Scripting

This page will describe how to use the scripting feature. To use the feature emDrive Configurator needs to have an open connection with an emDrive device. Scripts are written in C# (*.cs* files). To open the Script Window navigate to the menu bar, **Device > Script**.

User interface

ScriptWindow.png
Image not found or type unknown

The **Browse** button will open a *File Open window* to choose the script file. Once a file is loaded the file path will be displayed in the textbox next to it.

The **Run script** button will start the execution of the script.

Stop will stop the execution of the script.

The **Script has infinite loop** checkbox is used for scripts with long-running loops. This will automatically get set when the file is loaded if it contains the *Cancellation Token implementation*.

Scripts can be added to **Shortcuts** by selecting the desired shortcut in the **Set script to shortcut button** drop-down menu and clicking the **Set** button.

> Long-running scripts cannot be used with Shortcuts.

The window also comes with a simple script editor that can be opened with the **Edit script**.

Below these controls is the **Command Overview section** which will be populated when certain API calls are made.

To the right of it is the **Log section** which will print text if the **Logging API** calls are made.

Creating a script file

Script files are written in C#. Create a .cs file in your desired text editor or download one of the templates provided in the attachments.

Every script needs the `using System;` and `using emDrive_Configurator;` using statements as well as a `public class Script` with the `public void Main()` method.

> Version 2.5.4.0 and lower use `using eDrive_Configurator;` instead of `using emDrive_Configurator;`.

If long-running or infinite loops are used the user must add `using System.Threading;` and change the main method to public `void Main(CancellationToken)`.

> This is available only from version 2.9.0.0 and onwards. Long-running scripts should not be used in versions below 2.9.0.0.

Basic template

The basic template is used for simple scripts that don't contain long-running loops.

```
/*
  Author:     Amadej Arnus
  Date:       2024-05-08
*/

using System;
using emDrive_Configurator;

public class Script
{
    public void Main()
    {
        // Your code goes here
    }
}
```

Long-running template

This template is used for scripts that are expected to contain long-running or infinite loops. This allows the user to click **Stop** in the Script Window which will cancel the cancellation token and exit the script execution. The user must check for `CancellationToken.IsCancellationRequested` value in their loops for this to work.

```
/*
  Author:     Amadej Arnus
```

```
    Date:        2024-05-98
*/


using System;
using emDrive_Configurator;


using System.Threading;


public class Script
{
    public void Main(CancellationToken cancellationToken)
    {
        // Your code goes here


        // Example of loop with cancellation token
        while (true)
        {
            // Your code goes here


            if (cancellationToken.IsCancellationRequested)
            {
                break;
            }
        }
    }
}
```

Scripting API

The following section will list and describe all of the available API calls.

Script - Watch Interoperability

> This functionality is only available for emDrive Configurator 2.13.0.0 and later.

These API calls allow the user to add *virtual script objects* to **Watch** and write to them, as well as request the last value read for a specific object already in Watch. This can be achieved by using the

`WatchScriptObject` class or the method calls described below.

WatchScriptObject class

The `WatchScriptObject` class uses the API calls described below but is a more user-friendly approach to this functionality. Unless specified the `index` and `subindex` properties will be set automatically. `index` will always be `0xFFFF` and `subindex` will increment from `0` onwards. If the user wishes to reset the subindex counter they must call the `ResetSubIndexCounter` method.

The `TryToAddToWatch` method will attempt to add the virtual object to **Watch**. See **TryAddScriptEntryToWatch** method for possible errors.

The `TryToAddValueToWatch` method will attempt to add the value to the already created virtual object in Watch. See **TryAddScriptEntryValue** method for possible errors.

```
public class WatchScriptObject
{
    // Properties
    public string Name { get; }
    public int NodeId { get; } // Default: 0
    public uint Index { get; } // Default: 0xFFFF
    public ushort SubIndex { get; }
    public ushort DataType { get; }

    // Constructors
    public WatchScriptObject(string name, ushort datatype);
    public WatchScriptObject(string name, ushort subindex, ushort datatype);
    public WatchScriptObject(string name, uint index, ushort subindex, ushort datatype);

    // Methods
    public bool TryToAddToWatch();
    public bool TryToAddValueToWatch(dynamic value);
    public static void ResetSubIndexCounter();
}
```

TryAddScriptEntryToWatch method

Attempts to create a new script virtual object in **Watch** with the specified `name`, `nodeId`, `index`, `subindex`, and `datatype`. Returns `true` if successful or `false` if failed.

```
public static bool TryAddScriptEntryToWatch(string name, int nodeId, uint index, ushort subindex, ushort datatype)
```

Fail reasons:

- Object with specified `nodeId`, `index` and `subindex` already exists in **Watch**.

TryAddScriptEntryValue method

Attempts to write `value` to the script virtual object in **Watch**. Returns `true` if successful or `false` if failed.

```
public static bool TryAddScriptEntryValue(int nodeId, uint index, ushort subindex, dynamic value)
```

Fail reasons:

- Object with specified `nodeId`, `index` and `subindex` does not exist in **Watch**.
- Failed to convert value to the declared datatype.

TryGetLastWatchValue method

Attempts to get the last read value of the object specified with `nodeId`, `index`, `subindex` and saves it to `value`. Returns `true` if successful or `false` if failed.

```
public static bool TryGetWatchValue(int nodeId, uint index, ushort subindex, out double value)
```

Fail reasons:

- Object with specified `nodeId`, `index` and `subindex` does not exist in **Watch**.
- Failed to convert object value to type `double`.

Example file

Example file for Script - Watch Interoperability (also available as download)

```
/*
  Author:     Amadej Arnus
  Date:       2024-04-23
*/


using System;
using System.Windows.Forms;
using emDrive_Configurator;
using System.Threading;
using Timer = System.Threading.Timer;
```

```csharp
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Globalization;
using System.IO;
using System.IO.Ports;
using System.Linq;
using System.Management;
using System.Text;
using System.Text.RegularExpressions;

public class Script
{
    // Prepare error string
    string err = string.Empty;

    // Create object that we will attempt to read from watch
    CO_Object HwMonitorUdc = new CO_Object(0x3071, 0x00);

    public void Main(CancellationToken cancellationToken)
    {
        // This will run the example for the WatchScriptObject
        ObjectBasedVersion(cancellationToken);

        // This will run the example for the API calls
        //ObjectBasedVersion(cancellationToken);
    }

    private void ObjectBasedVersion(CancellationToken cancellationToken)
    {
        // Create new objects for watch-script interoperability
        Procedure.WatchScriptObject intObject = new Procedure.WatchScriptObject("ScriptObject_Int32",
0x0004);
        Procedure.WatchScriptObject floatObject = new Procedure.WatchScriptObject("ScriptObject_Real32",
0x0008);

        // Add int object to watch
        if (intObject.TryToAddToWatch())
```

```
    {
        LogResult("ScriptObject_Int32 added to watch");
    }
    else
    {
        LogResult("ScriptObject_Int32 not added to watch");
    }

    // Add float object to watch
    if (floatObject.TryToAddToWatch())
    {
        LogResult("ScriptObject_Real32 added to watch");
    }
    else
    {
        LogResult("ScriptObject_Real32 not added to watch");
    }

    int i = 0;

    // Do the loop until cancellationToken is requested
    while (!cancellationToken.IsCancellationRequested)
    {
        double hwMonitorValue = 0;

        // Read last value of HwMonitorUdc object
        if (HwMonitorUdc.TryGetWatchValue(1, out hwMonitorValue))
        {
            i++;

            // Set new value to int and float objects
            if (!intObject.TryToAddValueToWatch(i))
            {
                LogResult("Failed to write value to ScriptObject_Int32");
            }

            if (!floatObject.TryToAddValueToWatch(hwMonitorValue * 1.5))
            {
                LogResult("Failed to write value to ScriptObject_Real32");
```

```csharp
                }
            }
            // If not successful, log the error
            else
            {
                LogResult("Failed to get HwMonitorUdc value");
            }


            Procedure.Delay(200);
        }
    }


    private void ApiBasedVersion(CancellationToken cancellationToken)
    {
        // Add int object to watch
        if (TryAddScriptEntryToWatch("ScriptObject_Int32", 1, 0xFFFF, 0x01, datatype: 0x0004))
        {
            LogResult("ScriptObject_Int32 added to watch");
        }
        else
        {
            LogResult("ScriptObject_Int32 not added to watch");
        }


        // Add float object to watch
        if (TryAddScriptEntryToWatch("ScriptObject_Real32", 1, 0xFFFF, 0x02, datatype: 0x0008))
        {
            LogResult("ScriptObject_Real32 added to watch");
        }
        else
        {
            LogResult("ScriptObject_Real32 not added to watch");
        }


        int i = 0;


        // Do the loop until cancellationToken is requested
        while (!cancellationToken.IsCancellationRequested)
        {
```

```csharp
        double hwMinutorValue = 0;


        // Read last value of HwMonitorUdc object
        if (TryGetWatchValue(1, 0x3071, 0x00, out hwMinutorValue))
        {
            i++;

            // Set new value to int and float objects
            if (!TryAddScriptEntryValue(1, 0xFFFF, 0x01, i))
            {
                LogResult("Failed to write value to ScriptObject_Int32");
            }


            if (!TryAddScriptEntryValue(1, 0xFFFF, 0x02, hwMinutorValue * 1.5))
            {
                LogResult("Failed to write value to ScriptObject_Real32");
            }
        }
        // If not successful, log the error
        else
        {
            LogResult("Failed to get HwMonitorUdc value");
        }


        Procedure.Delay(200);
    }
}


// The following methods are just wrappers to shorten the method calls
bool TryAddScriptEntryToWatch(string name, int nodeId, uint index, ushort subindex, ushort datatype)
{
    return Procedure.TryAddScriptEntryToWatch(name, nodeId, index, subindex, datatype);
}
bool TryAddScriptEntryValue(int nodeId, uint index, ushort subindex, dynamic value)
{
    return Procedure.TryAddScriptEntryValue(nodeId, index, subindex, value);
}
bool TryGetWatchValue(int nodeId, uint index, ushort subindex, out double value)
{
```

```csharp
            return Procedure.TryGetWatchValue(nodeId, index, subindex, out value);
    }


    void LogResult(string LogString)
    {
        LogString = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss") + ": " + LogString;
        Procedure.Log(LogString);
    }
}


// CO_Object for readability
class CO_Object
{
    public uint index;
    public ushort subindex;
    public dynamic value;


    public CO_Object(uint index, ushort subindex)
    {
        this.index = index;
        this.subindex = subindex;
    }


    public bool TryGetWatchValue(int nodeId, out double value)
    {
        return Procedure.TryGetWatchValue(nodeId, index, subindex, out value);
    }
}
```

Script Features

Script features are extensions to the scripting functionality.

In order to use script features, the user must add

```
using emDrive_Configurator.Scripts.Features;
```

at the start of the script file.

Modbus TCP

> This functionality is only available for emDrive Configurator 2.13.1.0 and later.

Modbus_TCP.png
Image not found or type unknown

ModbusTCP currently supports connection with Modbus Servers (Client Mode) through TCP. In order to connect to the device, you need to provide the IP Address, Port and Modbus Server ID (Bus Address).

In the following Example, the IP is 127.0.0.1, the Port is 502 and the Modbus Server ID is 1:

```
ModbusTCP ModbusDevice1 = new ModbusTCP("127.0.0.1", 502, 1);
```

Alternatively, the device endianness can also be specified in the connection constructor:

```
ModbusTCP ModbusLocalhost = new ModbusTCP("127.0.0.1", 502, 1, ModbusEndianness.LittleEndian);
```

> Default Endianness is set to Big Endian.

Currently Read and Write Holding registers operation is supported using generic functions:

```
short i16_var = 0;

// Read Holding Register at Address 100
if(ModbusLocalhost.ReadHoldingRegister<short>(100, out i16_var) == true)
{
  // Read Successful
}
else
{
  // Read Failed
}

// Write Holding Register at Address 100
ModbusLocalhost.WriteHoldingRegister(100, i16_var);
```

Supported data types:

- *short*
- *ushort*
- *int*
- *uint*
- *float*