MIKE HYDRO Basin Engine

Interface Programming Guide

DHI A/S headquarters
Agern Allé 5
DK-2970 Hørsholm
Denmark

+45 4516 9200 Telephone
mike@dhigroup.com
www.mikepoweredbydhi.com

Company Registration No.: DK36466871

CONTENTS

MIKE HYDRO Basin Engine
Interface Programming Guide

1 Introduction

The Purpose of the present document is to introduce the use of the MIKE HYDRO Basin Engine .NET interface in programming environments. In particular we will consider the use of Visual Studio Integrated Development Environment (https://www.visualstudio.com/) together with the special package Visual Studio Tools for Office (VSTO) for interaction with MS Excel and all functionality available from the MIKE HYDRO Basin Engine interface. The MIKE HYDRO Basin Software uses MIKE HYDRO Basin's computational core (named the "MikeBasin engine", or simply the "Engine" in the present document), which can be accessed programmatically. The interface may be used with any programming language that supports .NET (C#, Iron Python, Visual Basic .NET, C++/CLI, etc.). In the present document, it is expected that the reader is familiar with the use of MIKE HYDRO Basin software and the water domain knowledge behind it. In case of doubts, please refer to the MIKE HYDRO Basin User's Guide. More specifically, the Engine interface actually consists of two interfaces objects, defined in the assembly DHI.MikeBasin.Engine.dll, namely

1. DHI.MikeBasin.Engine.Engine
2. DHI.MikeBasin.Engine.ModelObject

The Engine is the core object that contains the entire model setup and simulation information. The model setup itself is composed of ModelObject's. ModelObject's can be physical features or network elements, such as nodes, reaches and catchments. Additional types of ModelObject's are logical or computational entities, such as allocation rules. The ModelObject interface also provides methods to return the rules applicable for a feature-type ModelObject. Both interfaces contain methods that refer to some enumerations defined in the assembly DHI.Mike.Basin.Common.dll (namespace DHI.MikeBasin.Common). This assembly needs to be included in the project references if the user uses one of the enumerations defined by the assembly. This is not necessary in most of the simple models.

1.1 Technical Requirements for Using the Interface

In order to use the programming interface, the following tools need to be installed in your PC:

1. MIKE Zero 2017 or later releases (only available in 64 bits)
2. Any .NET framework 4.5.1 or later - compatible programming language (C#, VB, Iron Python, C++/CLI, etc.)
3. Visual Studio (2013 or above)
4. Visual Studio Office Development Tools (optional, for enabling interaction with MS Excel 64 bits)
5. 64-bit version of MS Excel 2013 or later (optional).

Remark: Due to architecture compatibility constraints, no version of MS Excel 32 bits can interact with any version of MIKE Zero 64 bits. Therefore, it is not possible to use two such binaries in a common environment. Thus, it is mandatory to use a 64-bit version of MS Excel for this purpose.

2 Using VSTO and C# Programming

Microsoft releases “Visual Studio Tools for Office” (VSTO, https://visualstudio.microsoft.com/vs/features/office-tools/) as an extension for Visual Studio for interacting with Office products. These tools provides a comprehensive way of interacting with the MikeBasin engine. It is a free tool if you have already Visual Studio installed in your PC. One of the most recommended languages for .NET applications during the last decade is C#. Therefore we will consider C# for the most of the document. Nevertheless, any language supporting .NET is suitable for programming using the MikeBasin engine interface and we will show a couple of examples with other .NET languages as well (see section 12). If you are new to the .NET Framework, there are plenty of .NET-based programming tutorials and courses for all languages (C#, Iron Python, VB, C++/CLI, etc.) and levels available on the internet. Any basic one is good enough for the requirements needed to use the MikeBasin Engine interface.

2.1 MIKE HYDRO Basin Engine with C#

If you want to use the Engine interface in a C# project in Visual Studio, it is necessary to include the reference to the dll assembly (DHI.MikeBasin.Engine.dll, usually located on the bin folder of your MIKE Zero installation, typically

C:\Program Files (x86)\DHI\MIKE Zero\2022\bin\x64

We also need to add a reference to the DHI.Mike.Install.dll assembly located in the GAC (After installing a MIKE product). In a Windows machine it is located typically at:

C:\Windows\Microsoft.NET\assembly\GAC_MSIL\DHI.Mike.Install\v4.0_1.0.0.0__c513450b5d0bf0bf

When you have done so, you can start coding your Console Application project, more specifically in your main program you can type:

```
1  using System;
2  using DHI.MikeBasin.Engine;
3  using DHI.Mike.Install;
4  namespace ConsoleApplication1
5  {
6    class Program
7    {
8      static Program()
9      {
10       if (!MikeImport.SetupLatest(MikeProducts.MikeCore))
11         throw new Exception("Cannot find a MIKE installation");
12     }
13     static void Main(string[] args)
14     {
15       Engine myEngine = new Engine();
16       if (myEngine != null)
17       {
18         Console.WriteLine("MikeBasin engine created succesfully!!");
19       }
20     }
21   }
22 }
```

The line 15,

```
        Engine myEngine = new Engine();
```

Which creates a new instance of the DHI.MikeBasin.Engine.Engine object (simplified syntax by the “using” statement in line 2). Then, line 10,

```
        if (!MikeImport.SetupLatest(MikeProducts.MikeCore))
```

links the code execution to the Mike Installation folder, so the built project can access the right Mike assemblies at run time.
After creating a new instance of the Mike Basin engine object, the next step is to initialize it. For that purpose, the method Initialize must be called, passing as arguments, the path to the MIKE HYDRO file location folder, and the MIKE HYDRO setup file name:

```
string workDir = @"C:\path\to\MikeHydro\file\location\";
string mikeHydroFileName = "MIKEHYDRO1.mhydro";
myEngine.Initialize(workDir, mikeHydroFileName);
```

After this, the simulation is ready to start. For that matter, a call to the simulate method would complete the execution:

```
myEngine.Simulate();
```

This line runs the simulation, and writes the result file.
Summarizing, a minimalistic program running a MIKE HYDRO Basin engine simulation would contain only three real code lines, besides the namespace and class definition and other minimal required code:

```
using System;
using DHI.MikeBasin.Engine;
using DHI.Mike.Install;

namespace ConsoleApplication1
{
  class Program
  {
    static Program()
    {
      if (!MikeImport.SetupLatest(MikeProducts.MikeCore))
        throw new Exception("Cannot find a MIKE installation");
    }
    static void Main(string[] args)
    {
      Engine myEngine = new Engine();
      myEngine.Initialize(@"C:\path\to\file\", "MIKEHYDRO1.mhydro");
      myEngine.Simulate();
    }
  }

}
```

These three lines inside the main program are equivalent to just open the MIKE HYDRO Basin file from the GUI and click on the “Run” button. Nevertheless, the MIKE HYDRO

Basin interface comes along with a number of methods to customize the execution of a simulation, allowing the user to change programmatically initial conditions and default parameters, as well as changing some of the state variables along the simulation. This also means that a user is able to set for instance, the time step size, and is able to run a single time step, retrieve the important results, and based on these, change one or more input parameters, run another single time step, and so on. In order to do this, we can use the method SetBasicSimulationTiming:

```
DateTime simStart = new DateTime(1980, 1, 1); // January 1, 1980
DateTime simEnd = new DateTime(1980, 7, 1);   // July 1, 1980
double timeStep = 86400; // one day in seconds
myEngine.SetBasicSimulationTiming (simStart, simEnd, timeStep);
```

The method requires as arguments, the start time of the simulation, the end time of the simulation and the time step. To run a single time step, you need to call a method called SimulateTimeStep, which uses as an argument the starting date time for the time step to be executed:

```
DateTime currDate = myEngine.SimulationStart;
myEngine.SimulateTimeStep(currDate);
```

If you want to extract a result from the recently executed time step, you can do that by using a ModelObject object and a method called GetCurrentResult. The modelObject instance specifies the element in the entire model setup from which we will extract a result data value. A ModelObject instance could be a river node, a water user, a reservoir, a catchment, an extraction rule, etc. The method GetCurrentResult requires as argument a string with name of the required quantity item:

```
ModelObject N2 = myEngine.GetModelObject("N2");
double flow = N2.GetCurrentResult("Water leaving model area");
```

In this example, flow is the required result data, the modelObject used is extracted using its modelObject identifier (“N2”), which hints us that it is about a river node (further explanation in section 3). The argument of the method ("Water leaving model area") is also a string identifier, specifying the item we want to extract (the flow leaving the model area) from the model object. If you want to have good control of the model, you need to know all possible object identifiers. In order to have full overview of what you can change and what result you could extract (see section 3).

If you want to change an input parameter before running a time step, you need a ModelObject instance and two function calls. The first function is called FindInputIndex, which returns an integer that will allow us to set the value of the input parameter. For example, if we want to modify the flood control level of a reservoir related to a rule with identifier “FCL 5”, we could do something like this:

```
ModelObject feature = myEngine.GetModelObject("FCL_5");
int iItemIndex = feature.FindInputIndex("TimeSeries", "R5|Water Level");
feature.SetInput(iItemIndex, currDate, currDate, 540.0);
```

Here we obtain a model object (feature) related to the identifier “FLC_5”, then we obtain an index related to the quantity “R5|water Level” and in the next line we set the value to 540.0 (using your default water level default unit). As we mentioned before, the next section will explain how to obtain these text fields to include in the function calls related to all model objects.

3 Model Object Identifiers

The ShowStatus() method output can be used to list the model objects that are part of the model setup, which is has been initialized. Furthermore, it also shows us, which input parameters, time series or look-up tables that are modifiable. The text in the dialogue will also list the properties that are accessible for each model object. To call the ShowStatus() method, you simply type in your program:

```
myEngine.ShowStatus();
```

Here it is an example of the output of the ShowStatus() dialogue, with line numbers added for helping referencing specific lines:

```
1  DHI_Engines_MIKEBASIN.Engine status
2  ============================================================
3
4  Working directory:
5  C:\work\Main\Products\Source\MIKEBASIN\Engine\TestingData\TwoReservoirs_GuideCurve
6
7
8  File/database output is enabled
9  01-01-1981
10 Results from the following network elements can be used (accessed):
11
12 River Node 12 (E22, 0)
13 River Node 2 (E22, 12057,97)
14 River Node 13 (E32, 0)
15 River Node 7 (E31, 5411,59)
16 River Node 3 (Catchment1)
17 River Node 14 (E28, 0)
18 River Node 11 (E27, 0)
19 River Node 9 (E33, 5833,33)
20 River Node 1 (E2, 0)
21 Water User Node 4 (industries)
22 Water User Node 6 (City)
23 Water User Node 5 (Irrigation)
24 Reservoir Node 16 (DownStr Res)
25 Reservoir Node 15 (Reservoir15)
26 Reach 22 (E22)
27 Reach 32 (E32)
28 Reach 31 (E31)
29 Reach 30 (E30)
30 Reach 28 (E28)
31 Reach 27 (E27)
32 Reach 33 (E33)
33 Reach 2 (E2)
34 Reach 21 (E21)
35 Reach 16 (E16)
36 Reach 6 (E6)
37 Reach 29 (E29)
38 Reach 12 (E12)
39 Reach 23 (E23)
40 Reach 18 (E18)
41 Reach 26 (E26)
42 Catchment 1 (Catchment1)
43
44
45 Inputs from the following network elements can be accessed:
```

```
46
47 Water User Node 4 (industries): "WaterUseTS", "Water demand" ("Time series
WaterUseTS|Water demand")
48 Water User Node 4 (industries): "DemandFactor", "" ("Parameter DemandFactor")
49 Water User Node 4 (industries): "WaterUseTS", "Deficit carry-over fraction" ("Time
series WaterUseTS|Deficit carry-over fraction")
50 Water User Node 6 (City): "WaterUseTS", "Water demand" ("Time series WaterUseTS|Water
demand")
51 Water User Node 6 (City): "DemandFactor", "" ("Parameter DemandFactor")
52 Water User Node 6 (City): "WaterUseTS", "Deficit carry-over fraction" ("Time series
WaterUseTS|Deficit carry-over fraction")
53 Water User Node 5 (Irrigation): "WaterUseTS", "Water demand" ("Time series
WaterUseTS|Water demand")
54 Water User Node 5 (Irrigation): "DemandFactor", "" ("Parameter DemandFactor")
55 Water User Node 5 (Irrigation): "WaterUseTS", "Deficit carry-over fraction" ("Time
series WaterUseTS|Deficit carry-over fraction")
56 Reservoir Node 16 (DownStr Res): "CharacteristicLevelsTS", "Bottom level" ("Time
series CharacteristicLevelsTS|Bottom level")
57 Reservoir Node 16 (DownStr Res): "CharacteristicLevelsTS", "Top of dead storagel"
("Time series CharacteristicLevelsTS|Top of dead storagel")
58 Reservoir Node 16 (DownStr Res): "LevelAreaVolumeTable", "x" ("Lookup table item
LevelAreaVolumeTable|x")
59 Reservoir Node 16 (DownStr Res): "LevelAreaVolumeTable", "Area" ("Lookup table item
LevelAreaVolumeTable|Area")
60 Reservoir Node 16 (DownStr Res): "LevelAreaVolumeTable", "Volume" ("Lookup table item
LevelAreaVolumeTable|Volume")
61 Reservoir Node 16 (DownStr Res): "InitialWaterLevel", "" ("Parameter
InitialWaterLevel")
62 Reservoir Node 16 (DownStr Res): "CharacteristicLevelsTS", "Dam crest level (if any)"
("Time series CharacteristicLevelsTS|Dam crest level (if any)")
63 Reservoir Node 15 (Reservoir15): "CharacteristicLevelsTS", "Bottom level" ("Time
series CharacteristicLevelsTS|Bottom level")
64 Reservoir Node 15 (Reservoir15): "CharacteristicLevelsTS", "Top of dead storagel"
("Time series CharacteristicLevelsTS|Top of dead storagel")
65 Reservoir Node 15 (Reservoir15): "LevelAreaVolumeTable", "x" ("Lookup table item
LevelAreaVolumeTable|x")
66 Reservoir Node 15 (Reservoir15): "LevelAreaVolumeTable", "Area" ("Lookup table item
LevelAreaVolumeTable|Area")
67 Reservoir Node 15 (Reservoir15): "LevelAreaVolumeTable", "Volume" ("Lookup table item
LevelAreaVolumeTable|Volume")
68 Reservoir Node 15 (Reservoir15): "InitialWaterLevel", "" ("Parameter
InitialWaterLevel")
69 Reservoir Node 15 (Reservoir15): "CharacteristicLevelsTS", "Dam crest level (if any)"
("Time series CharacteristicLevelsTS|Dam crest level (if any)")
70 Reservoir Node 15 (Reservoir15): "MakeWaterLevelPrioriryTS", "Reservoir Water level"
("Time series MakeWaterLevelPrioriryTS|Reservoir Water level")
71 Reservoir Node 15 (Reservoir15): "MakeWaterLevelPrioriryTS", "Reservoir Water level"
("Time series MakeWaterLevelPrioriryTS|Reservoir Water level")
72 Reach 22 (E22): "Width", "" ("Parameter Width")
73 Reach 32 (E32): "Width", "" ("Parameter Width")
74 Reach 31 (E31): "Width", "" ("Parameter Width")
75 Reach 30 (E30): "Width", "" ("Parameter Width")
76 Reach 28 (E28): "Width", "" ("Parameter Width")
77 Reach 27 (E27): "Width", "" ("Parameter Width")
78 Reach 33 (E33): "Width", "" ("Parameter Width")
79 Reach 2 (E2): "Width", "" ("Parameter Width")
80 Reach 21 (E21): "Width", "" ("Parameter Width")
81 Reach 16 (E16): "Width", "" ("Parameter Width")
82 Reach 6 (E6): "Width", "" ("Parameter Width")
83 Reach 29 (E29): "Width", "" ("Parameter Width")
84 Reach 12 (E12): "Width", "" ("Parameter Width")
85 Reach 23 (E23): "Width", "" ("Parameter Width")
86 Reach 18 (E18): "Width", "" ("Parameter Width")
87 Reach 26 (E26): "Width", "" ("Parameter Width")
88 Catchment 1 (Catchment1): "AssignedArea", "" ("Parameter AssignedArea")
```

```
89 Catchment 1 (Catchment1): "TotalRunoffTS", "Specific runoff" ("Time series
TotalRunoffTS|Specific runoff")
90 Rule RF_4_14: "TimeSeries", "Return flow fraction" ("Time series TimeSeries|Return
flow fraction")
91 Rule RF_6_12: "TimeSeries", "Return flow fraction" ("Time series TimeSeries|Return
flow fraction")
92 Rule RF_5_13: "TimeSeries", "Return flow fraction" ("Time series TimeSeries|Return
flow fraction")
93 Rule FCL_15: "TimeSeries", "Flood control level" ("Time series TimeSeries|Flood
control level")
94 Rule R_MIN_15: "TimeSeries", "Minimum release" ("Time series TimeSeries|Minimum
release")
95 Rule R_MAX_15: "TimeSeries", "Maximum release" ("Time series TimeSeries|Maximum
release")
96 Rule MOL_15: "TimeSeries", "Flood control level" ("Time series TimeSeries|Flood
control level")
97 Rule FCL_16: "TimeSeries", "Flood control level" ("Time series TimeSeries|Flood
control level")
```

It is a good advice to run the program, call the ShowStatus() method once and then save the output to a text file. By doing this you will not need to call the method again, unless you edit the model setup by adding or removing some of the model objects.

Beside the information provided by the ShowStatus() method, it is important to take into consideration that to retrieve a model object it is necessary to have either its string identifier or its zero-based integer identifier.

The string identifier is composed by a string ID related to the model object type and an integer number. The integer number is a unique identifier of the model object. The string ID for model object types are provided the following table:

Table 3.1 String ID for each model object type

Model Element	String ID
River node	"N"
Water User	"W"
Irrigation node	"I"
Hydro Power node	"H"
Reservoir node	"R"
Reach	"E"
Catchment	"C"
Rule	"L"
External Updater	"U"
WQ model	"Q"
WQ Parameter	"P"
WQ Parameter Location	"O"

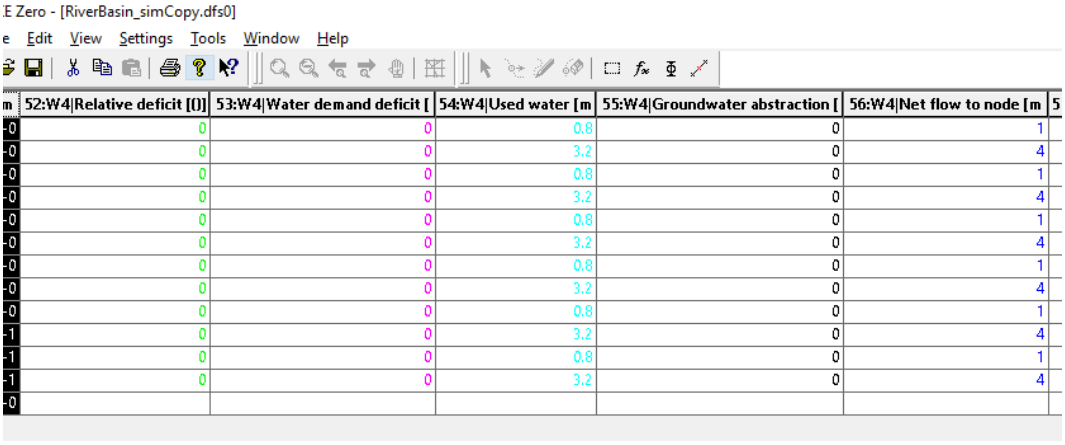
Therefore, if we want to obtain the modelObject related to a water user, we can look at the list of water users from the showStatus() method output:

```
21 Water User Node 4 (industries)
22 Water User Node 6 (City)
23 Water User Node 5 (Irrigation)
```

Since we are looking at water users, we use the string ID related to water users from the table: “W” (second row). If we are interested in getting the first water user “water user 4 (industry)” at line 21, the model object will have a name ID = “W4”. Following this, we can call the GetModelObject() method for obtaining the water user 4 object:

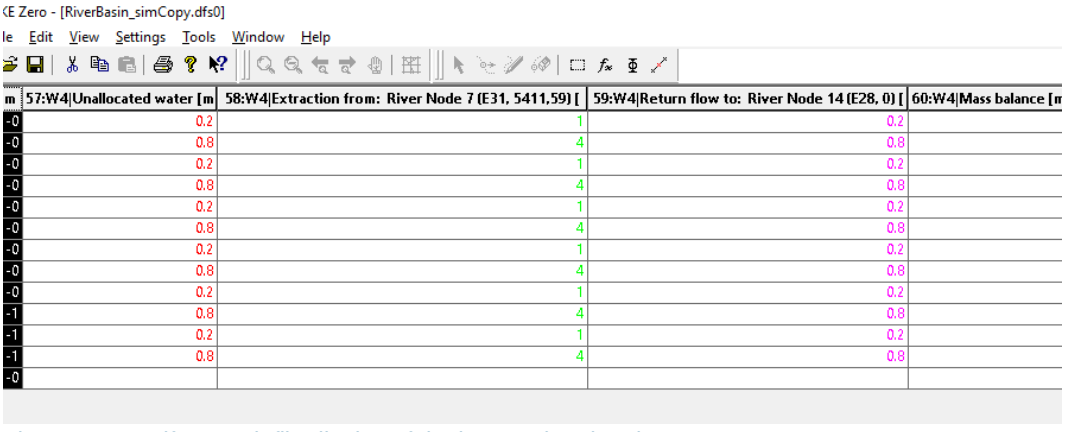
```
ModelObject Wu4 = myEngine.GetModelObject("W4");
```

If now we want to obtain any of the result values, we can look at a dfs0 result file related to the water user 4, displayed in the MIKE Zero time series editor:



m	52:W4|Relative deficit [(])	53:W4|Water demand deficit [	54:W4|Used water [m	55:W4|Groundwater abstraction [	56:W4|Net flow to node [m	5
-0	0	0	0.8	0	1	
-0	0	0	3.2	0	4	
-0	0	0	0.8	0	1	
-0	0	0	3.2	0	4	
-0	0	0	0.8	0	1	
-0	0	0	3.2	0	4	
-0	0	0	0.8	0	1	
-0	0	0	3.2	0	4	
-0	0	0	0.8	0	1	
-1	0	0	3.2	0	4	
-1	0	0	0.8	0	1	
-1	0	0	3.2	0	4	
-0						

Figure 3.1 dfs0 result file display of the items related to the water user 4, part 1



m	57:W4|Unallocated water [m	58:W4|Extraction from: River Node 7 (E31, 5411,59) [	59:W4|Return flow to: River Node 14 (E28, 0) [	60:W4|Mass balance [m
-0	0.2	1	0.2	
-0	0.8	4	0.8	
-0	0.2	1	0.2	
-0	0.8	4	0.8	
-0	0.2	1	0.2	
-0	0.8	4	0.8	
-0	0.2	1	0.2	
-0	0.8	4	0.8	
-0	0.2	1	0.2	
-1	0.8	4	0.8	
-1	0.2	1	0.2	
-1	0.8	4	0.8	
-0				

Figure 3.2 dfs0 result file display of the items related to the water user 4, part 2

The possibilities are, as you can see in the item names: "Relative deficit, "Water demand deficit", "Used water"", "Groundwater abstraction", "Net flow to node", "Unallocated water", "Extraction from: River Node 7 (E31, 5411,59)", "Return flow to: River Node 14 (E28, 0)", and "Mass balance". So, we could for example extract the amount of used water for the water user 4 using the lines:

```
double Flow = waterUser4.GetCurrentResult("Used water");
```

If we furthermore wanted to manipulate one of the Rule objects, we should look at the list of Rule objects in the ShowStatus() output text:

```
90 Rule RF_4_14: "TimeSeries", "Return flow fraction" ("Time series
TimeSeries|Return flow fraction")
91 Rule RF_6_12: "TimeSeries", "Return flow fraction" ("Time series
TimeSeries|Return flow fraction")
92 Rule RF_5_13: "TimeSeries", "Return flow fraction" ("Time series
TimeSeries|Return flow fraction")
93 Rule FCL_15: "TimeSeries", "Flood control level" ("Time series
TimeSeries|Flood control level")
94 Rule R_MIN_15: "TimeSeries", "Minimum release" ("Time series
TimeSeries|Minimum release")
95 Rule R_MAX_15: "TimeSeries", "Maximum release" ("Time series
TimeSeries|Maximum release")
96 Rule MOL_15: "TimeSeries", "Flood control level" ("Time series
TimeSeries|Flood control level")
97 Rule FCL_16: "TimeSeries", "Flood control level" ("Time series
TimeSeries|Flood control level")
```

We can just pick up one of these, for instance the second one (RF_6_12) at line 91, and we use the line:

```
ModelObject rule6 = myEngine.GetModelObject("RF_6_12");
```

Let us suppose we want to modify an input quantity related to this model object rule6. Looking again at the line defining the rule information:

```
91 Rule RF_6_12: "TimeSeries", "Return flow fraction" ("Time series
TimeSeries|Return flow fraction")
```

It is a rule defining a return flow fraction. Let us say we can to set this return flow fraction for the DateTime simTime to retFrac = 0.3. Then we can use the following couple of lines:

```
int iItemIndex = Rule4->FindInputIndex("TimeSeries", "Return flow
fraction");
Rule4->SetInput(iItemIndex, simTime, simTime, 0.3);
```

The first line gets an integer index to use in the second line, which is the method SetInput(..) for setting the desired value.

4 Using VSTO for Interacting with MS Excel

For the installing procedure, please refer to Visual Studio web page:

https://visualstudio.microsoft.com/vs/features/office-tools/.

Once you have installed VSTO on your PC, you can proceed to use it in your Visual Studio projects. The easiest way to use VSTO, is to create a MS Excel 2013 (or later) workbook project in Visual Studio, which will include the necessary references to start the project. Next section includes a step by step guide for creating a project for MS Excel interacting with the .NET interfaces in DHI.MikeBasin.Engine.dll. Here we will focus on the programming part on Visual Studio for manipulating Excel sheets in a simple and basic way. Once you have created the project, you can create an excel sheet object by typing the following line:

```
Excel.Worksheet actSheet =
((Excel.Worksheet)Globals.ThisWorkbook.Application.ActiveSheet);
```

This object will allow you to interact with the spreadsheet. For instance, if you want to get the value of a given cell in the active sheet, you can simply type, for example:

```
var cellValue = ((Excel.Range)activeSheet.Cells[2, 6]).Value2;
```

This line obtains the value of the cell in row 2, column 6 and store it on the variable “cellValue”. On the other hand, if you want to set the value of a cell, you can type:

```
((Excel.Range)activeSheet.Cells[10, 2]).Value2 = 12.5;
```

This will write the value 12.5 on the cell at row 10, column 2. These basic features will allow us to write/read data used to modify our simulation accordingly. For instance, we could have observed data related to the water level of a reservoir, and we would like to overwrite the current data from the simulation with this observed data. We could also want to take the result value of any give variable and write it down into a given column of the spreadsheet. There are many more options for the use of VSTO tools, if you are eager to learn more tricks from VSTO, feel free to refer to the documentation of this tool from the Visual Studio documentation.

4.1 Creating a VS Project with MikeBasin Engine/MS Excel 64 bits (MS Windows)

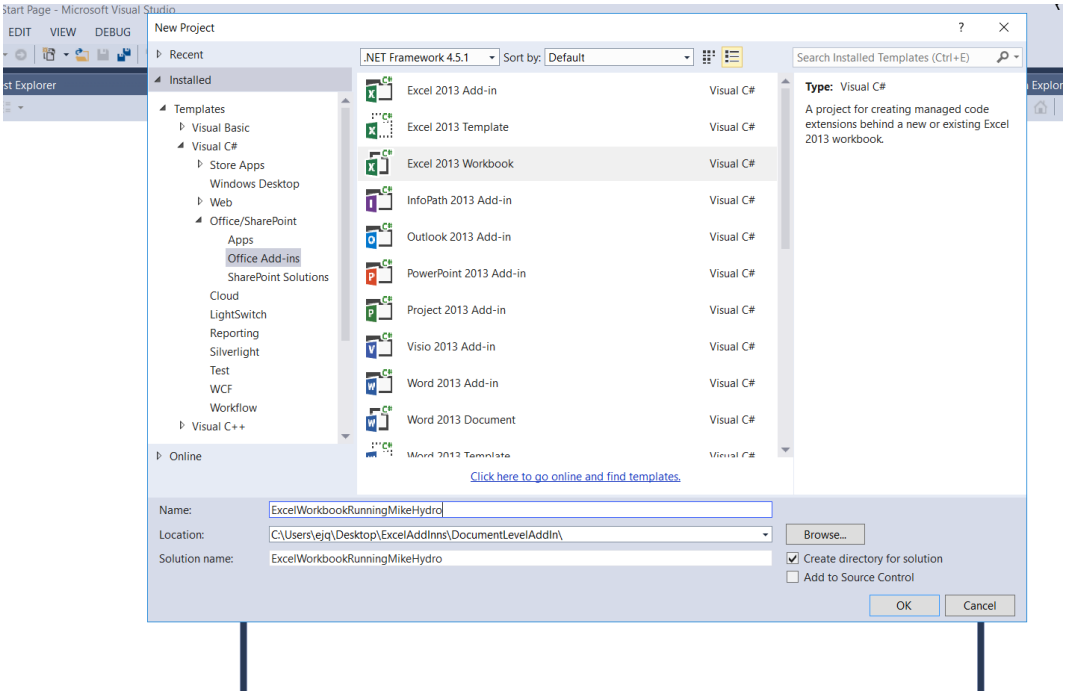
This step-by-step guide will help you create a Visual Studio project (Visual Studio 2013 or newer) where the MikeBasin Interface is used and controlled from the same project and the capabilities of MS Excel are also incorporated. You are required to have installed on your machine the following components:

- MS Windows 7 or later.
- MIKE Zero 2017 or later (64 bits).
- Visual Studio (2013 or later).
- Visual Studio Office Development Tools.
- MS Excel 2013 or later, 64 bits (MS Excel 32 bits is not compatible)

You are expected to be running Windows 7 or later, but this procedure may also work with other OS.

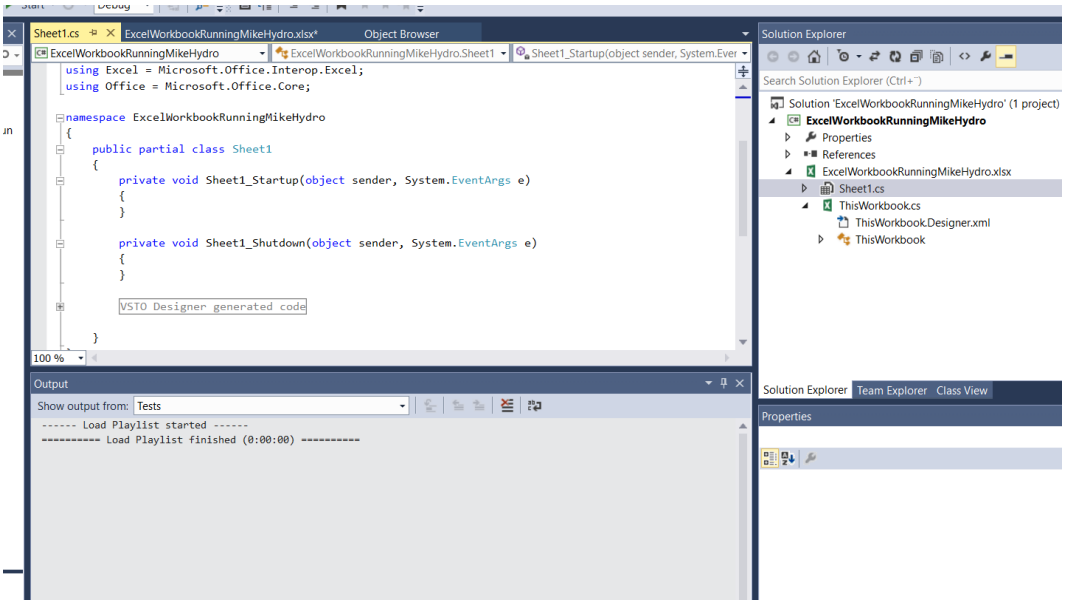
4.2 Creating a Visual Studio Project Step-by-Step

1. Open the Microsoft Visual Studio 2013 (or newer), click the “File => New project”. Under Visual C#, go to the Office/SharePoint Subtree, and select “Excel 2013 WorkBook”:



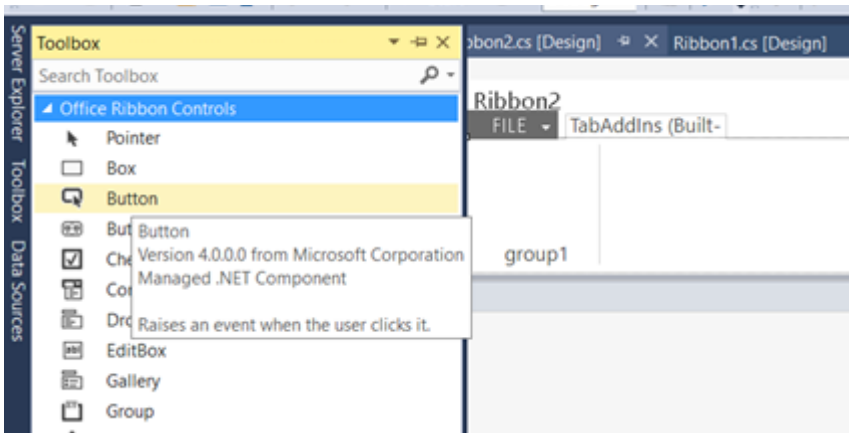
Choose a directory location and a name for your Visual Studio project

2. You will have a project with a few pre-created classes:

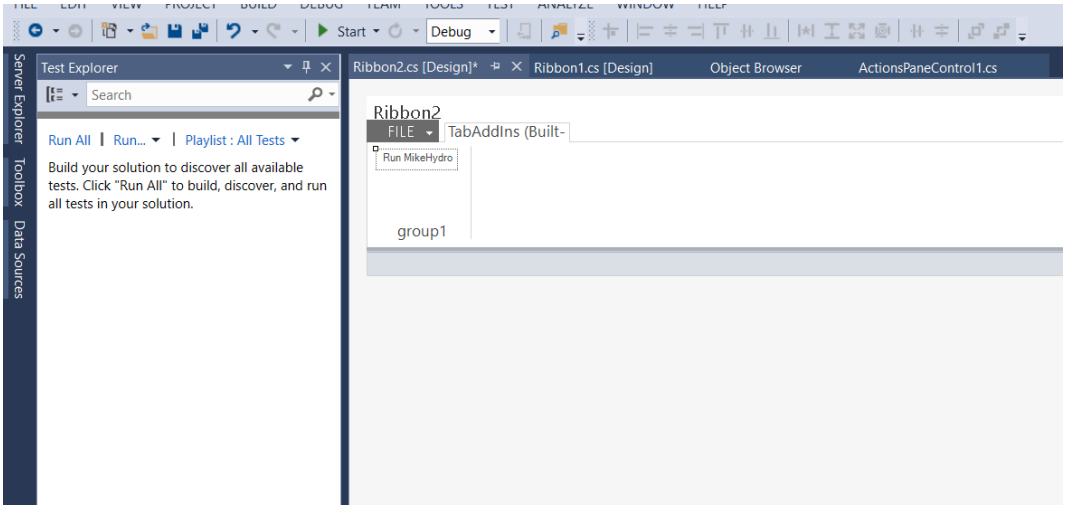


3. Right-click in the project name and add a new item. Select “Ribbon (Visual Designer)”.

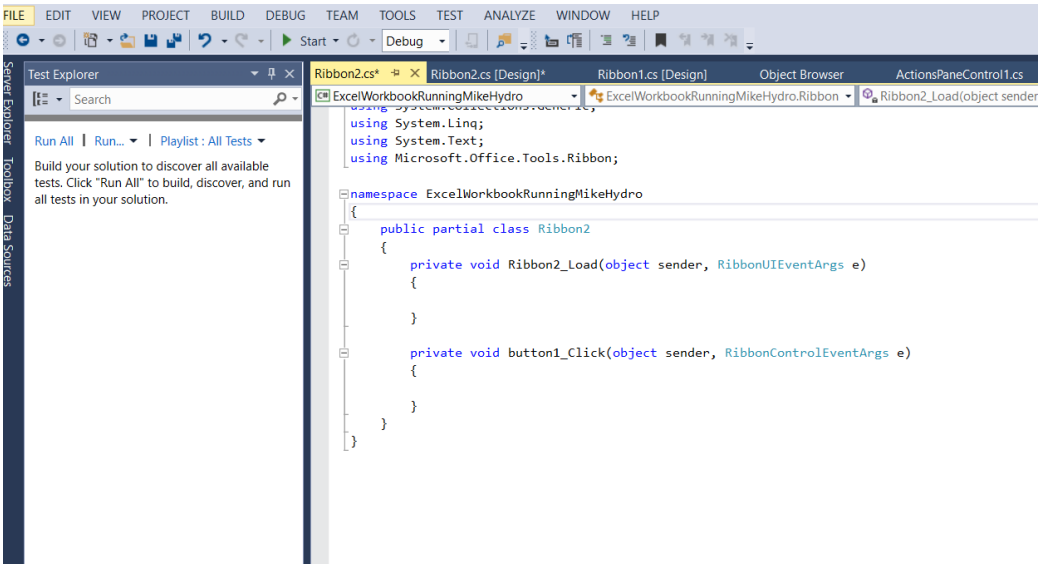
4. In the toolbox tab at the left of the window, select and drag a button into the ribbon:



5. Right click in the recently created button. Select “Properties”. Change the field “Label” to “Run MIKE HYDRO” or any other convenient name:



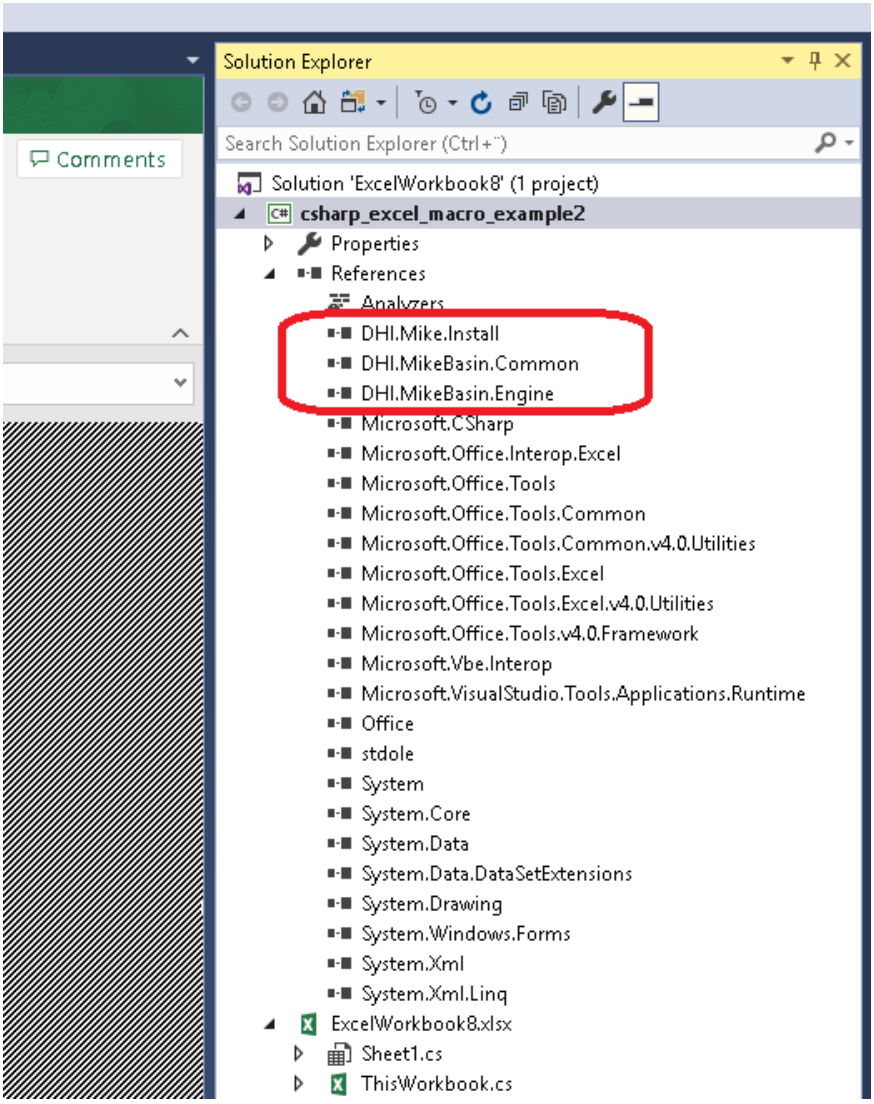
6. Double click in the created button. This action will automatically create the code related to the event of clicking the created button:



7. Add a reference to DHI.MikeBasin.Engine.dll, DHI.MikeBasin.Common.dll (also needed most of the times), and DHI.Mike.Install.dll. This reference is located in a subfolder, which, for windows machines is located in the bin folder or in the GAC (global assembly cache) folder:

   C:\Program Files (x86)\DHI\YEAR\Bin\x64, and
   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\DHI.Mike.Install,

   Where YEAR should correspond to the year of the MIKE version you have installed. Your Visual Studio project reference list may look something like this:



The highlighted in red are the added references to DHI assemblies. These references hold the interfaces for all MIKE HYDRO Basin Objects. Now you can add the logic running MIKE HYDRO Basin to the click button event created at step 6.

8. Add the static variables holding simulation data, the logic of the simulation and the rest of the logic loading and running a MIKE HYDRO Basin Model. You would usually want to link the simulation to the button clicking event handler (`button1_Click(object sender, RibbonControlEventArgs e)` in the example below). In the following example we show a program using the VSTO tools, the MIKE HYDRO Basin engine, where the code: 1) Opens a MIKE HYDRO Basin file; 2) Creates an initialize an engine with the data; 3) Performs a time step simulation; 4) Retrieves the resulting water level of the reservoir in the model; 5) Writes the value into the MS Excel spreadsheet, return to step 3 unless the current time reaches the end of the simulation. But of course it is up to the programmer to decide the ultimate logic and architecture of the program. This is just a suggestion to start with.

9.

Take a look at the code:

```
//Ribbon.cs //
using System;
using Microsoft.Office.Tools.Ribbon;
using Excel = Microsoft.Office.Interop.Excel;
using DHI.MikeBasin.Engine;

namespace ExcelWorkbookRunningMikeHydro
{
    public partial class Ribbon1
    {
        static List<double> reservoirWaterLevels = new List<double>();
        static String ReservoirName = "R15"; // identifier of the object
        static String ReservoirWLItemName = "Water level"; // item name
        // in the result section of Mike Hydro it appears as "R2|Water Level"
        static String waterUserName = "W4";
        static String waterUserTSIdentifier = "WaterUseTS";
        static String waterUserDemandIdentifier = "W4|Water demand";


        private void Ribbon1_Load(object sender, RibbonUIEventArgs e)
        {
        }

        private void button1_Click(object sender, RibbonControlEventArgs e)
        {

            // Set paths to simulation file
            string workDir = @"C:\path\to\MikeHydro\file\location\";
            string dataBaseFileName = "MIKEHYDRO1.mhydro";

            Engine myEngine = new Engine();
            myEngine.Initialize(workDir, dataBaseFileName);
            SetSimulationTimeData(ref myEngine);
            RunEngine(myEngine);
        }

        private static void SetSimulationTimeData(ref Engine myEngine)
        {
            DateTime simStart = new DateTime(1980, 1, 1); // Simulation start time
            DateTime simEnd = new DateTime(1980, 7, 1);   // Simulation end time
            double timeStep = 86400;
            DateTime badTime = new DateTime(100, 1, 1);
// Requires an invalid tof, so Data assimilation is not considered
            myEngine.SetSimulationTiming(simStart, badTime, simEnd, timeStep, 0);
        }

        static void RunEngine(Engine myEngine)
        {

            DateTime currDateTime = myEngine.SimulationStart;

            int counter = 0;
            bool areEqual = DateTime.Equals(currDateTime, new DateTime(1899, 12, 30));
            ModelObject waterUser = myEngine.GetModelObject(waterUserName);
            WriteValueInActiveSheet("WaterLevel", 1, 4);
            double WaterLevel = double.MaxValue;
```

```
            while (!areEqual)
            {
                if (WaterLevel < 538.5)
                {
                    int year = currDateTime.Year;
                    int month = currDateTime.Month;
                    DateTime monthBeginnig = new DateTime(year, month, 1, 0, 0, 0);
                    double modifiedDemand = 1.2; // leave industry with limited water
//  in dry periods
                    int iItemIndex = waterUser.FindInputIndex(waterUserTSIdentifier,
                                                              waterUserDemandIdentifier);
                    double dTime = monthBeginnig.ToOADate();
                    waterUser.SetInput(iItemIndex, dTime, dTime, modifiedDemand);
                }

                myEngine.SimulateTimeStep(currDateTime);
                currDateTime = myEngine.AdvanceTimeStep(true);
                counter++;
                areEqual = DateTime.Equals(currDateTime, new DateTime(1899, 12, 30));
                WaterLevel = GetResult(myEngine, ReservoirName, ReservoirWLItemName);
                WriteValueInActiveSheet(WaterLevel, counter + 1, 4);
            }

        }

        static double GetResult(Engine engine, String modelObjectName, String quantityName)
        {
            ModelObject modelobj1 = engine.GetModelObject(modelObjectName);
            double currenresult = modelobj1.GetCurrentResult(quantityName);
            return currenresult;
        }

        static void WriteValueInActiveSheet(object val, int row, int col)
        {
            Excel.Worksheet activeSheet =
            ((Excel.Worksheet)Globals.ThisWorkbook.Application.ActiveSheet);
            ((Excel.Range)activeSheet.Cells[row, col]).Value2 = val;
        }
    }
}
```

10. Build and run. Go to the Add-in tab and click on the “Run MIKE HYDRO” button.

    Remark: At build time, Visual Studio creates a new excel sheet containing the add-in. The Excel File is identical to the original one, with the addition of the created add-in. You should open the created one in the bin folder of the project.

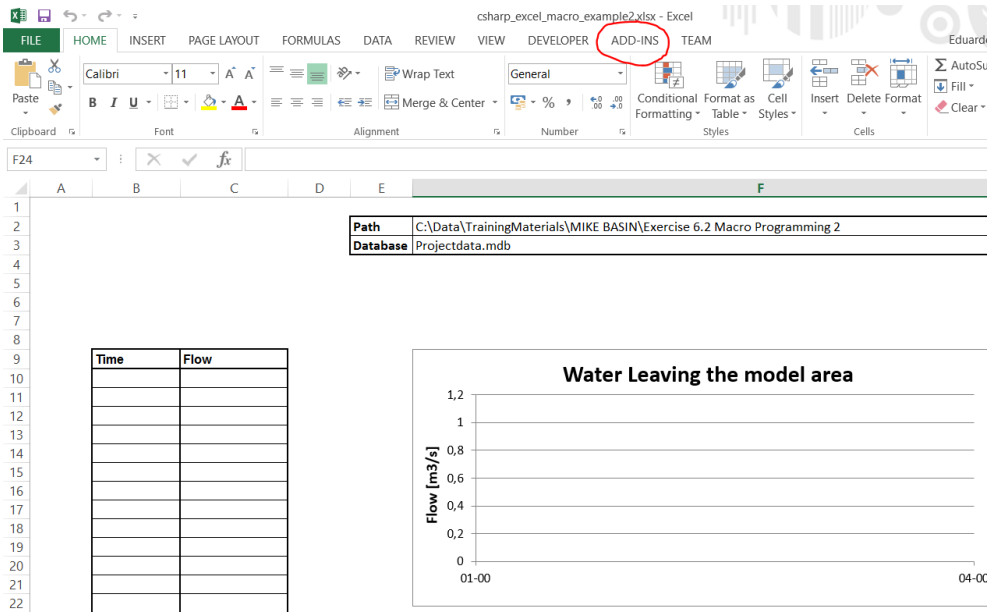
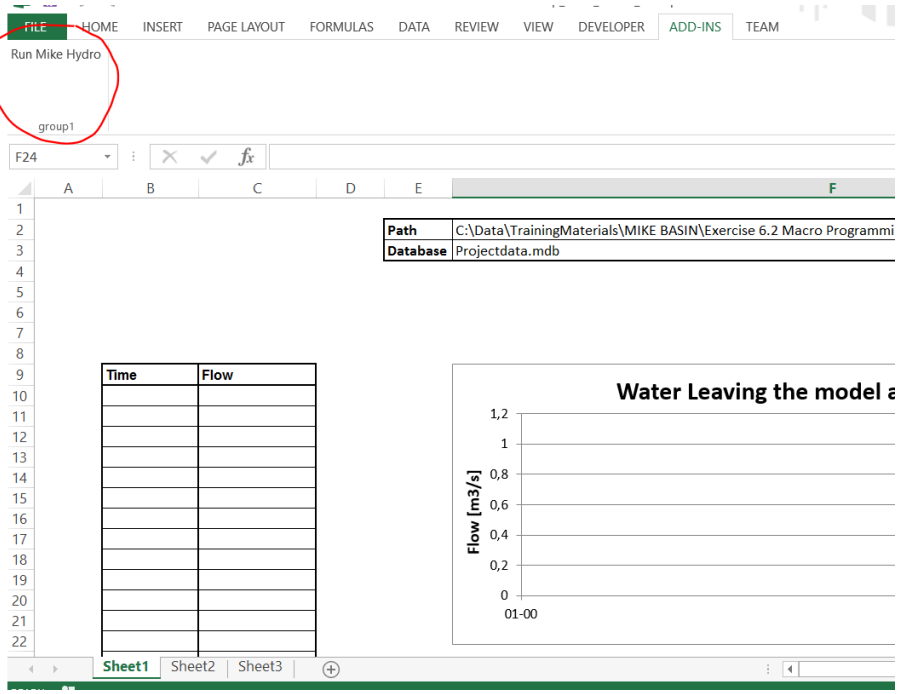


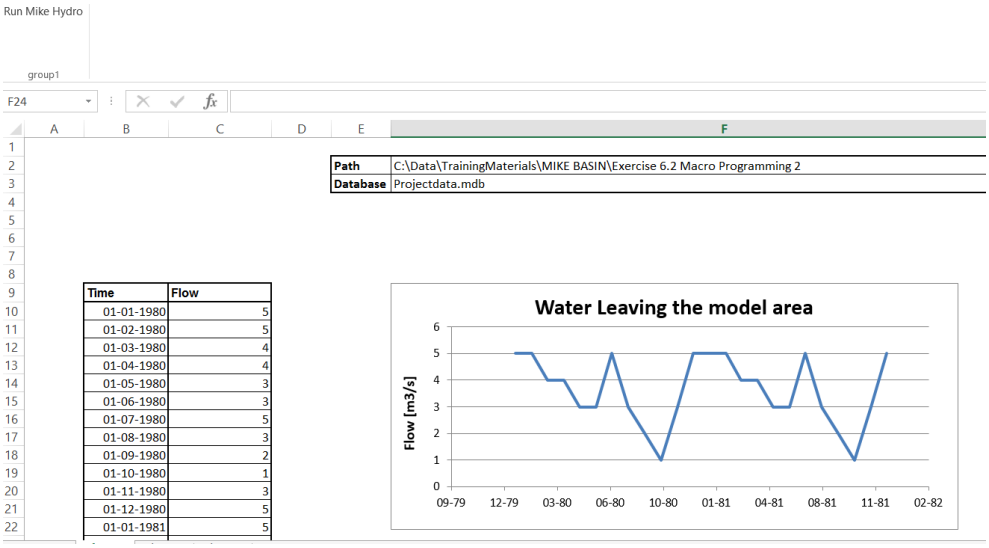
Figure 4.1 Excel sheet with Add-in created by the Visual Studio tool for Office

You move to the “ADD-INS” tab (encircled in red in the Figure 4.1). The excel sheets contains a couple of tables and a graph for input/output of relevant simulation data. These tables and graphs have been included beforehand, just to set a storing location to the result data. In the table on top, you can see a path to a model, so it could be read by the add-in in case you want to extend your application to make the program automatically read the date from the spreadsheet (exercise for the reader). There is a table that will be filled in by the Add-in with the results of the simulation (flow computed by the engine).

11. After you click in the Add-ins you will see the “Run MIKE HYDRO” Button inside the ribbon:



12. Click the button to run the simulation:



In this case the Add-in writes the values of the flow in a column of cells, which was already prepared to plot the graph with the values. As the presented one here, you can make many other customizations with the VSTO tools package in MS Excel Sheets.

5 Migrating from VBA to C# - VSTO

This section is intended for those who want to start programming in C# using the Engine interface, but also for those who had used VBA in the past and now will replace their existing VBA code for an equivalent in C# + VSTO. If you are not familiar with C# a few important details need to be considered, and we will focus on those details in this section. We intend to clarify the minimum concepts to make C# function calls, and the basics to define your own methods. A deep understanding of the C# language is way beyond the scope of this document. For such an extensive explanation of the C# language, refer to the official C# documentation.

5.1 Calling Modes in C# and VBA

5.1.1 VBA Calling modes

In VBA, an argument of a function may be called by reference or by value. If a method is called by reference, it means that the variable may be changed by the calling method, giving to the argument a status of output variable. Take for instance the method GetCurrTimeStepInfo from the Engine interface. We present here its signature and give an example of a function call (we use the special notation “byref” to represent call by reference, otherwise it is understood to be an argument by value):

Here it is the “signature” using our convention:

GetCurrTimeStepInfo *ByRef* Date CurrTimeStep, *ByRef* Double CurrTimeStepLength

And an example of such a function call:

```
‘ Function call example. – this is a VBA comment, therefore ignored by the compiler:
Dim CurrTimeStep As Date
Dim CurrTimeStepLength As Double
engine.GetCurrTimeStepInfo CurrTimeStep, CurrTimeStepLength
```

VBA can also use an argument for a method as an argument “by value”, meaning that the argument is only as an input and will not be modified by the calling method. Here is a VBA code signature and call example:

```
‘Signature:
SimulateTimeStep Date SomeDate

‘function call example:
Date  CurrTimeStep
CurrTimeStep = #12/15/1981#;
engine.SimulateTimeStep CurrTimeStep
```

Note that the call using an argument as by value does not include any keyword.

5.1.2 C# Calling Modes

In C#, there are four ways to use an argument in a function call. The arguments could be:

1) By value,
2) Of type “ref”, (reference)
3) Of type “out” (output reference) and
4) As unsafe pointer.

In sections 6, 7, 9, and 10, the signature of all functions are presented, so the user can know which keyword must be used in each function call.

A C# function call by value (case 1) does not use any keyword when making a function call (nor at function definition):

```
DateTime CurrTimeStep;
CurrTimeStep = new DateTime (1981, 15, 12);
engine.SimulateTimeStep(currDate);
```

The `DateTime` variable CurrTimeStep will still be December 15, 1981 after the call to SimulateTimeStep, since it was called by value, and we do not need to know the implementation details of the method to confirm this fact.

The cases 2) and 3) use each a special keyword in the signature and in the function calls. These keywords are respectively, “ref” and “out”. If the keyword “ref” is used (case 2), it means that the argument could be modified by the callee before returning it to the caller, with the requirement that the caller of the method needs to initialize the ref object before the method call. On the other hand, if the keyword “out” is used, it is understood that the callee must initialize the object and return it to the caller. Therefore we can say that there is a slightly more complex way to use the method arguments in C#: Furthermore, most of the time, the “byref “arguments get translated as “out” arguments in C#. For example, the GetBasicInfo(..) method, called with “byref” arguments in VBA, in the C# language has a signature using the “out” keyword for its arguments:

```
//signature
void GetBasicInfo(out int ObjectTypeAsInt, out int ObjectID, out string
UserDefName);

// function call:
int objectType;
int objectId;
string userName;
ModelObject feature = myEngine.GetModelObject("FCL_5");
feature.GetBasicInfo(out objectType, out objectId, out userName);
```

However, some other methods would also be called “byref” in VBA, its equivalent method in C# would have a signature including the “ref” keyword:

```
private void SetSimulationTimeData(ref Engine myEngine).
```

The case 4) represents a case where, the .NET interface is forced to use directly the original C++ pointer used in the C++/CLI interface (language in which the interface is written). VBA does not make this difference and the call is understood as done “byref”. For instance:

GetCurrTimeStepInfo *ByRef* Date CurrTimeStep, *ByRef* Double CurrTimeStepLength

Its corresponding signature in C# uses directly the C++ pointer, for which a "*" symbol is used (C++ dereferencing operator):

```
void GetCurrTimeStepInfo(DateTime* CurrTimeStep, double*
CurrTimeStepLength);
```

In order to call this method, the C# project needs to be set to allow unsafe code, and define de class where the method is called with the "unsafe" keyword:

```
Public unsafe class MyClass
{
    Public static Main(args[])
    {
        DateTime* CurrTimeStep = new DateTime(2001, 1, 1);
        double* CurrTimeStepLength = 86400;
        GetCurrTimeStepInfo(CurrTimeStep, CurrTimeStepLength);
    }
}
```

Among other differences with VBA, notice the different way in which a function call is made. In C#, the arguments are listed between parenthesis, and there is a comma "," between two consecutive arguments, unless a keyword is used (in which case the "," go between the argument and the keyword).

Properties

Properties are "shorthand" methods with one argument or return value, respectively. A property can be set, get (read-only), or both. An example of a property is

engine.Silent = True; // set property
bool bSilent;
bSilent = engine.Silent; // get property
The syntax used to document the property in the example is *bool* Silent {get; set;};

Explicit Declaration of variables

In C#, unlike VBA, variables must be declared specifying its type, unless the use of the keyword "var", so it is legal to write, e.g.:

```
DateTime currTimeStep;
currTimeStep = new DateTime (2005,12, 15);
var nextTimeStep = new DateTime (2005,1, 1);
myEngine.SimulateTimeStep(currTimeStep);
```

Local versus Static (Global) variables

In C#, you should most of the times prefer to declare your variables as local variables. C# possess an automatic garbage collector, which is in charge of free all resources that are gone out of scope during the execution of a program. In situations, however, where you want a program to run many simulations, you can save time by declaring the Engine as a static (global) variable of the outer class (for a Visual Studio tools project, it is the Ribbon class).

Static variables persist as long as the calling application (MS Excel) is open. A typical example is Optimization using Solver in Excel, working on a function like:

```
public static void MikeBasinEval(double ref someVariableToBeOptimized)
{
  if(myEngine == null) //then the first-time call only
  {
    myEngine = new Engine();
    myEngine.Initialize("c:\\example", "ProjectData.mhydro")
  }
  // code to set some inputs for engine
  myEngine.Simulate();
  // code to retrieve some results from the engine
}
```

The first time the above function is called, all variables are uninitialized (null). The `if` statement detects whether the engine variable has been created and the setup is initialized (see details on that method below). Any subsequent time the function is called, engine is no longer null, and the time-consuming call to Initialize" can be skipped.

Daisy-chaining vs temporary variables

For many common tasks, you may need to call more than one method. For example, a very common pattern is to have created a DHI.MikeBasin.Engine.Engine variable, then request a ModelObject from it, and finally retrieve a result for that ModelObject. The "daisy-chaining" code for this task is:

```
double result = myEngine.GetModelObject("N1").GetAverageResult("Net flow
to Node", new DateTime(1981, 15, 12), new DateTime(1982, 1, 1));
```

The alternative is to create temporary variables to hold the results of the first method:

```
DateTime dateStart = new DateTime(1981, 12, 15);
DateTime dateEnd = new DateTime(1981, 1, 1);
ModelObject Mn1 = myEngine.GetModelObject("N1");
double result = Mn1.GetAverageResult("Net flow to node", dateStart,
dateEnd);
```

This Daisy-chaining keeps the code shorter, but is more difficult to debug and often also slower. Whenever you want to call multiple methods for the feature above, it is much faster to only "get" it only once and then store the result for subsequent calls.

6 DHI.MikeBasin.Engine.Engine Interface Methods

In the following sections, the most commonly used methods are outlined.

Initialize

```
void Initialize(string WorkingDir, string MHydroFileName);
```

This method initializes the Engine. That is, it establishes the model setup as created in the user interface. Basically, the principle behind is to first create a setup in the regular user interface, run a “base” simulation, and then do any modifications in the program. Accordingly, this method must be called before any simulation. The argument WorkDir is the working directory for the simulation, which is the directory that contains the mhydro document, which is the second argument (MikeHydroFileName).

Simulate

```
void Simulate();
```

Runs the simulation for the same time period as specified in the engine at the time of calling the method.

SimulateTimeStep

```
void SimulateTimeStep(DateTime StepStart);
```

This method runs a single time step in the simulation, starting at `StepStart`. Stepwise simulation is an alternative to Simulate(), giving the possibility to control the simulation. For example, one can connect a user to a reservoir and change that user’s water demand dynamically based on the current time step’s simulation results or even on several previous time steps’ results (for example, the reservoir water level). Make sure the first call to SimulateTimeStep is done with the proper start date/time. SimulateTimeStep is a very flexible method, because the argument `StepStart` does not have to be continuously increasing as in a regular simulation.

The options are:

- Date input in the same sequence as it has in a regular simulation or the Simulate method.
- Date repeated (several iterations for the same time step, the iteration loop controlled by the user’s Main program).
- Date as an earlier time step for which initial conditions were remembered (‘hotstart’, allowing controlled iterations of sub-periods in the simulation). For more information, see RememberForHotstart.

AdvanceTimeStep

```
DateTime AdvanceTimeStep(bool bSaveResults);
```

Because the time stepping for SimulateTimeStep can include repeated iterations of the same time step, this method is needed to advance the simulation to the next time step once a user-defined iteration criterion is fulfilled. The argument `bSaveResults` is obsolete and retained only for backward compatibility. The returned value is the next time step's date, as determined by the sequence in a regular simulation. A value of 0 is returned when the simulation is finished.

ShowStatus

```
int ShowStatus();
```

This method is for reference only. Calling this method will cause a pop up dialogue to be displayed. It shows the working directory and mhydro file specified in "Initialize" and whether or not the Engine is ready to run simulations. Moreover, if the simulation has been initialized, it will list all model objects used in the model as well as the results that can be retrieved. The text can be used to find out which input can be modified for the initialized model setup. See ModelObject Identifiers section (section 3) for more details.

GetModelObject

```
ModelObject GetModelObject(object UserDefNameOrIDNameOrZeroBasedIndex);
```

A setup consists for a collection of ModelObjects. That is, the four types of nodes, reaches, catchments and allocation rules. GetModelObject retrieves any such object, as described by either:

- Its user-defined name, as entered in the "Name" field in feature's property dialogue, the Water Quality parameter set dialogue, or the "Comment" field of an allocation rule, depending on the type of model object.
  Note that such name has to be unique for this method to succeed.
- The short ID also shown in results time series (e.g., "N15" for the river node with feature ID 15). The ID system is different for rules, see comment in ModelObject interface, Section 3.
- A zero-based index within all ModelObjects in the setup. The ModelObject interface is documented separately.

7 Lesser used DHI.MikeBasin.Engine.Engine Methods

The following sections outline other methods available to you, but they are typically less commonly used.

RunAll

```
bool RunAll(string Directory, string MSAccessFileName, bool Silent, bool
RegressionTest);
```

This is a single method that “does it all”. In essence, it is a combination of “Initialize” and “Simulate”.
For details on the first two arguments, see Initialize. The argument Silent can be set to true to suppress any dialogs. The argument RegTest can enable regression tests (not publicly available). The return value is true if the method has finished successfully, false otherwise.

RememberForHotstart

```
void RememberForHotstart(bool saveToFile);
```

This method causes the Engine to remember all state variables for the current time step *t*, such that all initial conditions (e.g., reservoir levels, groundwater levels, solute masses, volumes in river branches with routing) are given when the simulation later goes back in time to *t*. RememberForHotstart must be called immediately after calling AdvanceTimeStep(..), before any call to SimulateTimeStep for the new time step (as such a call generally changes the state variables). Set the argument saveToFile to true to also save the state to a file.

GetIthRuleForNode

```
ModelObject GetIthRuleForNode(ModelObject NodeObject, int
iRelativeZeroBased, bool bWhereNodeIsUpstream, bool
bWhereNodeIsDownstream);
```

This method returns the i-th rule relevant for NodeObject. That is, a ModelObject that is a node, or 0 if the i-th rule does not exist. The returned rule is also a ModelObject. Set bUpstream and bDownstream to true to retrieve only those rules where NodeObject is the upstream and downstream end, respectively, of a rule. To retrieve all rules for a node, call this method in a while loop, incrementing iZeroBased until the return value is 0.

SetSimulationOptions

```
void SetSimulationOptions(DHI.MikeBasin.Common.WqSimulationModes wqMode,
bool bGroundwater);
```

This method allows you to overwrite the simulation settings otherwise defined in the Options dialogue in the user interface. To be effective, this method must be called before “Initialize”. The argument WqMode is not used in MIKE HYDRO. The argument bGroundwater refers to the groundwater module that can be enabled or disabled. The call

to this function requires that the project includes besides the engine interface assembly, also a reference to the DHI.MikeBasin.Common.dll assembly.

SetSimulationTiming

```
void SetSimulationTiming(DateTime SimStart, DateTime TimeOfForecast,
DateTime SimEnd, double TimeStepInSeconds, int StochasticPeriodInYears);
```

With this method, the simulation timing can be changed relative to the values specified in the simulation period and simulation time step. The most relevant dates that can be edited are simulation start and end date times. The time step must be given in seconds, or -1.0 to indicate monthly time steps. This method should be called immediately after “Initialize”, before any call to Simulate or SimulateTimeStep. All specs relevant to simulation timing are thus collected in a single multi-argument method rather than provided as individual properties to set. This design makes it easier to check the various constraints related to timing. Individual get-only properties for timing specs are provided, however. The arguments TimeOfForecast and StochasticPeriodInYears refer to options not available from the user interface. StochasticPeriodInYears can be set to 1 in the Time step control page when checking the “Stochastic analysis” option. However, intervals other than 1 year are also possible. The TimeOfForecast argument is related to the Engine’s ability to perform data assimilation. Forecast can be disabled by setting timeOfForecast for 1st January of year 100.

SetInputTimeSeriesValue

```
void SetInputTimeSeriesValue(string ObjectUserDefNameOrIDName, string
TSAbsOrRelativeFileName, string TSItemName, DateTime When, double
valueInUserUnits);
```

Changes an input value in a time series associated with a node, reach, or catchment feature as identified by the argument UserDefNameOrIDName, which can be:

1. The user-defined name, as entered in the “Name” field in feature’s property dialogue. Note that such name has to be unique in order for this method to succeed.

2. The short ID also shown in results time series (e.g., “N15” for the river node with feature ID 15). Allocation rules have a peculiar identifier since they are not directly represented in the results.

For this feature, it affects the time series specified by its file name (argument TSFileName), which can be either relative to the WorkingDirectory (recommended for portability of the code) or absolute. For this time series, it affects the entry for in the column with name equal to the argument TSItem-Name. In that column, the value at time When is set to valueInUserUnits.

SetInputTimeSeriesValue is maintained mostly for backward compatibility with pre-2005 versions of MIKE HYDRO BASIN macros. Accordingly, this method will also scan the time series for all rules associated with a node if UserDefName-OrIDName refers to a node. Currently, rules are ModelObjects in their own right, and can be retrieved using GetModelObject.

SetInputTimeSeriesValue is not as fast as its newer alternatives, but it is a shorthand method that “does it all” in a single line of code.

GetCurrTimeStepInfo

```
void GetCurrTimeStepInfo(DateTime* CurrTimeStep, double*
CurrTimeStepLength);
```

Get information on the current time step, both start time and length (in seconds). This method can be useful when running a simulation time step by time step, i.e., using AdvanceTimeStep. That method returns the next time steps start, but not its length. If you require that latter information, use GetCurrTimeStepInfo. Note that the arguments of these functions are classical C++ pointers handled by C# as unsafe pointers. Therefore, to use this method you need to allow unsafe code in the project settings, and the class making the function call should be prefixed with the “unsafe” keyword, as explained in Section 5.

ShowAnyWarnings

```
int ShowAnyWarnings();
```

Shows the list of warnings for the above simulation (if any). Unlike errors, warnings do not cause exceptions that stop the program, but contain useful information on inputs that probably are incorrect. This method will cause a text box to pop up.

FinishSimulation

```
void FinishSimulation(bool bForceWrite);
```

This method need generally not be called explicitly. It writes out the results time series and makes any feature - time series associations. Generally, FinishSimulation is automatically called after the last AdvanceTimeStep call, i.e., when a simulation has reached its end time. Only in rare cases where you may want to override the default behavior can this method become useful. Furthermore, the argument bForceWrite is not relevant for MIKE HYDRO.

GetTemplateModelObject

```
ModelObject GetTemplateModelObject(DHI.MikeBasin.Common.ObjectTypes
objType);
```

Return a “typical” instance of a type of model object. This method is mainly relevant if you want to see the relevant results and inputs for any ModelObject of a particular type. This method can be called before “Initialize”, but after any call to SetSimulationOptions. The call to this function requires that the project includes besides the engine interface, a reference to the DHI.MikeBasin.Common.dll assembly.

Engine

```
public Engine(bool bOpenMiLinkable);
```

Alternative constructor used by the OpenMI wrapper for the Engine.

FindModelObject

```
public ModelObject FindModelObject(DHI.MikeBasin.Common.ObjectTypes
objType, int withinTypeDHI_ID);
```

This method is an alternative to GetModelObject. Get a reference to a ModelObject, found by its type and primary key value in its attributes table. Should be called after “Initialize”. The call to this function requires that the project also includes a reference to the DHI.MikeBasin.Common.dll assembly.

GetRulesForNode

```
public ModelObject[] GetRulesForNode(ModelObject NodeObject, bool
bWhereNodeIsUpstream, bool bWhereNodeIsDownstream);
```

This method is an alternative to GetIthRuleForNode. It returns an array of all rules for a node rather than the i-th one.

RestoreFromHotstartFile

```
DateTime RestoreFromHotstartFile(DateTime DesiredTime, string
HotStartSubDirectory, bool bExactTime);
```

Read all states from hotstart file (i.e., initial conditions), as generated by a call to RememberForHotstart with argument true, when the simulation was at DesiredTime. The argument HotStartSubDirectory indicates on optional subdirectory of the WorkingDirectory to search for hotstart files in. Leave this argument empty if you have not moved hotstart files after a call to RememberForHotstart. If no hotstart file is available for DesiredTime, the argument bExactTime becomes relevant. If it is set to false, the method will try to use the closest hotstart file instead, otherwise it will fail. The return value is the date for which the hotstart file is valid, 0 if none. After this method, you can call Simulate() or SimulateTimeStep(). Internally in the Engine, the current time step is not changed by a call to RestoreFromHotstartFile.

Optimize

```
bool Optimize(DHI.MikeBasin.Common.OptimizationModes OptimMode);
```

This method is not relevant for MIKE HYDRO.

8 DHI.MikeBasin.Engine.Engine Interface Properties

Initialized

bool Initialized [get]
Indicates whether “Initialize” has finished successfully or not.

WorkingDirectory

string WorkingDirectory [get; set]

The directory where the MIKE HYDRO file is located.

WriteOutput

bool WriteOutput [set]
Determines whether simulation results should be written to a file.

NumberOfNetworkElements

long NumberOfNetworkElements [get]

Returns the total number of features (nodes, reaches, and catchments) in a model setup.

NumberOfObjects

long NumberOfObjects [get]

Returns the total number of ModelObjects in a setup. Besides features and network elements, this number also includes allocation rules.

SimulationStart

Datetime SimulationStart [get]

The start of the simulation as specified in the user interface. See also SetSimulationTiming.

SimulationEnd

Datetime SimulationEnd [get]

The end of the simulation as specified in the user interface. See also SetSimulationTiming.

TimeOfForecast

Datetime TimeOfForecast [get]

The time of forecast up to which simulated values can be corrected by observations, and beyond which the error model can estimate corrections. See also SetSimulationTiming.

TimeStep

double TimeStep [get]

The simulation time step as specified in the user interface, -1.0 for monthly. See also SetSimulationTiming.

StochasticPeriod

long StochasticPeriod [get]

The interval for resetting states (in years). The user interface only allows this value to be set to 1. See also SetSimulationTiming.

NumberOfTimeSteps

long NumberOfTimeSteps [get]

The number of time steps in the simulation, as follows from SimulationStart, SimulationEnd, and TimeStep. This number is first computed in the first time step, i.e., is not available until after the first call to SimulateTimeStep or Simulate.

Silent

bool Silent [get; set]

Indicates whether any dialogs should be displayed.

OptimizationMode

OptimizationModes OptimizationMode [get]

Not relevant for MIKE HYDRO.

SimulationDescription

string SimulationDescription [get; set]

The simulation description as specified in the user interface, “Simulation description” page. The description determines the name of the results output. The name of the output file should be changed only when the default name for river basin result file is used in “Storing of results” page. If you want to set this property, you must do so before calling “Initialize”.

9 DHI.MikeBasin.Engine.ModelObject Interface Methods

The following sections outline the most commonly used interface methods for the assembly DHI.MikeBasin.Engine.ModelObject. Note: You cannot create a new instance of a ModelObject. Instances can only be returned from the Engine interface, methods GetModelObject or FindModelObject.

In the latter model, rules were listed in a single table, it was therefore possible to identify rule with a unique ID composed of a letter and an integer value (e.g. 'L9'). In MIKE HYDRO rules are part of model objects such as reservoir or water users. The identifier is therefore different, and to be unique it is composed of: an ID (unique for each rule type), the upstream (if it exists) and downstream node identifiers.

This implies that a MIKE HYDRO Basin C# program using geo-database needs to be updated with regards to the rule identification. Using ShowStatus() method can help to generate the correct script, see Section 3 for more information. The list of Rule type IDs to be used for each rule type is given in Table 9.1 and the String IDs for each model object type are given in Table 3.1.

Table 9.1 Rule type IDs

Type	ID	Has upstream node	Has downstream node	Example of identifier
StorageOwnership_FloodControlLevel	FCL	No	Yes	"FLC_5"
Release_MinRequirement	R_MIN	No	Yes	
Release_MaxLimit	R_MAX	No	Yes	
StorageOwnership_MinOperationLevel	MOL	No	Yes	
StorageOwnership_GuideCurveLevel	GCL	No	Yes	
RemoteFlowControl_MinRequirement	RFC_MIN	No	Yes	
RemoteFlowControl_MaxLimit	RFC_MAX	No	Yes	
InstreamFlow_MinRequirement	F_MIN	No	Yes	
NaturalFlowSplit_Bifurcation	BIF	Yes	Yes	
Supply_FractionalRight	SF	Yes	Yes	
Supply_Demand	SD	Yes	Yes	
Supply_ManagedDemand	SMD	Yes	Yes	
StorageOwnership_ReservoirPool	RP	Yes	Yes	
Call_Demand	CD	Yes	Yes	"CD_1_2"
Call_FractionOfDemand	CF	Yes	Yes	
NaturalFlowSplit_ReturnFlow	RF	Yes	Yes	
Call_StorageDemand	CSD	Yes	Yes	

GetCurrentResult

```
double GetCurrentResult(object NameOrZeroBasedIndex);
```

Get the current (iteration's or time step's) value of a results item for the ModelObject, which must be a feature / network element. Thus, GetCurrentResult is relevant only when the program takes control of the simulation time loop by using SimulateTimeStep or AdvanceTimeStep on the engine. The return value is in user units, which are the units shown in the results time series. GetCurrentValue is very useful when modelling feedback, e.g., the sensitivity of water demand to availability. Given the "Relative deficit" current result, you could use SetInput to manipulate the "Water demand" input time series, then run the current time step again until some convergence criterion you define is attained, and finally call AdvanceTimeStep. The argument NameOrZeroBasedIndex can be a string (as used in FindResultIndex) or a Long int (as would be returned by FindResultIndex). Using an argument of type Long is faster.

GetAverageResult

```
double GetAverageResult(object NameOrZeroBasedIndex, DateTime StartTime,
DateTime EndTime);
```

Get any period's (reasonably ending before the current time in the simulation) time-weighted average value of a results item. The value is in user units. The argument NameOrZeroBasedIndex can be a string (as used in FindResultIndex) or a long (as would be returned by FindResultIndex). Using an argument of type Long is faster. The arguments StartTime and EndTime delimit the period of interest.

FindResultIndex

```
int FindResultIndex(string Name);
```

Simulation results are generated for every ModelObject that is a feature or network element. FindResultIndex returns the index (zero-based) within all result items available for a ModelObject. This index can be stored in a local variable and used repeatedly to access results in a fast manner. For example, all nodes have a result time series item "Net flow to node". Furthermore, in the output, the item name is prepended an ID letter and the feature ID for easier identification, e.g. "N15|Net flow to node". This method should be called without the prepended part, i.e., with "Net flow to node" alone. ShowStatus() will list of model objects, once the model set up has been initialized. Moreover, the exact names of the result items can be copied from the result file (See Section 3).

FindInputIndex

```
int FindInputIndex(string DatabaseFieldName, string TSGroupItemName);
```

This method is similar to FindResultIndex, but for inputs. Inputs are more complex than results, because they can be not only time series, but also lookup tables and parameters. The method ShowStatus() will list of model objects, once the model set up has been initialized. The output of ShowStatus() can be used to get the two arguments to this method. The argument DatabaseFieldName indicates the field in the attribute table where

an input is set. The argument TSGroupItemName indicates the time series item name (if any) within a time series input group. The return value is the zero-based index found. The output of the ShowStatus() can be used to get the two arguments to this method.

SetInput

```
void SetInput(int iQuantityZeroBased, object
TimeSeriesStartTimeOrTableRow, object TimeSeriesEndTime, double
valueInUserUnits);
```

This is the single method for setting any input to a simulation. The index iQuantityZeroBased is the index found with FindInputIndex or GetInputSpecs. If the quantity is a time series, you can pass Date values as the arguments TimeSeriesStartTimeOrTableRow and TimeSeriesEndTime to manipulate a sub-period only. If the quantity is a lookup table, you can pass a Long int value in the argument TimeSeriesStartTimeOrTableRow to manipulate a single row (zero-based). The value to be set is in the same unit as displayed in the user interface. Note that if the quantity is a time series, only values in existing time steps can be changed. If the arguments TimeSeriesStartTimeOrTableRow and/or TimeSeriesEndTime do not match perfectly any existing time step in the input time series, the closest ones will be used instead. Thus, be aware that if you would like to use a program to set daily inputs, you cannot use a monthly time series in the regular simulation input. Also note that there is no recycling for SetInput. In other words, while it is generally fine to specify an input time series for any year (as long as it covers a whole year), you must call SetInput with the actual year in the time series. Furthermore, when a parameter or a value in a time series is changed, the modification will only apply for the current simulation. The updated variables are not saved into the model setup or in the time series files.

10 Lesser used DHI.MikeBasin.Engine.ModelObject Methods

GetBasicInfo

```
void GetBasicInfo(out int ObjectTypeAsInt, out int ObjectID, out string
UserDefName);
```

Returns information common to all ModelObjects. ObjectTypeAsInt is the index in the enumeration ObjectTypes (0 = RiverNode, etc.). ObjectID is not relevant for MIKE HYDRO. UserDefName is the name of the object if it has one.

GetInputSpecs

```
bool GetInputSpecs(int iQuantityZeroBased, out string
AttributeTableFieldName, out string TsOrXyItemName, out string
DisplayName, out int EumUnit, out string QuantityDescription);
```

This method is similar to GetResultSpecs, but for inputs. The return value is true if a result item with the requested index exists, false otherwise. Thus you can use a while loop to retrieve information on all inputs for the ModelObject. DatabaseFieldName is set to the name of the field in the object's attribute table that - in a regular simulation - defines the input quantity. If this field refers to a time series group or XY lookup table, TSGroupItemName is set to the name of the item within the group. It is set to an empty string if the input quantity is a parameter. DisplayName and QuantityDescription are set to a textual descriptions, the latter containing more details. EumUnit is set to the DHI-internal code for the unit of the input quantity.

GetInputOriginalValue

```
double GetInputOriginalValue(int iQuantityZeroBased, object
TimeSeriesStartTimeOrTableRow, object TimeSeriesEndTime);
```

Return the value of a quantity with index iQuantityZeroBased as it would be in a regular simulation, i.e. as set during "Initialize". The value is in the same unit as displayed in the user interface. If the quantity is a time series, you can pass Date values in the arguments TimeSeriesStartTimeOrTableRow and TimeSeriesEndTime to retrieve a value for a sub-period only. If the quantity is a lookup table, you can pass a Long value in the argument TimeSeriesStartTimeOrTableRow to indicate the row (zero-based). GetInputOriginalValue is useful, for example, when wanting to set an input to a multiple of its otherwise user-defined "base" value, or to find an initial guess in an optimization.

GetExtendedInfo

```
void GetExtendedInfo(out DHI.MikeBasin.Common.ObjectTypes ObjectType, out
string UserDefName, out string UserDefCategory, out int ObjectID, out int
FeatureID, out DHI.MikeBasin.Common.RuleTypes RuleType);
```

This is an extended version of GetBasicInfo. The ObjectType is returned as the (more indicative) enumeration element. ObjectID and UserDefName are returned as in GetBasicInfo. UserDefCategory is not relevant for MIKE HYDRO. FeatureID is only relevant for features.

GetResultSpecs

```
bool GetResultSpecs(int iItemIndexZeroBased, out string ItemName, out int
EumDataType, out int EumUnit);
```

In a way, this is the inverse of FindResultIndex, in as much that for a given index, the result item name is returned. Also returned are DHI-internal IDs for the data type and unit, respectively. The return value is true if there exists a result item with the requested index, false otherwise. Thus you can use a while loop to retrieve information on all results for the ModelObject.

GetDayResult

```
double GetDayResult(object NameOrZeroBasedIndex, DateTime StartTime);
```

This method is mainly maintained for backward compatibility. It returns the same result as GetAverageResult with EndTime = StartTime + 1 day.

GetMonthResult

```
double GetMonthResult(object NameOrZeroBasedIndex, DateTime StartTime);
```

This method is mainly maintained for backward compatibility. It returns the same result as GetAverageResult with EndTime = StartTime + 1 month.

GetInputTSObject

TSObject GetInputTSObject Long iQuantityZeroBased

This method has been deprecated.

GetOverwriteableVariableSpecs

```
bool GetOverwritableVariableSpecs(int iVariableZeroBased, out string
VariableName, out ValueType VariableEumDataType, out ValueType
VariableEumUnit);
```

This method applies for OpenMI: what state variables can be overwritten? Loop through iVariableZeroBased from 0 until you get a return value of false. State variables are those otherwise calculated by this engine, e.g., flow. So in a sense they are "input" quantities, but only a small subset, because in general, input quantities are boundary conditions or parameters. Those are obtained from GetInputQuantitySpecs(..). Those can be set easily without affecting the simulation logic, whereas overwriting requires a lot more logic in the code. E.g., if you overwrite a water level in a reservoir, you also have to adjust the volume and the area. Furthermore, overwriting more than one variable may result in physically inconsistent states, e.g., when you overwrite a reservoir level, you cannot independently overwrite that reservoir's volume. Because of the extra logic required, only few variables are open for overwriting. Technically, over writable variables are presented to the user in the same way as result items. This makes sense, because results are the simulated values of internal variables. The analogy also makes identification of over writable variables intuitive, by using the same name as the corresponding result item. Also, over

writable variables just as results exist only for an Element, not an Object. The method returns whether iVariableZeroBased exists or not.

OverwriteVariableInComingTimeStep

```
void OverwriteVariableInComingTimeStep(int iVariableZeroBased, double
newValueInUserUnits);
```

This method applies for OpenMI: Overwrite the iVariableZeroBased-th over writable variable of an element.
For more info, see GetOverwritableVariableSpecs.

11 Skeleton of Program in C# using VSTO and the Engine Interface

The following skeleton can be used to run a simulation time step by time step. Inputs can be changed before executing the time step, and results can be retrieved afterwards.

Remark: It is also possible to execute the time step multiple times, before moving on to the next time step.

```
// file: Ribbon1.cs
using System;
using Microsoft.Office.Tools.Ribbon;
using Excel = Microsoft.Office.Interop.Excel;
using DHI.MikeBasin.Engine;

namespace csharp_excel_program_example21
{
    public partial class Ribbon1
    {
      private Engine myEngine;
      private string SimulationDirectory;
      private string SimulationFile;
      private double Flow;
      private int cellIndex;

      private void Ribbon1_Load(object sender, RibbonUIEventArgs e)
      {

      }

      private void RunMHButton_Click(object sender, RibbonControlEventArgs
e)
      {
        SimulationDirectory = ((Excel.Range) activeSheet.Cells[2,
6]).Value2;
        SimulationFile = ((Excel.Range) activeSheet.Cells[3, 6]).Value2;
        Simulate();
      }

      private void Simulate()
      {
        // initialize a mikeBasin Engine
        myEngine = new Engine();
        myEngine.Silent = false; //' no progress info, no separate dialogs
        myEngine.SimulationDescription = "C# Test";
        myEngine.Initialize(SimulationDirectory, SimulationFile);
        DateTime currDate = myEngine.SimulationStart;
        cellIndex = 10;
        bool areEqual = false;
        while (!areEqual)
        {
          // Simulate the current time step
          myEngine.SimulateTimeStep(currDate);
          // get current (this time step's) results (if any)
          GetSimResults(currDate);
```

```
            ModelObject feature = myEngine.GetModelObject("FCL_R5");
            int iItemIndex = feature.FindInputIndex("TimeSeries", "R5|Water
Level");
            feature.SetInput(iItemIndex, currDate, currDate, 540.0);

            //Set the simulation timing to the next time step
            currDate = myEngine.AdvanceTimeStep(true);
            areEqual = DateTime.Equals(currDate, new DateTime(1899, 12, 30));
          }
        }

        private void GetSimResults(DateTime currDate)
        {
          // define an object from which extract a result we might be
interested in
          ModelObject N2 = myEngine.GetModelObject("N2");
          Flow = N2.GetCurrentResult("Water leaving model area");
          // define the Excel sheet in which write the results
          object actSheetObj = Globals.ThisWorkbook.Application.ActiveSheet;
          Excel.Worksheet activeSheet = (Excel.Worksheet) actSheetObj;
          ((Excel.Range) activeSheet.Cells[cellIndex, 2]).Value2 = currDate;
          ((Excel.Range) activeSheet.Cells[cellIndex, 3]).Value2 = Flow;
          cellIndex = cellIndex + 1;
        }

        private void ChangeInput(DateTime currDate)
        {
           ModelObject feature = myEngine.GetModelObject("FCL_R5");
           int iItemIndex = feature.FindInputIndex("TimeSeries", "R5|Water
Level");
           feature.SetInput(iItemIndex, currDate, currDate, 540.0);
        }

    }
}
```

12 Limitations with 64 bits Installation and .NET Alternatives

As stated in the previous sections it is not be possible to use this interface with any version of MS Excel 32 bits. If you happen to have installed a version of MS Excel on 32 bits (which is the common case), you will not be able to use the MikeBasin Engine interface. One alternative to consider is to drop the use of MS Excel as interactive appliation, and use a simpler environment to control programatically the engine interface. You can still use any .NET language to connect to the engine interface, as for example, Visual Basic, C#, Iron Python, C++/CLI, etc.. If for example you decide to use C# (without VSTools for Office), the code would have to remove the VSTO references and related coded:

```
File: MikeBasinProgram.cs
using System;
using DHI.MikeBasin.Engine;

namespace CSharpCallingWrapper
{
  class Program
  {
    static Engine myEngine;
    static string SimulationDirectory;
    static string SimulationFile;

    static void Main(string[] args)
    {

      SimulationDirectory = @"C:\path\to\MikeHydro\file\";
      SimulationFile = "model.mhydro";
      myEngine = new Engine();
      myEngine.Silent = true; //' no progress info, no separate dialogs
      myEngine.SimulationDescription = "Test";
      myEngine.Initialize(SimulationDirectory, SimulationFile);
      DateTime currDate = myEngine.SimulationStart;

      bool areEqual = false;
      while (!areEqual)
      {
        myEngine.SimulateTimeStep(currDate);
        GetSimResults(currDate);
        ChangeInput(currDate);
        currDate = myEngine.AdvanceTimeStep(true);
        areEqual = DateTime.Equals(currDate, new DateTime(1899, 12, 30));
      }
    }

    private static void GetSimResults(DateTime currDate)
    {
      // define an object from which extract a result we might be interested
in
      ModelObject N2 = myEngine.GetModelObject("N2");
      double FloW = N2.GetCurrentResult("Water leaving model area");
    }

    private void ChangeInput(DateTime currDate)
    {
```

```
      ModelObject feature = myEngine.GetModelObject("FCL_R5");
      int iItemIndex = feature.FindInputIndex("TimeSeries", "R5|Water
Level");
      feature.SetInput(iItemIndex, currDate, currDate, 540.0);
    }
  }
}
```

12.1 Example Written on IronPython .NET

Another popular .NET language is Iron Python. If we decide to use IronPython to control a MIKE HYDRO Basin application. In Visual Studio 2013 for example, the procedure is very simple. The reference to the MikeBasin engine is included directly in the code. An example of an Iron Python Application would look like this:

```
file: callMikeBasin.py
import clr
clr.AddReference('DHI.Mike.Install.dll');

clr.AddReference('DHI.MikeBasin.Engine.dll')

import DHI.Mike.Install
from System import DateTime

DHI.Mike.Install.MikeImport.SetupLatest(DHI.Mike.Install.MikeProducts.MikeCo
re);
import DHI.MikeBasin.Engine
myEngine = DHI.MikeBasin.Engine.Engine()
SimulationDirectory = 'C:\\path\\to\\MikeHydro\\file\\';
SimulationFile = 'model.mhydro';

myEngine.Initialize(SimulationDirectory, SimulationFile);

currDate = myEngine.SimulationStart;
invalidTime = DateTime(1899, 12, 30);

areEqual = False;
while areEqual == False:
    myEngine.SimulateTimeStep(currDate);
    N2 = myEngine.GetModelObject("N2");
    headMessage = "Water leaving model area";
    Flow = N2.GetCurrentResult(headMessage);
    print('{0}: {1}'.format(headMessage, Flow))

    feature = myEngine.GetModelObject("FCL_R5");
    iItemIndex = feature.FindInputIndex("TimeSeries", "R5|Water Level");
    feature.SetInput(iItemIndex, currDate, currDate, 540.0);

    currDate = myEngine.AdvanceTimeStep(True);
    areEqual = DateTime.Equals(currDate, invalidTime);
```

the reference to DHI.Mike.Install is made using the line

```
clr.AddReference('DHI.Mike.Install.dll');
```

Remark: If you want to use any Python interpreter (which is not using .NET support by default), you can still add references to .NET assemblies, with the additional information about the qualified name of the assemblie):

```
clr.AddReference('DHI.Mike.Install, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=c513450b5d0bf0bf');
```

12.2 Example Written on Visual Basic .NET

As a third .NET alternative, you can use Visual Basic .NET. If we decide to use Visual Basic .NET, a similar procedure is done in Visual Studio. First you create a Visual Basic .NET console application, you should add a reference to the MIKE HYDRO Basin Engine Interface in the projects references. It is also adviced to add a reference to the DHI.Mike.Install.dll file. The code would look like this:

```
//file: ModuleMikeBasin.vb
Imports System
Module Module1
    Sub Main()

        Dim directory As String
        Dim mhydroFile As String
        directory = "C:\path\to\MikeHydro\file\"
        mhydroFile = "Model.mhydro"

        Dim engine As DHI.MikeBasin.Engine.Engine
        Dim feature As DHI.MikeBasin.Engine.ModelObject
        Dim iItemIndex, result, currDate

        ' initialize calculations
        engine = New DHI.MikeBasin.Engine.Engine
        engine.Silent = True ' no progress info, no separate dialogs
        engine.SimulationDescription = "Macro Test" 'make sure it does not
overwrite results

        engine.Initialize(directory, mhydroFile)

        ' run the simulation time loop
        currDate = engine.SimulationStart
        Dim invalidDate As DateTime = New DateTime(1899, 12, 30)

        While currDate <> invalidDate
            engine.SimulateTimeStep(currDate)

            ' get current (this time step's) results (if any)
            feature = engine.GetModelObject("N2")
            result = feature.GetCurrentResult("Water leaving model area")

            'Write the date and the results to spreadsheet
            currDate = engine.AdvanceTimeStep(True)

        End While
    End Sub
End Module
```

There are plenty of options to choose in case you want to control your MIKE HYDRO Basin Model from any .NET programming environment and customize the simulation according to your needs.