Scripts Module How-to

Extend solutions with Python, C# and VB.NET.

Overview

Scripts Module (How-to Guide) walks you through configuring the Scripts module for custom automation and business logic. You'll create reusable classes, automated tasks, simple expressions, and integrate external libraries using C#, VB.NET, or Python.

Prerequisites:

- Basic programming knowledge (C#, VB.NET, or Python)
- Understanding of event-driven programming
- Tags configured for script interaction

Configuration Workflow
Step 1: Create ScriptClasses
Step 2: Create ScriptTasks
Step 3: Add ScriptExpressions
Step 4: Add External References
Step 5: Test and Debug
Code Editor Features
Common Issues
Best Practices
Scripts Module Links

Configuration Workflow

1. **Create ScriptClasses** - Build reusable function libraries
2. **Create ScriptTasks** - Implement automated processes
3. **Add ScriptExpressions** - Simple one-line actions
4. **Add ScriptReferences** - Include external DLL libraries
5. **Test and Debug** - Verify script execution

Step 1: Create ScriptClasses

ScriptClasses are libraries of reusable methods available throughout your solution.

Creating a Class

1. Navigate to **Scripts Classes**
2. Click **Plus** icon
3. Configure:
    - **Name**: Class identifier
    - **Code**: Language (C#, VB.NET, Python)
    - **Domain**: Server or Client execution
    - **ClassContent**: Methods (standard) or Namespace (advanced)
4. Click **OK**
5. Double-click to open Code Editor

Built-in Classes

Class	Purpose	Domain
ServerMain	Methods for all server tasks	Server
ClientMain	Methods for all client displays	Client

Example Class Code

Important: ScriptClasses with Methods type contain only method definitions - NO class declaration wrapper

```
// CORRECT - Methods directly, no class wrapper
public double CalculateEfficiency(double input, double output) {
if (input == 0) return 0;
return (output / input) * 100;
}

public void LogEvent(string message) {
@Alarm.AuditTrail.AddCustomMessage(message);
}

public string GetStatus(double value, double limit) {
return value > limit ? "High" : "Normal";
}
```

Step 2: Create ScriptTasks

ScriptTasks execute code based on triggers or schedules.

Creating a Task

1. Go to **Scripts Tasks**
2. Click **Plus** icon
3. Configure:
    - **Name**: Task identifier
    - **Code**: Language selection
    - **Domain**: Server (default) or Client
    - **Trigger**: Execution condition
    - **Period**: Time interval (if periodic)
    - **InitialState**: Enabled or Disabled
4. Click **OK**
5. Double-click to edit code

Built-in Tasks

Task	Trigger	Purpose
ServerStartup	Solution starts	Initialize server resources
ServerShutdown	Solution stops	Cleanup operations
ClientStartup	Client connects	Setup client environment
ClientShutdown	Client disconnects	Client cleanup

Trigger Options

Trigger Type	Example	Use Case
Tag Change	`@Tag.Temperature`	React to value changes
Period	`60000` (ms)	Run every minute
Condition	`@Tag.Level > 100`	Monitor thresholds
Startup	Built-in	One-time initialization

Example Task Code

Important: ScriptTasks contain only the method body - NO function declarations or wrappers

```
// CORRECT - Code directly, no method wrapper
// Task: MonitorProduction
// Trigger: @Tag.ProductionRun

if (@Tag.ProductionRun == 1)
{
    @Tag.StartTime = DateTime.Now;
    @Tag.BatchNumber = @Tag.BatchNumber + 1;

    // Call class method
    Script.ServerMain.LogEvent("Production Started");
}
else
{
    @Tag.EndTime = DateTime.Now;
    TimeSpan duration = @Tag.EndTime - @Tag.StartTime;
    @Tag.BatchDuration = duration.TotalMinutes;
}
```

Step 3: Add ScriptExpressions

Expressions are single-line VB.NET statements for simple operations.

Creating an Expression

1. Navigate to **Scripts Expressions**
2. Add new row with:

Property	Description	Example
Object	Target tag to receive result	`Tag.Result`
Expression	VB.NET calculation/action	`Tag.Input1 + Tag.Input2`
Execution	When to run	OnChange
Domain	Server or Client	Server
Trigger	Optional trigger tag	`Tag.Calculate`

> **Expressions Syntax**
>
> Expressions use VB.NET syntax. The @ prefix on tag names is optional in Expressions (there are no local .NET variables declared in the expression context that could shadow a tag identifier).

Expression Methods

IIF Method (all parameters evaluated):

```
Tag.Status = IIF(Tag.Value > 100, "High", "Normal")
' Both "High" and "Normal" are evaluated
```

TIF Method (conditional evaluation - recommended):

```
Tag.Result = TIF(Tag.Mode = 1, Script.Class.ServerMain.Method1(), Script.Class.ServerMain.Method2())
' Only executes Method1() OR Method2(), not both
```

Common Expression Examples

```
' Mathematical
Tag.Average = (Tag.Value1 + Tag.Value2) / 2

' Conditional
Tag.Alarm = Tag.Temperature > Tag.Setpoint

' Method call
Tag.Efficiency = Script.Class.ServerMain.CalculateEfficiency(Tag.Input, Tag.Output)

' String manipulation
Tag.FullName = Tag.FirstName + " " + Tag.LastName

' Using TIF for conditional execution
Tag.Status = TIF(Tag.Running, "Active", "Stopped")
```

Step 4: Add External References

Adding DLL References

1. Go to **Scripts References**
2. Click **Plus** icon
3. Browse and select DLL file
4. Configure:
    - **Name**: Reference identifier
    - **Domain**: Server or Client
5. Click **OK**

DLL Location Best Practices

Location	Macro	Use For
ThirdParty folder	`_ThirdParty_`	Server-side DLLs
WpfControls folder	`_WpfControls_`	Client-side controls
Solution folder	`_SolutionPath_`	Solution-specific DLLs

Adding Namespaces

1. Open Code Editor
2. Click **Namespace Declarations** button
3. Add required namespaces in dialog:

```
System.Data
System.Linq
MyCustomLibrary
```

> **Never add using/import statements directly in code** - always use Namespace Declarations dialog

---

Step 5: Test and Debug

Enable Debugging

1. Go to **Runtime Build and Publish**
2. Enable **Debug Information**
3. Save solution

Using the Debugger

1. Start runtime (F5)
2. Go to **Runtime Startup**
3. Click **Connect**
4. Open script in Code Editor
5. Click **Attach .NET Debugger**
6. Set breakpoints by clicking line numbers
7. Use step controls when stopped

Monitoring Execution

Check script performance:

```
// Access in expressions or displays
@Script.Task.MyTask.ExecutionCount    // Number of executions
@Script.Task.MyTask.LastCPUTime       // Last execution time
@Script.Task.MyTask.State             // Running or Idle
```

Code Editor Features

IntelliSense

- Type `@` to see all namespaces
- Type `.` after objects for properties
- Automatic method completion
- Parameter hints

Productivity Tools

Tool	Action	Purpose
Format Document	Toolbar button	Auto-format code
Comment Out	Toolbar button	Comment selection
Uncomment	Toolbar button	Uncomment selection
Compile	Save or toolbar	Check for errors

Toolkit Methods

Common operations via `TK` namespace:

```
// Type conversion
double value = TK.ConvertTo<double>("123.45");

// Tag to DataTable
DataTable dt = TK.TagsToDataTable("Tag.*");

// Dynamic property access
object val = TK.GetObjectValue("Tag.MyTag");

// Obtaining asset values
int val = TK.AssetInt("/Folder1/Alias/Cities/Barcelona/Panel_1/Current")
// Note: TK.ToInt(TK.Asset("")) will perform better if used multiple times.
```

Common Issues

Script Not Executing

- Check trigger configuration
- Verify InitialState is Enabled
- Review BuildStatus for errors
- Confirm domain (Server/Client) matches deployment

Compilation Errors

- Check BuildStatus column (red X indicates errors)
- Double-click to see error details
- Verify namespace declarations
- Ensure correct code structure (no wrappers)

DLL Not Found

- Use macros instead of absolute paths
- Place in correct folder (ThirdParty/WpfControls)
- Restart Designer after DLL updates
- Check Resolved status in References

Performance Issue

- Monitor LastCPUTime property
- Avoid infinite loops
- Use appropriate trigger type
- Consider execution frequency

Best Practices

- **Code Structure** - Remember: no class/method wrappers in Tasks and Classes
- **Use descriptive names** - Clear task and class identification
- **Handle exceptions** - Platform provides automatic protection, but validate inputs
- **Reuse code**- Create methods in ScriptClasses
- **Use @ symbol correctly** - Required in Tasks/Classes, not in Expressions
- **Test incrementally** - Verify BuildStatus after each change
- **Monitor performance** - Check execution metrics regularly
- **Use TIF over IIF** - Avoid unnecessary method execution in expressions

---

Scripts Module Links

Explanation - to understand concepts

Modules / Business Operations / Scripts Module

Tutorials - to learn by doing

Tutorials / Business Operations / Scripts Module Tutorial

How-to Guides - to accomplish specific tasks

How-to Guides / Business Operations / Scripts Module How-to Guide

Reference - technical details

Technical Reference / Business Operations / Scripts Module Reference

---

In this section...
