pattern and proper event handling in .NET applications. The Windows Forms version particularly showcased how event-driven architecture enables responsive user interfaces with background processing.