Displays CodeBehind How-to

Add code logic to displays for dynamic behavior.

How-to **The Four Pillars** **User Interactions** **Displays Module** CodeBehind | How-To | Reference

---

Overview

This guide walks you through adding custom logic to displays using CodeBehind, enabling interactive behaviors, data manipulation, and dynamic responses to user actions. CodeBehind allows you to write .NET code (C# or VB.NET) that executes alongside your display.

Prerequisites:

- Understanding of display creation
- Basic .NET programming knowledge
- Familiarity with tag operations

Understanding CodeBehind

What is CodeBehind?

CodeBehind is display-specific code that:

- Executes on the client side (where display runs)
- Responds to display lifecycle events
- Handles user interactions from display elements
- Accesses and manipulates display objects
- Manages local display variables (Custom Properties)

Language Support

- **Portable Displays**: C# or VB.NET (switchable)
- **HTML5-only Displays**: JavaScript also available
- **Code Editor**: Same interface as Scripts module

Pre-defined Methods

Display Lifecycle Methods

Every display CodeBehind includes these entry points:

```
public void DisplayOpening()
{
    // Called when display is opening
    // Initialize variables, setup handlers
}

public void DisplayIsOpen()
{
    // Called periodically while display is open
    // Update values, check conditions
}

public void DisplayClosing()
{
    // Called when display is closing
    // Cleanup, save state
}
```

Dialog-Specific Method

For dialog displays:

```
public int DialogOnOK()
{
    // Called when OK button pressed
    // Return 1 to allow close
    // Return 0 to prevent close

    if (ValidateInput())
        return 1;
    else
        return 0;
}
```

Accessing Display Elements

Using GetControl Method

Access any display element by its Uid:

```
// Get reference to specific element
private Rectangle statusRect;

public void DisplayOpening()
{
    // Access element by Uid
    statusRect = this.CurrentDisplay.GetControl("22") as Rectangle;

    // Modify properties
    statusRect.Fill = Brushes.Green;
    statusRect.Visibility = Visibility.Visible;
}
```

Typed getters and the cast rule

`GetControl` returns a base control type, so you cast the result to the element's real type with `as` (as above). Common controls also have a dedicated getter that returns the concrete type directly, so no cast is needed:

```
// Typed getter returns the concrete type -> no cast
TDataGrid grid = this.CurrentDisplay.GetDataGrid("gridUid");

// No dedicated getter -> use GetControl and cast
TTrendChart trend = this.CurrentDisplay.GetControl("trendUid") as TTrendChart;
```

Rule of thumb: if the method already returns the specific type (for example `GetDataGrid` returns `TDataGrid`), no cast is needed; if it returns the base type (`GetControl`), you cast. In the code editor, type `this.CurrentDisplay.` and let IntelliSense list every accessor — `GetControl` plus each typed getter such as `GetAlarmWindow`, `GetDataGridWindow`, and `GetFormControl` — with its return type.

Finding Element Uid

1. Select element in Draw environment
2. Check Properties sidebar for **Uid** field
3. Or set custom Uid for easier reference

Handling User Actions

Method 1: Action Dynamics

1. Select element in display
2. Add **Action** dynamic
3. Set Event: MouseLeftButtonDown
4. Set Action: RunScript
5. Create method in CodeBehind:

```
public void Button1_Click(object sender, InputEventArgs e)
{
    // Your code here
    @Tag.Motor1.Start = 1;
    @Info.Trace("Motor started");
}
```

Method 2: Using Sender Parameter

Handle multiple elements with one method:

```
public void AnyButton_Click(object sender, InputEventArgs e)
{
    // Get the element that triggered event
    var button = sender as Button;

    // Use element properties
    string buttonName = button.Name;
    string uid = button.Uid;

    // Take action based on which button
    switch(uid)
    {
        case "StartBtn":
            @Tag.Motor.Start = 1;
            break;
        case "StopBtn":
            @Tag.Motor.Start = 0;
            break;
    }
}
```

Working with Symbols

Accessing Symbol Properties

```
private TSymbol mySymbol;

public void DisplayOpening()
{
    // Get symbol by Uid
    mySymbol = this.CurrentDisplay.GetControl("SymbolUID") as TSymbol;

    // Access symbol properties
    mySymbol.SetPropertyValue("State", 1);

    // Note: #Source property not accessible via code
}
```

Dynamically Updating Symbols

```
public void UpdateSymbol(string uid, int state)
{
    TSymbol symbol = this.CurrentDisplay.GetControl(uid) as TSymbol;
    if (symbol != null)
    {
        symbol.SetPropertyValue("State", state);
        symbol.SetPropertyValue("Value", @Tag.Temperature);
    }
}
```

Moving Elements Programmatically

Using Margin Property

```
private Ellipse movingElement;

public void DisplayOpening()
{
    movingElement = this.CurrentDisplay.GetControl("MovingDot") as Ellipse;
}

public void MoveElement(double x, double y)
{
    // Move element to new position
    movingElement.Margin = new Thickness(x, y, 0, 0);
}

// Called from button click
public void AnimateMovement(object sender, InputEventArgs e)
{
    double newX = @Tag.XPosition;
    double newY = @Tag.YPosition;
    MoveElement(newX, newY);
}
```

Display Navigation

Opening Displays from Code

```
// Method 1: Using Display namespace (auto-updates if renamed)
@Display.AlarmScreen.Open();

// Method 2: Using Client namespace (manual string)
@Client.OpenDisplay("AlarmScreen");

// Method 3: Open as popup
@Display.MotorDetails.NewPopup();

// Method 4: With parameters
@Client.NewPopup("MotorDetails", "Motor1");
```

Closing Displays

```
// Close current display
this.CurrentDisplay.Close();

// Close with dialog result
this.CurrentDisplay.DialogResult = true;
this.CurrentDisplay.Close();
```

Custom Properties

Local Display Variables

Custom Properties are display-specific variables:

```
public void DisplayOpening()
{
    // Set custom property
    this.CurrentDisplay.SetCustomPropertyValue("LastUpdate", DateTime.Now);
    this.CurrentDisplay.SetCustomPropertyValue("UserLevel", 2);
}

public void CheckProperty()
{
    // Get custom property
    object updateTime = this.CurrentDisplay.GetCustomPropertyValue("LastUpdate");
    int level = Convert.ToInt32(
        this.CurrentDisplay.GetCustomPropertyValue("UserLevel"));
}

// Get all as string (for debugging)
string allProps = this.CurrentDisplay.GetCustomPropertiesAsString();
```

Advanced Event Handling

Mouse Wheel Zoom

```
public void DisplayIsOpen()
{
    // Attach to mouse wheel event (only once)
    if (!wheelAttached)
    {
        this.CurrentDisplay.GetThis().PreviewMouseWheel += PreviewMouseWheelChanged;
        wheelAttached = true;
    }
}

private bool wheelAttached = false;

private void PreviewMouseWheelChanged(object sender, MouseWheelEventArgs e)
{
    e.Handled = true;

    // Zoom based on wheel direction
    if (e.Delta > 0)
        @Display.MainPage.ZoomLevel += 0.1;
    else if (e.Delta < 0)
        @Display.MainPage.ZoomLevel -= 0.1;
}
```

Performance Considerations

Avoiding UI Blocking

Problem: CodeBehind runs on UI thread, long operations freeze display.

Solution 1: Use async methods

```
public async void LongOperation()
{
    await Task.Run(() =>
    {
        // Heavy processing here
    });
}
```

Solution 2: Move to Scripts

- Use Script Tasks for server-side processing
- Use Script Classes for shared logic
- Keep CodeBehind lightweight

Best Practices

- **Keep methods short** – under 100ms execution.
- **Avoid loops** in `DisplayIsOpen()`.
- **Cache element references** in `DisplayOpening()`.
- **Use Scripts** for heavy processing.
- **Minimize property access** in loops.

Common Patterns

Toggle Button State

```
private bool motorRunning = false;

public void ToggleMotor(object sender, InputEventArgs e)
{
    motorRunning = !motorRunning;
    @Tag.Motor.Running = motorRunning ? 1 : 0;

    Button btn = sender as Button;
    btn.Content = motorRunning ? "STOP" : "START";
}
```

Validate Input Before Closing

```
public int DialogOnOK()
{
    TextBox input = this.CurrentDisplay.GetControl("InputBox") as TextBox;

    if (string.IsNullOrEmpty(input.Text))
    {
        MessageBox.Show("Please enter a value");
        return 0; // Prevent close
    }

    @Tag.UserInput = input.Text;
    return 1; // Allow close
}
```

Dynamic Element Creation

```
public void DisplayOpening()
{
    // Access the main canvas
    Canvas canvas = this.CurrentDisplay.GetControl("MainCanvas") as Canvas;

    // Create new rectangle
    Rectangle rect = new Rectangle();
    rect.Width = 100;
    rect.Height = 50;
    rect.Fill = Brushes.Blue;

    // Position and add to canvas
    Canvas.SetLeft(rect, 100);
    Canvas.SetTop(rect, 100);
    canvas.Children.Add(rect);
}
```

Troubleshooting

Display freezes during CodeBehind execution

- Use async/await for long operations
- Move heavy logic to Script Tasks
- Add @Info.Trace() to identify bottlenecks

Cannot find element with GetControl

- Verify Uid is correct
- Ensure element exists before accessing
- Check element type casting

Changes don't appear

- Ensure code runs (add trace statements)
- Check if running on correct thread
- Verify property changes are valid

Symbol properties not updating

- #Source property not accessible via code
- Use SetPropertyValue for other properties
- Verify property names match exactly

This guide covered adding custom logic to displays through CodeBehind, from basic event handling to advanced element manipulation, enabling rich interactive behaviors in your HMI application.

In this section...
