INCA V7.5 Instrument Integration Development Kit V1.5

www.etas.com

Copyright

The data in this document may not be altered or amended without special notification from ETAS GmbH. ETAS GmbH undertakes no further obligation in relation to this document. The software described in it can only be used if the customer is in possession of a general license agreement or single license. Using and copying is only allowed in concurrence with the specifications stipulated in the contract.

Under no circumstances may any part of this document be copied, reproduced, transmitted, stored in a retrieval system or translated into another language without the express written permission of ETAS GmbH.

© Copyright 2024 ETAS GmbH, Stuttgart

The names and designations used in this document are trademarks or brands belonging to the respective owners.

INCA V7.5 Instrument Integration Development Kit V1.5 I Tutorial R01 EN I 03.2024

Contents

1 Introduction 5

2 What You Need to Know 6

2.1 What You Have Learned after Completing the Tutorial 6

2.2 Prerequisites of the System 6

2.3 Prerequisites to Use the Tutorial Successfully 6

3 Lesson 1 - Creating a Model Instrument – “MyInstrument“ 7

3.1 Objectives 7

3.2 Integrating Project Templates for Instruments in Visual Studio 2019 7

3.3 The INCA Instrument Wizard 8
- 3.3.1 Creating a New Instrument 8
- 3.3.2 Run Through the INCA Instrument Wizard – The First Definitions 8

3.4 Integrating MyInstrument in INCA 15

4 Lesson 2 - Customizing MyInstrument 16

4.1 Objectives 17

4.2 Changing the Instrument User Interface 17
- 4.2.1 Adding Components to MyInstrument 17
- 4.2.2 Modifying the Window Size of MyInstrument 17

4.3 The Main Classes of MyInstrument 19

4.4 The Initialize Method of MyInstrument 19

4.5 Implement Event Handlers” 20

4.6 Modifying InstrumentProperties.xml 23
- 4.6.1 Objectives 23
- 4.6.2 Adding Properties in InstrumentProperties.xml 23
- 4.6.3 Adding an Enumeration Item 25
- 4.6.4 Defining SignalProperties 26
- 4.6.5 Adding Collection and Complex Properties 28
- 4.6.6 Defining a Custom Property Editor 31
- 4.6.7 Define PropertyDependencies 40

5 Lesson 3 - Access Variable Values 44

5.1 Objectives 44

5.2 Changing Properties in InstrumentType.xml 44
- 5.2.1 Modifying VariableCount 44
- 5.2.2 Adjusting the Supported VariableClasses 45
- 5.2.3 Adjusting the Supported VariableTypes 45
- 5.2.4 Changing ElementDataType 46
- 5.2.5 ValueAccess for MyInstrument 46
- 5.2.6 Survey in Read Only and Read/Write Access of Variables 47
- 5.2.7 The InitializeValueAccess Method 47
- 5.2.8 GetValue Method 49
- 5.2.9 SetValue Method 51

6 Lesson 4 - Context Menus 53

6.1 Objectives ... 53
6.1.1 Enable the Default Context Menu ... 53
6.1.2 Creating Your Own Context Menu Items ... 57

7 Lesson 5 - Making MyInstrument Multilingual ... 61

7.1 Objectives ... 61

7.2 Completing InstrumentResources.resx ... 61
7.2.1 Creating a New InstrumentResources.resx ... 61
7.2.2 Making MyInstrument Multilingual ... 62

8 Lesson 6 - Implementing Drag & Drop ... 63

8.1 Objectives ... 63

8.2 The Drag & Drop Function ... 63
8.2.1 Implementing Drag & Drop ... 63

9 Lesson 7 - Deploy Instruments on Other Systems ... 66

9.1 Objectives ... 66

9.2 Porting MyInstrument to Another System ... 66

10 Lesson 8 - How to Access Maps, Curves and Axes ... 67

10.1 Objectives ... 67

10.2 Creating MyMapInstrument ... 67

10.3 Using the VariableType Map ... 67
10.3.1 Filling the <label> in InitializeValueAccess Method ... 68
10.3.2 Accessing DataGrids and Error Messages ... 69
10.3.3 Understanding Interpolation Effects ... 77

10.4 Displaying WorkingPoint Information ... 78

10.5 Axis Properties ... 80
10.5.1 Which AxisType Bases on Which DataType? ... 81
10.5.2 Calibrating the Shared Axes Types ... 81

11 Lesson 9 - Register Signals ... 84

11.1 Objectives ... 84

11.2 Registering a Label in MyMapInstrument ... 84

11.3 Specifying the Device of a Variable ... 85

12 Migration: Compatibility of Instruments between INCA V7.2, V7.3, V7.4 ... 86

13 Troubleshooting ... 87

13.1 Objectives ... 87

13.2 Red Instrument in the INCA Experiment ... 88
13.2.1 Instrument Properties Do Not Show in INCA ... 88
13.2.2 Property Modification in InstrumentProperties.xml Does Not Show in INCA 88

13.3 The INCA Test Environment ... 89

14 Further Documentation ... 90

15 ETAS Contact Information ... 91

1 Introduction

ETAS designed the INCA Instrument Integration Development Kit V1.5 to enable the creation of customer specific instruments for the Experiment Environment of INCA V7.5.

These instruments extend the existing library of instruments available today:

- Measure Window
- Calibration Window
- YT Oscilloscope
- Combined Editor

Key features are the design of customized user interfaces and a broad scale of read and / or write access to the measure or calibration variables offered by INCA.

The tutorial gives an exemplary introduction for all offered options. It does not give a complete overview of all possible deployments.

Included in the INCA Instrument Integration Development Kit are

- the INCA Instrument Wizard supporting the development of your instrument, designed as a plug-in to Visual Studio 2022
- the new test environment
- two reference instruments with their complete source code

Find Further Information

You find a glossary explaining technical terms and abbreviations in the INCA Users Guide and a detailed overview of the complete documentation provided with the SDK in chapter 14 Further Documentation.

2 What You Need to Know

2.1 What You Have Learned after Completing the Tutorial

This tutorial enables you to:

- create and modify your own individual instruments
- modify their properties
- implement new context menu items
- access INCA variables by broad scale options
- install instruments on other systems
- customize the Variable Selection Dialog Display Configuration of your instrument

Moreover, this tutorial:

- presents all enhancements related to the new release
- defines the new technical requirements
- shows how to migrate former version instruments to the new environment
- provides tested sample code with which the intended function can be implemented
  The code can directly be copied into the respective source code passage.

2.2 Prerequisites of the System

- Program an instrument in a .NET language.
- Use Windows Forms only for the user interface.

NOTE

The system does not support WPF.

- Make definitions of variables and properties in XML.
- Make sure your operating system supports the tools and standards listed above.

2.3 Prerequisites to Use the Tutorial Successfully

This tutorial is for experienced software developers. Successful work with this tutorial requires the following knowledge:

- Measurement and calibration technology
- Control unit technology
- Object-oriented programming
- INCA technology
- INCA MC concepts
- INCA ASAP2 Structure
- INCA two-page calibration concept
- INCA wording
- Visual Studio 2022 (basic knowledge)
- .NET Framework 4.8

3 Lesson 1 - Creating a Model Instrument – “MyInstrument“

3.1 Objectives

You learn how to create instruments, here called `MyInstrument` and `MyMapInstrument`, using the INCA Instrument Wizard.

You learn how to integrate and start your `MyInstrument` in INCA. `MyInstrument` will also consider some of the new features that are available without changes to the initial layout.

`MyMapInstrument` introduces you to new features of INCA V7.1 with new variable types.

3.2 Integrating Project Templates for Instruments in Visual Studio 2022

Using Visual Studio 2022 you can use a project template which is used by the Instrument Wizard included in the Instrument Integration Development Kit installation.

If Visual Studio 2022 is installed after the Instrument Integration Development Kit, you need to manually integrate the project template in Visual Studio 2022.

To manually integrate the project template into Visual Studio

1. Copy the `INCA Custom Instrument.zip` file from the following path (default): `C:\Program Files\ETAS\INCA Instruments Integration SDK 1.5):`
   `<INST-DK Install Path>\VS2022Template\`
2. Paste it into
   `<My Documents>\Visual Studio 2022\Templates\ProjectTemplates\Visual C#`
3. In Windows 10 you find your `<My Documents>` folder by default under:
   `C:\Users\<userID>\Documents\`

NOTE

Copy the zip file, but do not unzip the file.

4. Restart Visual Studio 2022.
5. Select **File › New Project.**
6. Select **INCA Custom Instrument.**
7. Change the project name at the bottom to `MyInstrument`.
8. Click **OK.**

   The INCA Instrument Wizard opens.

4 INCA Instrument Wizard

The INCA Instrument Wizard allows a quick set-up of a new instrument and guarantees a working default design. The wizard creates all required features and files. Comments to indicate the different parts of the code are also automatically implemented for orientation.

4.1.1 Creating a New Instrument

To create a new instrument

1. Open Visual Studio 2022.
2. Select File `New Project`.
3. Search for `INCA Custom Instrument`.
4. Click **Next**.
5. Change the project name to `MyInstrument`.
6. Click **Create**.

NOTE

The INCA Instrument Wizard verifies the name of your instrument to be unique.

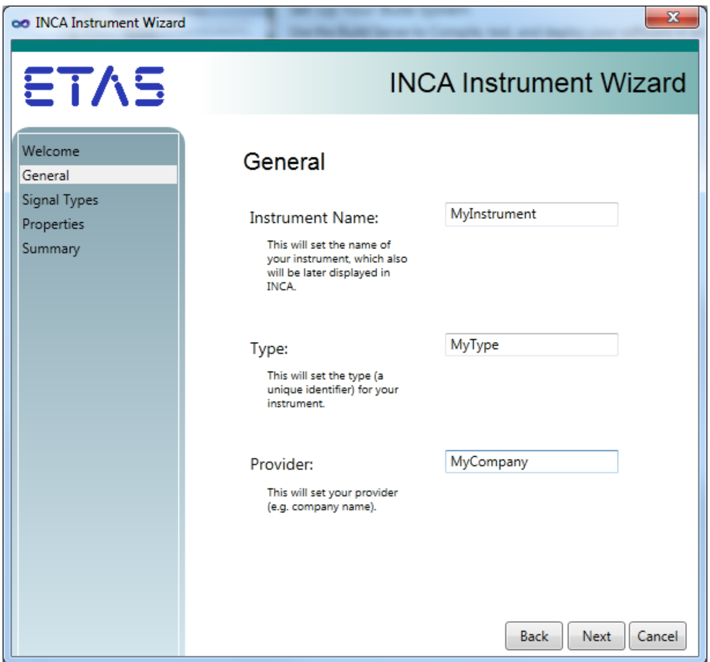
Choose unique names, otherwise your former instrument with the same name will be overwritten.

The INCA Instrument Wizard opens.

4.1.2 Run Through the INCA Instrument Wizard – The First Definitions

To define the basic instrument definition

1. Click **Next.**



NOTE

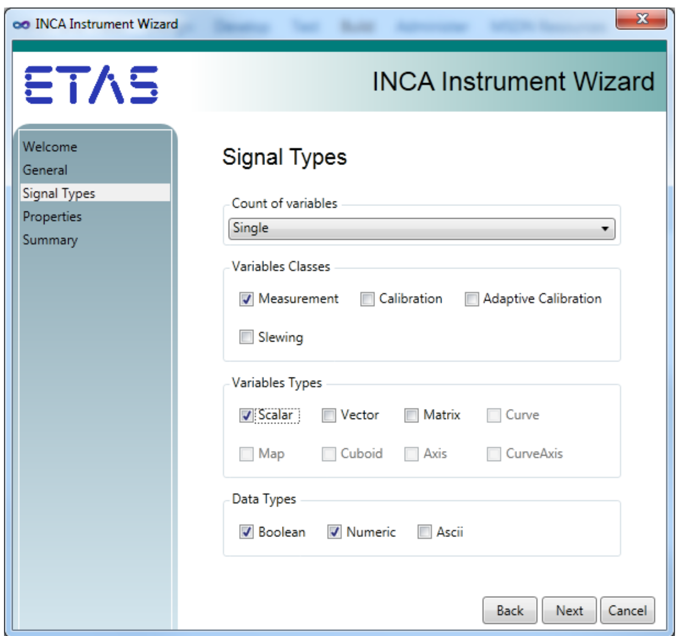
Do not enter dots, blanks, and numbers at the beginning. The system does not support this.

2. Define the instrument as shown in the screenshot above.
3. Click **Next.**

NOTE

The instrument name defines the namespace of the instrument classes. The instrument name, type, and provider must be unique within one namespace.

The “Signal Types” window opens.



Your settings on this page will define the type and number of the variables that can be assigned to `MyInstrument` in the INCA Variables Selection Dialog.

4. Define the Signal Types as shown in the screenshot above.
5. Click **Next**.

NOTE

All settings made here can be changed later directly in the `InstrumentType.xml`, see chapter 0.

Count of Variables includes three options:

- `Single:` Assignment of exactly one variable
- `Multiple:` Assignment of a maximum of 1000 variables
- `None:` No assignment of any variable

The feature `None` can be used to create instruments for INCA showing information other than variable values. You may use this option if your instrument is creating the variable accesses dynamically, see chapter 11 Lesson 9 - Register Signals.

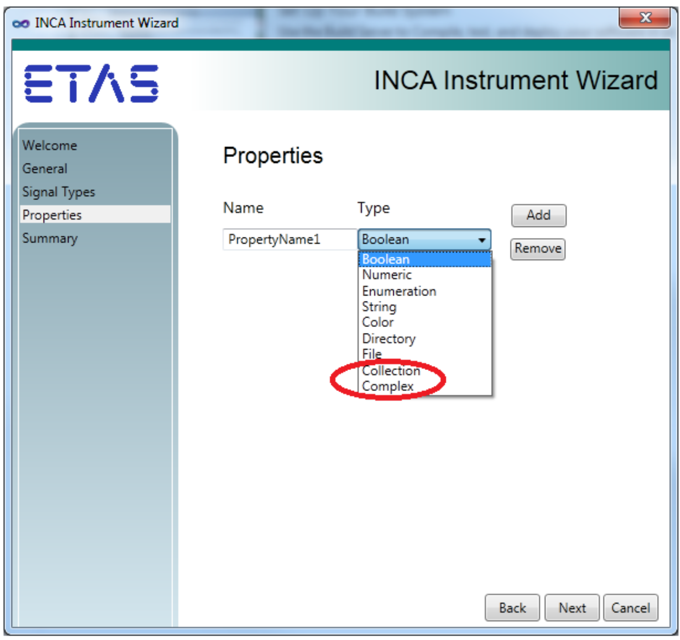
The Variable Class includes two new options:

- `Measurement` (read only access)
- `Calibration`
- `Adaptive Calibration`
- `Slewing`

The Variable Types allows eight different variable types:

- `Scalar:` single value variables
- `Vectors:` a list of values and
- `Arrays:` multi-dimensional array
- `Curve:` one dimensional look-up table
- `Map:` 2D loop-up table

– `Cuboid:` 3D and 4D look-up table
– `Axis:` distribution of axis points
– `CurveAxis:` special curve used as axis for look-up tables



InstrumentProperties consist of two parts:

– `Name` and
– `Type`

They are assigned to each other.

`Name` is the property identifier. It is a free text field and can be defined to your needs. `Type` can take one of nine optional values (ordered by complexity):

– `Boolean`
– `Numeric`
– `Enumeration`
– `String`
– `Color`
– `Directory`
– `File`
– `Collection`
– `Complex`

The properties you specify here are displayed in the Variable Selection Dialog on the **Display Configuration** tab in INCA.

You will therefore be able to edit the property values of your instrument in INCA.

NOTE

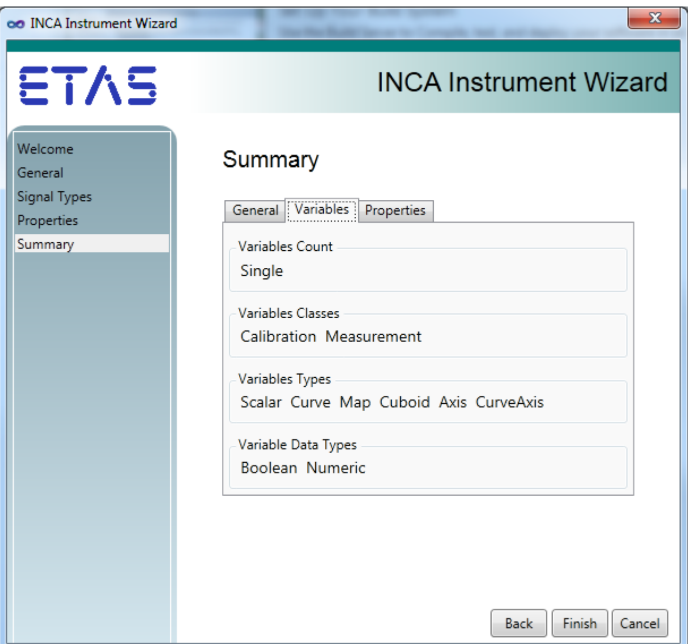
The properties can be changed in `InstrumentProperties.xml,` see chapter 0.

To define a new instrument property

1. Rename `PropertyName1` to `BackgroundColor`.
2. Change the `PropertyType` to `Color.`
3. Click **Add.**
4. Create the property `MyProperty2` with type `Boolean`.

5. Click **Next.**

   In the last step of the Instrument Wizard "Summary", all previously defined settings for `MyInstrument` are displayed for review.

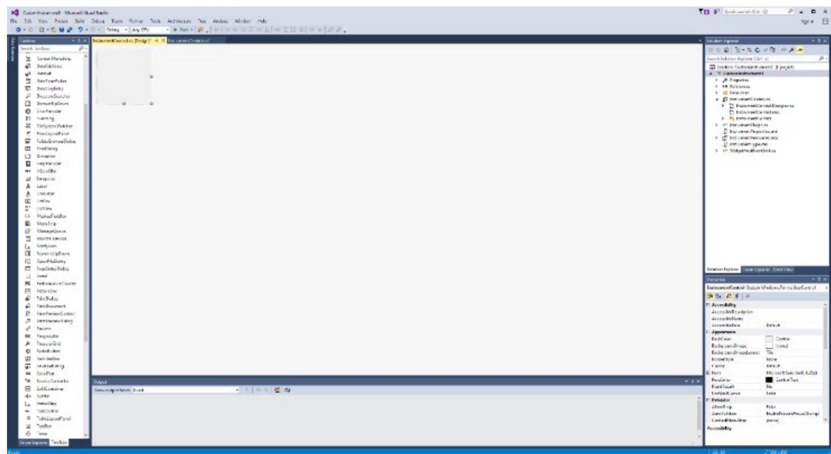


6. Check the instrument settings.
7. If you want to change something, click **Back.**

or

If you do not want to change anything, click **Finish**.

The generated `MyInstrument` control is created in Visual Studio:



The user interface (UI) is accessible via InstrumentControl.Designer.cs".

Visual Studio offers WYSIWYG design, so `MyInstrument` can be designed in the most suitable way to your needs.

The size of 150;150 is a default value in Visual Studio.

INCA, however, takes the value 100, 100 as default.

How to modify the window size of `MyInstrument` will be explained in chapter 4.2.2 Modifying the Window Size of `MyInstrument`.

The second part of the main user interface is the main class `InstrumentControl.cs`, containing the source code. The modification of this main class is explained in

NOTE

To integrate `MyInstrument` manually, see Chapter 9 Lesson 7 - Deploy Instruments on Other Systems

NOTE

It is possible to open the same instrument in INCA several times.

This is a useful feature to attribute different variables to the same instrument via the Variable Selection Dialog.

The different versions of the instrument will by default be indicated by incremented numbering.

Lesson 2 - Customizing `MyInstrument`.

NOTE

Visual Studio cannot process a build of your modifications if INCA is open.

Please make sure that INCA is always *closed* before you start build or rebuild.

To build the instrument

1. Make sure INCA is closed.
2. Select **Build** > **Build Solution**.

NOTE

The reference folder of your Visual Studio project contains a reference to the `ETAS.OpenEE.dll`.

This is the DLL file containing all interface classes for the communication with INCA.
You find it under:

`C:\ETAS\INCA7.5\bin\ETAS.OpenEE.dll`.

Now you can use the new `MyInstrument` in INCA. Note that INCA displays a blank user interface because no user controls have been added yet.

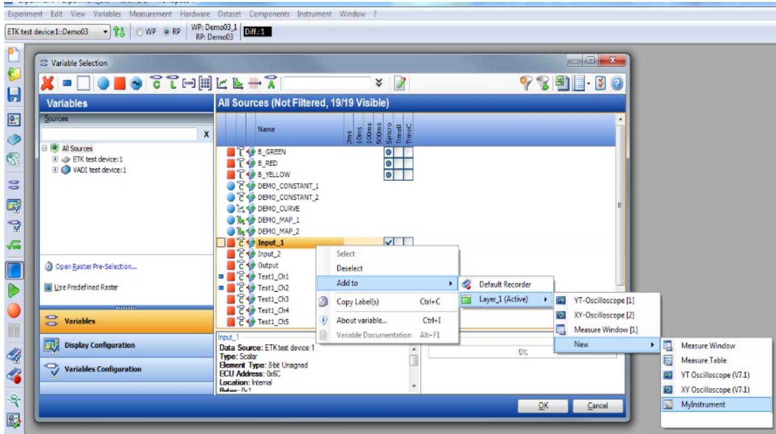
4.2 Integrating MyInstrument in INCA

With the Build, Visual Studio automatically copies the solution into the INCA plugin folder. If your ETASData folder is `D:\ETASData`, then it will be copied to `D:\ETASData\INCA7.5\Instruments`. INCA loads the instrument from this folder.

The instrument does not open in INCA unless you assign a variable or create the instrument in the Variable Selection Dialog.

To create variables in the Variable Selection Dialog

1. Restart INCA.
2. Open the Variable Selection Dialog.
3. In the "Variables" tab, select a variable you want to use in `MyInstruments`.
4. Right-click on desired variable. **Add to** > **Layer_1 (Active)** > **New** > `MyInstrument`.



5. Click **OK.**
6. The UI of `MyInstrument` is activated.
7. Choose the option **Variables,** see figure below.
8. Choose a variable you want to use in `MyInstrument`.
9. Right-click on the variable.
10. Follow the path as described in the screenshot Fig 3-11 below.

11. Click on `MyInstrument` to establish the link.
12. Click **Ok** in the Variable Selection Dialog to activate the UI of your `MyInstrument`.
    An instrument without controls in default size opens.

NOTE

To integrate `MyInstrument` manually, see Chapter 9 Lesson 7 - Deploy Instruments on Other Systems

NOTE

It is possible to open the same instrument in INCA several times.

This is a useful feature to attribute different variables to the same instrument via the Variable Selection Dialog.

The different versions of the instrument will by default be indicated by incremented numbering.

5 Lesson 2 - Customizing MyInstrument

5.1 Objectives

In this lesson, you learn the following:

- modifying an existing instrument via the Visual Studio UI Developer
- getting acquainted with the instrument's main classes and the `Property.xml` file
- getting an introduction to how the default code is organized

5.2 Changing the Instrument User Interface

There are two options to change the layout of the instrument's UI:

- via the Visual Studio UI Developer
- by programming components into the source code

The UI Developer of Visual Studio is a plugin to support the modification of instrument UIs.

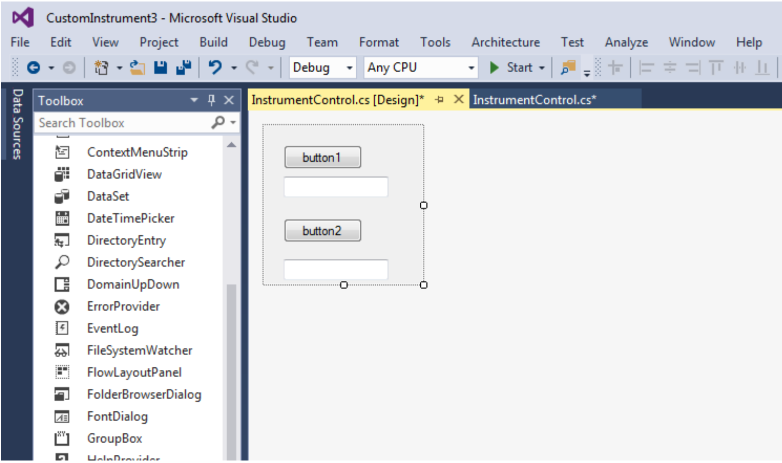
This tutorial describes the modification of the UI via the UI Developer of Visual Studio.

When you open the `InstrumentControl.cs` in the designer, you can use the Visual Studio Toolbox to add user controls to your UI.

5.2.1 Adding Components to MyInstrument

To add components to MyInstrument

1. Open the file `InstrumentControl designer` to display the UI.
2. Open the **Toolbox (Windows Forms)** of Visual Studio.
3. Choose two buttons and two textboxes from the **Toolbox** and place it at MyInstrument via drag and drop.



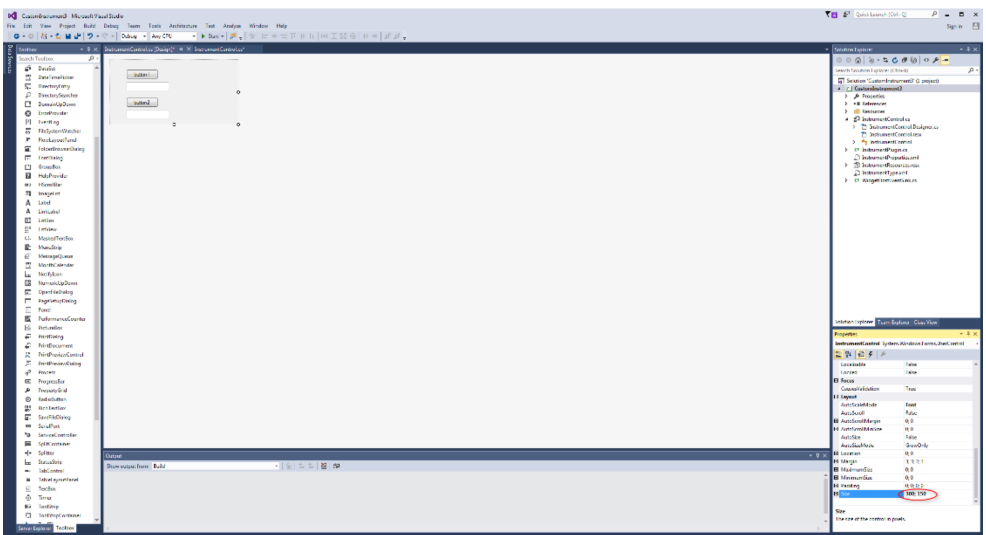
`MyInstrument` now looks like this in Visual Studio.

4. Change the size of the window.

5.2.2 Modifying the Window Size of MyInstrument

The window size is defined via the Instrument Control designer in Visual Studio when the instrument is first established.

The initial size of an instrument in INCA takes the default value of `PreferredSize` in `InstrumentType.xml`.



As displayed in the screenshots below, changes in the “Properties” window do not have an immediate effect on `PreferredSize` in `InstrumentType.xml`.

To modifying the window size of MyInstrument

You must modify the value `PreferredSize` in the `InstrumentType.xml` file if you want INCA to open `MyInstrument` in the size defined in Visual Studio UI Designer.

1. Go to the code line `PrefferedSize`.

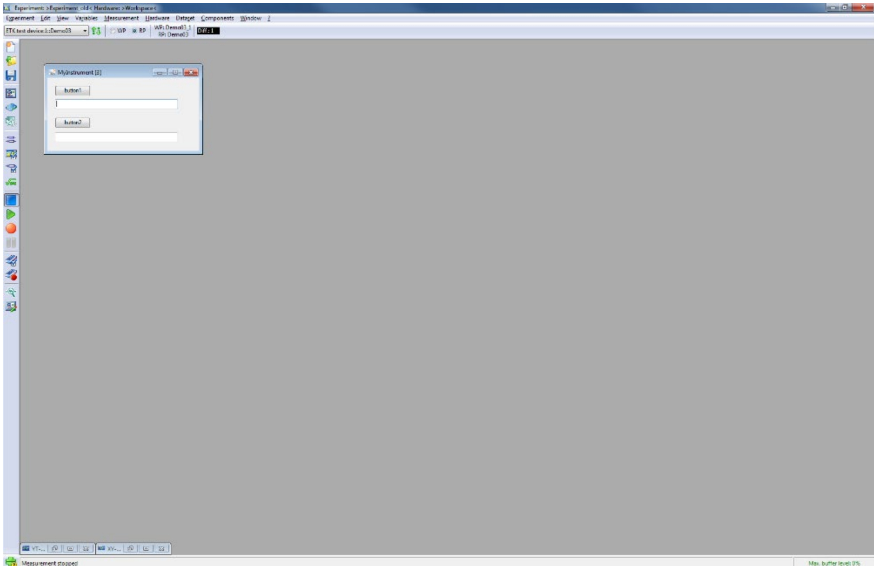
```
<?xml version="1.0" encoding="UTF-8"?>
<Widget Provider="MyCompany" Type="MyType"
DisplayName="Str_MyType" IconName="Ico_InstrumentIcon"
PreferredSize="100,100"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="c:\ETAS\INCA7.5\XML\schem
a\Etas.OpenEE.Widget.xsd">
```

2. Change the value to "`300,150`"

   You may also copy the values of the size property from the “**Properties**” window in the `Instrument Control designer` to `InstrumentType.xml`.

3. Click **Save** to store the layout.
4. Make sure INCA is closed.
5. Select **Build** > **Build Solution**.
6. Open INCA.
7. Create MyInstrument in Variable Selection Dialog.
8. Click **Ok.**

   Your modifications are now active in `MyInstrument`.



5.3 The Main Classes of MyInstrument

Instruments are organized in partial classes.

One partial class is the `InstrumentControl.Designer.cs` where user interface components are defined via graphical interface.

The second partial class is the `InstrumentControl.Designer.cs` which represents the code view of `InstrumentControl.Designer.cs` and handles all interaction between **INCA** and the instrument, such as actions, events, and the communication.

5.4 The Initialize Method of MyInstrument

The `Initialize` method is called to initialize instruments,

- i.e., by creating an instance of the instrument or
- when initializing the Experiment with an already existing instance.

The `WidgetHost` class is the main access of your instrument to INCA.

```
#region Implementation of IWidget
…
public void Initialize(IWidgetHost host)
{
    _WidgetHost = host;
    …

this.InitializeValueAccesses(initializedElementReferences,
null);
}
```

The `Initialize` method passes the main interface class `WidgetHost` to `MyInstrument`

To initialize MyInstrument

1. Open `InstrumentControl.cs`
2. Find the method `Initialize`.

The `InitializeValueAccess` method handles the creation of value accesses of assigned variables (see Fig 4-6 below). This method will be used and modified in Chapter 5 Lesson 3 - Access Variable Values.

3. Find the `InitializeValueAccesses` method.

```
private void InitializeValueAccesses(IList<IElementReference>
addedElements,
            IList<IElementReference> removedElements)
{ …
```

5.5 Implement Event Handlers"

The event handler `InstrumentControl_AssignedElementsChanged` in this example is called by the `IWidgetData.AssignedElementsChanged` event whenever variables are added or removed.

To attach the event handler

1. Find the highlighted code lines.

```
private void InstrumentControl_AssignedElementsChanged(object
sender,
            AssignedElementsChangedEventArgs e)
{
    this.InitializeValueAccesses(e.AddedElements,
e.RemovedElements);
}
```

The method host_SaveConfiguration handles all SaveConfiguration events initiated by WidgetHost.

To call the method

1. In the INCA experiment, click **Save**.
2. Open the Variable Selection Dialog, the Display Configuration Dialog, or the Variable Configuration Dialog.
3. Close the INCA Experiment.
4. Copy or move variables from one instrument to another.

   As an example, we want to save the value of the instrument's background color that is accessible through the `Windows.Forms.Control` property `BackColor`.

   By default, these code lines are part of an out commented passage.
5. Remove the outward comment indications /* and */ below
6. Verify if the correct passage is activated
7. Out comment again what is not needed by adding "//":

```
void host_SaveConfiguration(object sender,
System.EventArgs e)
{
…
 // save the configuration of the BackgroundColor.
 _WidgetHost.Configuration.Properties.SetProperty("BackgroundColor",
BackColor);
…
}
```

The Event Handler `host_LoadConfiguration` manages all `LoadConfiguration` events initiated by the `WidgetHost`.

The `host_LoadConfiguration` method is called whenever the instrument configuration has been changed (e.g., via editing in the Variable Selection Dialog) and the instrument shall adapt its representation.

In case an INCA Experiment gets loaded, the restoration of a formerly stored instrument configuration must be loaded within the `Initialize` method. To do so, call the `host_LoadConfiguration` also in the `Initialize` method.

In the `host_LoadConfiguration` Event Handler you describe the `Properties` you want to load. An example is given in an out commented passage.

To load the instrument's background color

1. Find the default method `host_LoadConfiguration`.
2. Remove the outward comment indications /* and */ below.
3. Check that the correct passage is activated.
4. Comment out what is not needed by adding "//".
5. To verify correct insertion, open `Instrument.Properties.xml`.
6. Find respective `PropertyDef Key`.
7. Copy the `PropertyDef Key` and insert accordingly in `InstrumentControl.cs` as described by highlighted code passage below.

```
…
    <PropertyGroup Type="MyTypeProperties">
      <PropertyDef Key="BackgroundColor"
DisplayKey="Str_BackgroundColor_DisplayKey">
        <Color DefaultValue="#CCCCCC" />
        <Description
DisplayKey="Str_BackgroundColor_Description" />
…
```

8. Insert the correct `PropertyDef Key` at the highlighted location".

```
void host_LoadConfiguration(object sender,
System.EventArgs e)
        {
…
            Color color;

if(_WidgetHost.Configuration.Properties.TryGetProperty("
BackgroundColor", out color))
            {
                BackColor = color;
            }
…
        }
```

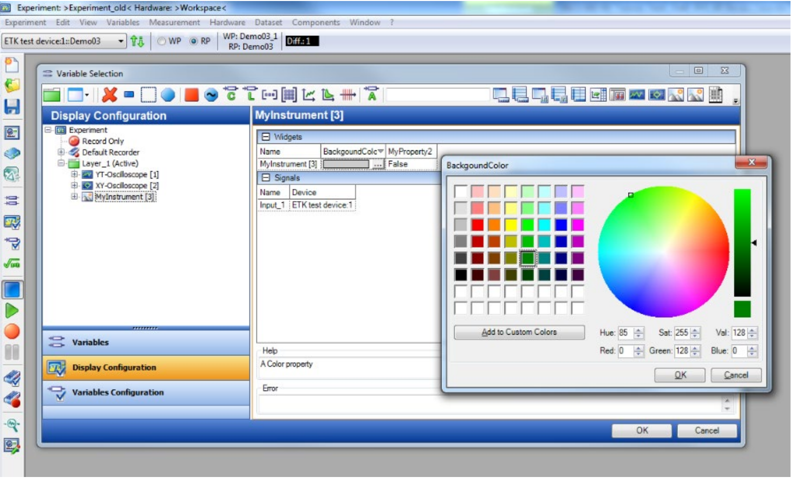
Make sure INCA is closed.

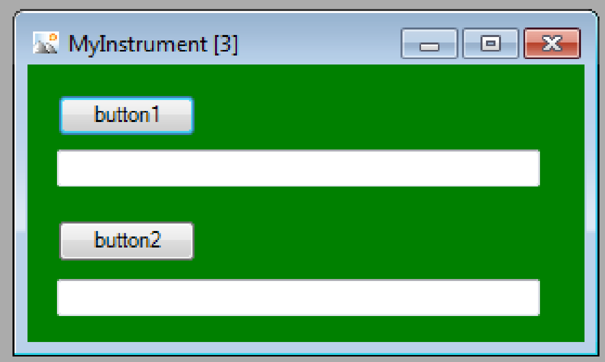
9. In Visual Studio, select **Build** › **Build Solution.**
10. Open INCA.
11. In the Variable Selection Dialog, call `MyInstrument`.
12. Click **Ok.**

    Your modifications are now active in `MyInstrument` and can be used in INCA.

To modify the properties

1. In the Display Configuration Dialog, call `MyInstrument`.
2. Click on `MyInstrument`.
3. Click into the “Color” field.
4. Change the `BackgroundColor` to green.
5. Click **OK.**





The `BackgroundColor` can now be changed at runtime in the VARIABLE SELECTION DIALOG.

5.6 Modifying InstrumentProperties.xml

All properties of your instrument plugin are defined in `InstrumentProperties.xml`. All properties defined in this file are visible in the INCA Variable Selection Dialog and can be edited at runtime of INCA.

To make the defined properties effective in the instrument user interface you must read and write the property values in the `LoadConfiguration` and `SaveConfiguration` methods of `InstrumentControl.cs` (see chapter above).

The values will automatically be loaded and saved with the INCA experiment.

5.6.1 Objectives

You will learn to add and change `Properties` and to change the properties values at runtime.

5.6.2 Adding Properties in InstrumentProperties.xml

The definition of instrument `Properties` are performed in XML. The `PropertyGroup` types inside `InstrumentProperties.xml` are referenced in the `InstrumentType.xml`.

To add an instrument property

1. In the Visual Studio Solution Explorer, double-click on `InstrumentProperties.xml`.
2. Find `PropertyDef Key “MyProperty2”`.

```
<PropertyDef Key="BackgroundColor"
DisplayKey="Str_BackgroundColor">
    <Color DefaultValue="#CCCCCC" />
    <Description
DisplayKey="Str_BackgroundColor_Description" />
</PropertyDef>
<PropertyDef Key="MyProperty2"
DisplayKey="Str_MyProperty2_DisplayKey">
    <Boolean BooleanFormat="TrueFalse"
DefaultValue="false" />
    <Description DisplayKey="Str_MyProperty2_Description" />
</PropertyDef>
```

NOTE

A `DisplayKey` is unique. INCA will ignore all `Properties` if you use it twice.

You add `Properties` by adding a new `Property` definition in XML syntax. Each `Property` needs a

- unique `Key`
- `DisplayKey`
- `DefaultValue`

NOTE

The types `Boolean` and `Numeric` have more attributes than the default value (you can see these attributes in the example above).

NOTE

To employ multilingualism as an option for your instrument please make sure to complete `InstrumentResources.resx` with the new `Properties`.

If not, INCA will always take the hard coded `DisplayKey` values.

3. Add a new `Property`.
4. Insert the highlighted code as given in the figure below:

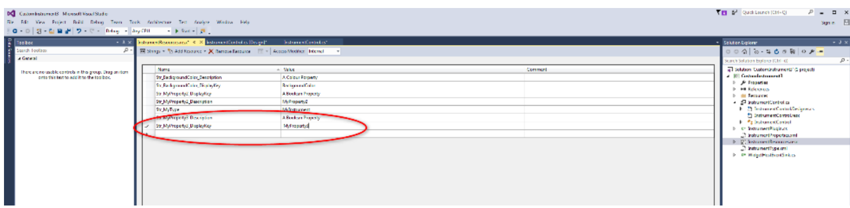
```
<PropertyDef Key="MyProperty3"
DisplayKey="Str_MyProperty3_DisplayKey">
          <Boolean DefaultValue="false" BooleanFormat="TrueFalse"/>
          <Description DisplayKey="Str_MyProperty3_Description"/>
</PropertyDef>
```

`MyInstrument` now provides a third property of type `boolean` for usage.

To keep localization in mind when customizing MyInstrument

If localization is an option for your instrument, do not forget to complete the `InstrumentResources.resx` file:

1. Double-click on `InstrumentResources.resx`.
2. Insert the new display keys in the `InstrumentResources.resx` table.



3. Make sure INCA is closed.
4. In Visual Studio, select **Build** > **Build Solution**.
5. Open INCA.
6. In the Variable Selection Dialog, call `MyInstrument`.
7. Click **Ok.**

   Your modifications are now active in INCA.

5.6.3 Adding an Enumeration Item

To add an enumeration item

1. Find the highlighted code lines in the commented-out default code.
2. Copy them into the active `PropertyDef`.

```
<PropertyDef Key="CalibrationColor"
DisplayKey="Str_CalibrationColor_DisplayKey">
    <Enumeration DefaultValue="0">
        <EnumerationItem Value="0" DisplayKey="red"/>
        <EnumerationItem Value="1" DisplayKey="green"/>
        <EnumerationItem Value="2" DisplayKey="blue"/>
    </Enumeration>
</PropertyDef>
```

NOTE

If changes are directly performed in the `InstrumentProperties.xml` file, Visual Studio will validate them directly against the current definitions to report errors.

5.6.4 Defining SignalProperties

`Properties` can be edited on the “Widget” level and on the “Signal” level. By default, a simple editor is already integrated.

INCA offers the option to define `Properties` at “Signal” level in more detail.

To define `SignalProperties` in `InstrumentProperties.xml`

1. Open `InstrumentProperties.xml`.
2. Implement the highlighted code lines:

```
…
   <PropertyGroup Type="SignalProperties">
      <PropertyDef Key="SignalColor"
DisplayKey="Str_SignalColor_DisplayKey">
        <Color DefaultValue="#CCCCCC" />
        <Description DisplayKey="Str_SignalColor_Description" />
      </PropertyDef>
    </PropertyGroup>
…
```

All `Properties` can be assigned to `Signals`. It is possible to assign several `Properties` to one `Signal`.

To reference `SignalProperties` in `InstrumentType.xml`

1. Open `InstrumentType.xml`.
2. Implement the highlighted code line.

```
<?xml version="1.0" encoding="UTF-8"?>
<Widget Provider="MyCompany" Type="MyInstrumentType"
DisplayName="Str_MyInstrument"
IconName="Ico_InstrumentIcon" PreferredSize="300,150"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="c:\ETAS\INCA7.5\XML\schem
a\Etas.OpenEE.Widget.xsd">
  <Properties RefType="MyTypeProperties" />
  <WidgetDataTypes>
    <WidgetData>
      <PossibleSignals MinVariableCount="1"
MaxVariableCount="1">
        <Properties RefType="SignalProperties"/>
          <Signal>
…
          </Signal>
      </PossibleSignals>
    </WidgetData>
  </WidgetDataTypes>
</Widget>
```

3. Make sure INCA is closed.
4. In Visual Studio, select **Build** > **Build Solution**.
5. In Visual Studio, select **Build** > **Build Solution**.
6. Open INCA.
7. In the Variable Selection Dialog, call `MyInstrument`.

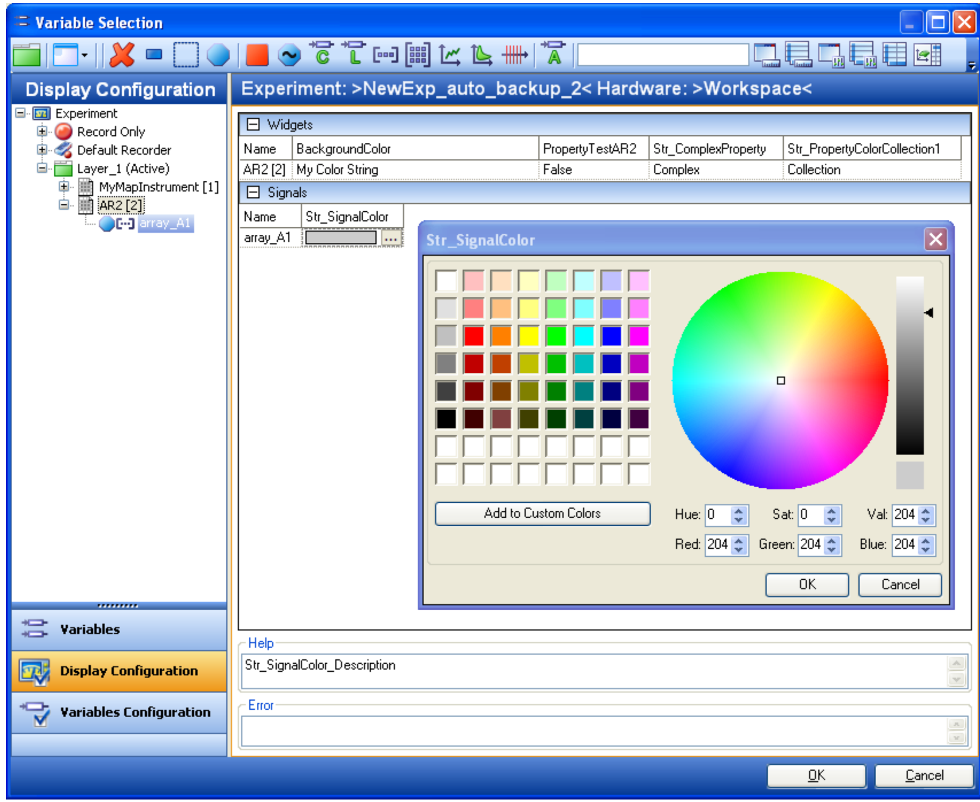
8. Click **Ok.**
   Your modifications are now active in `MyInstrument`.

All signals assigned to `MyInstrument` will now show an individual property in the Variable Selection Dialog at runtime, in our example a color.

The Variable Selection Dialog now presents a new main area:

– The “Widget” level as usual (top).
– The “Signal” level with an additional header.

The `SignalProperty Color` is displayed as presented in INCA in the figure below:



In the source code of the instrument, you can access the value of the `SignalProperties` at the `IElementReference.Properties` element of the assigned variables.

5.6.5 Adding Collection and Complex Properties

INCA V7.1 introduces two new properties:

- `Collection`
- `Complex`

In this chapter, you learn to implement these features and what options they offer.

NOTE

The Collection Editor and the Complex Editor can be attached to the "Widget" or to the "Signal" level (see `SignalProperty` editor definition chap. 4.6.5) related to the `PropertyGroup` in which they will be implemented in `InstrumentProperties.xml`.

Our example shows the `Widget level`.

To implement a collection property in `InstrumentProperties.xml`

1. Implement the highlighted code lines.

```
      <PropertyDef Key="PropertyColorStringCollection1"
DisplayKey="Str_PropertyColorStringCollection1">
          <Collection>
            <CollectionItem Key="ColorStringCollection1">
              <PropertyDef Key="Color" DisplayKey="Str_Color">
                <Color DefaultValue="#CCCCCC" />
                <Description DisplayKey="Str_Color_Description" />
              </PropertyDef>
              <PropertyDef Key="String" DisplayKey="Str_String">
                <String DefaultValue="MyCompany" />
                <Description DisplayKey="Str_String_Description" />
              </PropertyDef>
            </CollectionItem>
          </Collection>
        </PropertyDef>
```

   A property has been defined which represents a list of `Color` and `String` property pairs.

2. Make sure INCA is closed.
3. In the Visual Studio, select **Build > Build Solution**.
4. Open INCA.
5. In the Variable Selection Dialog, call `MyInstrument.`
6. Click **Ok.**
   Your modifications to `MyInstrument` are now active in INCA.

NOTE

A `Collection` can contain more than one `PropertyType`.

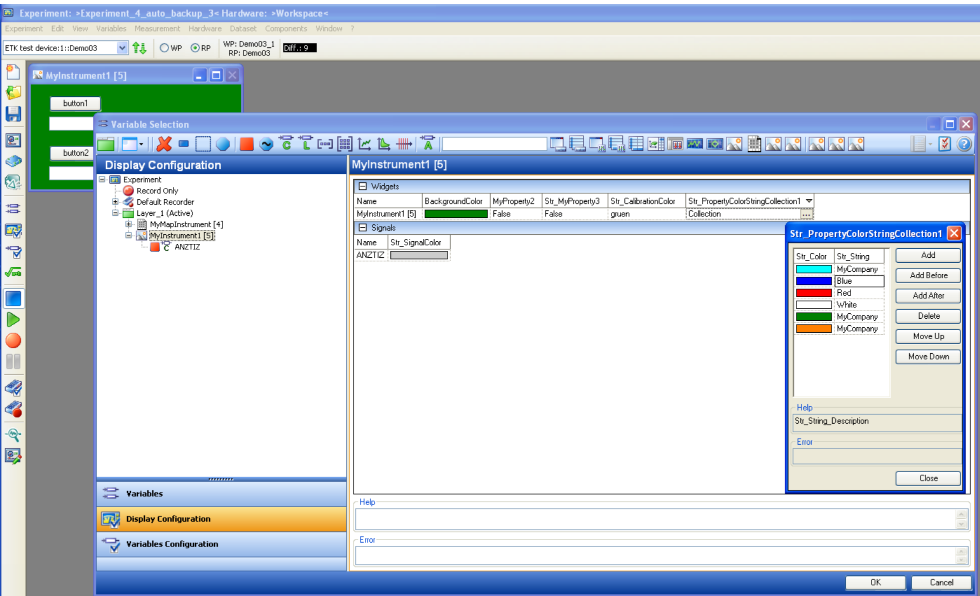
NOTE

Make sure to attribute unique `Keys` and `DisplayKeys`. This feature does not work otherwise.

NOTE

Make sure to integrate the new `DisplayKeys` into `InstrumentResources.resx` if multilingualism is an option or if you want to see speaking names for the properties in the Variable Selection Dialog.

`MyInstrument` now displays the new Collection Editor on “Widget” level in the Variable Selection Dialog:



As displayed in the figure above, INCA offers a new editor listing the items defined in `ColorStringCollection1`. The options **Add**, **Delete,** and **Move** are offered within the collection list.

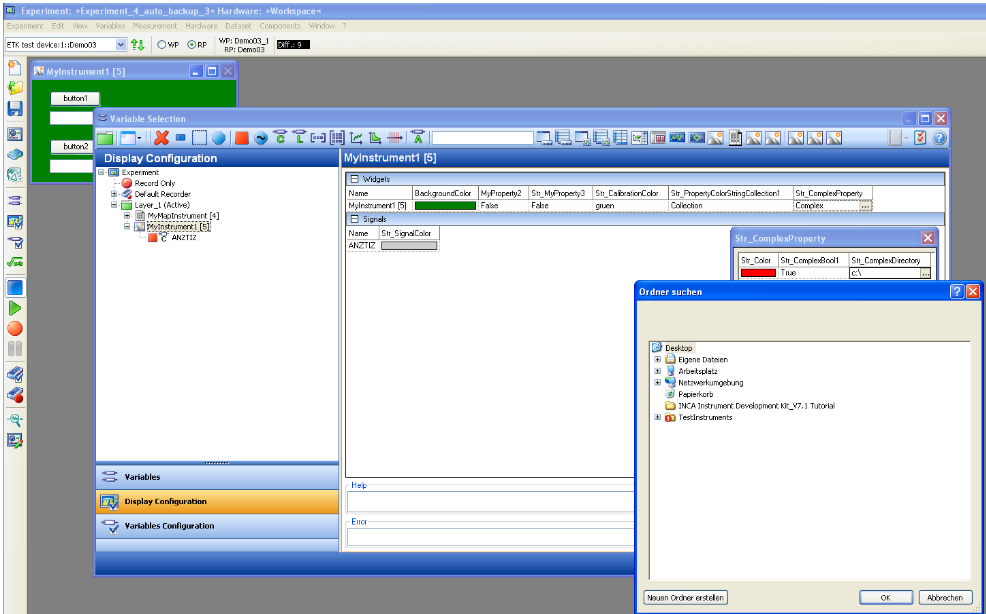
The Collection Editor is working at runtime. You can perform modifications during the use of `MyInstrument`.

To define a complex property in `InstrumentProperties.xml`

1. Implement the highlighted code lines:

```
<PropertyDef Key="ComplexProperty" DisplayKey="Str_ComplexProperty">
          <Complex>
             <PropertyDef Key="Color" DisplayKey="Str_Color">
                <Color DefaultValue="#CCCCCC" />
                <Description DisplayKey="Str_Color_Description" />
             </PropertyDef>
             <PropertyDef Key="ComplexBool1"
DisplayKey="Str_ComplexBool1">
                <Boolean BooleanFormat="TrueFalse" DefaultValue="false"
/>
                <Description
DisplayKey="Str_ComplexBoolean1_Description" />
             </PropertyDef>
             <PropertyDef Key="ComplexDirectory"
DisplayKey="Str_ComplexDirectory">
                <Directory DefaultValue="c:\"/>
                <Description
DisplayKey="Str_ComplexDirectory_Description" />
             </PropertyDef>
          </Complex>
</PropertyDef>
```

   A structure of several different `Properties` has now been defined that will be edited together in one editor.
2. Make sure the new `DisplayKeys` is integrated in `InstrumentResources.resx` if multilingualism is an option.
3. Make sure INCA is closed.
4. In the Visual Studio, select **Build** > **Build Solution.**
5. Open INCA.
6. In the Varaiable Selection Dialog, select `MyInstrument.`
7. Click **Ok.**
   Your modifications are now active in `MyInstrument`.
   `MyInstrument` now displays the new Complex Editor in the Variable Selection Dialog:



As displayed in the figure above, the Complex Editor groups the defined `Properties`. The example above displays the option `Directory`, offering access to the Windows Explorer.

The editor options differ in relation to the defined `Properties`.

The Complex Editor is working at runtime. You can perform modifications during the use of `MyInstrument`.

5.6.6 Defining a Custom Property Editor

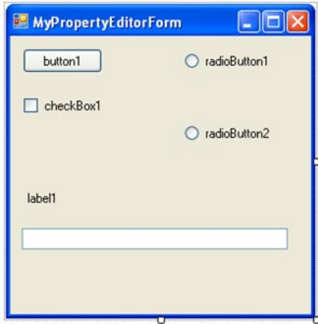
In this chapter, you learn to define a custom property editor. The custom property editor can be designed for every property type. The feature is especially useful for `Complex` properties. In our example we attribute a custom editor to the `Color` property to concentrate on the basic program structure.

The necessary steps are:

- implement `MyPropertyEditorForm`
- declare the `UIEditor` “`MyPropertyEditor`” for the `Property`
- implement the respective `Class` and `Converter`
- define display options

To create a customer editor: MyPropertyEditorForm

1. In the Solution Explorer, right-click on `MyInstrument`.
2. Click **Add.**
3. Choose `Windows.Forms`.
4. Name Forms file `MyPropertyEditorForm`.
5. Click **Add.**
6. Drag controls into the form, see example below.
   Visual Studio creates a new Windows Forms class.
   The Editor has a default value in Visual Studio of 300;300.



7. Open `MyPropertyEditorForms` code view.
   Right-click on UI.
   Choose the "View Code" option in the Context Menu.
8. Insert the highlighted code lines.
9. Verify the code.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace MyCompany.MyInstrument
{
    public partial class MyPropertyEditorForm :
UserControl
    {
        public MyPropertyEditorForm()
        {
            InitializeComponent();
        }
        public Color Color { get; set; }
    }
}
```

`MyInstrument` has a new class: `MyPropertyEditorForm`

To implement MyPropertyEditor

1. To create a new class right-click on `MyInstrument` in Visual Studio
2. Select **Solution Explorer** > **Add** > **Class.**
   A window opens.
3. Name it `MyPropertyEditor.`
4. Verify your selection.
5. Click **Add.**
6. `MyPropertyEditor.cs` opens.
7. Let your class inherit from the interface `IPropertyUIEditor`.
8. Implement the highlighted code lines.

```
…
using Etas.OpenEE;
using System.Drawing;
…
namespace MyCompany.MyInstrument
{
    public class MyPropertyEditor : IPropertyUIEditor
    {
        private ISimpleProperty<Color> m_Color;
        private MyPropertyEditorForm m_Editor;
        public bool CompleteEditing(bool isCanceled)
        {
            if (!isCanceled)
                m_Color.Value = m_Editor.Color;
            return true;
        }
        public System.Globalization.CultureInfo CultureInfo
        {
            set { }
        }
        public PropertyUIEditorDisplayType DisplayType
        {
            get { return PropertyUIEditorDisplayType.Popup; } //
"Popup" is 1 of 3 options!
        }
        public event EventHandler<PropertyUIEditorClosedEventArgs>
EditorClosed;

        public System.Windows.Forms.Control EditorControl
        {
            Get
            {
                if (m_Editor == null)
                    m_Editor = new MyPropertyEditorForm();
                    m_Editor.Color = m_Color.Value;
                return m_Editor;
```

```
                }
            }
            public IConfigObjectPath ObjectPath
            {
                set { }
            }
            public IProperty Property
            {
                set
                {
                    m_Color = (ISimpleProperty<Color>)value;
                }
            }
            public void StartEditing(string initialValue)
            {
            }
            public void Dispose()
            {
            }
        }
}
```

`MyInstrument` has a new class: `MyPropertyEditor.cs`

To implement MyPropertyConverter

1. Create a new class.
2. Name it `MyPropertyConverter`.
3. Implement the highlighted code lines.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Etas.OpenEE;
using System.Drawing;

namespace MyCompany.MyInstrument
{
    public class MyPropertyConverter :
IPropertyConverter<Color>
    {
        public bool CanConvertPropertyValueType(Type
typeOfPropertyValue)
        {
            return true;
        }
        public void Initialize(IPropertyConverterContext context)
        {
        }
        public bool TryGetDisplayValue(IPropertyConverterContext
context, IProperty property, Color propertyValue, out string
displayValue)
        {
            displayValue = "My Color String";
            return true;
        }
        public bool TryGetPropertyValue(IPropertyConverterContext
context, IProperty property, string displayValue, out Color
propertyValue)
        {
            propertyValue = Color.Red;
            return false;
        }
    }
}
```

`MyInstrument` has a new class: `MyPropertyConverter.cs`.

To declare the new classes in InstrumentProperties.xml

1. Implement the highlighted code lines.

```
…
<PropertyDef Key="BackgroundColor"
DisplayKey="Str_BackgroundColor_DisplayKey">
        <Color DefaultValue="#CCCCCC" />
        <Description
DisplayKey="Str_BackgroundColor_Description" />
        <UIEditor Type="MyPropertyEditor"/>
        <Converter Type="MyPropertyConverter"/>
</PropertyDef>
…
```

Assigned to the property is now the specific `UIEditor MyPropertyEditor`.

`MyPropertyConverter` is needed to convert the `property` into a `string` so it can be displayed in the field without being in edit mode.

To implement the necessary extensions to InstrumentPlugins.cs

1. Let your `InstrumentPlugin.cs` inherit form `IPropertyUIEditorFactory` and `IPropertyConverterFactory`.
2. Implement the highlighted code lines.

```
using System.Resources;
using System.Xml.Linq;
using Etas.OpenEE;

namespace MyCompany.MyInstrument
{
    public class InstrumentPlugin : IWidgetPlugin,
IPropertyConverterFactory, IPropertyUIEditorFactory
    {
        #region IWidgetPlugin Members
…
        public XElement TypeDescription
        {
            get;
            private set;
        }
        public XElement PropertiesDescription
        {
            get;
            private set;
        }
        public void Initialize(IWidgetPluginHost
pluginHost)
        {
            TypeDescription =
XElement.Parse(InstrumentResources.InstrumentType);
```

```
            PropertiesDescription =
XElement.Parse(InstrumentResources.InstrumentProperties)
;
        }
        public void DeInitialize()
        {
        }
        public ResourceManager Resources
        {
            get { return new
ResourceManager(typeof(InstrumentResources)); }
        }
        public IWidget CreateInstrument()
        {
            return new InstrumentControl();
        }
        #endregion
        public IPropertyUIEditor CreatePropertyUIEditor(string
editorName, System.Collections.Generic.IDictionary<string, string>
parameters)
        {
            return new MyPropertyEditor();
        }
        public IPropertyConverter CreatePropertyConverter(string
converterName)
        {
            return new MyPropertyConverter();
        }
    }
}
```

`MyInstrument` now has the `UIEditor` `MyPropertyEditor` for the feature `BackgroundColor` property.

To define UIEditorDisplayTypes

The `DisplayType` property of the `MyPropertyEditor` class defines the `DisplayType` of the `PropertyUIEditor`:

1. Find the highlighted code passage:

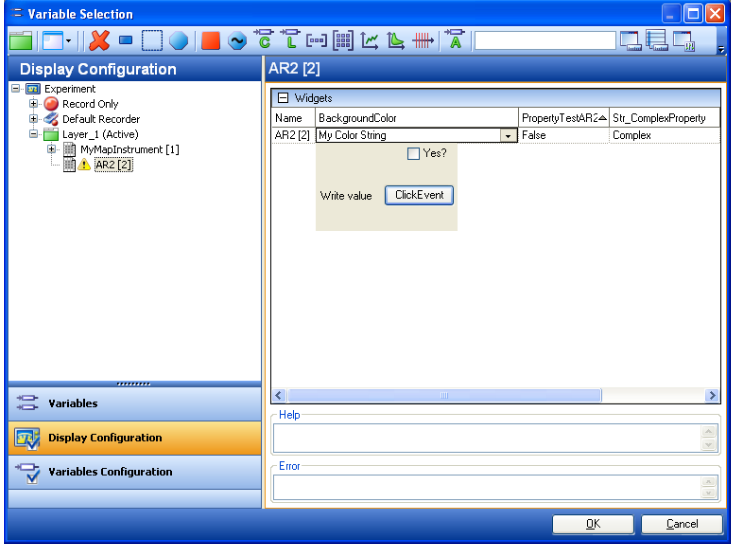
```
        public PropertyUIEditorDisplayType DisplayType
        {
            get { return PropertyUIEditorDisplayType.Popup; }
// "Popup" is 1 of 3 options!
        }
```

   The following options are available:
   - `Popup`
   - `Inline`
   - `Dialog`

2. Find the following code snippet:

```
...
            get { return
PropertyUIEditorDisplayType.Popup; } // "Popup" is 1 of 3
options!
...
```

3. Make sure INCA is closed.
4. In Visual Studio, select **Build** > **Build Solution**.
5. Open INCA.
6. In the Variable Selection Dialog, call `MyInstrument`.
7. Click **Ok.**
   The `MyPropertyEditor` is displayed:

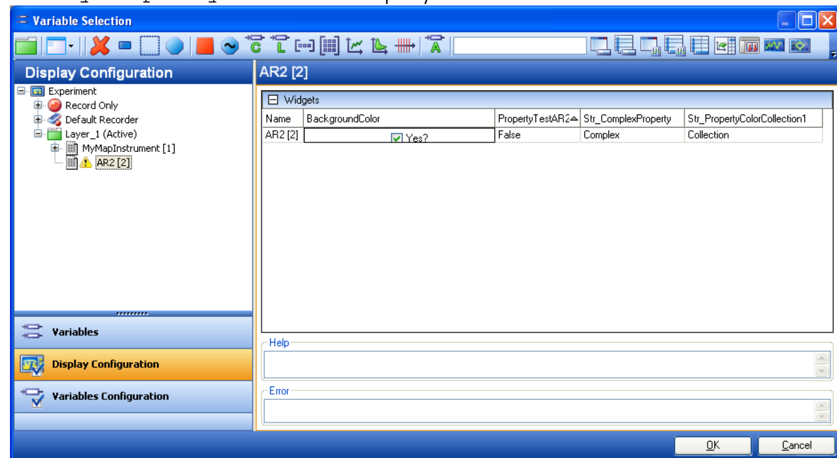


8. Modify `UIEditorDisplayType` to `Inline:`

```
...
            get { return
PropertyUIEditorDisplayType.Inline; }
...
```

9. Make sure INCA is closed.
10. In Visual Studio, select **Build** > **Build Solution**.
11. Open INCA.
12. In the Variable Selection Dialog, call `MyInstrument`.
13. Click **Ok.**

    The `MyPropertyEditor` is displayed



    As presented in the two figures above the `Inline` option requires special care in usage.

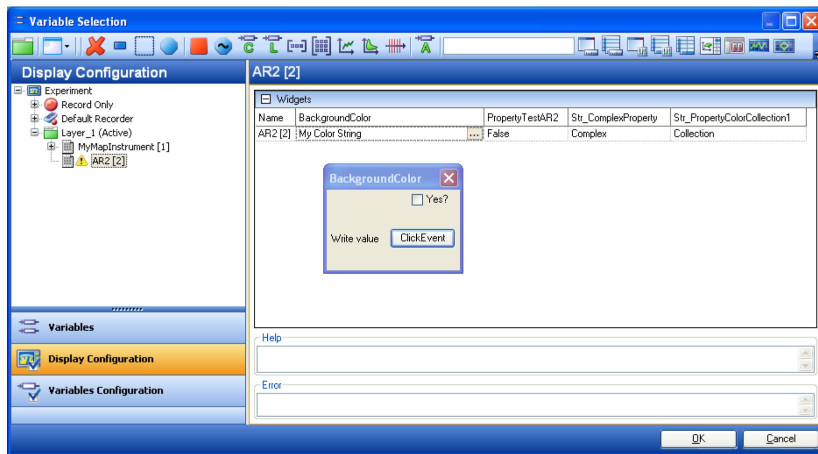
    `Inline` option places the `MyPropertyEditor` directly into the line of the property in the Display Configuration. If the editor size exceeds this limited space (as in our example), the overlap of the editor will not be displayed and therefore will remain inaccessible for usage.

14. Modify `PropertyUIEditorDisplayType` to `Dialog:`

```
...
            get { return
PropertyUIEditorDisplayType.Dialog; }
...
```

15. Make sure INCA is closed.
16. In Visual Studio, select **Build** > **Build Solution**.
17. Open INCA.
18. In the Variable Selection Dialog, call `MyInstrument`.
19. Click **Ok.**

`MyPropertyEditor` is displayed.



As presented in the figure above, the `Dialog` option creates a dialog box displaying the defined `MyPropertyEditor`.
Make sure to define a proper size for the dialog box in Visual Studio (see chapter 4.2.2 Modifying the Window Size of **MyInstrument**).

5.6.7 Define PropertyDependencies

INCA offers the option to define dependencies between different `Properties`. By this means it is possible to define that e.g., a certain color setting for one `Property` switches to another `Property` to read only access, disables or enables certain submenus etc.

In this chapter, you learn to define `Dependencies` between `Properties`.

To declare the necessary classes

1. To create a new class in Visual Studio right-click on the instrument.
2. Select **Solution Explorer** › **Add** > **Class.**
   A dialog window opens.
3. Name one class `MyConfiqHandler.cs.`
4. Verify your selection.
5. Click **Add.**

Visual Studio creates the default class **MyConfigHandler**.

To configure MyConfigHandler.cs

1. Open `MyConfigHandler.cs.`
2. Modify to the highlighted code passages:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Etas.OpenEE;

namespace MyCompany.MyInstrument
   public class MyConfigHandler : IWidgetConfigHandler
   {
```

```
        private IWidgetConfig _widgetConfig;
        private IWidgetConfigContext _widgetConfigContext;

        private ISimpleProperty<bool> _myProperty2;
        private ISimpleProperty<bool> _myProperty3;

        private ISimplePropertyInfo _myPropertyInfo2;
        private ISimplePropertyInfo _myPropertyInfo3;

        public void Initialize(IWidgetConfig configuration,
EWidgetConfigState widgetConfigState, IWidgetConfigContext context)
        {
            _widgetConfig = configuration;
            _widgetConfigContext = context;

            if
(_widgetConfig.Properties.TryGetSimpleProperty("MyProperty2", out
_myProperty2) &&

_widgetConfig.Properties.TryGetSimpleProperty("MyProperty3", out
_myProperty3))
            {
                IPropertyInfo propInfo2;
                if
(context.PropertyInfoAccess.TryGetPropertyInfo(_myProperty2, out
propInfo2) &&
                   propInfo2 is ISimplePropertyInfo)
                {
                    _myPropertyInfo2 = propInfo2 as
ISimplePropertyInfo;
                }

                IPropertyInfo propInfo3;
                if
(context.PropertyInfoAccess.TryGetPropertyInfo(_myProperty3, out
propInfo3) &&
                   propInfo3 is ISimplePropertyInfo)
                {
                    _myPropertyInfo3 = propInfo3 as
ISimplePropertyInfo;
                }
                _myProperty2.ValueChanged +=
_myProperty2_ValueChanged;
            }
        }
        void _myProperty2_ValueChanged(object sender, EventArgs e)
        {
            var newValue = _myProperty2.Value;
```

```
            // Change the read only state of MyProperty3
according
            // to the value of MyProperty2
            _myPropertyInfo3.IsReadOnly = newValue;
        }
        public void DeInitialize()
        {
            if (_myProperty2 != null)
            {
                _myProperty2.ValueChanged -=
_myProperty2_ValueChanged;
            }
        }
    }
}
```

You have now implemented a `dependency` between `MyProperty2` and `MyProperty3` setting `MyProperty3` to read only if `MyProperty2` changes.

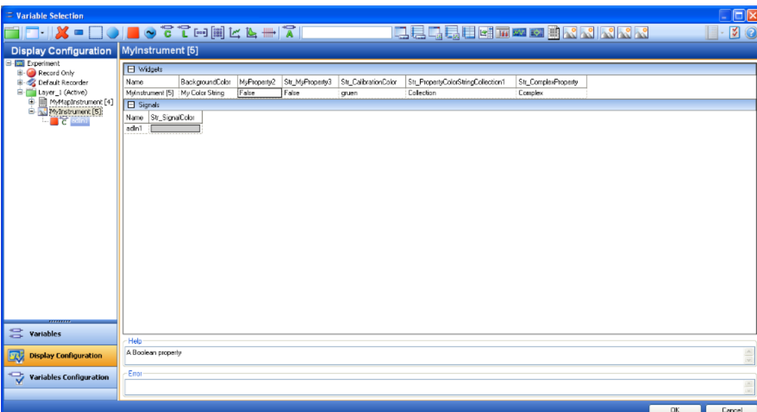
To declare the new classes in InstrumentPlugin.cs

1. Let your plugin inherit from `IWidgetConfigHandlerFactory`:

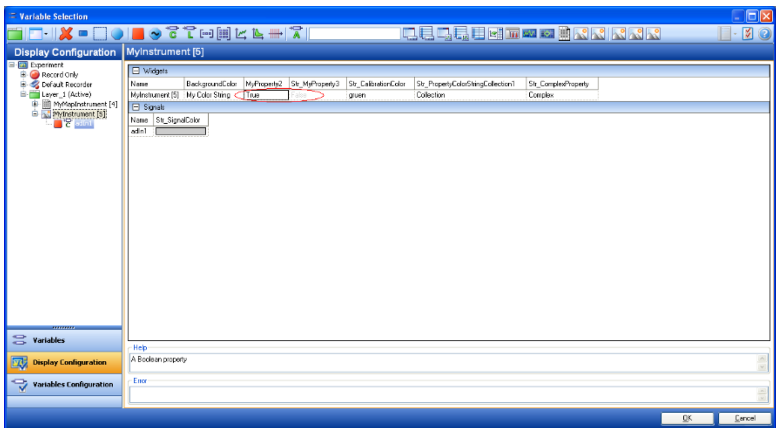
```
InstrumentPlugin : IWidgetPlugin,
IPropertyUIEditorFactory, IPropertyConverterFactory,
IWidgetConfigHandlerFactory
…
#endregion
…
public IWidgetConfigHandler CreateWidgetConfigHandler()
         {
             return new MyConfigHandler();
         }
     }
```

2. Make sure INCA is closed.
3. In Visual Studio, select **Build** > **Build Solution.**
4. Open INCA.
5. In the Variable Selection Dialog, call `MyInstrument`.
6. Click **Ok.**

   In INCA the `dependency` now works as represented below:



7. Change value of `MyProperty2` to `True`



6 Lesson 3 - Access Variable Values

6.1 Objectives

In this chapter, you learn to access `VariableValues` from `MyInstrument`.

This is the central feature of the interface. This lesson and the following exemplify all basic options.

6.2 Changing Properties in InstrumentType.xml

The `InstrumentTypes` are organized in XML like the `InstrumentProperties`. In the `InstrumenType.xml` you can describe which types and classes of variables from INCA should be supported in `MyInstrument`.

To open the `InstrumentType.xml` select
File > **Open** > **File** > `InstrumentType.xml`.

6.2.1 Modifying VariableCount

In the source code, the three different options of the wizard are represented as shown below:

```
...
<PossibleSignals MinVariableCount="0" MaxVariableCount="0">
...
```

> `VariableCount` option `None`.

```
...
<PossibleSignals MinVariableCount="1" MaxVariableCount="1">
...
```

> `VariableCount` option `Single`.

```
...
<PossibleSignals MinVariableCount="1" MaxVariableCount="10">
...
```

> `VariableCount` option `Multiple`.

> Set the `MaxVariableCount` to 10.

NOTE

Changing the `InstrumentType.xml` directly overwrites the settings defined in the wizard when creating the `MyInstrument`.

6.2.2 Adjusting the Supported VariableClasses

In the `InstrumentType.xml` the definitions of the `VariableClass` that `MyInstrument` supports are listed. At least one definition must added, but also several or all are possible:

```
<VariableClasses>
<VariableClass Value="Calibration"/>
<VariableClass Value="Measurement"/>
</VariableClasses>
```

Further options are:

- `Adaptive Calibration` and
- `Slewing`
  - Add `AdaptiveCalibration`

```
…
<VariableClass Value="AdaptiveCalibration"/>
…
```

6.2.3 Adjusting the Supported VariableTypes

Since the 7.1 version INCA supports eight different types. They are:

- `Scalar`
- `Vector`
- `Array`
- `Curve`
- `Map`
- `Cuboid`
- `Axis`
- `Curve Axis`

The `VariableType` defines the type of values `MyInstrument` will be able to contain.

- Find the following code lines

```
<VariableTypes>
   <VariableType Value="Scalar"/>
</VariableTypes>
```

- Add `Vector` and `Array` as `VariableType`

```
…
   <VariableType Value="Vector"/>
   <VariableType Value="Array"/>
…
```

6.2.4 Changing ElementDataType

The `ElementDataType` describes the `DataType` of a `Variable MyInstrument` supports. Three options are possible:

- `Numeric:` the variable supports `numeric` values
- `Boolean:` the variable can take the values `true` or `false`
- `Ascii :` the variable contains `strings`
- Find the following code lines

```
<ElementDataTypes>
   <ElementDataType Value="Numeric" />
   <ElementDataType Value="Boolean" />
</ElementDataTypes>
```

- Add `ElementDataType` “`Ascii`”

```
…
   <ElementDataType Value="Ascii" />
…
```

The combination of `VariableClass`, `VariableType` and `ElementDataType` is specifying which variables `MyInstrument` can contain.

In this example, you can assign up to 10 `Measurement` or `Calibration` variables of type `Scalar, Vector` or `Array` with the data type `Numeric, Boolean` or `Ascii`.

1. Make sure INCA is closed.
2. In Visual Studio, select **Build** > **Build Solution.**
3. Open INCA.
4. In the Variable Selection Dialog, call `MyInstrument`.
5. Click **Ok.**
   Your modifications are now active in `MyInstrument`.

6.2.5 ValueAccess for MyInstrument

WARNING

Risk of unexpected vehicle behavior

Calibration activities influence the behavior of the ECU and the systems that are connected to the ECU.
This can lead to unexpected vehicle behavior, such as engine shutdown as well as breaking, accelerating, or swerving of the vehicle.
Only perform calibration activities if you are trained in using the product and can assess the possible reactions of the connected systems.

`ValueAccess` options require a declaration of the respective object in the `global declaration`.

To activate the ValueAccess declaration

1. Open the `InstrumentControl.cs.`

2. Find the highlighted code lines.
3. Remove comment indication.

```
              // global declaration
…
private ISignalValueAccessScalar<double> _valueScalarAccessExample;
…
```

6.2.6 Survey in Read Only and Read/Write Access of Variables

The `ETAS.OpenEE` library offers interfaces for read only or read / write access. The options are related to the `VariableType`s.

Variable-Type	Read only ValueAccess	Read / Write ValueAccess
`Scalar`	`IValueAccessScalar<T>`	`ISignalValueAccessScalar<T>`
`Array`	`IValueAccessArray<T>`	`ISignalValueAccessArray<T>`
`Ascii`	`IValueAccessAscii<T>`	`ISignalValueAccessAscii<T>`

Tab. 5-1 The ValueAccess interfaces for read only and read/ write access

The more complex `DataTypes` / `VariableTypes` are also mapped to the `ValueAccess` interfaces listed above (see also Lesson 8 - How to Access Maps, Curves and Axes).

6.2.7 The InitializeValueAccess Method

The method `InitializeValueAccess` manages the specific `ValueAccess` of all `variables` assigned to `MyInstrument`.

To activate the access on variable values

1. Find the following code lines

```
private void
InitializeValueAccesses(IList<IElementReference>
addedElements,
              IList<IElementReference> removedElements)
{
…
```

2. Remove the comment out signs of the passage
   The default code is activated.

To define the value to be modified

With the following command lines, you can view the `type` (`Scalar, Array, Ascii`...) and the `class` (`Measurement`, `Calibration`...) of the given element.

`Type` and `class` of the `ElementInfo` are taken to distinguish between several `types` and `classes` to get the correct `value`.

1. Find the following code lines

```
    // With this command lines, you can get the type
(Scalar, Array, Ascii,...) and the class (calibration,
         // measurement) of the given element reference.
    EElementType elementType =
addedElements[0].ElementInfo.Type;
    EElementClass elementClass =
addedElements[0].ElementInfo.Scalar.ElementClass;
```

> **NOTE**
>
> When trying to access a type specific property member of the `ElementInfo` (e.g. `ElementInfo.Scalar, ElementInfo.Array, ElementInfo.Ascii`, ...) that does not correspond to respective `EElementType`, INCA will throw a `NullReferenceExeption`.

EElementType	ElementInfo Member
`EElementType.Scalar`	`IElementInfo.Scalar`
`EElementType.Ascii`	`IElementInfo.Ascii`
`EElementType.Array`	`IElementInfo.Array`
`EElementType.LookupTable`	`IElementInfo.LookupTable`
`EElementType.Axis`	`IElementInfo.Axis`
`EElementType.None`	No access possible

Tab. 5-2 Type specific members of ElementInfo

For additional information on this topic, see chapter 10 Lesson 8 - How to Access Maps, Curves and Axes.

To create a ValueAccess for a variable

1. Find the following code lines

```
...
        _valueScalarAccessExample =
addedElements[0].ElementInfo.Scalar.GetValueAccess<double>(EA
ccessType.Phys, EPageId.Current);
...
```

You have now created a `ScalarValueAccess` to access the variable's value:

- as a physical value,
- in double format and
- on the current page.

Survey on the different `GetValueAccess` parameters:

GetValueAccess() parameters	ValueAccess type	Read only for calibrations?
`EAccessType.Phys`	Physical values	N/A
`EAccessType.Impl`	Implementation values	N/A
`EPageId.Current`	currently selected page	Yes, if `current` = RP No, if `current` = WP
`EPageId.Reference`	Reference page	Yes

Tab. 5-3 GetValueAccess survey

To attach a handler to the ValueChangedEvent

1. Find the highlighted code lines.
2. Remove comment indication /* and */

```
...
        _valueScalarAccessExample.ValueChangedEvent += new
EventHandler(ExampleValueChangedEvent);
...
```

You have now activated a default code passage that attaches an Event Handler for the `ValueChangedEvent` and links it to the `ExampleValueChangedEvent` method.

Each time the value of the `Variable` changes, the `ExampleValueChangedEvent` method will be called. By this event mechanism, there is no need to poll INCA for the current values.

Your instrument will automatically get informed about changed values.

6.2.8 GetValue Method

The method `GetValue()` of the `ISignalValueAccessScalar` object is used to receive the current value of the element. A default method is provided by the **wizard.**

Make sure to have assigned a default value. The parameter of the `GetValue` method will be returned if there is no valid value available.

To get the variable value

As an example, we define the type `double.`

1. Find the highlighted code lines

```
...
private void exampleValueChangedEvent(object sender, EventArgs e)
{
...
```

2. Remove the comment indication /* and */ below the found code

```
...
        double actualValue = _valueScalarAccessExample.GetValue(0.0);
```

   You have now activated the default `GetValue` method.

3. Insert the highlighted code lines below. To display the actual value write it into a textbox

```
        textBox1.Text = actualValue.ToString();
```

```
}
```

MyInstrument can now read the value of the respective Scalar variable.

To get the value the TryGetValue() method can be used. In this method, the value is available if the return value of the method is true.

To display the current value, use one of the buttons and textboxes already created in chapter 4.2.1. This value is written into textBox2 not used so far.

4. Add the following method to the InstrumentControl class

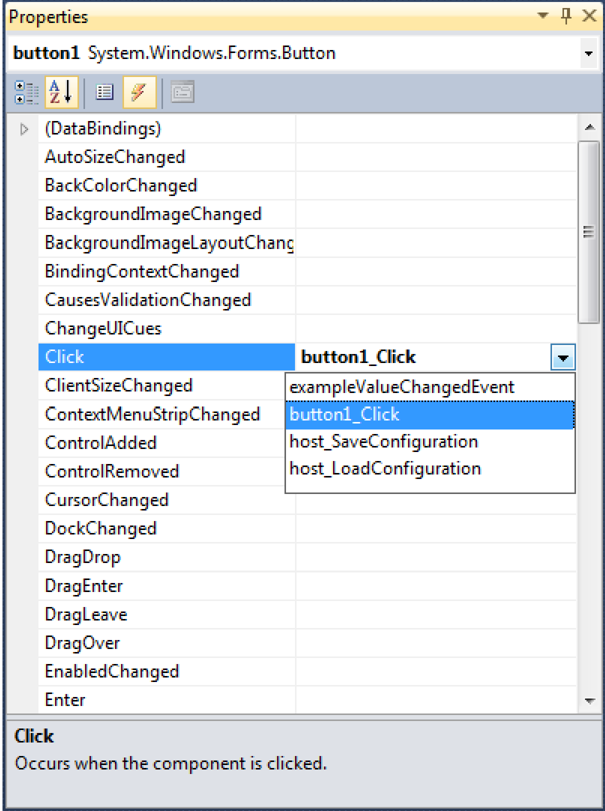
```
private void button1_Click(object sender, EventArgs e)
{
```

Writing current value at time of the click event into textBox2

```
        textBox2.Text = _valueScalarAccess.GetValue(0.0).ToString();
}
```

To attribute button1 to the button1_click event:

1. Open the InstrumentControl.Designer.cs.
2. Click on button1.
3. Go to **Event** tab.
4. Go to “Click” and select the button1_click event out of the drop-down list.



MyInstrument now assigns the constantly read values to textBox1 and on each click on button1 the value, frozen at the time of the ClickEvent, to textBox2.

5. Make sure INCA is closed.
6. In Visual Studio, select **Build** > **Build Solution.**

7. Open INCA.
8. In the Variable Selection Dialog, call `MyInstrument`.
9. Click **Ok**.
   Your modifications are now active in `MyInstrument`.
10. In INCA, check if `MyInstrument` is working for `Scalar Calibration` values as well as for `Scalar Measurement` values.
    i. Open a corresponding INCA editor (measure window/calibration window).
    ii. Check if the `MyInstrument` is updating correctly when working on the same variables

NOTE

In our example, always the last variable assigned to your instrument will be active, since we initialized the local variable `_valueScalarAccess` with `addedElements[0`.

6.2.9 SetValue Method

Calibration variables are modified by the `SetValue` method. This method is an essential part of the `ValueAccess` interface.

As an example, a `ClickEvent` of `button2` reads the string from the `textBox2` and writes it into the `ValueAccess` object. The `SetValue()` method returns an object with type `IRetVal`. This object contains information on the success of the operation.

To set the variable value

1. Add the following method to the `InstrumentControl` class.

```
private void button2_Click(object sender, EventArgs e)
{
        try
        {
            // write the given value from the textbox into
the value access object
            IRetVal val
=_valueScalarAccess.SetValue<double>(BoundaryModes.RespectAllBounds,
Convert.ToDouble(textBox2.Text));
        }
        catch (Exception ex)
        {
            Logging.Log(ELoggingSeverity.Error, ex.ToString());
        }
}
```

2. Attach the method as button2_Click event handler of button2.

NOTE

The conversion from string to <double> can throw an Exception.

As a simple error handling we just use a try...catch block in the example.

3. Make sure INCA is closed.
4. In Visual Studio, select **Build** > **Build Solution.**
5. Open INCA.
6. In the Variable Selection Dialog, call MyInstrument.
7. Click **Ok.**

   Your modifications are now active in MyInstrument.

Your MyInstrument is now able to

– read values,
– get values under certain conditions, here by button1_click
– write values into the variable, here by button2_click, thus: to calibrate

NOTE

Measurement Values cannot be modified.

NOTE

The variable you assign to MyInstrument must be of class Calibration.

If you use the SetValue method with a Measurement Variable you will get a return value which contains that the Variable is not editable.

NOTE

Be careful with the BoundaryModes option IgnoreAllBounds.

The value will be written in your project's file independently of the boundaries specified.

7 Lesson 4 - Context Menus

INCA supports to display context menus in instruments. A default set of context menu items is available. To enable the context menu items, the instrument must provide some additional information by an implementation of `IWidgetHostEventSink`.

7.1 Objectives

In this chapter, you learn to enable the **INCA** default **context menu** and to implement and add your own **context menu items**.

7.1.1 Enable the Default Context Menu

The generated `Initialize` method calls the `InitializeDefaultContextMenus` by default. By that the context menu will be opened at runtime when doing a right mouse click on the `InstrumentControl`.

To activate `WidgetHostEventSink`

1. Open `InstrumentControl.cs`.
2. Find the following code lines.

```
public void Intitialize(IWidgetHost host)
{
     …
     _WidgetHost.InitializeDefaultContextMenus(this);
     …
}
```

3. Remove comment indication from the highlighted code passages.

```
public partial class InstrumentControl : UserControl,
IWidget
    {
        //private IWidgetHost _WidgetHost; //is supposed
to connect to and to react on all of the events included
        //private WidgetHostEventSink _EventSink;
```

4. Remove comment indication.

```
/// <summary>
        /// Initializes this instrument component.
        /// </summary>
        public InstrumentControl()
        {
            InitializeComponent();
            //_EventSink = new WidgetHostEventSink(this);
            //an instance of the new WidgetHostEventSink
object with this InstrumentControl as parameter
        }
```

5. Find the following source code lines below.
6. Delete the highlighted code lines.

7. Remove comment indication

```
...
Public IWidgetHost WidgetHost

{
    get { return null; }
    // get { return(_WidgetHost;}
}
```

The `WidgetHostEventSink` must provide the additional information to enable the default context menu items.

To provide additional information from `WidgetHostEventSink`

1. Open `WidgetHostEventSink.cs.`
2. Find the following code lines.
3. Delete the highlighted code lines.

```
public bool CanComputeParameter(IParameterDescription
parameterDescription)
{
    return false;
    /*
     return parameterDescription.Key == ParameterIDs.ElementSelection
||
             parameterDescription.Key == ParameterIDs.ValueSelection;
   */
}
```

   This method returns which additional information can be provided to compute the enabled states for the context menu items.

4. Find the following code lines.
5. Delete highlighted code lines and the comment indication.

```
public bool ComputeParameter(IParameter parameter)
{
      return false;
...
     /*switch (parameter.Description.Key)
     {
          case ParameterIDs.ElementSelection:
               parameter.Value =
_instrumentControl._WidgetHost.Configuration.WidgetDatas[0].ElementRef
erences;
               return true;
          case ParameterIDs.ValueSelection:
               parameter.Value = new[] {
new ValueSelection(_instrumentControl.ValueScalarAccess,
EValueFormat.String) };
               return true;
     }
     return false; */
```

```
}
```

The method `ComputeParameter` provides different information specified with the `ParameterID` and `Description`. For example, the "`About variable…`" context menu item in INCA will show the "About Variable" dialog for the `ElementReference` provided in the `ElementSelection` parameter.

6. Find the following code lines.
7. Verify that highlighted code is set to `true`.

```
            public bool CanShowProperties()
{
    return true;
}
```

The method `CanShowProperties` determines what happens when the "Properties…" context menu item is clicked.
The default value `false` will display the VSD Display Configuration for the respective instrument.
If the method returns the value `true`, the `ShowProperties` method is called.

8. Find the following code lines.

```
public void ShowProperties()
{
//TODO implement
}
```

9. Replace `TODO` by the highlighted code lines.

```
…
    MessageBox.Show("Enable UI updates?", "Properties",
MessageBoxButtons.YesNoCancel);
…
```

The method `ShowProperties` allows you to show the properties for the currently selected instrument. In this case, a message box appears.

10. Find the following code lines.

```
public System.Drawing.Size GetOptimalWindowSize()
{
    //This is an example for an optimal window size:
    System.Drawing.Size size = new
System.Drawing.Size(285, 150);
    return size;
}
```

The method `GetOptimalWindowSize` returns the optimal window size required by the instrument.

11. Find the following code lines.

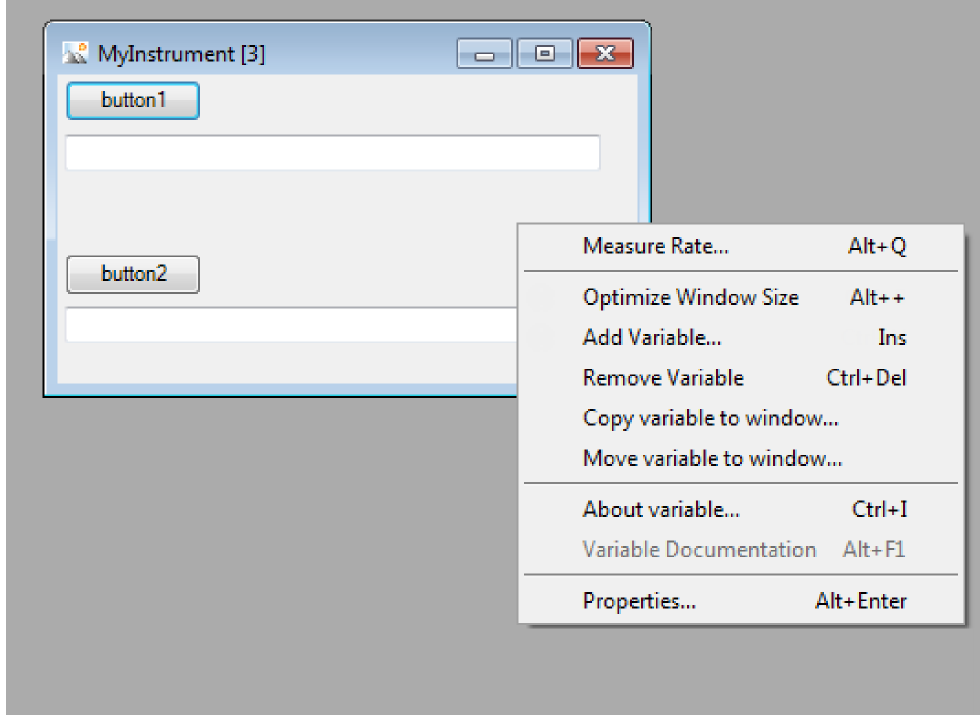
```
public bool HasOptimalWindowSize()
{
    return true;
}
```

The method `HasOptimalWindowSize` is called to enable or disable the `OptimizeWindowSize` context menu.

12. Make sure INCA is closed.
13. In Visual Studio, select **Build** > **Build Solution.**
14. Open INCA.
15. In the Variable Selection Dialog, call `MyInstrument`.
16. Click **Ok.**

    Your modifications are now active in `MyInstrument`.

The default context menu of INCA should now look like this:



The code in `ComputeParameter` above is returning the currently selected variables. By this, INCA knows which variables are selected and the context menu items like “Move variable to window...” will apply to the selected variables. If the selection in your instrument changes during runtime, you will need to call `_WidgetHost.EventSink.SelectionChanged()` in your `InstrumentControl` class to trigger INCA again to call `ComputeParamter`.

7.1.2 Creating Your Own Context Menu Items

In this chapter, you learn how to implement and enable your own context menu items. You create a new class, which represents the new context menu item.

You connect the new context menu item to the default Event Handler.

To create the BeepAction class

A new context menu item is declared as a new `class`. The class is derived from the `IActionDescription` interface.

This interface describes the actions that can be triggered by the context menu or a shortcut.

1. To create a new class, right-click on the instrument in Visual Studio.
2. Select **Explorer › Add › Class**.

   A dialog window opens.
3. Rename the class `BeepAction`.
4. Verify your selection.
5. Click **Add.**

   Visual Studio creates a default empty class including the basic header declarations.
6. Open `BeepAction.cs`.
7. Verify the declaration in the head.
8. Complete to below example if necessary.
9. Implement the highlighted code lines.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Etas.OpenEE;

public class BeepAction : IActionDescription
{
    // global variables
    private readonly int _frequency;
    private readonly int _length;
    private readonly Keys _shortcut;

    // constructor
    public BeepAction(int frequency, int length, Keys shortcut)
    {
        this._frequency = frequency;
        this._length = length;
        this._shortcut = shortcut;
    }
```

```
    public string Text // the description of the context
menu item
    {
        get { return "Beep"; }
    }
    public Keys Shortcut // the shortcut will be declared in
the "InstrumentControl.cs"
    {
        get
        {
            return _shortcut;
        }
    }
    public string ShortcutDisplayString
    {
        get { return String.Empty; }
    }
    public Icon Icon // here, you can specify the icon
    {
        get { return SystemIcons.Asterisk; }
    }
    public bool Checked
    {
        get { return false; }
    }
 // return true to enable the context menu item
    public bool Enabled
    {
        get { return true; }
    }
// return true to activate the context menu item
    public bool Active
    {
        get { return true; }
    }
    internal void DoIt()
    {
        Console.Beep(_frequency, _length);
    }
}
```

Your class `BeepAction` is implemented.

With a few further steps you can trigger different “beeps” via `MyInstrument`.

To modify the InstrumentControl.cs

1. Open `InstrumentControl.cs.`

You must declare the new context menu items in the `InstrumentControl.cs`.

Add `BeepAction` definition as given in the example below

2. Find the following code lines.
3. Implement the highlighted passages.

```
public InstrumentControl()
{
…
  private List<IActionDescription> _actions = new
List<IActionDescription>{new BeepAction(1000, 500, Keys.Control |
Keys.B), new BeepAction(500, 500, Keys.Control | Keys.N)};
…
}
```

The method `GetActionDescriptions` defines the actions of `MyInstrument`. In this case we return the list of the Context Menu actions.

4. Find the following code lines.
5. Replace `return null;` by the highlighted code lines.

```
public IList<IActionDescription> GetActionDescriptions()
{
     return _actions;
}
```

6. Find the following code lines.

```
public void ExecuteAction(IActionDescription
actionDescription)
        {
…
```

7. Integrate the highlighted code passages.
8. Comment out `System.Diagnostics.`

```
…
              if (actionDescription is BeepAction)
              {
                   ((BeepAction)actionDescription).DoIt();
              }
              // (default command to remind missing
execution if method is called)
              // System.Diagnostics.Debug.Assert(false);
        }
```

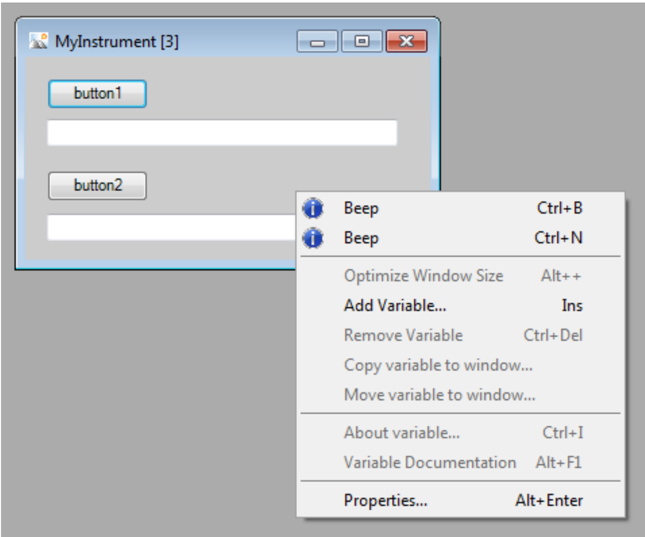
9. Make sure INCA is closed.
10. In Visual Studio, select **Build** > **Build Solution**.
11. Open INCA.
12. In the Variable Selection Dialog, call `MyInstrument`.
13. Click **Ok.**

    Your modifications are now active in `MyInstrument`.

NOTE

`Beep` is system sensitive. If you have silenced your system, there may be nothing to hear.

Now you can use your new content menu items:



8 Lesson 5 - Making **MyInstrument** Multilingual

8.1 Objectives

In this chapter, you learn how to create and modify language files.

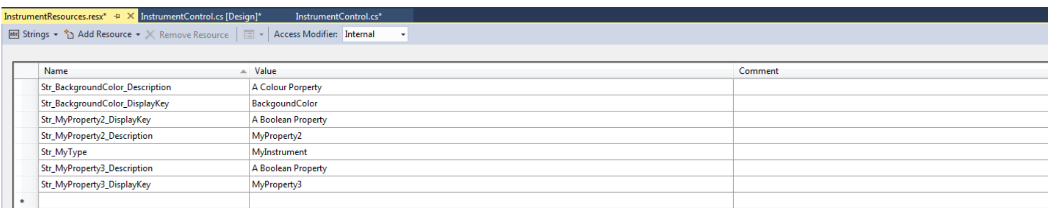
NOTE

You must create one `InstrumentResource.resx` file for each language.

8.2 Completing **InstrumentResources.resx**

The `InstrumentResources.resx` (default name) is an XML based file specifying objects and strings by entries in XML tags. To modify or copy the file a text editor (such as Notepad or Word) is needed. In this tutorial the editor provided by Visual Studio is used.

Visual Studio edits the entries in a table as shown in the figure below:



8.2.1 Creating a New **InstrumentResources.resx**

To create a resx for a new language

1. Right-click on the file `InstrumentResources.resx.`
2. Select **Copy.**
3. Right-click on `MyInstrument.`
4. Select **Paste.**
5. Rename copied file to `InstrumentResources.de.resx.`
6. Press ‹ENTER›.

There are several options how to create a new `InstrumentResources.resx`.

An alternative is to use a translation tool, which automatically creates `InstrumentResources.resx`.

NOTE

The naming of the resource file follows the following convention: `InstrumentResources.languagecode.resx`.

For example: `InstrumentResources.en-us.resx`.

Further information about the language codes can be found in the msdn library at: http://msdn.microsoft.com/en-us/library/kx54z3k7%28vs.71%29.aspx

8.2.2 Making MyInstrument Multilingual

To add translations

1. Open the `InstrumentRessources.de.resx` file.
2. Translate the entries.
3. Save the file.

Visual Studio creates for every language a new folder within `MyInstrument's` main folder.

4. Make sure INCA is closed.
5. In Visual Studio, select **Build** > **Build Solution**.
6. Open INCA.
7. In the Variable Selection Dialog, call `MyInstrument`.
8. Click **Ok.**
   Your modifications are now active in `MyInstrument`.

NOTE

The language settings of INCA are loaded during initial installation.

Therefore, modifications to the language have no visible effect, unless you reinstall INCA and modify your language settings.

Multilingual UI interface is now implemented.

9 Lesson 6 - Implementing Drag & Drop

9.1 Objectives

In this chapter, you learn to implement the Drag & Drop function to or out of `MyInstrument`.

9.2 The Drag & Drop Function

In the INCA Experiment Environment variables can be moved from one window to another by drag & drop. INCA offers dropping variables by default, so you only must implement the drag function.

NOTE

The variable can be dragged into or out of `MyInstrument` under two conditions:

- *`MyInstrument`* / the target instrument supports the dragged *`VariableType`*

the *`MaxCount`* of signals of *`MyInstrument`* / the target instrument is not exceeded.

9.2.1 Implementing Drag & Drop

Start a Drag & Drop operation when the user clicks and moves the mouse. For this purpose, event handlers for `MouseDown` and `MouseMove` must be registered. Moreover, we must remember the location where the user has clicked.

To register the mouse event handlers

1. Open `InstrumentControl.cs`.
2. Implement the highlighted code lines.

```
public InstrumentControl()
{…
   this.MouseDown += OnMouseDownControl;
   this.MouseMove += OnMouseMoveControl;
}
private Rectangle m_DragBoxFromMouseDown = Rectangle.Empty;
…

private void OnMouseDownControl(object sender, MouseEventArgs e)
{
     if (e.Button == MouseButtons.Left)
     {
          m_DragBoxFromMouseDown = GetDragBoxFromPoint(e.Location);
     }
     else
     {
          m_DragBoxFromMouseDown = Rectangle.Empty;
     }
}
```

```
private static Rectangle GetDragBoxFromPoint(Point p)
{
      Size dragSize = SystemInformation.DragSize;
      return new Rectangle(new Point(p.X - (dragSize.Width / 2), p.Y
(dragSize.Height / 2))) dragSize;
}


private void OnMouseMoveControl(object sender, MouseEventArgs e)
{
      if ((e.Button & MouseButtons.Left) != MouseButtons.Left)
      {
      }
            else if (m_DragBoxFromMouseDown != Rectangle.Empty &&
                      !m_DragBoxFromMouseDown.Contains(e.X, e.Y))
            {
        // reset the dragBox rectangle
                  m_DragBoxFromMouseDown = Rectangle.Empty;
                  // start drag and drop
                  _WidgetHost.EventSink.DoDragDropSelectedElements();
             }
        }
}
```

The method `OnMouseDownControl` contains the reaction on the `MouseButtons.Left` event. It is called every time you click into the instrument window.

The method `GetDragBoxFromPoint` is used to define a virtual `Rectangle` around the `Point` you clicked to decide when to start the `Drag` operation when moving the mouse with pressed mouse button.

The method `OnMouseMoveControl` manages the reaction to the `MouseMove` event.

The `DoDragDropSelectedElements` method called at the bottom of this method starts the Drag & Drop operation of the selected element reference.

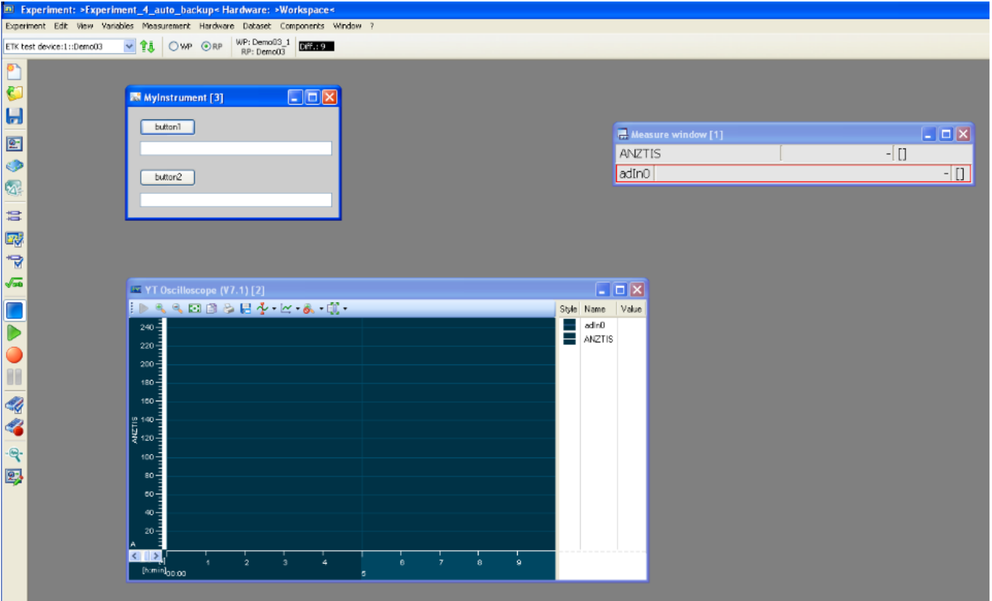
For this it is important that the instrument implemented the `WidgetHostEventSink` and computes the `ElementSelection` parameter (see chapter 6 Lesson 4 - Context Menus).

`MyInstrument` will now be able to trigger the dragging of the selected variables to another instrument.

To activate the modifications

1. Make sure INCA is closed.
2. In Visual Studio, select **Build** › **Build Solution**.
3. Open INCA.
4. In the Variable Selection Dialog, create `MyInstrument`.

5. Click **Ok.**
   Your modifications are now active in `MyInstrument`.
   Now you can use the Drag & Drop function of `MyInstrument` in the INCA-Experiment.



10 Lesson 7 - Deploy Instruments on Other Systems

10.1 Objectives

In this chapter, you learn how to deploy instruments on other systems.

10.2 Porting MyInstrument to Another System

After finishing the Wizard, Visual Studio copies `MyInstrument` folders to the following location: `<ETASData>\INCA7.5\Instruments`

To deploy instruments on another system

1. Make sure INCA is closed.
2. Copy the complete `MyInstrument` folder to one of the following plugin folders:
   - `<ETASData>\INCA7.5\Instruments`
   - `<CommonApplicationData>\ETAS\INCA\7.5\Instruments`
   - `<ApplicationData>\ETAS\INCA\7.5\Instruments`
3. Restart your device.
4. Start INCA.

`MyInstrument` is now deployed on the other system.

The recommended plugin folder is `<ETASData>\INCA7.5\Instruments`. If the instrument folder is not available on the system, you like to install the plugin, create the system on that you like to install the plugin. You can install instrument plugin DLLs to separate folder, but you can also install different plugins into the same folder. This might help if you would like to create different instruments using shared DLLs.

Example deployment structure:

```
<ETASData>\INCA7.5\Instrument\InsPlugin1\InsPlugin1.dll

               ...InsPlugin2\InsPlugin2.dll

               ...sPlugin2\InsPlugin3.dll
```

or

```
<ETASData>\INCA7.5\Instrument\InsPlugins\InsPlugin1.dll

               ...InsPlugin2.dll

               ...InsPlugin3.dll

               ...SharedDLL.dll
```

11 Lesson 8 - How to Access Maps, Curves and Axes

11.1 Objectives

In this chapter, you establish a new instrument containing new features.

You learn to use the new `VariableTypes.VariableType Map` is used as an example to introduce the new options.

Moreover, this chapter contains

- an introduction to fill and modify `DataGrids`,
- a description to display `system error messages`
- an introduction to `InterpolationModes`
- a description to display `WorkingPoints`
- an introduction to the different `AxisTypes` and their modification handling
- and an introduction to `interpolation` and `BoundaryModes`

11.2 Creating MyMapInstrument

To create an intrument for a map

1. Start a new project in Visual Studio 2022.
2. Create a new instrument via the Wizard.
3. Name the instrument `MyMapInstrument`.
4. Name the type of the instrument `MyMapType`.

NOTE

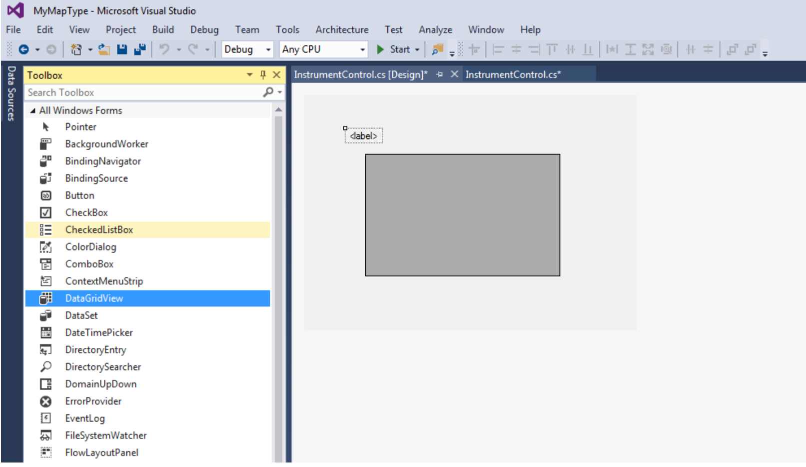
For each new instrument you create, you need to define a *unique* type name. If multiple instruments are using the same type name, they will not be loaded correctly by INCA

5. Assign VariableType › Map.

11.3 Using the VariableType Map

To use the instrument for a map

1. Create an UI with `AllWindowsForms` option `DataGridView`.
2. Add a `label`, call the variable `mapLabel`.
3. Set the text of the label to `<label>`.



You have now created a user interface for INCA which allows access to 2D maps.

4. Make sure INCA is closed.
5. In Visual Studio, select **Build** > **Build Solution**.
6. Open **INCA.**
7. In the Variable Selection Dialog, call `MyMapInstrument`.
8. Click **Ok.**
   Your modifications are now active in `MyMapInstrument`.

11.3.1 Filling the <label> in InitializeValueAccess Method

To get the name of the map

1. Open `InstrumentControl.cs.`
2. Implement the highlighted code lines in `InitializeValueAccess` (see Fig 10-4, mid-1st page):

```
…
mapLabel.Text = mapInfo.GetNaming().GetValue(""); //Name of Variable
to Label
…
```

   `MyMapInstrument` now displays the name of the currently assigned variable in `<lable>`.

If you call `MyMapInstrument` several times and assign different variables, the respective variable will now be displayed in the user interface of your `MyMapInstrument`.

11.3.2 Accessing DataGrids and Error Messages

In this chapter, you learn to access `DataGrid` indices and to display system error messages in the user interface.

To define the grid methods for the axes and the values

1. Implement the highlighted code lines:

```
public partial class InstrumentControl : UserControl,
IWidget
    {
        private IWidgetHost _WidgetHost;
        …
        private ISignalValueAccessAxis<double> mXAxisValueAccess;
        private ISignalValueAccessAxis<double> mYAxisValueAccess;
        private ISignalValueAccessArray<double> mValuesValueAccess;

        private int mapSizeX;
        private int mapSizeY;
        …
```

To define the values of the grid within the InitializeValueAccess

1. Verify the non-highlighted code passages.
2. Implement the highlighted code lines:

```
this.InitializeValueAccesses(e.Added, e.Removed);
        {
… // standard method needed to initialize and modify any
instrument
        private void
InitializeValueAccesses(IList<IElementReference>
addedElements,
            IList<IElementReference> removedElements)
        {
 …

            if (removedElements != null)
            {
                foreach (IElementReference er in
removedElements)
                {
                    //TODO implement
                }
            }
            // Then process any added elements
            if (addedElements != null)
            {
                foreach (IElementReference er in
addedElements)
                {
```

```
                         IElementInfo mapInfo =
addedElements[0].ElementInfo;
                         mapLabel.Text =
mapInfo.GetNaming().GetValue(""); //Name of Variable to
Label
                         mXAxisValueAccess =
mapInfo.LookupTable.Axes[0].GetValueAccess<double>(EAccessType.Phys,
EPageId.Current);

                         mYAxisValueAccess =
mapInfo.LookupTable.Axes[1].GetValueAccess<double>(EAccessType.Phys,
EPageId.Current);

                         mValuesValueAccess =
mapInfo.LookupTable.GetValueAccess<double>(EAccessType.Phys,
EPageId.Current);

                         mXAxisValueAccess.ValueChangedEvent += new
EventHandler(mXAxisValueAccess_ValueChangedEvent);

                         mYAxisValueAccess.ValueChangedEvent += new
EventHandler(mYAxisValueAccess_ValueChangedEvent);

                         mValuesValueAccess.ValueChangedEvent += new
EventHandler(mValuesValueAccess_ValueChangedEvent);

                         UpdateAll();
                    }
               }
          }
          // defining the space of the axes and the values
               //declaration of methods

          void mValuesValueAccess_ValueChangedEvent(object sender,
EventArgs e)
          {
               UpdateValues();
          }
          void mYAxisValueAccess_ValueChangedEvent(object sender,
EventArgs e)
          {
               UpdateYAxisValues();
          }
          void mXAxisValueAccess_ValueChangedEvent(object sender,
EventArgs e)
          {
               UpdateXAxisValues();
          }
          private void UpdateAll()
```

```
          {
               UpdateXAxisValues();
               UpdateYAxisValues();
               UpdateValues();
          }
          private void UpdateXAxisValues() // definition of x
axis indices
          {
               IValueArray<double> myValues =
mXAxisValueAccess.GetValue(null);
               mapSizeX = myValues.Dimensions[0];
               mapValues.ColumnCount = mapSizeX;
               for (int col = 0; col < this.mapValues.ColumnCount;
col++)
               {
                    mapValues.Columns[col].HeaderCell.Value =
myValues.Values[col].ToString();
               }
          }
          private void UpdateYAxisValues() // definition of y
axis indices
          {
               IValueArray<double> myValues =
mYAxisValueAccess.GetValue(null);
               mapSizeY = myValues.Dimensions[0];
               mapValues.RowCount = mapSizeY;
               for (int row = 0; row < mapSizeY; row++)
               {
                    mapValues.Rows[row].HeaderCell.Value =
myValues.Values[row].ToString();
               }
          }
          private void UpdateValues() // definition of the grid
values
          {
               double[] values =
mValuesValueAccess.GetValue(null).Values;
               for (int i = 0; i < mapSizeX; i++)
                    for (int j = 0; j < mapSizeY; j++)
                         mapValues.Rows[j].Cells[i].Value = values[j *
mapSizeX + i];
          }
…
          private void exampleValueChangedEvent(object
sender, EventArgs e) // standard method
          public void DeInitialize() //standard method
          …
```

To write values back into the table

WARNING

Risk of unexpected vehicle behavior

Calibration activities influence the behavior of the ECU and the systems that are connected to the ECU.
This can lead to unexpected vehicle behavior, such as engine shutdown as well as breaking, accelerating, or swerving of the vehicle.
Only perform calibration activities if you are trained in using the product and can assess the possible reactions of the connected systems.

1. Implement a `textbox` labeled "Value".
2. Define three `buttons`.

   The buttons will call a click method triggering the writing back of the inserted values to defined places of the grid.

3. Name the buttons as follows:
   i. Left button
      xAxis at index3: calibrates the respective index to the value in the textbox
   ii. Middle button
      yAxis at index2: calibrates the respective index to the value in the textbox
   iii. Right button
      Grid value at index 3;4: calibrates the respective index to the value in the textbox
4. Insert a field `<result>`.

   This field will display system messages on the latest performed operation.

5. Implement the highlighted code lines below the standard code:

```
private void label3_Click(object sender, EventArgs e)
          {
          }

          private void buttonSetXAxis_Click(object sender, EventArgs
e)
          {
               double[] value = new double[] {
Double.Parse(textBoxValue.Text) };
               // this example sets 4th x axis value to
textBoxValue
               IRetVal retval =
mXAxisValueAccess.SetValues<double>(BoundaryModes.IgnoreAllBounds,
value, 3, InterpolationMode.None, null);
               labelResult.Text = retval.Success + " ; " +
retval.ErrorMessage + " ; " + retval.ReturnValueType;
          }
```

```
        private void buttonSetYAxis_Click(object sender, EventArgs
e)
        {
            double[] value = new double[] {
Double.Parse(textBoxValue.Text) };
            // this example sets 3rd y axis value to
textBoxValue
            IRetVal retval =
mYAxisValueAccess.SetValues<double>(BoundaryModes.IgnoreAllBounds,
value, 2, InterpolationMode.None, null);
            labelResult.Text = retval.Success + " ; " +
retval.ErrorMessage + " ; " + retval.ReturnValueType;
        }
        private void buttonSetValue_Click(object sender, EventArgs
e)
        {
            double[] value = new double[] {
Double.Parse(textBoxValue.Text) };
            int[] index = new int[] { 3, 4 };
            // length defines number of repetitions.
            // In case of other values than noted here,
make sure to offer an according number of values e.g. by
fill method
            int[] length = new int[] { 1, 1 };
            IRetVal retval =
mValuesValueAccess.SetValues<double>(BoundaryModes.IgnoreAllBounds,
value, index, length);
            labelResult.Text = retval.Success + " ; " +
GetErrorMessage(retval) + " ; " + retval.ReturnValueType;
        }
```

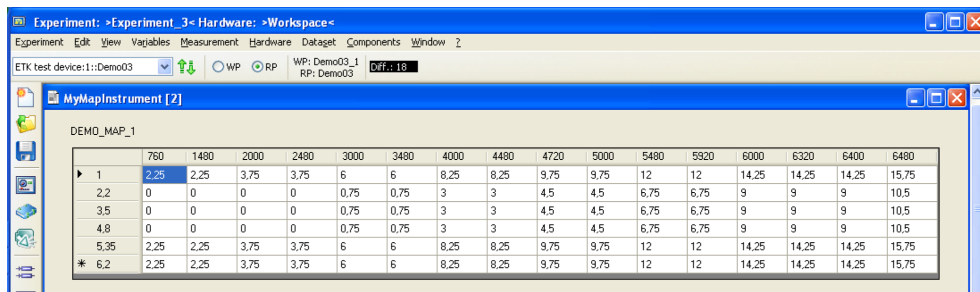
6. Make sure INCA is closed.
7. In Visual Studio, select **Build** > **Build Solution**.
8. Open INCA.
9. In the Variable Selection Dialog, assign a map **to** `MyMapInstrument`.
10. Click **Ok.**

    Your modifications are now active in `MyMapInstrument`.

NOTE

These actions can only be performed on the working page in INCA. If your current page is reference page the calibration action will fail and `false`

`MyMapInstrument` now looks like this in INCA:



To display error messages

1. Implement the highlighted code lines below.

```
// helper method to retrieve information in case of
failed calibration action
          private string GetErrorMessage(IRetVal returnValue)
          {
               var errorMessage = String.Empty;
               if (returnValue.ErrorMessage != String.Empty)
               {
                    errorMessage = returnValue.ErrorMessage + "\r\n";
               }
               switch (returnValue.ReturnValueType.ToString())
               {
                    case "None":
                         break;
                    case "BoundaryInfoScalar":
                         var boundaryInfoScalar =
((IRetValBoundaryInfoScalar)returnValue).BoundaryInfo;
                         errorMessage += "BoundaryInfo: " +
boundaryInfoScalar;
                         break;
                    case "BoundaryInfoArray":
                         var boundaryInfoArray =
((IRetValBoundaryInfoArray)returnValue).BoundaryInfo;
                         errorMessage += "BoundaryInfo: " +
boundaryInfoArray;
                         break;
                    case "SetScalarValue":
                         var setScalarValue =
((IRetValSetScalarValue)returnValue).SetValueActionsTaken;
                         errorMessage += "SetValueActionsTaken: " +
setScalarValue;
                         break;
                    case "SetArrayValue":
                         var setArrayValue =
((IRetValSetArrayValue)returnValue).SetValueActionsTaken;
```

```
                        errorMessage += "SetValueActionsTaken: " +
setArrayValue;
                        break;
                    case "SetScalarValueError":
                        var setScalarValueError =
((IRetValSetScalarValueError)returnValue).BorderViolations;
                        // Border Violations offer detailed
error information.
                        // Check in case you wish to return
more information than here indicated
                        errorMessage += "BorderViolations: " +
setScalarValueError;
                        break;
                    case "SetArrayValueError":
                        var setArrayValueError =
((IRetValSetArrayValueError)returnValue).BorderViolations;
                        errorMessage += "BorderViolations: " +
setArrayValueError;
                        break;
                    case "MaxRangeExceededError":
                        // Dimension Limits offer detailed
error information.
                        // Check in case you wish to return
more information than here indicated
                        var maxrangeExceededError =
((IRetValRangeError)returnValue).DimensionLimits;
                        errorMessage += "DimensionLimits: " +
maxrangeExceededError;
                        break;
                    case "MinRangeExceededError":
                        var minrangeExceededError =
((IRetValRangeError)returnValue).DimensionLimits;
                        errorMessage += "DimensionLimits: " +
minrangeExceededError;
                        break;
                    case "CanNotChangeSize":
                        break;
                    case "NotEditableError":
                        var editableState =
((IRetValNotEditableError)returnValue).EditableState;
                        errorMessage += "EditableState: " +
editableState;
                        break;
                    case "InvalidArgumentError":
                        var invalidArgumentError =
((IRetValInvalidArgument)returnValue).ArgumentName;
                        errorMessage += "ArgumentName: " +
invalidArgumentError;
                        break;
                    case "InvalidOperationError":
```

```
                    break;
                default:
                    break;
            }
            return errorMessage;
        }
    }
```

NOTE

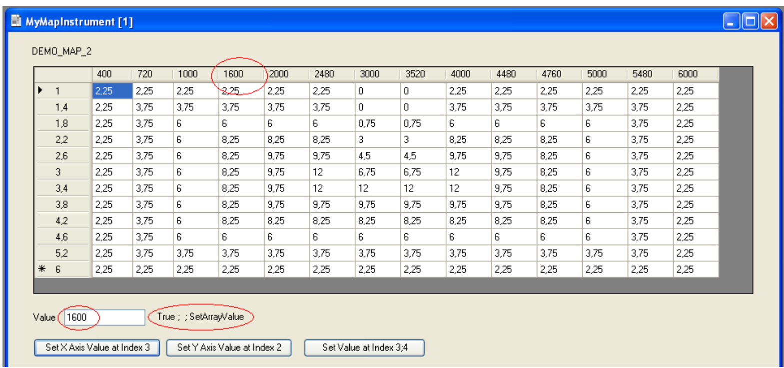
The code comments indicate that INCA offers more detailed error messages than used in the example. Esp. for calibration action further information is available.

2. Make sure INCA is closed.
3. In Visual Studio, select **Build** > **Build Solution.**
4. Open INCA.
5. In the Variable Selection Dialog, assign a map to `MyMapInstrument`.
6. Click **Ok.**

   Your modifications are now active in `MyMapInstrument`.

How the new read/write options work – with error messaging

1. In the **Value** field, enter `1600`.



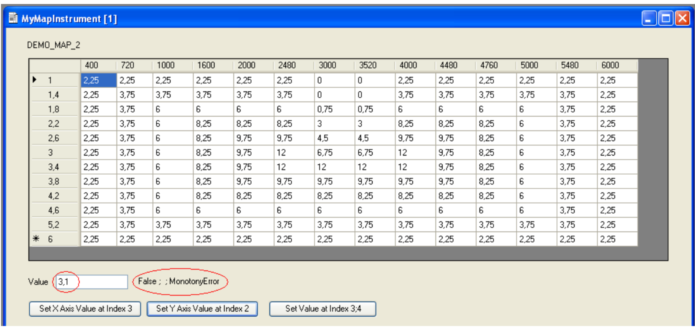
The figure above represents the result of the action:

– the values are processed correctly

– the value `True` is returned

– a positive error or system message is displayed

2. In the **Value** field, enter `3.1`.



The figure above represents the result of the action:

- the inserted value did not fulfill the necessary criteria to be processed
- the result variable returns a system error message, indicating the cause of the fault

11.3.3 Interpolation Effects

Any array of data can be treated according to interpolation modes.

There are four options:

- Normal
- To larger value
- To smaller value
- None

NOTE

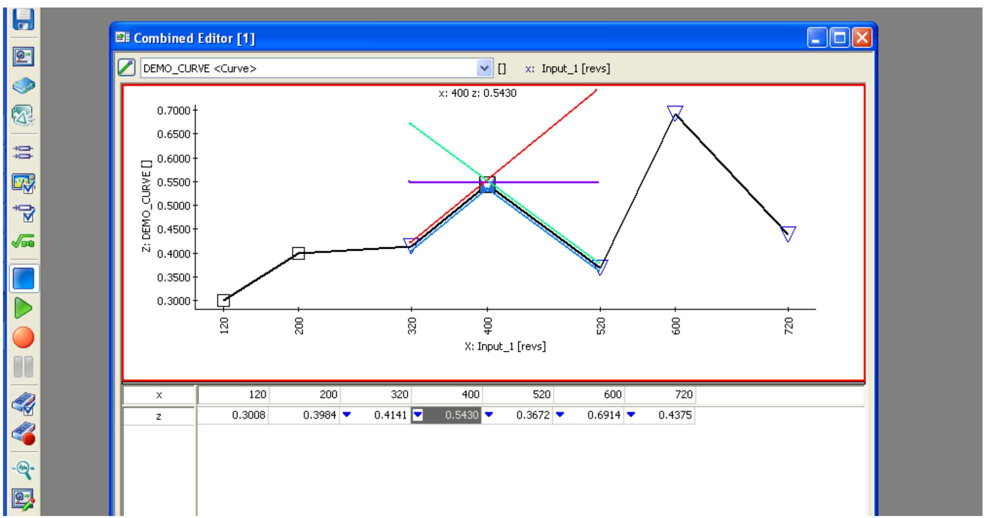
Interpolation does only take place between three points: the reference point, its predecessor, and its successor.

Normal: In the figure below indicated in blue (below the default curve):
All changes will be performed on the initially defined curve.
In case of a modification to an x axis index point, the interpolation mode will return the value on the y axis corresponding to this smaller or larger x axis index point on the normal (initial) curve.

To larger value: here indicated in green:
Reference point of the interpolation is the larger index point of the axis in question (here x-axis). Any modification of the curve will be processed in reference to this larger value point.
In this example, the modified preceding point would undergo interpolation in relation to larger point “520” on the x axis, therefore on the green line.

To smaller value: here indicated in red:
Reference point of the interpolation is the smaller index point of the axis in question (here x- axis). Any modification of the curve will be processed in reference to this smaller point.
In this example, the modified following point would undergo interpolation in relation to the preceding smaller point “320” on the x axis, therefore on the red line.

None: here indicated in purple:
Reference point of the interpolation is the indicated index point. The course of the line will now be set parallel to the axis in question (here x axis) on the level of the indicated index point of the other axis.



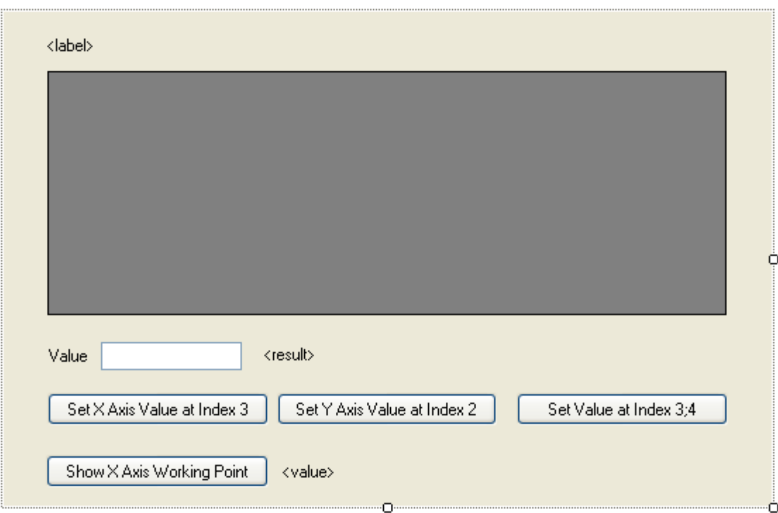
11.4 Displaying WorkingPoint Information

INCA supports the option to display the `WorkingPoint`.

In this chapter, you learn how to display the `WorkingPoints`.

<u>To expand the scope of MyMapInstrument to integrate WorkingPoints</u>

1. Open `InstrumentControl.cs`.
2. Insert a new button and a new label.
3. Rename the text of the button to „Show x Axis WorkingPoint“,
4. Rename the label test to ‹Value›.



5. Implement the highlighted code lines below.

```
//Implement WorkingPoint display
private void buttonShowWorkingPoint_Click(object sender, EventArgs e)
        {
            IDynamicElementReference processPoint;
            mXAxisInfo.TryGetProcessPointElement(out processPoint);
            processPointValueAccess =
processPoint.ElementInfo.Scalar.GetValueAccess<double>(EAccessType.Phy
s, EPageId.Current);
```

```
                processPointValueAccess.ValueChangedEvent += new
EventHandler(processPointValueAccess_ValueChangedEvent);
                UpdateWorkingPointValue();
           }
           void processPointValueAccess_ValueChangedEvent(object
sender, EventArgs e)
           {
                UpdateWorkingPointValue();
           }

           private void UpdateWorkingPointValue()
           {
                labelWorkingPoint.Text =
processPointValueAccess.GetValue(0.0).ToString();
           }
```

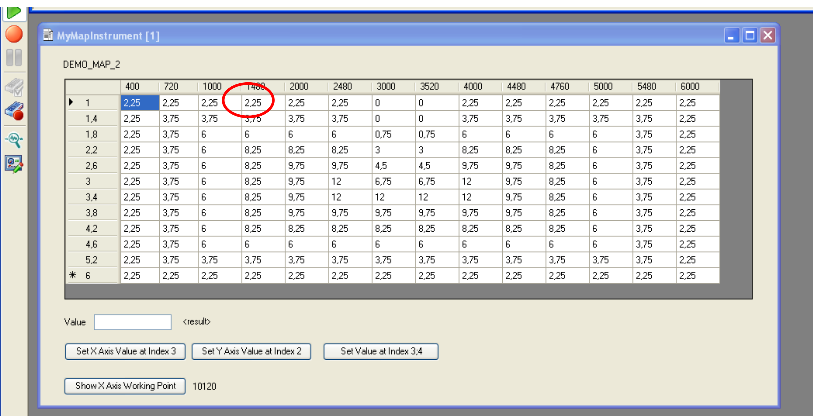
6. Make sure INCA is closed.
7. In Visual Studio, select **Build** > **Build.**
8. Open INCA.
9. In the Variable Selection Dialog, assign a map to `MyMapInstrument`.
10. Click **Ok.**

    Your modifications are now active in `MyMapInstrument`.

<u>To display a WorkingPoint</u>

`MyMapInstrument` should now look like this in INCA.

1. Activate a measurement for `MyMapInstrument`.



	400	720	1000	[illegible]	2000	2480	3000	3520	4000	4480	4760	5000	5480	6000
1	2,25	2,25	2,25	2,25	2,25	2,25	0	0	2,25	2,25	2,25	2,25	2,25	2,25
1,4	2,25	3,75	3,75	3,75	3,75	3,75	0	0	3,75	3,75	3,75	3,75	3,75	2,25
1,8	2,25	3,75	6	6	6	6	0,75	0,75	6	6	6	6	3,75	2,25
2,2	2,25	3,75	6	8,25	8,25	8,25	3	3	8,25	8,25	8,25	6	3,75	2,25
2,6	2,25	3,75	6	8,25	9,75	9,75	4,5	4,5	9,75	9,75	8,25	6	3,75	2,25
3	2,25	3,75	6	8,25	9,75	12	6,75	6,75	12	9,75	8,25	6	3,75	2,25
3,4	2,25	3,75	6	8,25	9,75	12	12	12	12	9,75	8,25	6	3,75	2,25
3,8	2,25	3,75	6	8,25	9,75	9,75	9,75	9,75	9,75	9,75	8,25	6	3,75	2,25
4,2	2,25	3,75	6	8,25	8,25	8,25	8,25	8,25	8,25	8,25	8,25	6	3,75	2,25
4,6	2,25	3,75	6	6	6	6	6	6	6	6	6	6	3,75	2,25
5,2	2,25	3,75	3,75	3,75	3,75	3,75	3,75	3,75	3,75	3,75	3,75	3,75	3,75	2,25
6	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25	2,25

The `WorkingPoint` values are now floating into the label `<value>` and are available for further processing, e.g., to trigger actions when reaching threshold values etc.

11.5 Axis Properties

INCA offers options for accessing axes values. The following table gives an overview:

Axis type	Local		Shared		
	Read Access	Write Access	Is shared?	VSD entry?	Write access
Standard	Yes	Yes	No	No	n/a
Fix	Yes	No	No	No	n/a
Rescale	Yes (phys only)	No	Yes	Yes	Yes
Curve	Yes (phys only)	No	Yes	Yes	Yes
Com	Yes	No	Yes	Yes	Yes

Tab. 10-1 Survey on axis options

The distinction between *local* and *shared* axis types is of vital importance in case of modification actions since the scope of the modification can be much broader than intended.

The above table indicates the cross references between axis type and scope of the axis:
`Standard` and `Fix` axes are limited to local usage,

- **Rescale, Curve** and **Combined** types can be used either locally or in a shared mode,
- **Com axes** are always shared.

These features are related to the way the axes are generated.

11.5.1 AxisType Based on DataType

The following table shows which axis type is based on which data type.

AxisType	DataType
`Standard axis`	`Array`
`Fix axis`	This type is defined by – a `starting point`, – an `offset` and – a `length`.
`Rescale axis`	`Data field`
`Curve axis`	`Curve`
`Com axis`	`Standard axis (array)` – but is always shared

Tab. 10-2 Axis type definition by data type

11.5.2 Calibrating the Shared Axis Types

The calibration of the type of `Standard axis` has already been described in the chapter How to fill the DataGrid / Step3: How to write values back into the table. The method of calibrating shared axis types described in this chapter is also valid for `Com axes`, since this type is based on the `Standard axis`.

Fix axes cannot be calibrated as the name suggests and are therefore not discussed.

To modify rescaleAxis values

1. Open `InstrumentControl.cs`.
2. Implement the highlighted code lines.

```
IElementInfo rescaleAxisElementInfo =
addedElements[0].ElementInfo;
// access to Rescale Axis Values
            // access to Rescale Axis Input Values
            IAxisInfo rescaleAxisInputValuesInfo =
rescaleAxisElementInfo.LookupTable.Axes[0];
            ISignalValueAccessAxis<double> rescaleAxisInputValueAccess
= rescaleAxisInputValuesInfo.GetValueAccess<double>(EAccessType.Impl,
EPageId.Current);
            // access to Rescale Axis Scale Values
            ILookupTableInfo rescaleAxisScaleValuesInfo =
rescaleAxisElementInfo.LookupTable;
            ISignalValueAccessArray<double>
rescaleAxisScaleValueAccess =
rescaleAxisScaleValuesInfo.GetValueAccess<double>(EAccessType.Impl,
EPageId.Current);
```

The code snippet above is in the `ValueAccess` part of the code.

NOTE

Local access to rescale axis values is limited to read only.

3. Verify in `InstrumentType.xml` that the respective axis type has been declared, if not `rescaleAxis` cannot be handled.
4. Make sure INCA is closed.
5. In Visual Studio, select **Build** > **Build Solution**.

6. Open INCA.
7. In the Variable Selection Dialog, select an according axis.
8. Assign the variable to your instrument.
9. Click **OK.**

To interpolate COM axes values

1. Open `InstrumentControl.cs`.
2. Implement the highlighted code lines:

```
            IElementInfo comAxisElementInfo =
addedElements[0].ElementInfo;
            // access to COM axis values
            IAxisElementInfo comAxisValuesInfo =
comAxisElementInfo.Axis;
            ISignalValueAccessAxis<double> comAxisValueAccess =
comAxisValuesInfo.GetValueAccess<double>(EAccessType.Phys,
EPageId.Current);
            // getting the objects for all lookup tables
that depend from this axis
            IGroupAxisDependency[] dependentLookupTables =
comAxisValuesInfo.DependentElements;

comAxisValueAccess.SetValues<double>(BoundaryModes.LowerUpperIgnoreBou
nds, values, 0, InterpolationMode.Normal, dependentLookupTables);
```

   `COM axis` type demands - along with the modification of its values - an indication of scope, i.e., a definition in which other tables this modification will be effective.

3. Define `IGroupAxisDependencies` [ ] as last argument of `SetValues` (our example passes all dependent lookup tables).
4. Define also `BoundaryModes`. It is possible to define several or all modes.
5. Define the interpolation mode.
6. Define the scope of the modification.
   According to your definitions, your `MyMapInstrument` will now process the defined options.
7. Make sure INCA is closed.
8. In the VSD, select **Build** > **Build Solution.**
9. Open INCA.
10. In the Variable Selection Dialog, select an according axis variable.
11. Assign the variable to your instrument.
12. Click **OK.**

To modify curve axes

1. Open `InstrumentControl.cs`.
2. Implement the highlighted code lines:

```
            IElementInfo curveAxisElementInfo =
addedElements[0].ElementInfo;
            // access to Axis Values
            IAxisInfo curveAxisValuesInfo =
curveAxisElementInfo.LookupTable.Axes[0];
            ISignalValueAccessAxis<double> curveAxisValueAccess =
curveAxisValuesInfo.GetValueAccess<double>(EAccessType.Impl,
EPageId.Current);
```

```
                // access to Values
                ILookupTableInfo curveValuesInfo =
curveAxisElementInfo.LookupTable;
                ISignalValueAccessArray<double> curveValuesAccess =
curveValuesInfo.GetValueAccess<double>(EAccessType.Impl,
EPageId.Current);
```

3. Make sure INCA is closed.
4. In Visual Studio, select **Build** > **Build Solution.**
5. Open INCA.
6. In the Variable Selection Dialog, select an according axis variable.
7. Assign the variable to your instrument.
8. Click **OK.**
   The modification is now active in your instrument.

NOTE

Access to `CurveAxis` values is done in the same way as access to any `Curve`.

12 Lesson 9 - Register Signals

It is possible to access a variable and its values directly in the instrument code without assigning it via the default Variable Selection Dialog mechanisms. For this, the variable's label must be known.

As the same label might occur within different control units, there is also the possibility to select the specific variable in the respective device.

NOTE

This assignment does not exclude other variables form being used together with the respective instrument.

Other variables can always be assigned at runtime.

12.1 Objectives

In this chapter, you learn to register a `label` in an instrument and access its values without assigning it via the Variable Selection Dialog.

12.2 Registering a Label in MyMapInstrument

The highlighted code can be placed wherever you think it fits within your instrument plugin. In the following example, we place the highlighted code to `InstrumentControl.cs` into the `Initialize` method.

To register a label in MyMapInsturment

1. Open `InstrumentControl.cs.`
2. Implement the highlighted lines:

```
        public void Initialize(IWidgetHost host)
        {
...
             IDynamicElementReference dynElRef =
_WidgetHost.ElementRegistry.RegisterElement("DEMO_MAP_2");
        }
```

   Note that `dynElRef` will be `null` when the label "`DEMO_MAP_2`" does not exist in the experiment. As an `IDynamicElementReference` has been registered, you are also able to create `ValueAccess` objects for it, like in the figure below.

```
…
ISignalValueAccessArray<string> valueAccess =
dynElRef.ElementInfo.LookupTable.GetValueAccess<string>(
EAccessType.Phys, EPageId.Current);
…
```

   For measurement `Variables` the raster will be the default raster.

   If you need `Variables` to be created in a specific measurement raster, you can implement the following option:

3. Implement the following code lines:

```
IDynamicElementReference dynElRef2 =
_WidgetHost.ElementRegistry.RegisterElement("MEASURE_T04", "ETK test
device", "Syncro");
```

```
_WidgetHost.ElementRegistry.EndRegistration();
```

4. Make sure INCA is closed.
5. In Visual Studio, select **Build** > **Build Solution**.
6. Open INCA.
7. Start your instrument.
   The modification is now active in your instrument.

12.3 Specifying the Device of a Variable

This option allows the choice of the device when registering a variable. This feature is useful, if you must deal with the same label occurring in multiple measurement devices.

To implement a DeviceChoiceHandler

1. Implement the following code lines

```
using System.Linq;
…

      public void Initialize(IWidgetHost host)
      {
…
      IDynamicElementReference dynElRef3 =
_WidgetHost.ElementRegistry.RegisterElement("DEMO_MAP_2",
SelectDevice);
      }

        private string SelectDevice(IEnumerable<string>
availableDeviceNames)
        {
            // This will get called if the variable label
occurs within multiple
            // devices. We just return the first given
device as choice.

            return availableDeviceNames.First();
        }
```

13 Migration: Compatibility of Instruments between INCA V7.2, V7.3, V7.4, and V7.5

Using INCA V7.3 and V7.4 instruments in INCA V7.5

You can use INCA V7.3 and V7.4 instruments in INCA V7.5 without any problems.

To use the instruments in INCA V7.5, copy the instruments to the instruments directory: `<ETASData>\INCA7.5\Instruments\`

Using INCA V7.2 instruments in INCA V7.5

You can use INCA V7.2 instruments in INCA V7.5 without any problems if the following condition is fulfilled:

– the instrument is built for the target platform “Any CPU”

To use the instruments in INCA V7.5, copy the instruments to the instruments directory: `<ETASData>\INCA7.5\Instruments\`

14 Troubleshooting

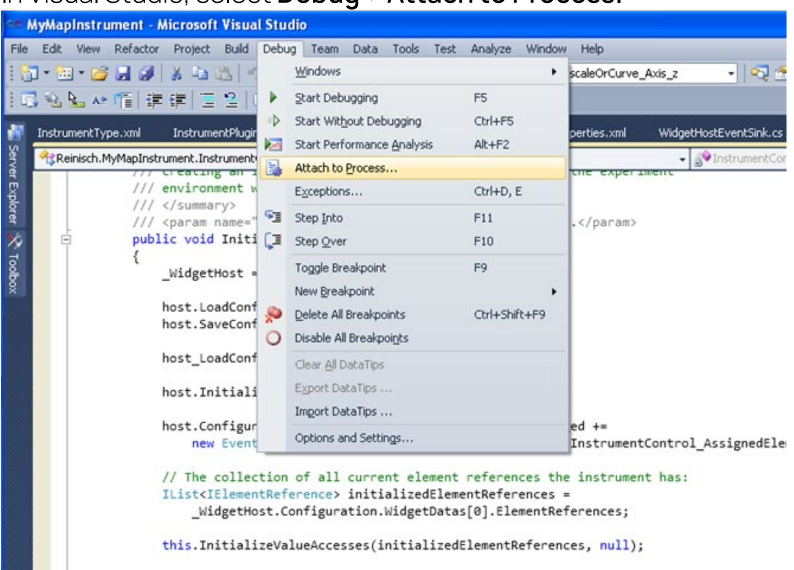
14.1 Objectives

In this chapter the most frequent problems that might occur during the creation and customizing of instruments and their solutions are described.

A short introduction to the debug tool of Visual Studio is given. You can debug your instrument at runtime using the Visual Studio debug functionality.

To debug an instrument

1. In Visual Studio, select **Debug** › **Attach to Process.**

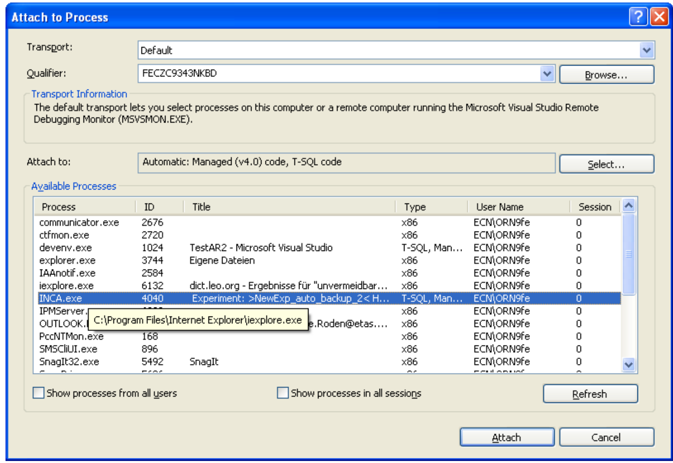


NOTE

This function only works if INCA is open (Runtime!).

The "Attach to Process" window with all active processes opens.

2. Go to `INCA.exe`.
3. Click **Attach.**

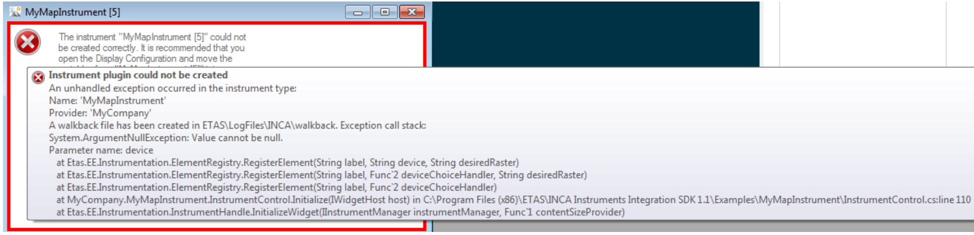


You can now add breakpoints to you source code.

The debugger will stop when your instrument reaches the corresponding line during execution.

14.2 Red Instrument in the INCA Experiment

INCA did not load the instrument correctly because there is an error in the source code:



If you move the mouse over the red cross in your instrument, INCA will show you the error message and Stack Trace of the Exception that occurred. Here you find the `method`, and `line number` where the `exception` occurs. Correct this exception in your instrument, rebuild, restart INCA and the error should be gone.

The INCA logfiles also contain this call stack.

14.2.1 Instrument Properties do not Show in INCA

To load the `Properties` into INCA, the `RefType` in the `InstrumentType.xml` must be identical to the string in the definition of the `PropertyGroup` in `InstrumentProperties.xml`.

To check the name of PropertyGroup

1. **Compare** the strings in `InstrumentType.xml` and in `InstrumentProperties.xml`.

```
<Properties RefType="InstrumentProperties" />
<WidgetDataTypes >
  <WidgetData >
   ...
```

2. Make them identical in case of difference.

```
<PropertyGroups >
  <PropertyGroup Type="InstrumentProperties" />
   ...
```

If the strings are identical, the problem should no longer prevail.

14.2.2 Property Modification in `InstrumentProperties.xml` does not show in INCA

After changing the properties, you always must load your modification to INCA.

To load the modifications

1. Make sure INCA is closed.
2. In Visual Studio, select **Build** > **Build Solution.**
3. Open INCA.
4. In the Variable Selection Dialog, call `MyInstrument`.

5. Verify if a respective variable has been assigned and assign it.
6. Click **Ok.**
   Your modifications are now active in `MyInstrument`.

14.3 INCA Test Environment

ETAS extended the Instrument SDK by a simple test environment that allows you to verify the function of your instrument without having to open INCA. The functionality of the test environment is limited, since most of INCA's behavior, regarding calibration access and configuration handling, cannot be simulated. However, the test environment allows you to validate the user interface specific implementations faster.

When you have opened your instrument project in Visual Studio, press <F5>. The test environment automatically opens with your instrument already loaded.

You find the test environment in the following folder of the SDK installation:

`Binaries\TestEnvironment\OpenEETestEnvironment.exe`

NOTE

The test environment only provides a limited set of variable types and does not provide the interface for `LoadConfiguration` and `SaveConfiguration`.

15 Further Documentation

For further information, open the Windows Start Menu:

`Start -> ETAS INCA Instruments Integration SDK 1.5. -> …`

16 ETAS Contact Information

Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the website: www.etas.com/hotlines

ETAS Headquarters

ETAS GmbH		
Borsigstraße 24	Phone:	+49 711 3423-0
70469 Stuttgart	Fax:	+49 711 3423-2106
Germany	Internet:	www.etas.com

Tables

Tab. 5-1 The ValueAccess interfaces for read only and read/ write access 47
Tab. 5-2 Type specific members of ElementInfo.. 48
Tab. 5-3 GetValueAccess survey.. 49
Tab. 10-1 Survey on axis options.. 80
Tab. 10-2 AxisType definition by DataType 81

Definitions and Abbreviations

ASAP2

Known in full name as ASAM-2MC by the Association of Standardization and Automation of Measuring Systems

CDM

Calibration Data Manager

EE

Experiment Environment

MC

Measurement and calibration

UI

User Interface

VSD

Variable Selection Dialog

WPF

Windows Presentation Foundation