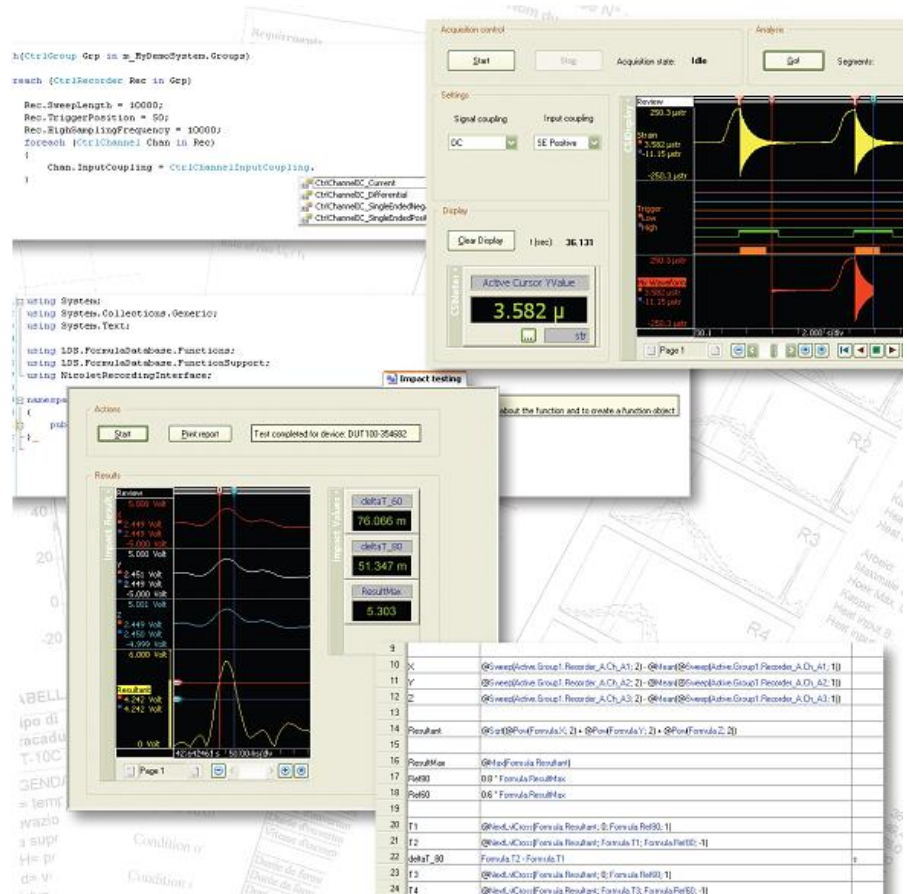


User Manual

English



CSI Programming for GEN series

Document version 5.0 – March 2020

For Perception 7.40 or higher

For HBM's Terms and Conditions visit www.hbm.com/terms

HBM GmbH
Im Tiefen See 45
64293 Darmstadt
Germany
Tel: +49 6151 80 30
Fax: +49 6151 8039100
Email: info@hbm.com
www.hbm.com/highspeed

Copyright © 2019

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

LICENSE AGREEMENT AND WARRANTY

For more information about LICENSE AGREEMENT AND WARRANTY refer to:

www.hbm.com/terms

Table of Contents	Page
1 GETTING STARTED	8
1.1 INTRODUCTION	8
1.2 INTENDED AUDIENCE	8
1.3 REQUIREMENTS AND INSTALLATION	8
1.3.1 <i>System requirements</i>	9
1.3.2 <i>Supported hardware</i>	9
1.3.3 <i>Installation</i>	9
1.4 STARTING CSI	10
1.4.1 <i>To quit the Perception CSI</i>	10
1.5 WHERE DO YOU START?	10
2 USING PERCEPTION CSI	11
2.1 WHAT CAN YOU EXPECT?	11
2.2 USAGE OF SUPPORT FILES	12
2.2.1 <i>To load the Perception CSI template</i>	12
2.3 YOUR FIRST CSI PROGRAM	13
3 PERCEPTION CSI TEMPLATE	17
3.1 THE FUNDAMENTALS OF SHEET INTERFACING	17
3.1.1 <i>Sheet Control Information Provider</i>	17
3.1.2 <i>Sheet Control Information Retriever</i>	19
3.1.3 <i>Sheet Control</i>	19
3.1.4 <i>Members</i>	19
3.1.5 <i>ISheet Members</i>	22
3.1.6 <i>Events</i>	22
3.1.7 <i>Properties</i>	25
3.1.8 <i>Methods</i>	27
3.1.9 <i>ISerializable Members</i>	31
3.1.10 <i>Disposing</i>	31
4 ACQUISITION CONTROL	32
4.1 BASIC ACQUISITION CONTROL	32
4.1.1 <i>Getting started</i>	32
4.1.2 <i>The user interface</i>	33
4.1.3 <i>Event handling</i>	35
4.1.4 <i>Add menu commands</i>	39
5 HARDWARE SETTINGS	40
5.1 HARDWARE ORGANIZATION	40
5.2 GET AND SET PARAMETERS	41
5.3 USER INTERFACE	41
5.4 GET AND SET PARAMETERS USING CAPABILITIES	42
5.5 INITIALIZE	43
5.6 USER INTERFACE WITH CAPABILITIES	43
5.7 GET CAPABILITIES	44
5.8 MODIFY A HARDWARE SETTING	48
5.9 CREATE A CLASS	49
5.9.1 <i>Modify the setting</i>	50
5.10 SYNCHRONIZATION	53
6 DATA VISUALIZATION	56
6.1 DATA DISPLAY	56
6.1.1 <i>User interface</i>	58
6.1.2 <i>The code</i>	58
6.2 METER	61
6.2.1 <i>User interface</i>	61
6.2.2 <i>The code</i>	62
7 DATA ANALYSIS - PART ONE	66

7.1	INTRODUCTION	66
7.2	THE DATA MANAGER	66
7.2.1	<i>Numerical data source</i>	67
7.2.2	<i>String data source</i>	68
7.2.3	<i>Waveform data source</i>	69
7.3	USER DATA SOURCES	69
7.3.1	<i>User numerical data source</i>	70
7.3.2	<i>User string data source</i>	71
7.3.3	<i>User waveform data source</i>	71
7.4	THE EXAMPLE	73
7.4.1	<i>User interface</i>	73
7.4.2	<i>The code - Getting started</i>	74
7.4.3	<i>Manipulate measurement cursor position</i>	75
7.5	DO SOME BASIC MATH	80
7.6	USING PERCEPTION WAVEFORM CALCULATORS	82
8	DATA ANALYSIS - PART TWO	89
8.1	FORMULA DATABASE AS CALCULATOR	89
8.1.1	<i>User interface</i>	89
8.1.2	<i>The code</i>	90
8.2	MAKE YOUR OWN FUNCTIONS	93
8.2.1	<i>Create and initialize the external class library</i>	94
8.3	THE FUNCTION INFORMATION	94
8.3.1	<i>Category</i>	96
8.3.2	<i>CreateFunction</i>	96
8.3.3	<i>Description</i>	96
8.3.4	<i>Example</i>	96
8.3.5	<i>MinimumParameterCount</i>	97
8.3.6	<i>Name</i>	97
8.3.7	<i>Parameters</i>	97
8.3.8	<i>ParameterDescriptions</i>	97
8.3.9	<i>ParameterNames</i>	97
8.3.10	<i>ParameterTypes</i>	98
8.4	GETTING IT ALL TO WORK	99
8.5	IMPLEMENT THE FUNCTION	99
8.5.1	<i>Using the comprehensive method</i>	100
8.5.2	<i>Using the intelligent method</i>	100
8.6	SUMMARY	109
9	AUTOMATION	110
9.1	EXAMPLE: POST-ACQUISITION ANALYSIS AND REPORTING	110
9.1.1	<i>Procedure</i>	110
9.1.2	<i>Before you begin</i>	110
9.1.3	<i>User interface</i>	112
9.1.4	<i>The code</i>	114
9.1.5	<i>Acquisition control</i>	114
9.1.6	<i>Print control</i>	119
9.1.7	<i>Calculations</i>	128
9.2	POINTS OF CONSIDERATION	133
10	USER-KEY SCRIPT ACTION	135
10.1	PERCEPTION.SCRIPTACTION	135
10.1.1	<i>To load the Perception CSI template</i>	135
10.2	YOUR FIRST USER KEY SCRIPT ACTION	136
10.2.1	<i>IScriptActionInfo</i>	139
10.2.2	<i>IScriptAction</i>	140
10.2.3	<i>IConfigurable</i>	141
10.3	EXAMPLE: CREATE A SCRIPT ACTION FOR AUTO SCALING ALL TRACES OF THE ACTIVE DISPLAY ..	141
10.4	ADD OPTION TO SELECT ALL OR ONLY ACTIVE TRACE TO BE AUTO SCALED	142
11	SUMMARY	145
12	APPENDIX: MULTITHREADING	146

1 Getting Started

Welcome to the Perception Custom Software Interface (CSI). CSI is a powerful technology which allows software integrators and Perception users to customize and automate (parts of) the Perception software. As opposed to the 'standard' API technology, programs written with CSI form an integral part of the Perception software and are fully integrated in the Perception user interface. They act like 'plug-ins'.

1.1 Introduction

While Perception and its options offer a satisfying solution for most measuring, processing and reporting tasks, there are still some areas where the supplied software is not tailored to your specific requirements.

A viable solution in this situation is to extend the Perception software with your own programs, using Perception CSI. CSI stands for **C**ustom **S**oftware **I**nterface. Perception CSI is your interface to the insides of Perception.

Writing your own program has the advantage that you have total control over your extensions, while you keep the flexibility and power of the Perception environment.

Other 'standard' interface techniques allow you to add an external program that works in parallel with the main application, either remotely or on the same machine. Using the Perception CSI you create plug-ins that become part of the Perception application on the same machine. These plug-ins have a user interface that is based on the Perception concept known as '**sheets**'. You will create a DLL that is linked into the Perception software at start-up, you will not create a stand-alone executable (*.exe)

Besides sheets it is also possible to create your own:

- **Functions:** these can be used in the formula database to do your own specific calculations
- **Automation Actions:** used to program special actions for the automation process
- **User-key script actions:** used to add special functionality behind user keys

Should you require external/remote control of the Perception software, then you should consider using the Perception Remote Control option (a.k.a. Remote API or COM-RPC).

This document describes a part of the CSI interface to Perception. It contains a command overview and examples. Examples are written in C# using the Microsoft Visual Studio 2015 development environment.

1.2 Intended audience

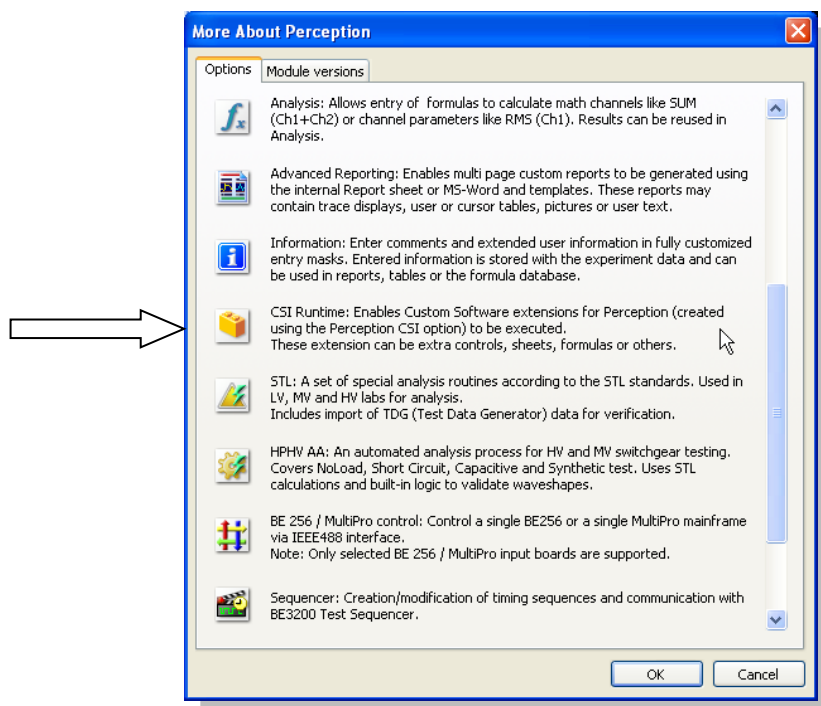
CSI is designed to be used by C# ("See-Sharp") programmers. You must be proficient in this programming language and Windows technology in order to write custom programs. This documentation assumes you understand your HBM equipment, software, and basic acquisition terminology.

Understanding acquisition terminology is vital to understanding digital recordings: trigger, sample rate, pre-/post trigger, etc.

1.3 Requirements and installation

The HBM CSI is an option that is enabled through the use of the HASP®4 USB Token.

This option is also listed as **CSI: Custom Software Interface** in the Perception menu:
Help > About Perception > More... > Options page



In addition you must install the required software modules as described below.

1.3.1 System requirements

- HBM Perception software with CSI option enabled
- Microsoft® Windows-7 or later
- Microsoft DirectX 9 or higher (included on media)
- Microsoft .NET 4.6
- 4GB of RAM memory - 4 GB or more recommended and required when working with more than one data acquisition mainframe.

1.3.2 Supported hardware

- HBM GEN Series Modular Data Acquisition System

1.3.3 Installation

Depending how you received your copy of the CSI Software Developers Kit do one of the following:

- For a zipped download: unzip the file in a separate folder. In that folder run setup.exe.
- When you received the CSI SDK on a CD you must install the SDK from the CD onto your hard disk; you cannot use CSI from the CD.

To install:

1. Start Windows and insert the CD in the CD-ROM drive.
 2. In the Windows Task Bar click the **Start** button, point to and click **Run....**
 3. In the **Run** dialog type d:\setup (or e:\setup, depending on your CD-ROM drive assignment) in the **Open:** text input field and click **OK**.
 4. Follow the on-screen instructions.
- When you received the CSI SDK as part of the Perception installation CD, locate the CSI folder on that CD. In that folder run setup.exe.

1.4 Starting CSI

CSI, when enabled, is part of the Perception software engine. When you start Perception you have direct access to the CSI commands and functions.

1.4.1 To quit the Perception CSI

The Perception CSI is automatically closed when Perception is closed.

1.5 Where do you start?

Investigate the sample projects; decide which one is closest to the functionality you want to provide. Run it in your software development environment and set breakpoints. The documentation provided is intended as a guide with detailed explanation only where appropriate; watching the information within the breakpoints is a good way to understand what happens. Feel free to explore and experiment with the CSI on your own once you're familiar with it, but please, resist the temptation to start from scratch until you're confident; you may end up repeating other developers' mistakes, including your own.

This manual gives you only an introduction to the CSI interface; there are a lot more topics which are not covered here e.g. Start Manager, Word Reporting, Sweep Deletion, Object Manager, Excel Interaction, Sensor database, extending the RPC commands etc.

It is however also possible to join a **Perception CSI training**, for more information contact the technical support or go to the HBM web site: [HBM Perception API](#) or [HBM Perception](#). Although the examples are created with Visual Studio 2017 it is also possible to use the free Visual Studio Community IDE. You can download this version for free from the Microsoft website.

2 Using Perception CSI

CSI is designed to give users access to the Perception internals. As opposed to standard API programming there is no layer between the user software and Perception. Since the user software is integrated, it is also created as a DLL instead of an executable. The user interface provided is based on the Perception sheet concept. On a sheet, the user is free to build their own interface. By default, a sheet comes with a proprietary menu.

One of the clear benefits of CSI is that it enables your organization to achieve a tight integration between your knowledge and requirements and the Perception platform. This approach also allows for customization on user request by independent software integrators, in close cooperation with HBM.

2.1 What can you expect?

In this document we will describe a variety of functions, but not all and we will also not treat every subject exhaustively.

The CSI SDK comprises:

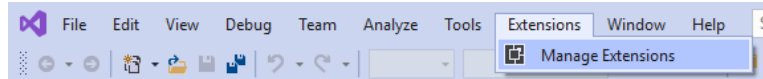
- Access to all public and documented features:
 - User modes
 - Components and sheet manager
 - Workbench:
 - Workbook(s)
 - File operations
 - Sheet operations
 - Perception DLL's, e.g.:
 - Formula database
 - Recordings
 - .NET DLL's

- Starter kit: C# template for Microsoft Visual Studio

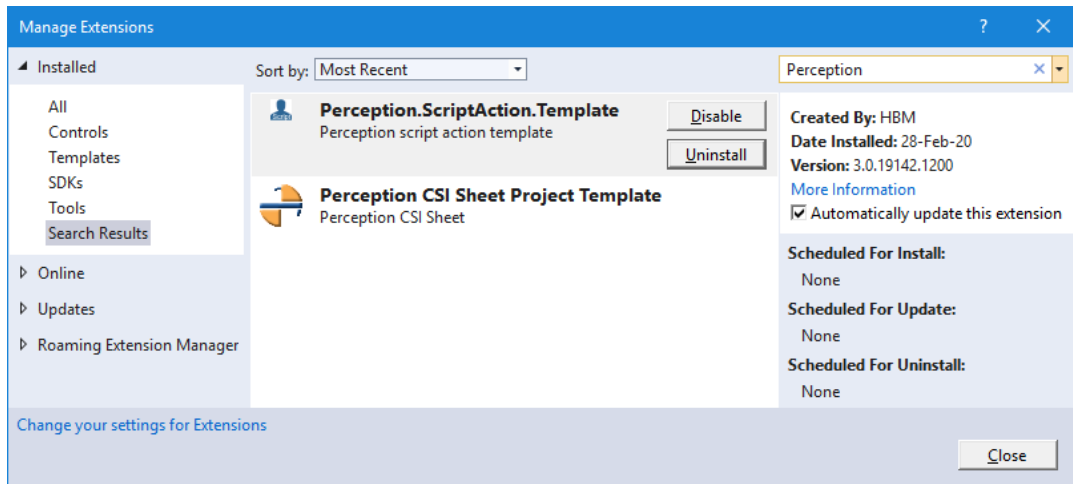
You cannot add functionality through other Perception components: you cannot modify existing menus, graphs or displays, etc. You do have access to standard features. All this will be explained in this document.

2.2 Usage of support files

For use with the Microsoft Visual Studio programming environment and C# a template is provided. Depending on how you installed your CSI SDK and your computer environment this file might already be installed. To verify, start your Microsoft Visual Studio 2019, and select **Extension >Manage Extensions**



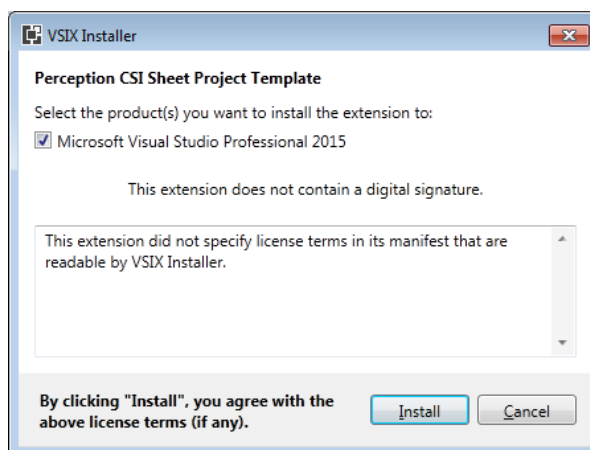
In the dialog that comes up type Perception in the right upper search box:



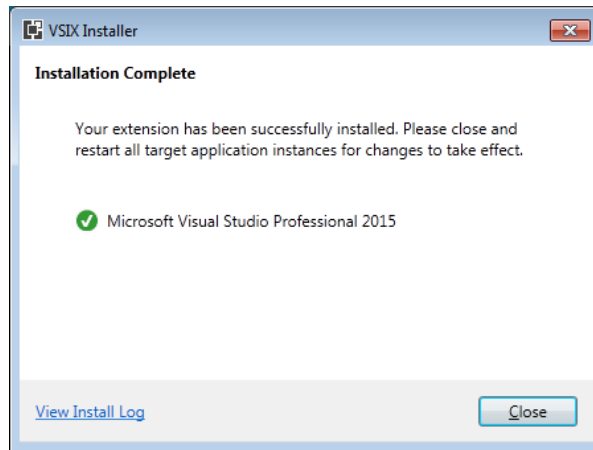
In the Template section, the **Perception CSI sheet** should be available. If not so, proceed as described below.

2.2.1 To load the Perception CSI template

1. Locate the file named **Perception CSI Sheet Project Template.vsix**
2. Double-click this file. The VSIX installer will be launched:



3. Click **Install** to install

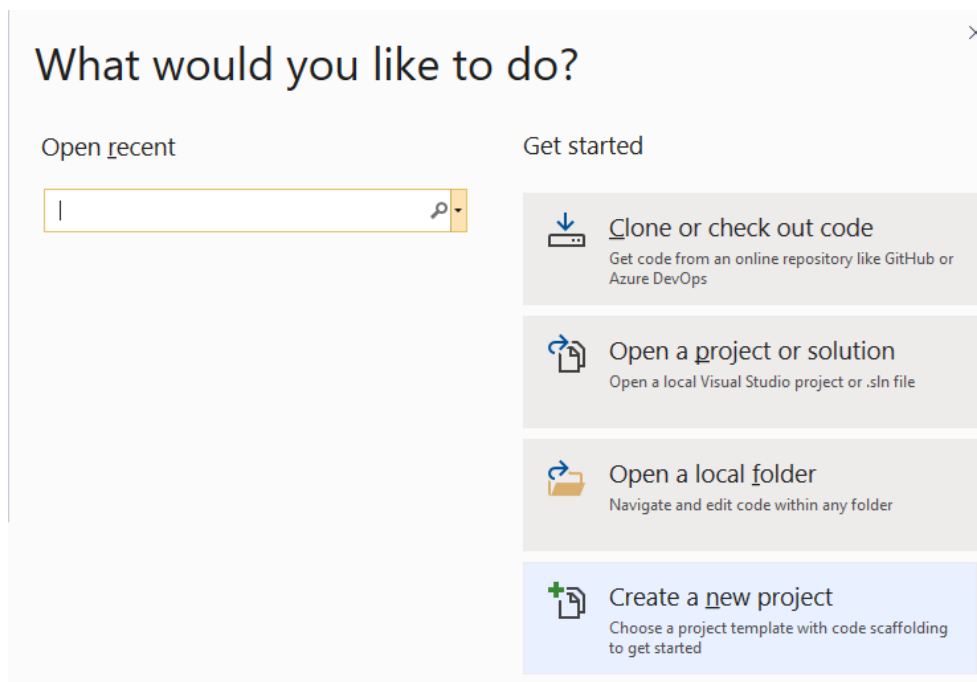


4. Click Close

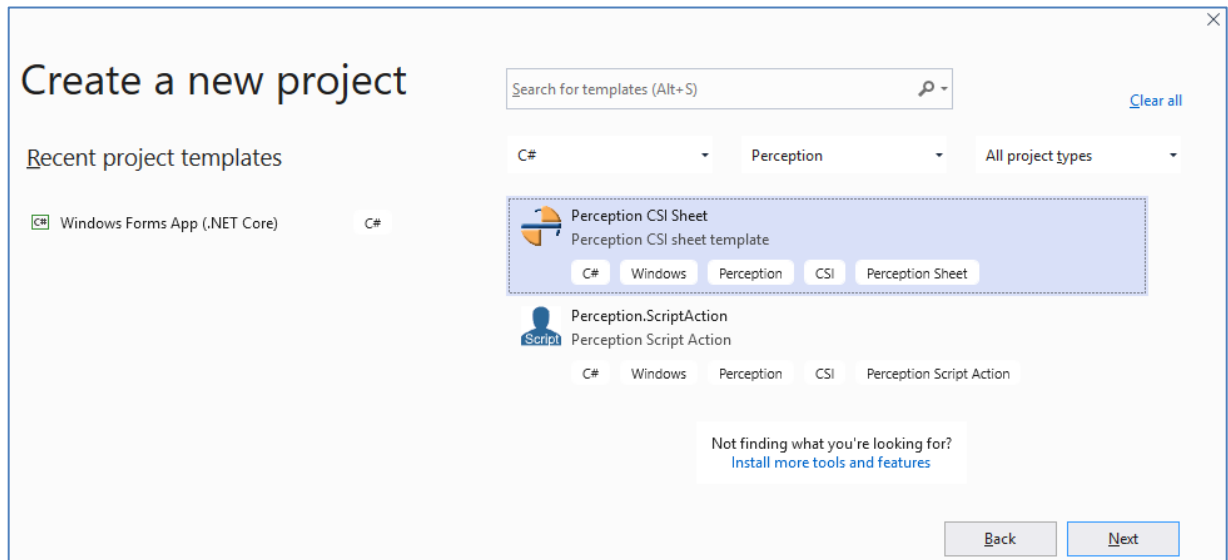
2.3 Your first CSI program

You now should be able to create, compile and run your first CSI program. To do so proceed as follows:

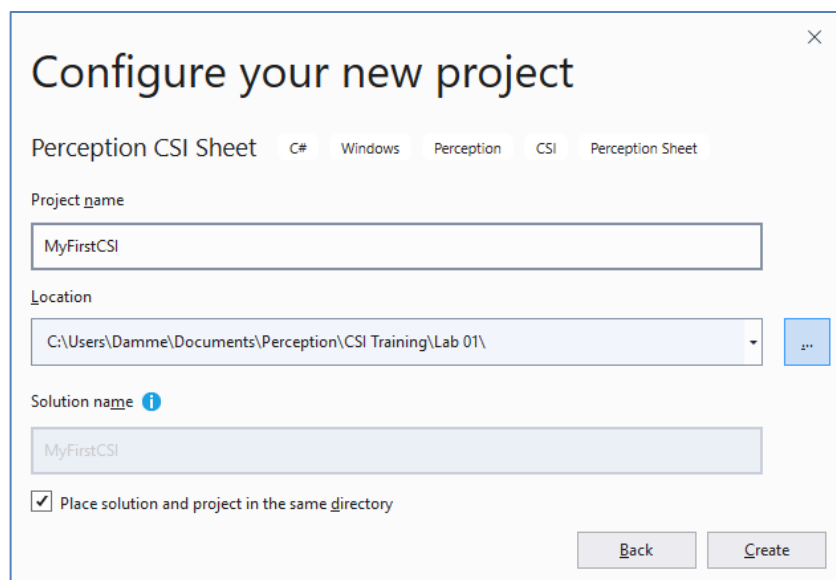
1. Start your Microsoft Visual Studio and select **Create a new project**.



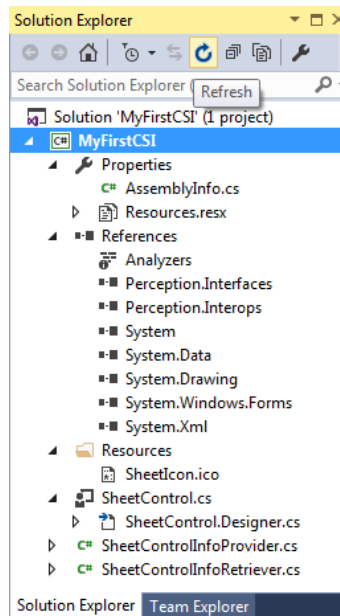
2. In the dialog that comes up select **C#** in the first selection box.
3. **Perception** in the second selection box.
4. **All project** types in third selection box
5. The **Perception CSI Sheet** template selection should now be visible:



6. Select the **Perception CSI Sheet** template
7. Enter a name **MyFirstCSI** and location for this project and click **OK**



The Solution Explorer will now include the following:



- A reference to the Perception Interfaces
- C# code for the **SheetControl**

This code is enough to create a sheet in Perception. Before we can build it, we need to add some more information to the project itself.

Optionally give the sheet a name and icon other than default:

1. Go to **Project > <Project Name> > Properties**
2. Go to **Resources > Strings** and modify the text from `IDS_USERNAME` into your sheet name and add a descriptive text for your sheet.
3. Go to **Resources > Icons** and select **Add Resource > Add Existing File**. Browse to and select your own icon file.
4. Remove the default icon
5. Rename your icon into "SheetIcon"
6. Go to **Debug-> Start external** program and enter the full name where Perception.exe is located, default this will be: "C:\Program Files\HBM\Perception\Perception.exe"

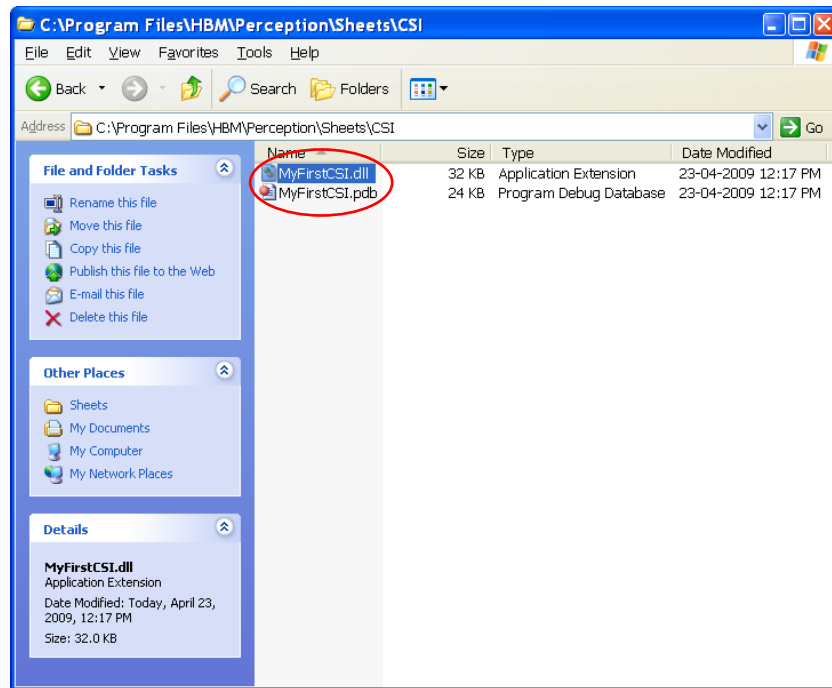
Mandatory:

1. Go to **Project > Project Name Properties**
2. Go to **Build > Output > Output path**
3. Verify the output path: **C:\Program Files\HBM\Perception\Sheets\CSI**
4. Go to **Build > Configuration** and select **Release**
5. In the main menu select **Build > Build Solution**

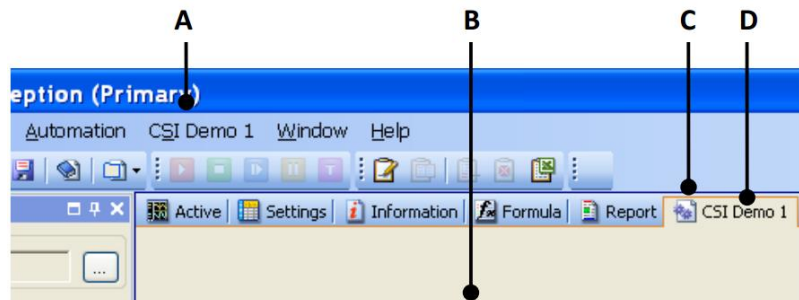
When all is OK, no error messages are generated. However, **ignore** warnings for the time being.

When error messages are generated verify all of the above steps. Also make sure you have the latest version of the template and the latest version of Perception.

Use the Windows Explorer to have a look in the output directory.



Between the already installed sheets you will see the one we just created. To verify the operation, start Perception.



You should now see the:

- A. Default menu for your sheet
- B. Empty sheet area
- C. Sheet icon
- D. Sheet name

Note: the output path as specified in the Build section of the Project Properties should be the same for each Configuration: debug as well as release.

Now we know that the basics are functioning. In the next chapter we will have a look into the code and structure of the template.

3 Perception CSI Template

This chapter of the document describes the fundamentals of the provided template. The template comprises all available functions required to communicate with Perception, as demonstrated already in the previous chapter: without writing one line of code we were able to create a sheet. However, the template is more: it is the starting point of complete integration of your application specific software and Perception.

3.1 The fundamentals of Sheet interfacing

Perception uses a Sheet Manager to control Sheets. The fundamental information of a sheet is provided by the code in **SheetControllInfoProvider.cs**.

3.1.1 Sheet Control Information Provider

In the **SheetControllInfoProvider.cs** the Perception Sheet Manager can find information like:

- Sheet descriptive name
- Sheet descriptive icon
- Preferred sheet index
- Can we create more than one instance of the sheet, automatic or manually

This information is used even before a sheet is created.

Before we create a sheet we must determine if we want the sheet to be automatically created or not. The following options are available:

- **Not manually creatable:** the sheet is created automatically by Perception at startup
- **Manually creatable:** the sheet is not created automatically by Perception at startup, you need to add the sheet manually. You can create multiple instances of your sheet.
- **Manually creatable only once:** the sheet is not created automatically by Perception at startup, you need to add the sheet manually. You can create only one instance of your sheet.

In the **SheetControllInfoProvider.cs** the code is as follows:

```

public ManuallyCreateType ManuallyCreatable
{
    get
    {
        return ManuallyCreateType.NotManuallyCreatable;
    }
}

```

This is the default situation as shown in the previous chapter. Now change the return value into:

```

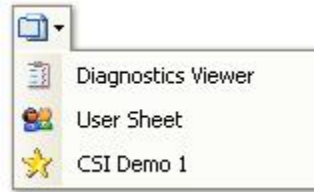
return ManuallyCreateType.ManuallyCreatableSingleInstance;

```

and build the solution again. If you now Start Perception you will see that no sheet is loaded.

To load the sheet, do one of the following

- In the **File** menu point to **New Sheet** and select the sheet in the submenu that comes up.
- In the toolbar click the **New Sheet** icon. In the menu that comes up select the sheet. (see below)



- Do a right mouse-click in the tab area of the sheets. Point to New Sheet and select the sheet in the submenu that comes up.

You can modify the position of the sheet tab in the row of sheet tabs, when the sheet is automatically created (default). You can do this in the following code:

```
public int PreferredSheetIndex
{
    get
    {
        return -1;
    }
}
```

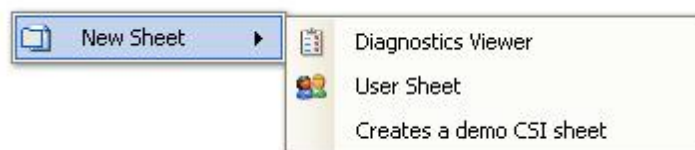
The first position (index = 0) is reserved for the Perception Active Sheet. Use a return value of 1 for the second position, 2 for the third position, etc. Use -1 for default.

You can modify the name and icon as shown in the New Sheet list by modifying the corresponding code, e.g.

```
public string Name
{
    get
    {
        return "Creates a demo CSI sheet";
    }
}

public Icon Icon
{
    get
    {
        return null;
    }
}
```

Now these settings are used for the New Sheet menus.



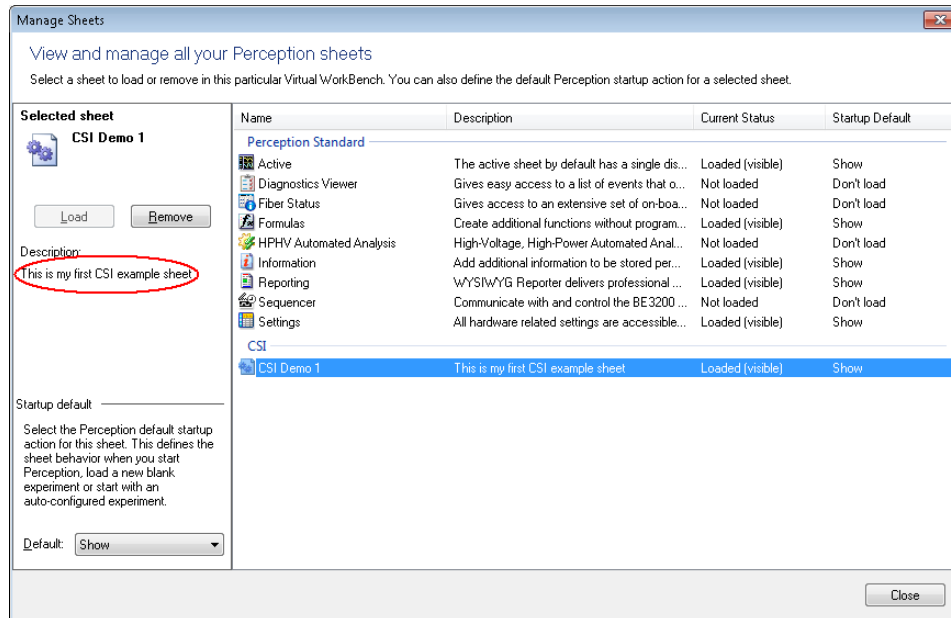
Note that the Sheet itself still has the name and icon as defined earlier.

3.1.2 Sheet Control Information Retriever

In the **SheetControlInfoRetriever.cs** the Perception Sheet Manager can find information like:

- Sheet description

The sheet description is used by the Sheet manager



3.1.3 Sheet Control

The file **SheetControl.cs** has all the fundamentals on board to communicate with the Perception application itself. The source code provides five main regions:

- Members
- Constructor
- Sheet Members
- Serializable Members
- Disposing

We will discuss these in the following sections.

3.1.4 Members

The members region comprises the members of the sheet interface:

- `m_strUserName`
- `m_iProgram`
- `m_UIState`
- `m_InitializeState`
- `m_bDisposed`

and are defined as follows:

```
private string m_strUserName = Properties.Resources.IDS_USERNAME;

private IProgram m_iProgram = null;
private UIState m_UIState = UIState.Invisible;
private InitializeState m_InitializeState = InitializeState.Unknown;
private bool m_bDisposed = false;
```

m_strUserName

This is the name we already have defined in the resources.

m_iProgram

The interface to the Perception program. This member interfaces to the **ComponentManager**, **Experiment**, **SheetManager**, **UserMode**, Workbench and **ApplicationSettings**. So far we have already used the **SheetManager** and in this section we will also see the use of the **UserMode**.

m_UIState

This member (User Interface State) defines the "visibility" of the sheet. Initially this state is set to invisible. When Perception loads the sheet, this state is set to visible, unless we decide not to do so, as will be explained later.

You can see the state-change in action. To do so:

- Make sure that you have set in the project properties:
 - **Debug > Configuration: Active** (debug)
 - **Debug > Start Action > Start external program:** <Perception>
- And the output window available.

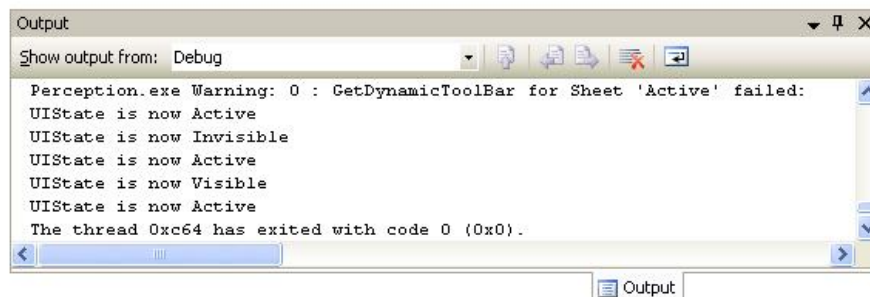
In the source code go to the **Events** region > **Properties** region and locate the code for the **UIState**. Now modify the code as follows:

```
public UIState UIState
{
    get
    {
        return this.m_UIState;
    }
    set
    {
        this.m_UIState = value;

        System.Diagnostics.Debug.WriteLine("UIState is now " +
            this.m_UIState.ToString());
    }
}
```

When Perception changes the UIState of the sheet, program execution will enter the UIState-set. Here we enter code to display the UIState in the debug output window.

Start debugging and see the various state-changes.



You can use this entry point for various functions. E.g. when you are running a video on your sheet, you can pause the video when the sheet becomes invisible and resume playing when the sheet becomes active again. (active = running + visible)

m_InitializeState

One of the first things Perception will do when it creates a sheet is verifying the initialization state. This is done before the sheet is created. The following states are currently supported:

- **InstalledIncorrectly**: currently not used
- **NotAllowed**: do not create the sheet
- **Succeeded**: all initialization is OK, create sheet
- **Unknown**: unknown

As an example, you could create a sheet that is always visible unless the user mode is 'review only', i.e. no control allowed.

In the source code go to the **Events** region > **Methods** region and locate the code for the **InitializeState**. Now modify the code as follows:

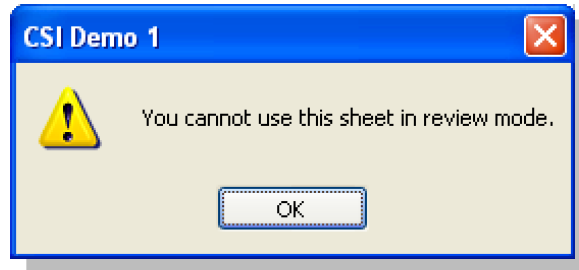
```
public InitializeState Initialize(IProgram iProgram)
{
    this.m_iProgram = iProgram;

    if (iProgram.UserMode == UserMode.Review)
    {
        this.m_InitializeState = InitializeState.NotAllowed;
        MessageBox.Show(this, "You cannot use this sheet in review mode.",
            "CSI Demo 1", MessageBoxButtons.OK, MessageBoxIcon.
            Exclamation);
    }
    else
    {
        this.m_InitializeState = InitializeState.Succeeded;
    }
    return this.m_InitializeState;
}
```

In this section you can also do other tests and initializations before the sheet is created.

Combine this with message boxes for user feedback. Note that the Initialize code is only executed once: the first time the sheet is created. When you hide and restore the sheet this code will not be executed.

Make sure that the sheet is `ManuallyCreatableSingleInstance` and do all the things we have done before, but now start Perception in Review mode. When you want to add the sheet the message box will come up



and the sheet will not be created.

m_bDisposed

This variable is used during the `CleanUp()` procedure to prevent you repeat the cleanup again.

Constructor

There is only one constructor that is required for Designer support and creates a sheet control object.

3.1.5 *ISheet Members*

These are the most widely used functions to interact between the sheet and the Perception application. These members are divided into the three main categories:

- **Events** provide a way for a class or object to notify other classes or objects when something of interest happens. The class that sends (or raises) the event is called the publisher and the classes that receive (or handle) the event are called subscribers.
- **Properties** are members that provide a flexible mechanism to read, write, or compute the values of private fields. Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- **Methods** are a code block containing a series of statements. In C#, every executed instruction is done so in the context of a method.

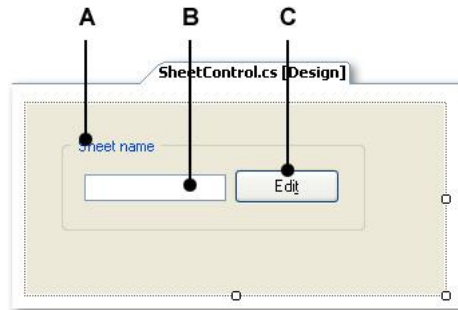
3.1.6 *Events*

Each sheet has its own menu and optional toolbar. When you go from one sheet to another, the corresponding menu/toolbar comes up. These are called dynamic menu and dynamic toolbar as opposed to the standard (static) menus and toolbars that are available within Perception. There are three events defined that are related to this dynamic behaviour:

- **UserNameChanged** Use this event when you modified within your 'application' the name of the current sheet.
- **ToolItemsUpdated** Use this event to notify Perception that something has happened to the dynamic toolbar.
- **RebuildDynamicMenuRequested** Notify Perception that you want to rebuild the menu.

In the next example we will demonstrate the use of `UserNameChanged` and `RebuildDynamicMenuRequested`.

In this example we will start with adding some user interface to our sheet. To do so, start the `SheetControl.cs` in Design mode and add two common controls: a `TextBox` and a `Button`, grouped in a `GroupBox` container as follows:



A GroupBox

B TextBox "textBox1"

C Button "button1"

The operation will be as follows: initially the `TextBox` is disabled and contains the sheet name. When you click on the `Button` the `TextBox` is enabled for editing and the text of the `Button` changes into "Enter". Now you can modify the sheet name. Click the `Button` again to conclude the modification.

The initialization code is placed in `SheetControl_Load`, the other code is placed in the `button1_click` event. The complete code could look like this:

```

private void SheetControl_Load(object sender, EventArgs e)
{
    m_bEditFlag = false;
    textBox1.Text = m_strUserName;
    textBox1.Enabled = false;
}

private void button1_Click(object sender, EventArgs e)
{
    if (m_bEditFlag == false)
    {
        button1.Text = "En&ter";
        textBox1.Enabled = true;
        m_bEditFlag = true;
    }
    else
    {
        UserName = textBox1.Text;
        button1.Text = "Edi&t";
        textBox1.Enabled = false;
        m_bEditFlag = false;

        // Fire the RebuildDynamicMenuRequested event to update the
        // dynamic menu name and toolbar
        if (this.RebuildDynamicMenuRequested != null)
            this.RebuildDynamicMenuRequested(this, new EventArgs());
    }
}

```

In the SheetControl_Load we initialize a state flag that has been defined as member in the Members > ISheet region as :

```

private Boolean m_bEditFlag = false;

```

The textbox is initialized as defined with user name and disabled. Note that the SheetControl_Load code is only executed once: the first time the sheet is loaded after creation. When you hide and restore the sheet this code will not be executed.

When you click on the button, initially the textbox is enabled, the button text changes and the flag is set to **true**.

Now you can edit the text.

When you click on the button for the second time, the text from the textbox is passed to "UserName" as argument.

UserName is defined in the **Events > Properties** region. It gets and sets the sheet name. When the sheet name is modified, it fires the **UserNameChanged** event. This allows Perception to react on this event and modify the sheet name in the tab of the sheet.

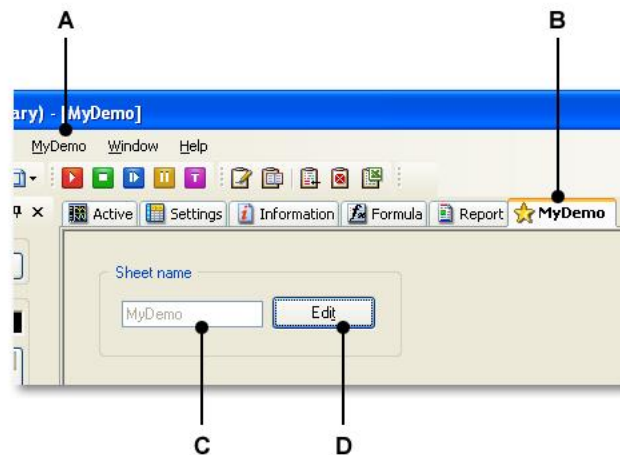
After this the button text is modified, the textbox is disabled and the flag is set to false again.

Although Perception will update the sheet name in the tab of the sheet, it will not automatically update the name of the dynamic menu at once. In order to do this, we need also to fire the **RebuildDynamicMenuRequested** event. To do this test if a change is made and if so, fire the event.

To test this piece of software set the sheet to be **ManuallyCreatable** in the **SheetControlInfoProvider**. This allows for multiple instances of the sheet.

Now you can start debugging.

When Perception is launched in any user mode - except Review only - you can add this sheet. Modify the name as described earlier.



- A Dynamic menu
- B Sheet tab
- C Text box with new name entered
- D Edit / Enter button

You can add multiple sheets and give each sheet its own name.

3.1.7 Properties

A sheet has several properties. Perception uses these properties. Therefore these properties must be available to Perception. Some of the properties are read-only, others are read-write properties.

These properties are defined in the **ISheet > Properties** region. Currently the following properties are available:

- **UserName** (R/W) The sheet name
- **MenuName** (R) The name of the dynamic menu, usually the same as the sheet name
- **Icon** (R) Sheet icon
- **DeleteActiveItemSupported** (R) When an object on a sheet is active it can be deleted or not (default = no)
- **UIState** (R/W) User interface state, e.g. visibility
- **InitializeState** (R) Gets the initialization state of the sheet

UserName

The User name / sheet name already has been discussed in detail in the ISheet Members > Events section.

MenuName

The menu name is a read only property. By default it is set to the user name. However, you

can modify it to be any other name. E.g. when you own a company that manufactures everything, called ACME, you would like the menu header to be ACME also. This has advantages when:

- You want your name to be on top of the menu, regardless of the name of the sheet
- You want your name to be on top of the menu, even if you have multiple instances of a sheet with different names
- You want your name to be on top of the menu, even if you have multiple types of sheets

Please note that although the name remains the same, the dynamic menu contents depends on the selected sheet.

You could modify the MenuName properties as follows:

```
public string MenuName
{
    get
    {
        if (String.IsNullOrEmpty(this.m_strUserName))
        {
            // Return a string from the resource table when no current
            // User Name is available
            return Properties.Resources.IDS_USERNAME;
        }
        else
        {
            // Return the current User Name or selected name
            return "ACME Company";
        }
    }
}
```

If you run the previous example (modify user name of sheet) with these modifications and make two instances of the same sheet, you will notice that the menu name remains the same and that the contents of the dynamic menu is related to the selected sheet.

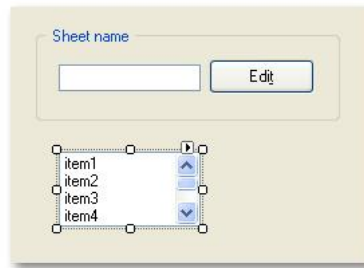
Icon

The Icon property is read only. You could choose to modify it, although there is no valid reason to do so.

DeleteActiveItemSupported

When you want to have the possibility to **Delete** items on your sheet, you must inform Perception to do so. This is done through the **DeleteActiveItemSupported** property. By default this is set to 'false'. When you want to have this option, you need to set this property to true. When the menu is pulled down, Perception interrogates this property and once the option is set to true, the Delete command in the Perception Edit menu comes available.

As an example add a listbox with some items to your sheet:



Enter some items and modify the property code as follows:

```

public bool DeleteActiveItemSupported
{
    get
    {
        return ((listBox1.SelectedIndex >= 0) &&
            (m_bEditFlag == false));
    }
}
    
```

When nothing is selected in the listbox, it will return false, otherwise true, i.e. when nothing is selected there is nothing to delete, so the command is not available. This is combined with the textbox edit: when the textbox edit is (also) active, the command is not available since a textbox has its own edit-handling.

Note that keyboard handling must be done by yourself. The Delete command in the Edit menu is merely a 'goodie' to get you started.

When a delete command is issued, Perception enters the DeleteActiveItem code that is defined in the ISheet Members > Methods region. Modify as follows:

```

public void DeleteActiveItem()
{
    listBox1.Items.Remove(listBox1.SelectedItem);
}
    
```

Here you will need to implement the code required to perform the delete action, i.e. you need to find out which object is selected and perform an adequate delete action. In the above example the selected item will be deleted from the listbox.

UIState

The UIState has been described extensively in the section on the sheet control members: m_UIState.

InitializeState

The InitializeState has been described extensively in the section on the sheet control members: m_InitializeState.

3.1.8 Methods

A sheet provides a number of methods. Perception uses these methods. Therefore these

methods must be available to Perception.

These methods are defined in the ISheet > Methods region. Currently the following methods are available:

- **UpdateMenuItems** Update the contents of the dynamic menu
- **DeleteActiveItem** Code executed when a delete command from the Edit menu is issued
- **AutoConfig** Code for the auto-configuration feature
- **Initialize** Code executed when Perception initializes the sheet
- **PostLoad** Executes code when a new settings file has been loaded
- **GetDynamicMenu** Here comes the code that takes care of the creation of a dynamic menu
- **GetDynamicToolbar** Here comes the code that takes care of the creation of a dynamic toolbar

This section describes these methods in a different order than provided in the template.

DeleteActiveItem

This method already has been discussed in the section on the **ISheet Members > Properties > DeleteActiveItemSupported**.

AutoConfig

When you start Perception or select **File > New**, you have the option to let Perception create a "default" layout based on the hardware it finds.

You can also use this auto-configuration feature. E.g. you can load the drop-down list in the previous example with values based on preferences or any other reason. Currently this only works for sheets that are automatically created, i.e. not manually creatable.

Assume you want to fill the list with all available recorders within the mainframe that was selected during auto-configuration:

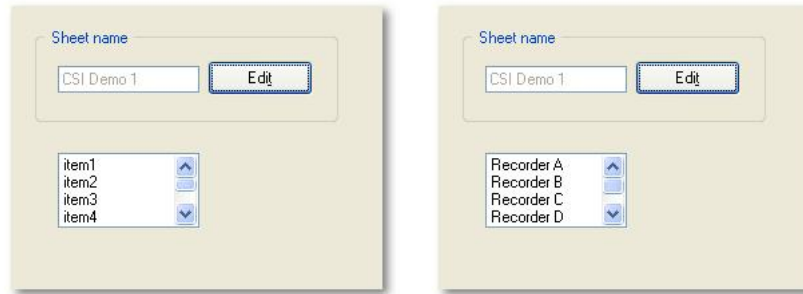
```
public void AutoConfig()
{
    listBox1.Items.Clear();
    CtrlAcquisitionSystem AcqSys = CtrlAcquisitionSystemFactory.Create();
    CtrlGroup GroupAll = AcqSys.Groups.GroupAll;
    foreach (CtrlRecorder Recorder in GroupAll.Recorders)
    {
        listBox1.Items.Add(Recorder.Name);
    }
}
```

First we clear the listbox contents. The next two lines will be explained in more detail in another section of this manual. For now: we create a group of recorders based on the group "All" of the acquisition system. This is the same group as you will find in the Perception Hardware Navigator.

Once we have this group we loop through all available recorders and display their name.

To test this piece of software set the sheet to be **NotManuallyCreatable** in the **SheetControlInfoProvider.cs**. This will create the sheet automatically.

First start Perception and do not use the auto configuration mode. Then select **File > New...** and select auto-configuration. In the Select Mainframe dialog that comes up select a mainframe.



Now the contents of the listbox will be updated. An example is shown in the diagram above: left is the situation before the auto-configuration, right you see the situation after the auto-configuration.

Initialize

This has been discussed in detail in the section on **Members > m_InitializeState**.

PostLoad

When Perception loads a new settings file it also executes the sheet code **PostLoad**. Here you can enter your code. This could be the same kind of code as you would do after an auto-configuration, e.g. loading a list with the available recorders.

GetDynamicMenu

This method is called when Perception want to display the dynamic menu. By default the dynamic menu has already two entries: move and delete sheet. The other entries are defined by the sheet.

To add items to the dynamic menu you must create a context menu strip in the sheet user interface. This menu is not actually deployed but used as a piece of menu transferred to the dynamic menu.

To add items to the dynamic menu

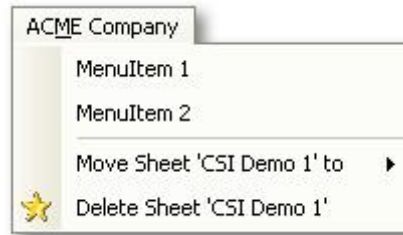
1. In the programming environment add a tool strip menu to your user interface: **Toolbox > Menus & Toolbars > ContextMenuStrip**
2. Use **Edit Items ...** to add/modify menu items
3. Add the following code:

```

public ToolStripItem[] GetDynamicMenu()
{
    ContextMenuStrip strip = contextMenuStrip1;
    if (this.IsDisposed || this.Disposing || (strip == null))
        return null;
    ToolStripItem[] aToolArray = new ToolStripItem[
        strip.Items.Count];
    strip.Items.CopyTo(aToolArray, 0);
    return aToolArray;
}
    
```

Here we create a 'toolstrip' item called **aMyItems** based on the size of the designed **contextMenuStrip**. After this we copy the contents of the contextMenuStrip to our toolstrip and return this toolstrip.

4. Run the program



You will see that the new commands are added. In a next chapter we will go into more detail in using the menu.

UpdateMenuItems

Menu items can be modified during program execution. E.g. when a start acquisition command is part of your menu, this command should be disabled while acquisition is active. A good moment to update your menu information is just before the menu becomes active, i.e. drops down. Now you are sure you have the latest information.

This function is provided through the **UpdatMenuItems** method.

Example:

```
public void UpdateMenuItems ()
{
    menuItem1ToolStripMenuItem.Enabled = !m_bEditFlag;
    toolStripButton1.Enabled = !m_bEditFlag;
}
```

Depending on the edit state of the sheet name as we have used so far, the second menu entry is disabled or enabled.

GetDynamicToolBar

In very much the same way as we created a dynamic menu, you can also create a toolbar related to your sheet. Instead of creating a context menu strip, you now create a toolstrip and modify its contents. Once created you can use the following code:

```
public ToolStripItem[] GetDynamicToolBar ()
{
    ToolStrip strip = this.toolStrip1;
    if (this.IsDisposed || this.Disposing || (strip == null))
        return null;
    ToolStripItem[] Result = new ToolStripItem[
        strip.Items.Count];
    strip.Items.CopyTo(Result, 0);
    return Result;
}
```

Make sure you have set the visibility to false.

3.1.9 *ISerializable Members*

This region contains the **GetObjectData** and **SheetControl** procedure that allows you to **save/load** sheet specific information when the **vwb** is saved/loaded. For each item that you want to save, choose a unique name and create an entry in the data stream to be saved as follows:

```
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("ModifiedSheetName", m_strUserName, typeof(string));
}
```

For each item add a line. In the above example the current sheet name *m_strUserName* is saved in the string variable named *ModifiedSheetName*.

Once saved you can restore the data when the same vwb is loaded as follows:

```
public SheetControl(SerializationInfo info, StreamingContext context): this()
{
    m_strUserName = Tools.GetValue<string>(info, "ModifiedSheetName",
                                           m_strUserName);
}
```

Here the saved string in variable **ModifiedSheetName** is restored into **m_strUserName**. When no data is available, the third parameter in **GetValue** is used as default. Typically this is the initialized version of the variable to be restored.

3.1.10 *Disposing*

This region contains the code where you can add your own dispose code. It contains the following generated lines:

```
private void SheetDisposed(object sender, EventArgs e)
{
    if (IsDisposed) return;
    if (m_bDisposed) return;

    try
    {
        // Add your clean up code
    }
    catch
    {
    }
    m_bDisposed = true;
}
```

Use this function if you want to release (e.g.) objects you are using.

```
private void SheetDisposed(object sender, EventArgs e)
{
    if (IsDisposed) return;
    if (m_bDisposed) return;
    m_Datamanager = null;
    m_bDisposed = true;
}
```

Later on in this manual we will come back to the meaning of the **DataManager**

4 Acquisition Control

One of the first actions when creating your own software is usually to see if you can control acquisition. In this chapter we will demonstrate the fundamentals of acquisition within the Perception software. Not only will we demonstrate how to create your own acquisition control, but also how to read acquisition status and how to implement a good user interface that complies with Perception as well as common guidelines for user interfaces.

4.1 Basic acquisition control

In this section we will describe the implementation of basic acquisition control as well as the steps required to implement hardware interfacing.

4.1.1 Getting started

Create a new project as described earlier with the correct name, icon and “creatability”. To get started we will:

- add a 'using' directive
- create an acquisition control object
- initialize program and object

Acquisition control is 'hosted' by the **Perception.ILO**. **ILO** stands for **Intermediate Layer Objects**, these layer objects are used to interface with the connected mainframe(s). Since we will be using the ILO multiple times, we will add a Using Directive at the beginning of the source code where already other directives are placed. Add this directive below the other CSI directives:

```
using Perception.Sheets;
using Perception;
using Perception.ILO;
```

Since we want to control an acquisition system, we will need to create an acquisition system object:

In the Members region add a region below the ISheet region as follows:

```
#region -> MyMembers

private CtrlAcquisitionSystem m_MyDemoSystem = null;
private CtrlGroup m_GroupAll = null;

#endregion
```

We now have a member that is a control of an acquisition system. The other member is **m_GroupAll**, the type of this member is **CtrlGroup**. This member variable will point to the GroupAll, this group is always available and contains all available recorders in the system.

Initialization will be done in the Initialize method as follows:


```

public InitializeState Initialize(IProgram iProgram)
{
    if (iProgram.UserMode == UserMode.Review)
    {
        m_InitializeState = InitializeState.NotAllowed;
    }
    else
    {
        try
        {
            m_iProgram = iProgram;
            m_MyDemoSystem = CtrlAcquisitionSystemFactory.Create();
            m_GroupAll = m_MyDemoSystem.Groups.GroupAll;
            m_InitializeState = InitializeState.Succeeded;
        }
        catch
        {
            m_InitializeState = InitializeState.NotAllowed;
            PerceptionMessageBox.Show(this, "Could not initialize sheet",
                "CSI: Catch", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    return this.m_InitializeState;
}
    
```

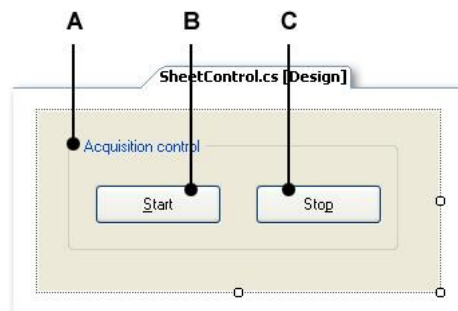
We start again with a simple test: when in review mode the sheet cannot be loaded.

In the following section we perform a basic test: when an error occurs within the **try** section, the **catch** code is executed, i.e. when the connection to Perception fails or when we cannot create an instance of the acquisition control object, the sheet is not loaded.

We can now start with the user interface.

4.1.2 The user interface

We will start with a very basic user interface. Also the code behind this user interface will be very elementary. Create a design as follows:



- A. GroupBox
- B. Button "StartCmd"
- C. Button "StopCmd"

For the time being we will initialize the buttons when the sheet becomes active: when no acquisition system is found, the buttons are disabled, otherwise the Start button is enabled.

```

public UIState UIState
{
    get
    {
        return this.m_UIState;
    }
    set
    {
        this.m_UIState = value;
        if (this.m_UIState == UIState.Active)
            // sheet becomes active
            {
                if (m_GroupAll.Recorders.Count == 0)
                    // no hardware -> no control
                    {
                        StartCmd.Enabled = false;
                        StopCmd.Enabled = false;
                    }
                else
                {
                    StartCmd.Enabled = true;
                    StopCmd.Enabled = false;
                }
            }
    }
}

```

The Start button is used to start an acquisition for all recorders. Also some acquisition parameters are set as example.

Enter the code for the Start button:

```

private void StartCmd_Click(object sender, EventArgs e)
{
    // for demo purposes -> set some acquisition parameters here
    // 10 kSamples at 10 kHz with trigger position at 50%
    foreach (CtrlGroup myGroup in m_MyDemoSystem.Groups)
    {
        if (myGroup.SupportsTimebase)
        {
            myGroup.SweepLength = 10000;
            myGroup.TriggerPosition = 50;
            myGroup.HighSamplingFrequency = 99990.0;
        }
    }
    m_GroupAll.Run();
    StartCmd.Enabled = false;
    StopCmd.Enabled = true;
}

```

For ease of use we start all recorders within all recorder groups at the same time with the same settings.

To do this we loop through the recorders collection within the **GroupAll** group of our acquisition system **m_MyDemoSystem**.

For each recorder we find we set the sweep length, trigger position and sample rate.

When done, the acquisition is started and the buttons are enabled or disabled as appropriate.

The Stop command is easier:

```
private void StopCmd_Click(object sender, EventArgs e)
{
    m_GroupAll.Stop();
    StartCmd.Enabled = true;
    StopCmd.Enabled = false;
}
```

When you run this program, you will notice some issues with respect to the update of the buttons. E.g. when you trigger an acquisition, the recording will stop but the buttons will not be modified. This is because we do not 'respond' to changes in the acquisition status. To do this properly we need an event handler that responds to the various event changes.

4.1.3 Event handling

As in real life it is sometimes not a bad idea to respond to events that happen instead of having to look all the time to a status.

In our environment we will do that by creating an event handler that responds to events fired by an object. In our situation we will respond to events fired by the **GroupAll** object. The **GroupAll** object is defined as follows in the **MyMembers** section:

```
private CtrlGroup m_GroupAll = null;
```

and initialized in the Initialize method:

```
this.m_GroupAll = m_MyDemoSystem.Groups.GroupAll;
```

after the initialization of m_MyDemoSystem.

We will always try to use hook and unhook functions to manage the event procedures in our program.

Therefore, you should create the following functions:

```
private void HookToGroupAll()
{
    UnHookFromGroupAll();
    if (m_GroupAll != null)
    {
        m_GroupAll.AcquisitionStateChanged +=
            GroupAllAcquisitionStateChanged;
    }
}

private void UnHookFromGroupAll()
{
    if (m_GroupAll != null)
    {
        try
        {
            m_GroupAll.AcquisitionStateChanged -=
                GroupAllAcquisitionStateChanged;
        }
        catch
    }
}
```

```

    {
    }
}

```

The complete code segment will now look as follows:

```

public InitializeState Initialize(IProgram iProgram)
{
    if (iProgram.UserMode == UserMode.Review)
    {
        this.m_InitializeState = InitializeState.NotAllowed;
    }
    else
    {
        try
        {
            this.m_iProgram = iProgram;
            this.m_MyDemoSystem = CtrlAcquisitionSystemFactory.Create();
            this.m_GroupAll = m_MyDemoSystem.Groups.GroupAll;
            HookToGroupAll();
            this.m_InitializeState = InitializeState.Succeeded;
        }
        catch
        {
            this.m_InitializeState = InitializeState.NotAllowed;
            PerceptionMessageBox.Show(this, "Could not initialize sheet",
                "CSI: Catch", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    return this.m_InitializeState;
}

void GroupAllAcquisitionStateChanged(object sender, int Running,
    int OneShot, int Stopping, int Paused, int Idle)
{
    throw new Exception("The method or operation is not implemented.");
}

```

It is good habit to clean up the code during the dispose of the sheet, therefore add the following code:

```

private void SheetDisposed(object sender, EventArgs e)
{
    if (IsDisposed) return;
    if (m_bDisposed) return;

    try
    {
        // Add your clean up code
        UnHookFromGroupAll();
        m_GroupAll = null;
        m_MyDemoSystem = null;
    }
    catch
    {
    }
    m_bDisposed = true;
}

```

Since the fundamentals are in place, we can modify the user interface to include a text label.

The label named **AcqStatus** is used to show the status of the acquisition.



The event handler can now be programmed:

```

void GroupAllAcquisitionStateChanged(object sender, int Running,
    int OneShot, int Stopping, int Paused, int Idle)
{
    this.InvokeOnUI(() => DoGroupAllAcquisitionStateChanged(sender,
        Running, OneShot, Stopping, Paused, Idle));
}
void DoGroupAllAcquisitionStateChanged(object sender, int Running,
    int OneShot, int Stopping, int Paused, int Idle)
{
    if (Running > 0 )
    {
        AcqStatus.Text = "Active";
        StartCmd.Enabled = false;
        StopCmd.Enabled = true;
    }
    else if (Paused > 0)
    {
        AcqStatus.Text = "Paused";
        StartCmd.Enabled = true;
        StopCmd.Enabled = true;
    }
    else if (OneShot > 0)
    {
        AcqStatus.Text = "Single shot mode";
        StartCmd.Enabled = false;
        StopCmd.Enabled = true;
    }
    else if (Stopping > 0)
    {
        AcqStatus.Text = "Stopping";
        StartCmd.Enabled = false;
        StopCmd.Enabled = false;
    }
    else if (Idle > 0)
    {
        AcqStatus.Text = "Idle";
        StartCmd.Enabled = true;
        StopCmd.Enabled = false;
    }
    else
    {
        AcqStatus.Text = "****";
        StartCmd.Enabled = false;
    }
}
    
```

```

        StopCmd.Enabled = false;
    }
}

```

The event handler calls the **InvokeOnUI()** for support multi-threading. This function takes care that the event request possibly coming from a background thread will be handled in the main UI thread. The **DoGroupAllAcquisitionStateChanged()** method updates the UI: for each state the text in the label is adapted and the command buttons are enabled/ disabled according the possibilities each state has. E.g. when the acquisition status is active you can only stop the acquisition, not start it.

The complete acquisition status and control handling is now event driven. Therefore we can omit the code we introduced in the **UIState** property.

For initialization purposes we still can add code in the SheetControl load:

```

private void SheetControl_Load(object sender, EventArgs e)
{
    AcqStatus.Text = "****";
    StartCmd.Enabled = false;
    StopCmd.Enabled = false;
}

```

Also the key handling in the button click routines can be removed, leaving:

```

private void StartCmd_Click(object sender, EventArgs e)
{
    // for demo purposes -> set some acquisition parameters here
    // 10 kSamples at 10 kHz with trigger position at 50%
    foreach (CtrlGroup myGroup in m_MyDemoSystem.Groups)
    {
        myGroup.SweepLength = 10000;
        myGroup.TriggerPosition = 50;
        myGroup.HighSamplingFrequency = 99990.0;
    }
    m_GroupAll.Run();
}

private void StopCmd_Click(object sender, EventArgs e)
{
    m_GroupAll.Stop();
}

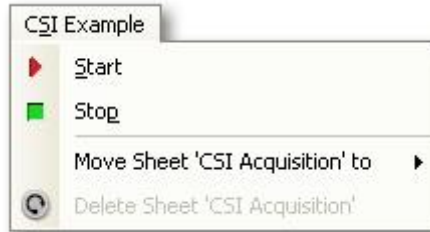
```

You can of course also add a button to trigger or set the system in pause mode.

Run the software and play with the commands on the sheet as well as the command buttons of the acquisition control in Perception.

4.1.4 Add menu commands

As discussed earlier you can also add menu commands to allow for an improved user interface and better keyboard support. Start with the implementation of the basic menu commands and add an image when available:



Once done two more things need to be done:

- Connect the commands to the correct functions
- Add code to make the behaviour of the menu commands identical to the behaviour of the buttons.

To connect the commands to the correct functions you need to modify the events of the **contextMenuStrip** we created.

For ease of reference name the menu commands "MenuStartCmd" and "MenuStopCmd". In the Properties dialog of the **contextMenuStrip** go to the **Events** section. In the menu select a command and modify the Click event:

- For the MenuStartCmd select "StartCmd_Click"
- For the MenuStopCmd select "StopCmd_Click"

Now add code in the event handler and the **SheetControl_Load** to control the behaviour of the menu commands, e.g.:

```

if (Running > 0 )
{
    AcqStatus.Text = "Active";
    StartCmd.Enabled = false;
    StopCmd.Enabled = true;
    MenuStartCmd.Enabled = false;
    MenuStopCmd.Enabled = true;
}

```

You can also add a toolbar with the same functionality to finalize this chapter. When you do so do not forget to include a call to "RebuildDynamicMenuRequested" or "ToolItemsUpdated" at the end of the event handler to make sure that the toolbar is updated correctly.

5 Hardware Settings

In the previous chapter we already have seen a small attempt to modify hardware settings. In this chapter we will demonstrate how to access hardware settings in general and how to read the capabilities of a specific channel and recorder. But first we will need to understand how hardware is organized in Perception.

5.1 Hardware organization

Data acquisition hardware within Perception is based on the concept of a **recorder**. A recorder consists of a number of acquisition channels that share the same basic recording parameters sample rate, sweep length and pre- and post-trigger length. Usually a single recorder is physically identical to a single acquisition card.

Multiple recorders can be placed in a single **mainframe**. The mainframe is the housing for the recorders, provides the power and includes the interface for the local area network. A mainframe has its own network address (IP address).

Within the Perception software recorders can be combined into logical **groups** for easy reference. Recorders within a group are not bound by mainframe.

Therefore the basic ways to access hardware is either through groups or through mainframes.

There can be one or more groups. The number of groups is defined by a **count**. You can use this count to index a specific group. All groups together are referenced by the **GroupAll** concept.

There can be one or more mainframes. The number of mainframes is defined by a count. You can use this count as an index to address a specific mainframe.

Each group and mainframe has one or more recorders. A recorder is part of both a group and a mainframe.

Recorders and channels also use a count to find out how many objects are available. Use these counts to index a specific recorder or channel.

The general method to access groups, recorders or channels is by using the **foreach** statement:

```
foreach (CtrlGroup myGroup in m_MyDemoSystem.Groups)
{
    myGroup.SweepLength = 1000;
    myGroup.TriggerPosition = 50;
    myGroup.HighSamplingFrequency = 99990.0;
    foreach (CtrlRecorder myRecorder in myGroup.Recorders)
    {
        foreach (CtrlChannel myChannel in myRecorder.Channels)
        {
            myChannel.InputCoupling = CtrlChannelInputCoupling.CtrlChannelIC_Current;
        }
    }
}
```

Here you see an actual example of accessing a specific channel setting. In this example the group concept is used. You could, however, also loop through the mainframes:


```

foreach (CtrlMainFrame myMainframe in m_MyDemoSystem.MainFrames)
{
    foreach (CtrlRecorder myRecorder in myMainframe.Recorders)
    {
        foreach (CtrlChannel myChannel in myRecorder.Channels)
        {
            myChannel.InputCoupling = CtrlChannelInputCoupling.CtrlChannelIC_Current;
        }
    }
}
    
```

5.2 Get and set parameters

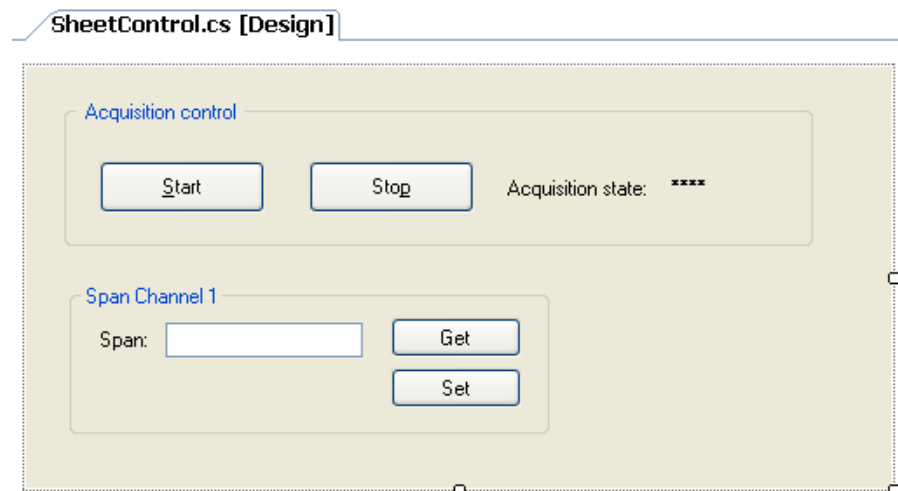
We will start with a simple example to show how to get and set the **span** of a **channel**. A channel is a member of a recorder and every recorder is a member of the **GroupAll**; therefore we can use the following line of code to get access to the first channel of the first recorder:

```
CtrlChannel myChannel = m_GroupAll.Recorders[1].Channels[1];
```

Attention: the recorder and channel collection indexes are 1 based.

5.3 User interface

The user interface is based on two list boxes with labels and grouped in a groupbox:



Put the following code behind the on-click events of the **Get** and **Set** buttons:

```

private void btnGetSpan_Click(object sender, EventArgs e)
{
    // Check if there are recorders available
    if (m_GroupAll.Recorders.Count < 1)
    {
        PerceptionMessageBox.Show(this, "No Channels connected", "CSI: Demo",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    // Check if the first recorder contains channels
    if (m_GroupAll.Recorders[1].Channels.Count < 1)
    {
    
```

```

        PerceptionMessageBox.Show(this, "No Channels connected",
            "CSI: Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
    return;
}

CtrlChannel myChannel = m_GroupAll.Recorders[1].Channels[1];
TextBoxSpan.Text = myChannel.Span.ToString();
}

private void btnSetSpan_Click(object sender, EventArgs e)
{
    // Check if there are recorders available
    if (m_GroupAll.Recorders.Count < 1)
    {
        PerceptionMessageBox.Show(this, "No Channels connected",
            "CSI: Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    // Check if the first recorder contains channels
    if (m_GroupAll.Recorders[1].Channels.Count < 1)
    {
        PerceptionMessageBox.Show(this, "No Channels connected",
            "CSI: Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    CtrlChannel myChannel = m_GroupAll.Recorders[1].Channels[1];

    double dSpan = FloatingPoint.ConvertStringToDouble(TextBoxSpan.Text);
    // Try to convert the text box string input into a double
    if (double.IsNaN(dSpan))
    {
        string cMsg = string.Format(
            "Error setting the Span of the channel: {0} to {1}",
            myChannel.Name, TextBoxSpan.Text);
        PerceptionMessageBox.Show(this, cMsg, "CSI: Demo",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    else
    {
        myChannel.Span = dSpan;
    }

    // Read back the Span and show it in the text box
    TextBoxSpan.Text = myChannel.Span.ToString();
}
}

```

5.4 Get and set parameters using capabilities

The Perception software is based on the concept that it has no knowledge of the hardware it therefore has to interrogate the hardware for its capabilities on a specific feature. Depending on the feature this can be either a list of possible settings, or minimum and maximum values including a step size.

In addition there are settings in hardware that go automatically to the nearest available value when a different value is entered. Therefore you should always verify these settings afterwards.

First we need to define when we can get hardware capabilities from a system. When no hardware is available we cannot interrogate settings. When recorders are added or removed we might need to take action. In order to do so we will "hook" onto the **RecorderAdded** (and **RecorderRemoved**) of the **GroupAll** event. I.e. when somewhere in the system a recorder is added or removed we can start interrogating its capabilities.

5.5 Initialize

Add the following line of code in the HookToGroupAll function of our program:

```
|| m_GroupAll.RecorderAdded += GroupAllRecorderAdded;
```

Add the following line of code in the UnHookFromGroupAll function of our program:

```
|| m_GroupAll.RecorderAdded -= GroupAllRecorderAdded;
```

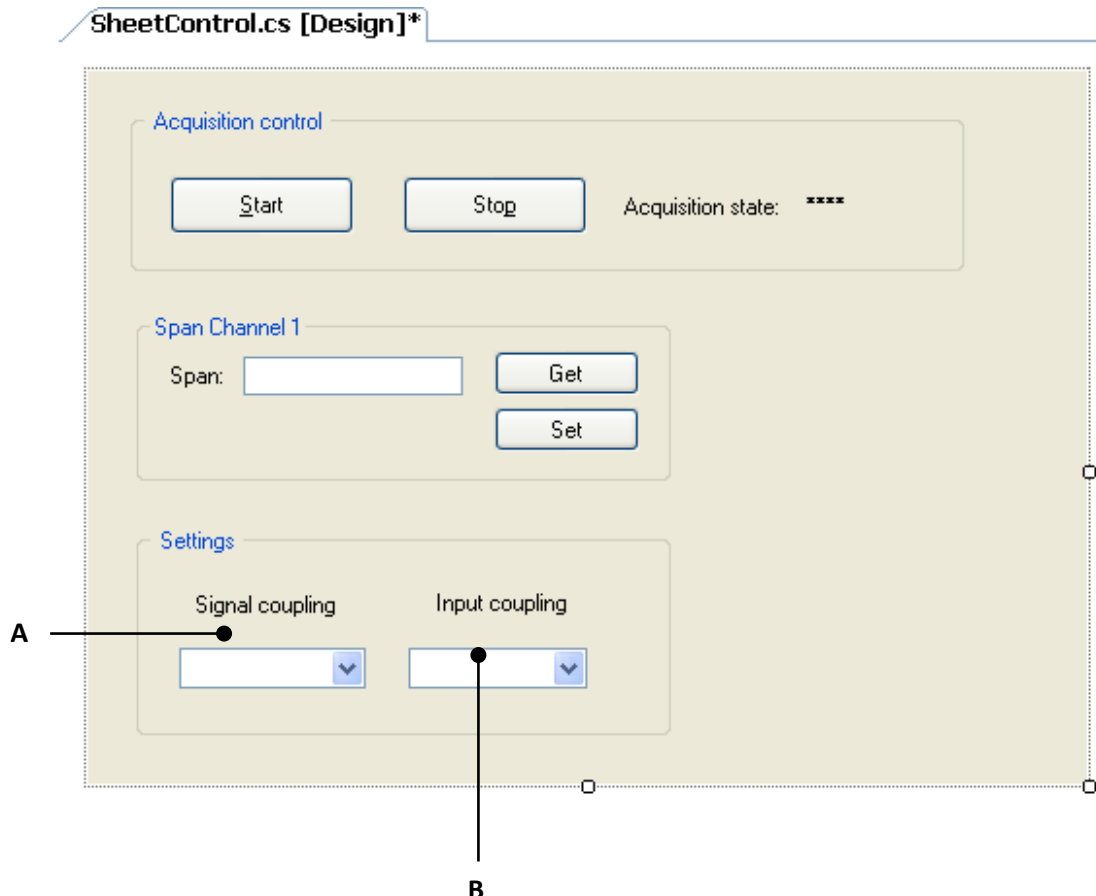
Add the following method to the program:

```
|| void GroupAllRecorderAdded(object sender, CtrlRecorder Recorder)
|| {
|| }
||
```

As an example we will create two drop down lists that allow us to get and set the signal and input coupling of the first channel of the first recorder. When done you can verify correct operation by looking at the settings sheet in Perception. In our example we will use a very straightforward approach. In real programming life you should be more object-oriented.

5.6 User interface with Capabilities

The user interface is based on two list boxes with labels and grouped in a groupbox:



- A. ComboBox for signal coupling: **listSignalCoupling**, with DropDownStyle set to DropDownList
- B. ComboBox for input coupling: **listInputCoupling**, with DropDownStyle set to DropDownList

5.7 Get capabilities

You can get the capabilities of a specific settings using the GetCapabilities procedure. This procedure requires as input the requested item and returns a structure of type Capabilities.

This structure has the following fields:

```
public struct Capabilities
{
    public CtrlEntryType EntryType;
    public object LowerBound;
    public CtrlSettingType SettingsType;
    public CtrlSettingUsage SettingUsage;
    public object StepSize;
    public object UpperBound;
    public Array ValueListItems;
    public Array ValueListItemsEnabled;
}
```

- **CtrlSettingsEntryType** Indicates the type of entry. E.g. CtrlSettingsEntry_List indicates a list.
- **LowerBound** Lowest value when the entry type is not a list, e.g. an **edit** or a **list edit**.
- **CtrlSettingsType** Defines the type of values for the upper and lower values, e.g. a CtrlSettingsType_I4 indicates a four-byte integer.
- **CtrlSettingsUsage** Are you allowed to modify it? E.g. CtrlSettingsUsage_Selectable indicates yes.
- **StepSize** Stepsize when entry type is a step edit.
- **UpperBound** Highest value when the entry type is not a list, e.g. an **edit** or a **list edit**.
- **ValueListItems** When entry type is a list, these are the possible 'values'.
- **ValueListItemsEnabled** For each list entry there is a value that indicates if a value is enabled or not, or only when acquisition is idle.

In the RecorderAdded event handler a possible scenario could be as follows:

```
243 void GroupAllRecorderAdded(object sender, CtrlRecorder Recorder)
244 {
    // Make user interface update thread safe by using InvokeOnUI()
245     this.InvokeOnUI(() => DoGroupAllRecorderAdded(sender, Recorder));
246 }

248 void DoGroupAllRecorderAdded(object sender, CtrlRecorder Recorder)
249 {
250     // this should be > 0, but you never know
251     if (m_GroupAll.Recorders.Count == 0)
252         return;
253
254     // now we have at least one recorder, but does it have channels?
255     if (m_GroupAll.Recorders[1].Channels.Count == 0)
```

```

256     return;
257
258     // did we already fill this list?
259     if (listSignalCoupling.Items.Count > 0)
260         return;
261
262     // Create variables
263     CtrlChannelSignalCoupling CCSC;
264     CtrlChannelInputCoupling CCIC;
265     string sResult;
266
267     // fetch signal coupling capabilities
268     Capabilities args;
269     m_GroupAll.Recorders[1].Channels[1].GetCapabilities(
270         CtrlChannelCapabilities.CtrlChannelCaps_SignalCoupling, out args);
271
272     // are we allowed to use this setting?
273     if (args.SettingUsage != CtrlSettingUsage.CtrlSettingUsage_Selectable)
274         return;
275
276     // is it a list?
277     if (args.EntryType == CtrlEntryType.CtrlEntryType_List)
278     {
279         // yes to all -> fetch values that we are allowed to use
280         listSignalCoupling.Enabled = true;
281         int vLB = args.ValueListItems.GetLowerBound(0);
282         int vUB = args.ValueListItems.GetUpperBound(0);
283         for (int nItem = vLB; nItem <= vUB; nItem++)
284         {
285             CCSC =
286                 (CtrlChannelSignalCoupling) args.ValueListItems.GetValue(nItem);
287
288             if (CtrlEntryEnabled.CtrlEntryEnabled_Never ==
289                 (CtrlEntryEnabled) args.ValueListItemsEnabled.GetValue(nItem))
290                 continue;
291
292             sResult = GetEnumSigConString(CCSC);
293
294             listSignalCoupling.Items.Add(Item);
295         }
296         // fetch the actual setting
297         CCSC = m_GroupAll.Recorders[1].Channels[1].SignalCoupling;
298         sResult = GetEnumSigConString(CCSC);
299         // show actual setting
300         listSignalCoupling.SelectedIndex =
301             listSignalCoupling.FindStringExact(sResult);
302     }
303
304     // continue with second list if not already done
305     if (listInputCoupling.Items.Count == 0)
306     {
307         // fetch input coupling capabilities
308         m_GroupAll.Recorders[1].Channels[1].GetCapabilities(
309             CtrlChannelCapabilities.CtrlChannelCaps_InputCoupling, out args);
310
311         // are we allowed to use this setting?
312         if (args.SettingUsage != CtrlSettingUsage.CtrlSettingUsage_Selectable)
313             return;
314
315         // is it a list?
316         if (args.EntryType == CtrlEntryType.CtrlEntryType_List)
317         {
318             // yes to all -> fetch values that we are allowed to use
319             listInputCoupling.Enabled = true;

```

```

318     int vLB = args.ValueListItems.GetLowerBound(0);
319     int vUB = args.ValueListItems.GetUpperBound(0);
320     for (int nItem = vLB; nItem <= vUB; nItem++)
321     {
322         CCIC =
            (CtrlChannelInputCoupling) args.ValueListItems.GetValue(nItem);
323
324         if (CtrlEntryEnabled.CtrlEntryEnabled_Never ==
            (CtrlEntryEnabled) args.ValueListItemsEnabled.GetValue(nItem))
325             continue;
326
327         sResult = GetEnumInpConString(CCIC);
328
329         listInputCoupling.Items.Add(Item);
330     }
331     // fetch the actual setting
332     CCIC = m_GroupAll.Recorders[1].Channels[1].InputCoupling;
333     sResult = GetEnumInpConString(CCIC);
334     // show actual setting
335     listInputCoupling.SelectedIndex =
        listInputCoupling.FindStringExact(sResult);
336     }
337 }
338 }

```

In the above example we fill the signal and input coupling lists. The code is not optimized for demonstration purposes.

In **lines 243 - 246** we do our multi-threading support by calling the **InvokeOnUI()** function

The actual event handler starts with some tests: are recorders added (**line 250 - 256**), if so, are there any channels available in the first recorder (**line 255 - 256**)?

Since this event probably is called multiple times (each recorder added fires this event), we must check one way or another if we already have filled the list or not. Here a test is done to verify if the list is filled in **line 259 - 260**.

The **lines 263 - 269** are used to declare and initialize variables we need in the rest of the code.

In **line 269** we make the actual call to the procedure that provides us with information about the signal coupling capabilities of the selected channel of the selected recorder.

Now we need to investigate these settings: **line 273 - 274** verifies if we are allowed to use this setting or not.

We continue if it is a list (**line 277**).

There are two arrays provided: **ValueListItems**, a list of values and **ValueListItemsEnabled**, a list of corresponding settings. For each entry in ValueListItems there is an entry in ValueListItemsEnabled that defines the behaviour of the value in ValueListItems.

In **line 283 - 295** we step through the ValueListItems and the ValueListItemsEnabled. For each ValueListItems item that we may use we add an entry in the signal coupling list. The entry in the list is a string. Therefore we make use of a procedure that converts the enumeration into an actual string.

Visual studio can provide a helping hand in the creation of such a procedure. The result can look like this:

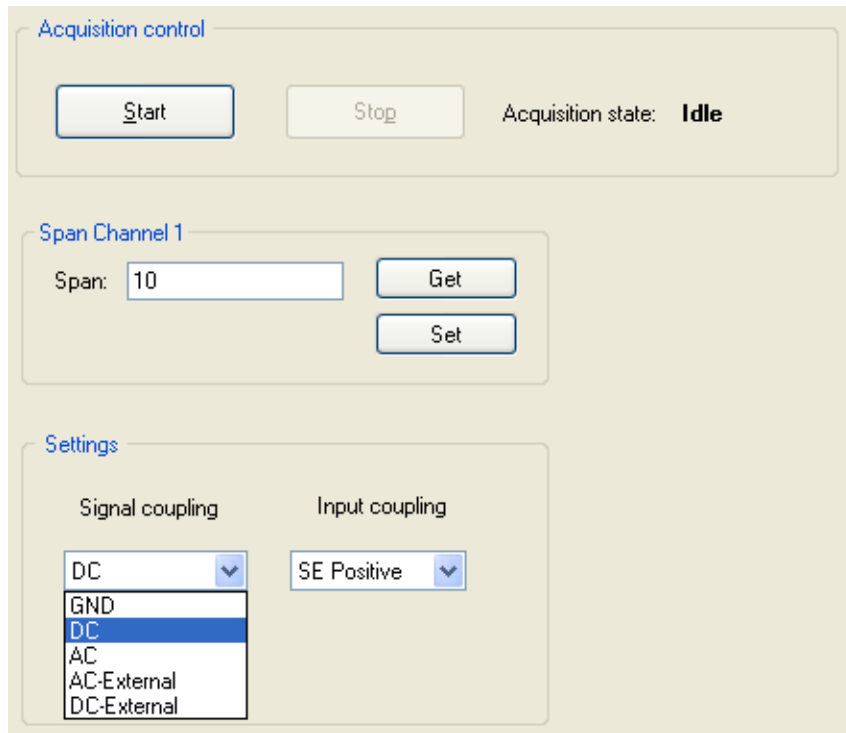
```

504 private string GetEnumSigConString(CtrlChannelSignalCoupling sc)
505 {
506     switch (sc)
507     {
508         case CtrlChannelSignalCoupling.CtrlChannelSC_AC:
509             return "AC";
510         case CtrlChannelSignalCoupling.CtrlChannelSC_AC_ExternalProbe:
511             return "AC-External";
512         case CtrlChannelSignalCoupling.CtrlChannelSC_AC_Frequency:
513             return "AC-Freq";
514         case CtrlChannelSignalCoupling.CtrlChannelSC_AC_RMS:
515             return "AC-RMS";
516         case CtrlChannelSignalCoupling.CtrlChannelSC_AC_TrueRMS:
517             return "AC-TrueRMS";
518         case CtrlChannelSignalCoupling.CtrlChannelSC_DC:
519             return "DC";
520         case CtrlChannelSignalCoupling.CtrlChannelSC_DC_ExternalProbe:
521             return "DC-External";
522         case CtrlChannelSignalCoupling.CtrlChannelSC_DC_Frequency:
523             return "DC-Freq";
524         case CtrlChannelSignalCoupling.CtrlChannelSC_DC_RMS:
525             return "DC-RMS";
526         case CtrlChannelSignalCoupling.CtrlChannelSC_DC_TrueRMS:
527             return "DC-TrueRMS";
528         case CtrlChannelSignalCoupling.CtrlChannelSC_GND:
529             return "GND";
530         default:
531             return "----";
532     }
533     return "****";
534 }

```

When the list is filled, the last thing we need to do is to find out the actual setting and show it. To do this we get the signal coupling property, convert it again to a string and use this as a pointer to set the index of our list. Below is an example of the user interface when the code is executed.

In lines **305 – 335** the input coupling list will be filed using similar steps.



The screenshot displays the CSI software interface with three main sections:

- Acquisition control:** Contains a **Start** button, a **Stop** button, and the text "Acquisition state: **Idle**".
- Span Channel 1:** Features a text input field labeled "Span:" containing the value "10", a **Get** button, and a **Set** button.
- Settings:** Contains two dropdown menus:
 - Signal coupling:** A dropdown menu with a list of options: DC (selected), GND, AC, AC-External, and DC-External.
 - Input coupling:** A dropdown menu with the option "SE Positive".

Items are added to the list when the corresponding setting is not equal to `CtrlEntryEnabled_Never` (line 324). However, some items can only be enabled when acquisition is idle. This means that they are in the list by default but should be 'disabled' when acquisition is not idle. We will discuss this issue in the next section where we will show you how to send a setting to the hardware.

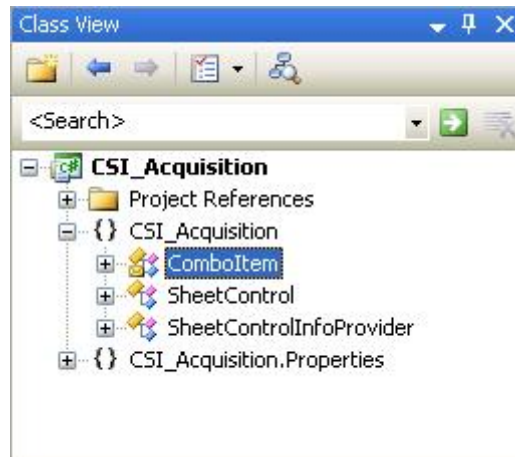
5.8 Modify a hardware setting

As mentioned in the previous section, some hardware settings are only allowed when acquisition is idle. However, after we have used a `GetCapabilities` function, this information is lost, unless we save this information.

One way to save this information is to create a class that contains all this information. We will do this for our signal coupling and input coupling items.

5.9 Create a class

To create a class in Visual Studio select Project > Add Class ... and add a class named ComboItem.



Go to the **class view** to see classes.

Initially this class should have the following code:

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Perception.ILO.Engine;
5
6  namespace CSI_Acquisition
7  {
8      class ComboItem
9      {
10         public string m_Text;
11         public int m_Value;
12         public CtrlEntryEnabled m_Enabled;
13
14         public ComboItem(string Text, int Value, CtrlEntryEnabled
15                             Enabled)
16         {
17             m_Text = Text;
18             m_Value = Value;
19             m_Enabled = Enabled;
20         }
21         public override string ToString()
22         {
23             return m_Text;
24         }
25     }
    
```

This code creates three public class members: one for the string that we use in our list, one for the enumerator and one for the 'enabled'-value.

The constructor fills these members with the values it gets.

The **ToString()** method returns the string.

Now we also need to adjust the code from the previous section a little bit. Once an item was found that could be added to the list we had the following code:

```
sResult = GetEnumSigConString(CCSC);
listSignalCoupling.Items.Add(sResult);
```

Now we also have to create the new object:

```
sResult = GetEnumInpConString(CCIC);

ComboItem Item = new ComboItem(sResult, (int) CCIC,
(CtrlEntryEnabled) args.ValueListItemsEnabled.GetValue (nItem));

listInputCoupling.Items.Add(Item);
```

The line where we show the actual setting should read:

```
listSignalCoupling.SelectedIndex = listSignalCoupling.
FindStringExact(sResult);
```

We need to do this because an object is added to the list, not a text string. So far operation of our drop-down lists will remain the same.

We repeat this also for the second list the signal input coupling:

```
sResult = GetEnumSigConString(CCSC);
listSignalCoupling.Items.Add(sResult);
```

Now we also have to create the new object:

```
sResult = GetEnumInpConString(CCIC);

ComboItem Item = new ComboItem(sResult, (int) CCIC,
(CtrlEntryEnabled) args.ValueListItemsEnabled.GetValue (nItem));

listInputCoupling.Items.Add(Item);
```

5.9.1 Modify the setting

Modification of the actual hardware setting is simple. However, we need to keep in mind that we cannot change every setting always. Therefore additional code is required. We will react on the **SelectionChangeCommitted** event of the list. This event is generated when *"an item is chosen from the drop-down list and the drop-down list is closed"*.

Here we need to verify if the selected option is allowed and, if so, modify the hardware setting.

Initially we will start with the following code:

```

617 private void listSignalCoupling_SelectionChangeCommitted(
        object sender, EventArgs e)
618 {
619     ComboItem Item = listSignalCoupling.SelectedItem as ComboItem;
620
621     if (Item.m_Enabled == CtrlEntryEnabled.CtrlEntryEnabled_WhenIdle)
622     {
623         // Check acquisition status
624         int Live;
625         int Pause;
626         int HoldNext;
627         int HoldLast;
628         int Idle;
629
630         m_GroupAll.GetAcquisitionState(out Live, out HoldNext,
            out HoldLast, out Pause, out Idle);
631         if (Idle != m_GroupAll.Recorders.Count)
632         {
633             listSignalCoupling.SelectedText = GetEnumSigConString
                (m_GroupAll.Recorders[1].Channels[1].SignalCoupling);
634             return;
635         }
636     }
637     m_GroupAll.Recorders[1].Channels[1].SignalCoupling =
        (CtrlChannelSignalCoupling) Item.m_Value;
638 }

```

Since we are working with ComboItem objects we first need to define a new object for this routine and at the same time copy the selected object values into it (line 302).

In line 621 verify the enabled setting. If this setting is enabled when idle we need to test, otherwise it is ok

Note: *only items that are enabled are in the list*

Check the acquisition status. When the acquisition status is not idle replace the selected item text with the text that represents the actual current situation, do nothing and return.

Otherwise send the new setting.

This should do the job.

When you are using the Genesis Firmware Simulator, or depending on your actual hardware, you might see another phenomenon:

Depending on the acquisition state the capabilities displayed in the settings sheet differ from the capabilities we have seen so far.

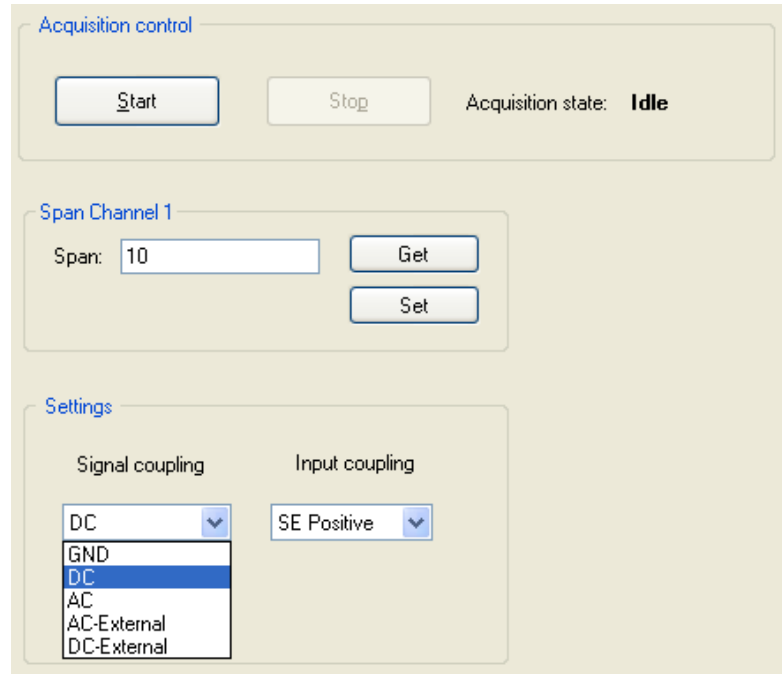
This is because some capabilities are listed differently when the system is idle or not.

To cope with this issue we should have written our code better structured to make it easier to respond to various events, e.g. the **m_GroupAll.Recorders[1].CapabilitiesChanged()** event.

For the time being we will solve this issue with a 'trick': when our list drops down, we clear the list and fill it again. To fill it we call the RecorderAdded event handler, because there is where our code is:

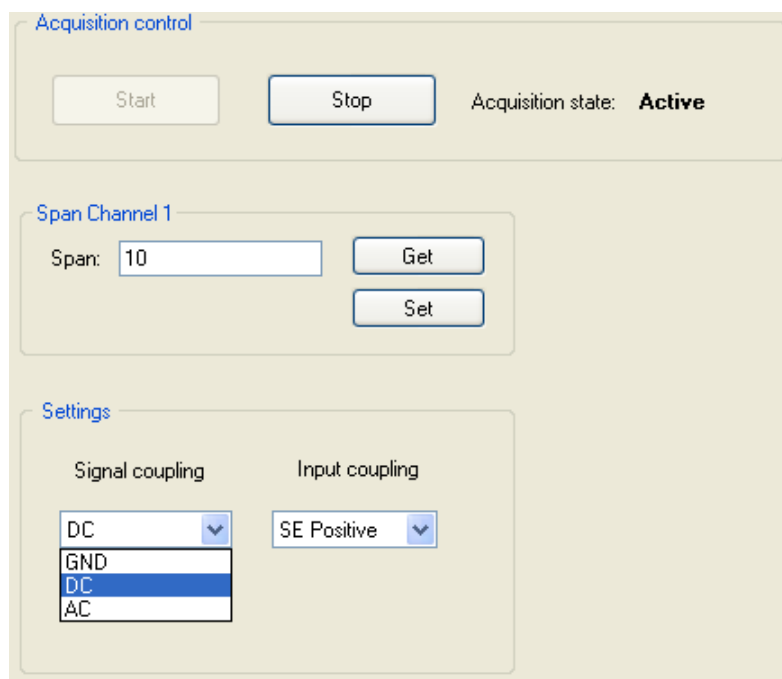
```
private void listSignalCoupling_DropDown(object sender, EventArgs e)
{
    listSignalCoupling.Items.Clear();
    m_GroupAll_RecorderAdded(null);
}
}
```

When using the Genesis Firmware Simulator you can see these effects. First make sure you are idle and display the list:



Here you will see 5 items, the last two are only available when the system is idle.

Now start an acquisition.



And display the list: only the top 3 items are available.

Now there is only one thing remaining: full synchronization between our sheet and Perception. When we modify a setting, this is reflected in the Perception software. However, when Perception modifies a setting, this is not reflected in our sheet. So we need another event handler.

5.10 Synchronization

We need to get notified when a setting has changed and react on this event. For this we need to create an object for a channel that generates events when something has changed. As a rule of thumb: create an object for each 'unit' that you want to modify. E.g. if you want to modify channel settings, create a channel object. This object then has properties, methods and events that we can use.

Example: create a channel member variable in the Members region:

```
#region -> MyMembers

private CtrlAcquisitionSystem m_MyDemoSystem = null;
private CtrlGroup m_GroupAll = null;
private CtrlChannel m_Channel = null;

#endregion
```

Before we can use this member variable we need to initialize it. This can be done the first time we access the channel setting: in the **RecorderAdded** event handler, just before we fetch the signal coupling capabilities.

```
// Connect the object and hook it to the correct event
if (m_Channel == null)
{
    m_Channel = m_GroupAll.Recorders[1].Channels[1];
    m_Channel.ScCouplingSettingsChanged +=
        m_Channel_ScCouplingSettingsChanged;
}
```

First we check if the object is already initialized, if not it is connected to the first channel of the first recorder and a corresponding event handler is defined.

The event handler itself can have the following code:

```

370 void DoChannelScCouplingSettingsChanged(object Sender,
CtrlCouplingSettingItem Mask, CtrlChannelInputCoupling InCoupling,
CtrlChannelSignalCoupling SigCoupling, double MeasuringPeriod, double
371 ReferenceLevel, double Impedance, double InputCapacityLow, double
372 InputCapacityHigh)
373 {
374 this.InvokeOnUI(() => DoChannelScCouplingChanged(sender, Mask, InCoupling,
SigCoupling, MeasuringPeriod, ReferenceLevel, Impedance, InputCapacityLow,
InputCapacityHigh));
}

375 void DoChannelScCouplingSettingsChanged(object Sender,
376 CtrlCouplingSettingItem Mask, CtrlChannelInputCoupling InCoupling,
377 CtrlChannelSignalCoupling SigCoupling, double MeasuringPeriod, double
378 ReferenceLevel, double Impedance, double InputCapacityLow, double
379 InputCapacityHigh)
{
380 // event handler. we only do the signal coupling changed here
381 // although we could also do the input coupling here
382 if (Mask != CtrlCouplingSettingItem.CtrlCouplingSettingItem_
383 SignalCoupling)
384 return;
385
386 string sResult = GetEnumSigConString(SigCoupling);
387 int scIndex = listSignalCoupling.FindStringExact(sResult);
388 if (scIndex < 0)
389 {
390 // item could not be found -> list is incorrect
391 // so rebuild and reset index
392 listSignalCoupling.Items.Clear();
393 m_GroupAll_RecorderAdded(null);
scIndex = listSignalCoupling.FindStringExact(sResult);
}
listSignalCoupling.SelectedIndex = scIndex;
}

```

As a starter: Use the InvokeOnUI() to for the **multi-threading** support.

After this we check which parameter fired this changed event. To keep it simple we only test for the signal coupling: when it is not the signal coupling we return (**line 382**).

As we have done earlier we fetch the text string that corresponds to the enumeration and find the index in the list that corresponds to this text.

Now we stumble into a typical behaviour. As mentioned earlier, capabilities can change depending on the acquisition state. This is what we try to catch in lines **384 - 391**: if we cannot find the correct string, the index returns -1 and therefore we need to rebuild the list with a trick as we have done before. Should we have dealt with this situation better in the first place, then this would have not been necessary.

As an example remove the test and proceed as follows with the Genesis Firmware Simulator 'connected':

- Make sure that the system is idle.
- Set the signal coupling to 'external'
- **Start** an acquisition.

- The drop-down list will now only show the two 'external' options.
- **Stop** the acquisition and switch to **AC**, **DC**, or **GND** in the settings sheet.
- The text entry field in 'our' list will be empty.
- Drop down the list and the list will be initialized by the drop-down event handler of the combo-box.

If you have the multiple workbook option in Perception and a dual-monitor system you can do your testing more easily: move the test sheet to a new workbook and place this on the second monitor. Now you can have both sheets active: the settings sheet and the test sheet and verify the correct interaction.

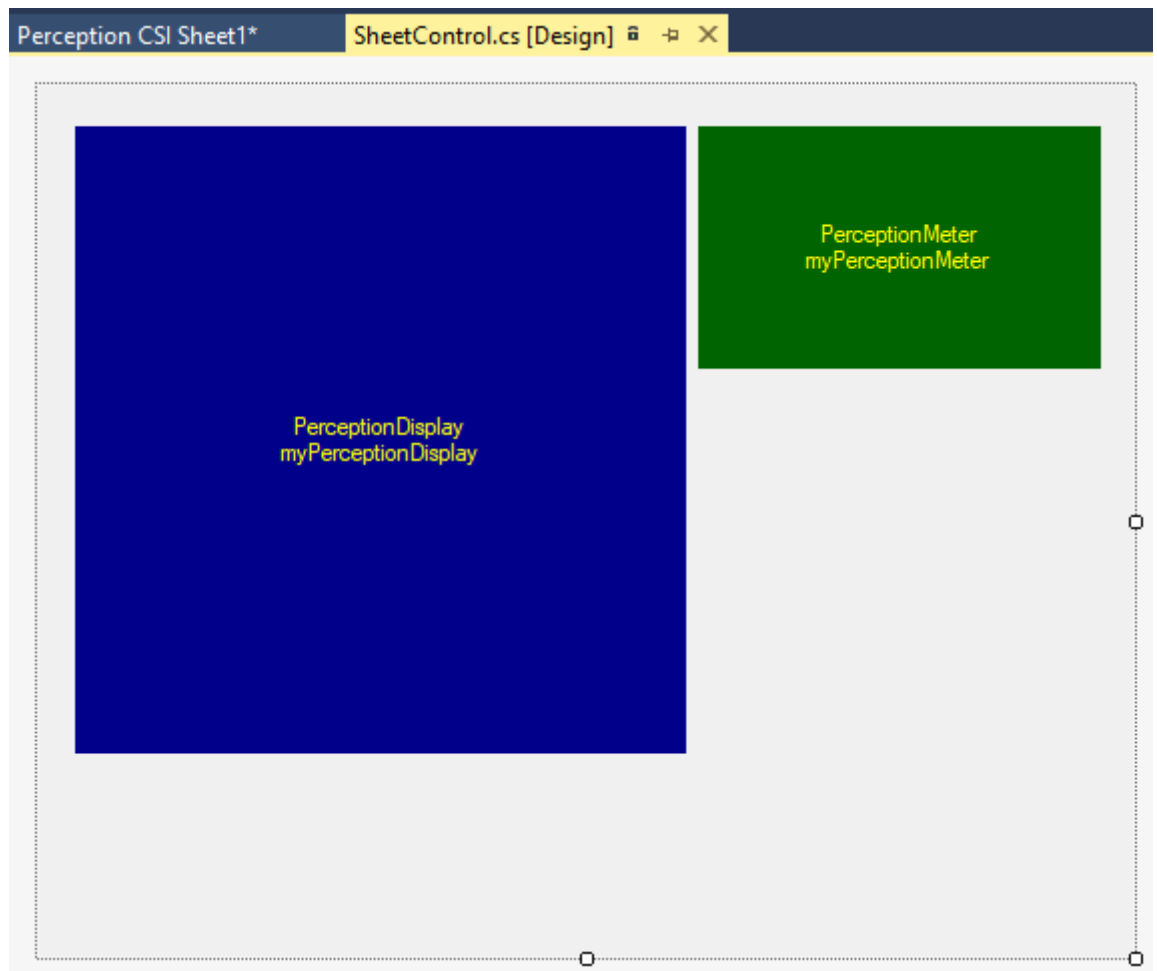
6 Data Visualization

Data acquisition is all about data: the acquisition, display, analysis, reporting and archiving of data. So far we have been occupied with sheet interfacing, acquisition control and modifying settings. So now it is time to do some visualization of recorded data and basic measurements within that data. Using the Perception CSI has the advantage that you have access to a variety of functions that you do not need to program yourself. One of them is the data display.

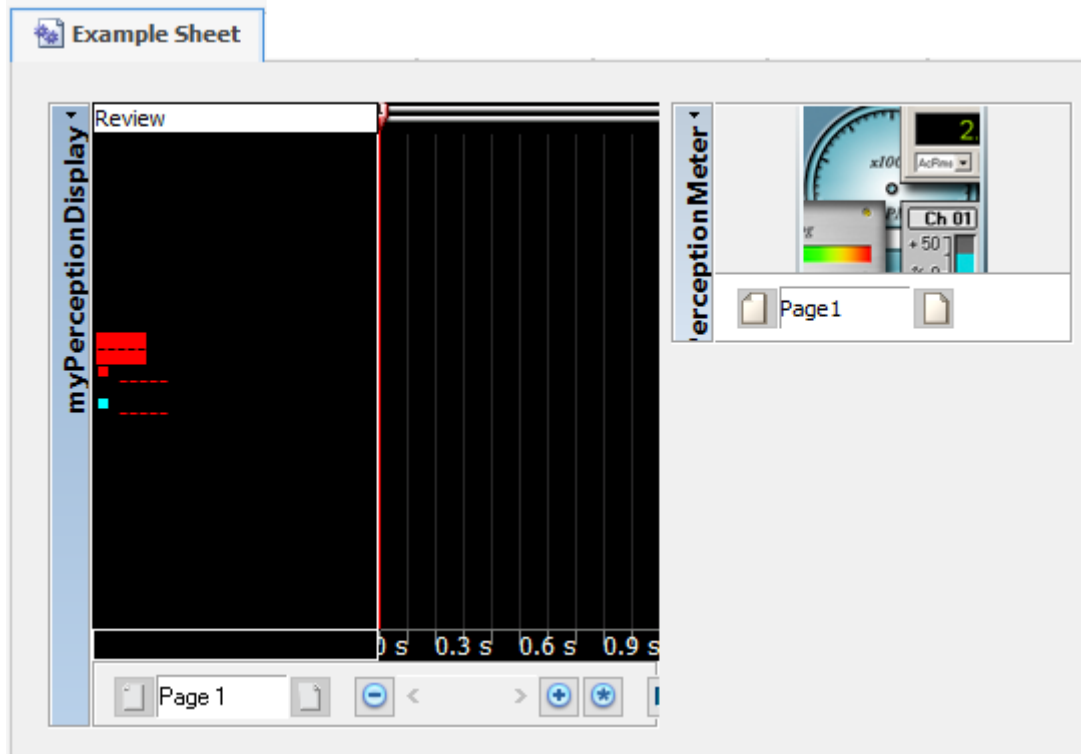
Note that when you use an "off-the-shelf" Perception component you also get all the additional stuff for free. E.g. the display also gives you zooming, cursors, measurements, drag-and-drop, etc.

6.1 Data display

The data display is part of a range of standard Perception components that can be added to your project. In the past (32-bit version of Perception) you could directly use the Perception Display component in the Visual Studio designer of your CSI project. However for the 64-bit version of Perception this is not possible anymore because Visual Studio is 32 bit and does not support 64-bit native code components in design mode. The Perception Display and Meter core is build using native code. To solve this problem there are two wrapper components added to Perception called **PerceptionDisplay** and **PerceptionMeter**, these components are in Perception.CSI.Support.dll. They do not show the meter or display in the visual studio designer but show a blue or green area with a yellow text like you see in the picture below.



During runtime this looks like:



To add the Perception components

1. Go to the **SheetControl** Design layout.
2. In the Toolbox select one of the tab headers and do a right mouse click.
3. In the context menu that comes up select **Add Tab** and give it a relevant name like "Perception Components".
4. With this tab selected do a right mouse click.
5. In the context menu that come up select **Choose Items ...**
6. In the dialog that comes up select **Browse...**
7. Navigate to the Perception folder. Typically: **C:\Program Files\HBM\Perception**.
8. Select the file **Perception.Components.dll** and click **Open**.
9. Click **OK** in the Choose Toolbox Items dialog.

Now the components are added to the toolbox.

To add the Perception wrapper components

1. Go to the **SheetControl** Design layout.
2. In the Toolbox select one of the tab headers and do a right mouse click.
3. In the context menu that comes up select **Add Tab** and give it a relevant name like "Perception CSI Support".

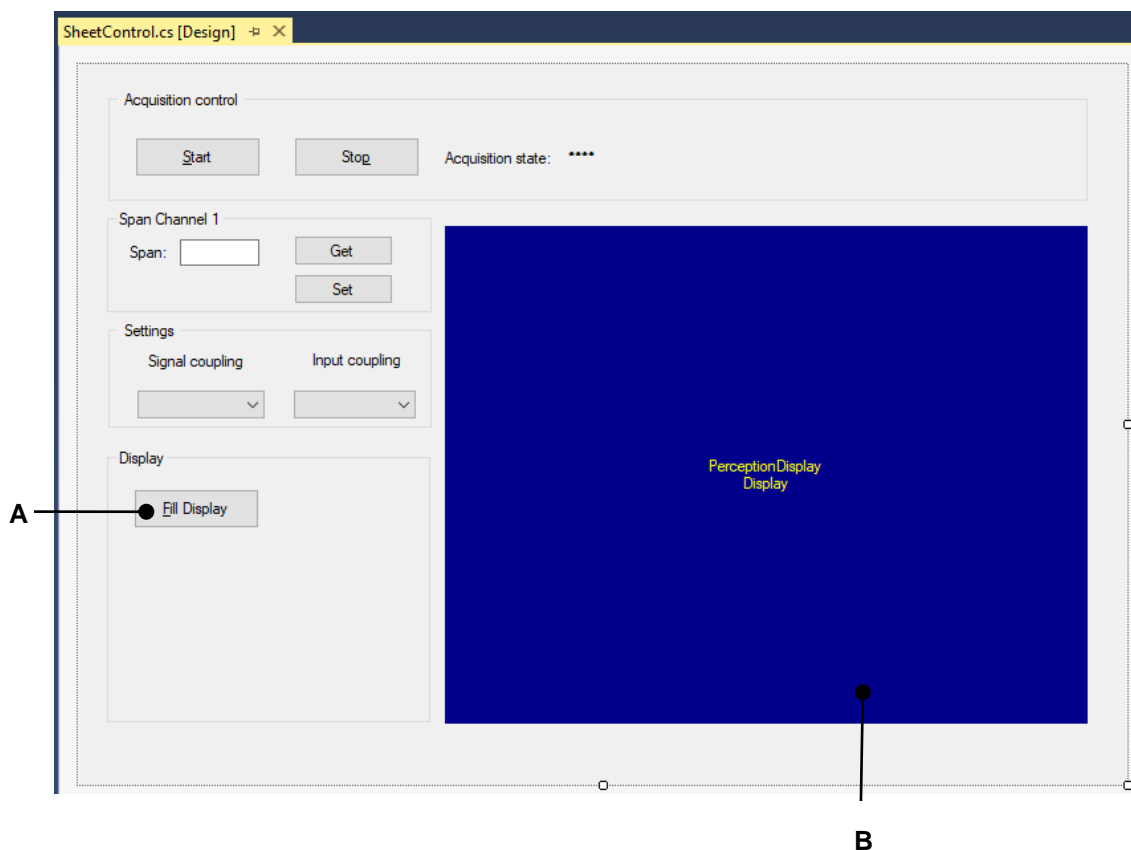
4. With this tab selected do a right mouse click.
5. In the context menu that come up select **Choose Items ...**
6. In the dialog that comes up select **Browse...**
7. Navigate to the Perception folder. Typically: **C:\Program Files\HBM\Perception.**
8. Select the file **Perception.CSI.Support.dll** and click **Open.**
9. Click **OK** in the Choose Toolbox Items dialog.

Now the components **PerceptionDisplay** and **PerceptionMeter** are added to the toolbox.

6.1.1 User interface

The user interface for this example is simple: a display and a command button. The command button is used to 'fill' the display with data sources.

When combined with our previous example the layout could look like this:



- A. Command button **FillCmd**
- B. Placeholder for the data display **display1**

6.1.2 The code

To gain access to the data sources for display we need to have access to the data manager. The data manager is the central part of the software that manages all data: waveforms

(analogue, digital), numerical values, system variables, etc. So we will start with the obvious:

```
#region -> MyMembers

private CtrlAcquisitionSystem m_MyDemoSystem = null;
private CtrlGroup m_GroupAll = null;
private CtrlChannel m_Channel = null;
private DataManager m_DataManager = null;

#endregion
```

All the required code is placed within the FillCmd_Click routine as follows:

```
712 private void FillCmd_Click(object sender, EventArgs e)
713 {
714     if (m_DataManager == null)
715     {
716         m_DataManager = new DataManager();
717     }
718     // clear display
719     for (int j = display1.pDisplay.TimeDisplay.CtlLayout.Pages.Count;
720         j > 0; j--)
721     {
722         display1.pDisplay.TimeDisplay.CtlLayout.Pages[j].Delete();
723     }
724     display1.pDisplay.TimeDisplay.Pages.InitPages();
725
726     int TraceCount = 1;
727     foreach (PoolEntry PE in m_DataManager.PoolEntries)
728     {
729         if (PE.DataSource == null)
730             continue;
731         if (PE.DataSource.DataType !=
732             RecordingInterface.DataSourceDataType.
733             DataSourceDataType_AnalogWaveform)
734             continue;
735         string[] aPoolEntry = new string[] { PE.Name };
736         if (TraceCount > 4)
737         {
738             display1.pDisplay.TimeDisplay.AddPage().Activate();
739             TraceCount = 1;
740         }
741         else
742         {
743             if (TraceCount != 1)
744             {
745                 display1.pDisplay.TimeDisplay.CtlLayout.Pages.
746                     ActivePage.Panes.AddPane().Activate();
747             }
748         }
749         display1.pDisplay.AddDataSources(aPoolEntry);
750         TraceCount++;
751     }
752     display1.pDisplay.TimeDisplay.CtlLayout.Pages[1].Activate();
753 }
```

We start with some housekeeping:

- **714 - 717:** Create a new data manager object.
- **719 - 722:** Delete all pages that exist.
- **723:** Create a single new page and make it the active page.

The main loop is used to step through all available data sources (pool entries). When an analogue waveform is found this waveform is added to the display. In this example we use a maximum of 4 traces per page, each trace in its own pane.

- **728 - 731:** Skip empty data sources and all data sources that are not analogue waveforms.
- **732:** A string array is created (with only one entry) that contains the identifier of an analogue waveform.
- **733 - 736:** Create a new page when number of traces on this page exceeds 4 and activate it.
- **738 - 743:** Add a new pane for a trace when it is not the first trace of a page and activate it.
- **745 - 746:** Add the data source to the display on the active page in the active pane. Increment the trace counter.

When all is said and done activate the first page to show it.

Now run the code.



Note: that the display is fully integrated in the Perception software: you can zoom and pan, modify the properties, drag and drop data sources from the navigator into the display, do cursor measurements, etc.

For better performance when looking for pool entries we recommend the use of the **PoolEntries** function **GetNames()**. This function returns an array of pool entry strings. As an input parameter you enter a string to define where to look in the pool, you can use the "*" character as a wildcard. The second parameter defines the type of the pool entry you want. The last parameter gives you back an array containing the found pool entries.

Code example:

```

Array aNames;
m_DataManager.PoolEntries.GetNames("Active.*",
    PoolEntryType.PoolEntryType_Waveform, out aNames);

foreach (string cName in aNames)
{
    string[] aPoolEntry = new string[] { cName };
    if (TraceCount > 4)
    {
        display1.pDisplay.TimeDisplay.AddPage().Activate();
        TraceCount = 1;
    }
    else
    {
        if (TraceCount != 1)
        {
            display1.pDisplay.TimeDisplay.CtlLayout.
                Pages.ActivePage.Panes.AddPane().Activate();
        }
    }
    display1.pDisplay.AddDataSources(aPoolEntry);
    TraceCount++;
}

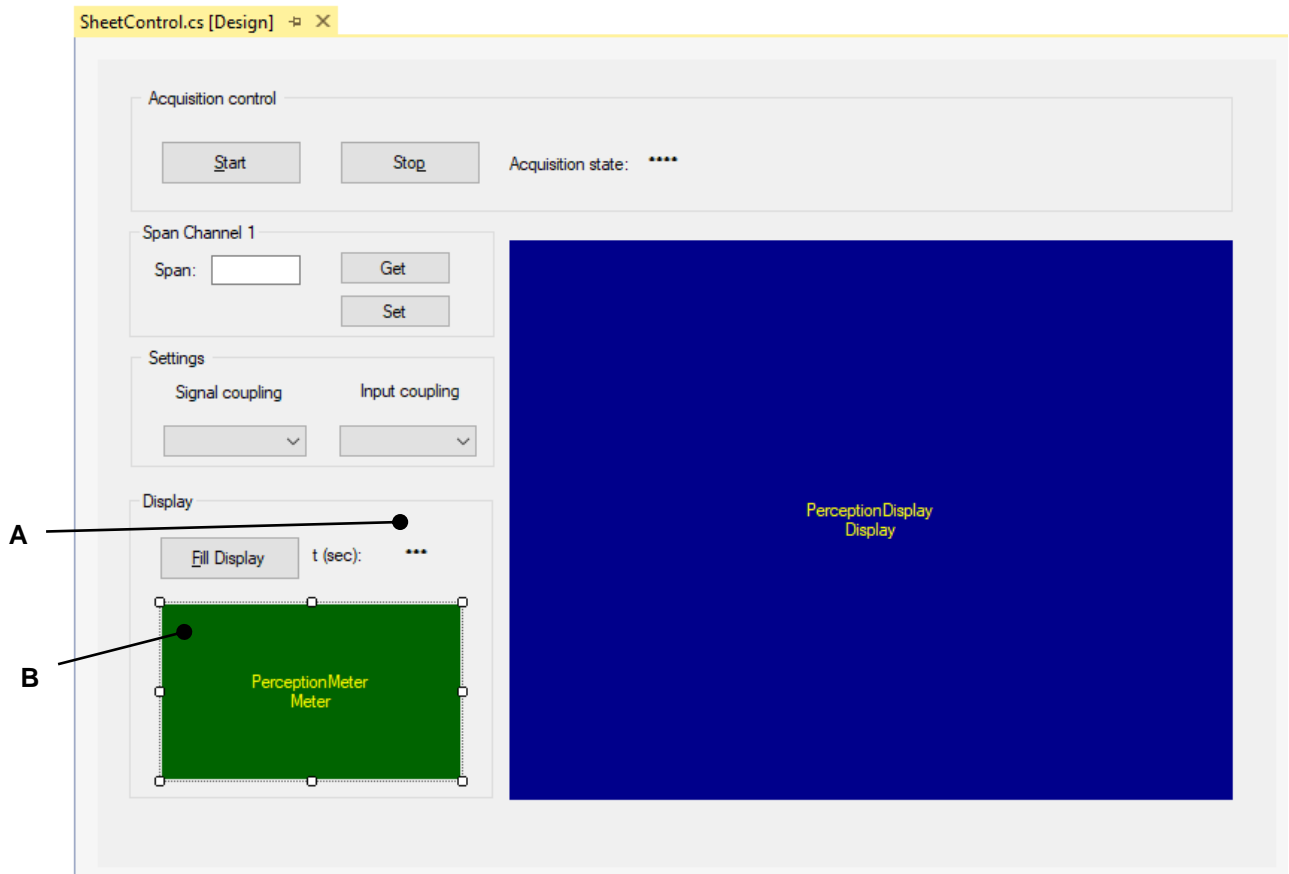
```

6.2 Meter

As standard the 'digital' meter is accessible. You can add a numerical meter the same way as a display. In the next example we will add a meter and 'connect' it to the measurement cursor Y-value. At the same time we will 'hook' to the event when the measurement cursor value changes and modify a text label to show the corresponding X-value.

6.2.1 User interface

The user interface consists of a meter and a text label. We need to make a little bit more room to make everything fit.



- A. Text label `XValueLbl`
- B. Placeholder for meter `meter1`

6.2.2 The code

As a starter add a new member:

```
private PoolEntry m_ActiveCursor = null;
```

Modify the `SheetControl_Load` to include some initialization. By now it could look like this:

```

561 private void SheetControl_Load(object sender, EventArgs e)
562 {
563     AcqStatus.Text = "****";
564     StartCmd.Enabled = false;
565     StopCmd.Enabled = false;
566     FillCmd.Enabled = false;
567     MenuStartCmd.Enabled = false; ToolStartCmd.Enabled = false;
568     MenuStopCmd.Enabled = false; ToolStopCmd.Enabled = false;
569     listSignalCoupling.Enabled = false;
570     listInputCoupling.Enabled = false;
571     display1.UserName = "CSIDisplay";
572     // set meter name ans connect meter to the correct pool entry
573     meter1.UserName = "CSIMeter";
574     string[] aPoolEntry = new string[] { "Display.CSIDisplay.
ActiveCursor.YValue" };
575     meter1.pMeter.LDSMeter.AddDataSources(aPoolEntry);
576 }

```

The new information starts at line 571.

In **lines 571** and **572** the name of the display and the meter are set. It is imperative that you use unique names throughout the Perception application, otherwise the navigators cannot make a distinction and use the first available object only.

In **lines 573** and **574** the data source is 'connected' to the display in the same way as you would connect to a display.

Again we want to connect to an event. Therefore we need to create an event handler. At the same time move some code to a better location.

We will start using the constructor section:

```

57 public SheetControl()
58 {
59     InitializeComponent();
60
61     if (m_DataManager == null)
62     {
63         m_DataManager = new DataManager();
64     }
65
66     if (m_DataManager != null)
67     {
68         m_ActiveCursor = m_DataManager.PoolEntries[
69             "Display.CSIDisplay.ActiveCursor.XPosition"];
70
71     }
72     HookToActiveCursor();
73     Disposed += SheetDisposed;
74 }
    
```

Line 61 - 64 is the moved code.

In **line 66 - 69** a new member is created: the x-position of the active cursor of our named display.

In **line 71** hook to the active cursor changed event.

Note that you need to create an event handler for each object that you want to use this way.

Also note that a pool entry with a name (line 68) always returns an object. This allows us to hook to events even before the actual variable is initialized.

```

private void HookToActiveCursor()
{
    UnHookFromActiveCursor();
    if (m_ActiveCursor != null)
    {
        m_ActiveCursor.DataChanged += ActiveCursor_DataChanged;
    }
}
    
```

```
private void UnHookFromActiveCursor()  
{  
    if (m_ActiveCursor != null)  
    {  
        m_ActiveCursor.DataChanged -= ActiveCursor_DataChanged;  
    }  
}
```

Our last action for this example is to create the corresponding event handler:

```
796 void ActiveCursor_DataChanged()  
797 {  
798     this.InvokeOnUI(()=>  
799     {  
800         if (m_ActiveCursor != null)  
801             double dxVal = (double) m_ActiveCursor.DataSource.Value;  
802             XValueLbl.Text = dxVal.ToString("##0.000");  
803         }  
804     }  
805 }
```

Of course we start with the `InvokeOnUI()` to make it possible to update the UI from a possible call coming from a sub thread

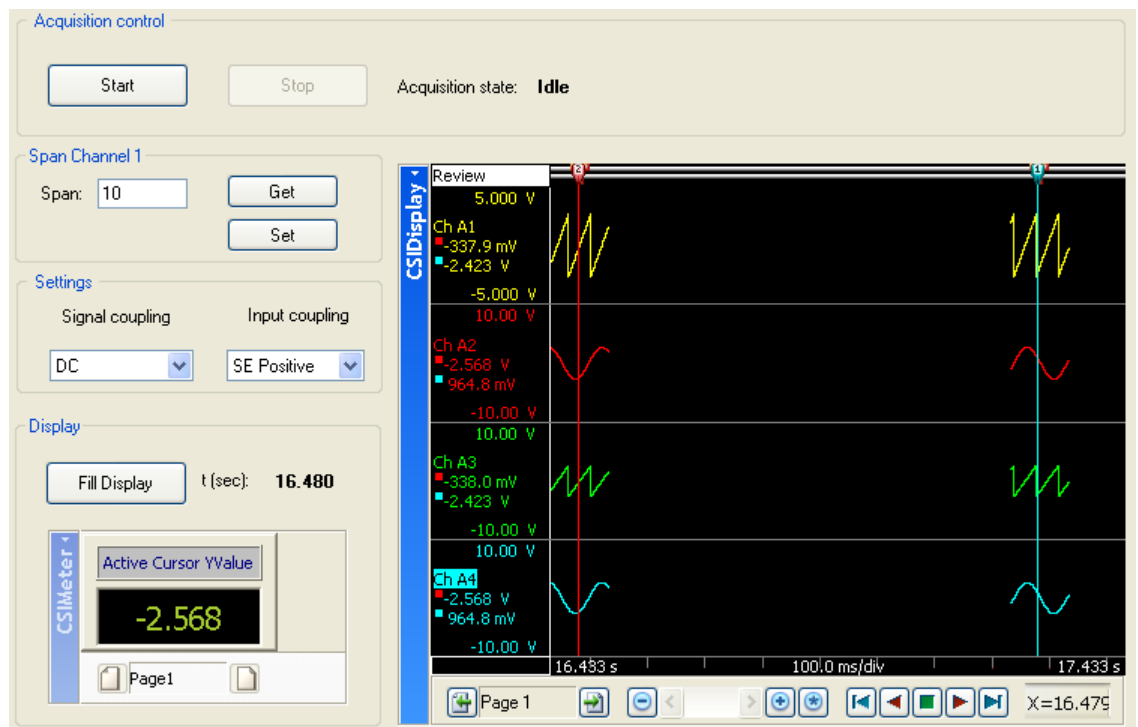
In **line 801** the actual value is fetched. Since an object is returned we need to convert it to a double before we can format the output. If no formatting was required we could have used the one-liner:

```
XValueLbl.Text = m_ActiveCursor.DataSource.Value.ToString();
```

Do not forget to call the `UnhookFromActiveCursor()` in the `SheetDisposed` method.

Also set the private member variable `m_ActiveCursor` to null in the `SheetDisposed` method.

Run the code to see it in action.



The meter shows the Y-value of the active trace at the position of the active (red) measurement cursor. The text label shows the X-position. Moving the cursor will update the values. Also selecting another trace will update the values.

7 Data Analysis - Part One

After discussing acquisition and visualization in the previous chapters it is now time to start working on the analysis of data. This topic is so extensive that we split it into two parts. In this first part we will describe the concepts of the data manager, data sources and user variables. In our examples we will demonstrate how you can use the measurement cursors to define a part of interest within a waveform, cut out that part of a waveform, do some basic operation on this data and put it back into the system and display it. We also will show how to use the internal Perception calculators to do some basic calculations on data. Display markers will be used to show the calculation results into a display.

7.1 Introduction

In this first part we will give an introduction on relevant concepts: data manager, data source and user variables.

7.2 The data manager

In previous chapters and examples we already have seen the data manager. The data manager was defined as:

```
protected DataManager m_DataManager = null;
...
if (m_DataManager == null)
{
    m_DataManager = new DataManager();
}
```

The data manager is the central part of the software that manages all data: waveforms (analog, digital), numerical values, system variables, etc.

To do so the **data manager**:

- keeps a list of ALL variables available in the system,
- maintains a list of event listeners for each variable,
- takes care of creating the link between data originators (data sources) and data consumers

Variables within the system:

- are of a specific type:
 - number
 - string (literal text)
 - waveform analog or digital
 - formula, i.e. the result of a calculation through the formula database,
 - etc.
- have a unique name, e.g.:
 - System.Constants.Pi

- Active.Group1.Recorder_A.Ch_A1
- have a reference to their data source (the IDataSrc interface)
- have a list of event subscribers. Events can be:
 - data added
 - data changed
 - etc.

All variables are controlled by the data manager and stored in the **data pool**. To gain access to a variable you need to fetch the variable out of the data pool through the data manager, i.e. you request from the data manager an entry out of the datapool, the **pool entry**. Using the aforementioned definition:

```

PoolEntry PE = m_DataManager.PoolEntries["System.Time"];

if (PE.EntryType == PoolEntryType.PoolEntryType_String)
{
}

```

Note that PoolEntries[<name>] always returns a PoolEntry object. This allows you to hook to events even before the variable is initialized.

At this point you have the pool entry, i.e. the gateway to the actual data source, but not yet the data itself!

For this we need to go one step further: the **IDataSrc** interface. This interface is the most important interface in Perception. It is the interface between the data originator and data consumer.

Typical data sources include numerical, string and waveform data sources.

7.2.1 Numerical data source

For the numerical data source the following 'settings' are in effect (there are more):

- Type = DataSourceDataType_Numerical
- Name is the user name of the variable, e.g. "Mean"
- Value is the actual value, returned as an object
- YUnits contains the technical unit string for this value

As an example have a look at the following code:

```
using RecordingInterface;
...

PoolEntry PE =
m_DataManager.PoolEntries["Display.CSIDisplay.ActiveCursor.Yvalue"];
if (PE.EntryType == PoolEntryType.PoolEntryType_Number)
{
    if (PE.DataSource.DataType == DataSourceDataType.
DataSourceDataType_Numerical)
    {
        if (PE.DataSource.Value != null)
        {
            double dValue = (double)PE.DataSource.Value;
            string sUnits = PE.DataSource.YUnit.ToString();
            string sName = PE.DataSource.Name.ToString();
        }
    }
}
```

The result can be:

- **dValue** = -0.31596969696967947
- **sUnits** = "Volt"
- **sName** = "Active Cursor YValue"

First we test the pool entry type. Then we test the actual data source type. Usually these are the same. However, there are situations with waveforms where the pool entry type returns an analogue waveform, while the data source type returns a more detailed description.

As mentioned the value itself is an object. When null it does not exist. In our example the active cursor could be outside the waveform range and return null. The value could also return an invalid number (NaN).

7.2.2 String data source

For the string data source the following 'settings' are in effect:

- Type = DataSourceDataType_String
- Name is the user name of the variable, e.g. "Local Time"
- Value is the actual string value, returned as an object

Example:

```

PoolEntry PE = m_DataManager.PoolEntries["System.Time"];
if (PE.EntryType == PoolEntryType.PoolEntryType_String)
{
    if (PE.DataSource.DataType == DataSourceDataType.DataSourceDataType_String)
    {
        if (PE.DataSource.Value != null)
        {
            string sTime = PE.DataSource.Value.ToString();
            string sName = PE.DataSource.Name.ToString();
        }
    }
}

```

The result can be:

- sTime = "11:55:50"
- sName = "Local Time"

In the above example the same procedure is followed as in the numerical data source example.

7.2.3 *Waveform data source*

Waveforms are more complex than the relatively simple numerical and string data sources. Important parameters / settings are:

- Status:
 - Static: the data has been recorded. No new data will be added to this waveform.
 - Dynamic: data is being recorded and DataAdded events will be fired.
- Sweeps: a collection of DataSweep objects that can be used to determine start and end times of sweeps in multi-sweep acquisitions.
- GetValueAtTime: a method to get a value at a specified time - even if that time is between two consecutive samples - using linear interpolation.

An important feature of waveforms is that they consist of one or more **segments**. When you retrieve the data, you will get initially a list of segments. Each segment is a piece of data with its own X- and Y- information as well as begin and end time. Data may be segmented due to timebase changes as well as amplifier range changes. Also gaps (e.g. caused by a temporarily pause of the acquisition) create segments.

For more detailed information on recordings, segments, etc. refer to the **HBM PNRF SDK User Manual** that is separately available.

7.3 User data sources

As opposed to the data sources described earlier there are also user variables. These user variables form a single collection of **user data sources**.

The user data sources are created and modified by the user, i.e. for specific types of variables the user can define its own content. You cannot do this with standard data sources, e.g. you cannot modify the value of string or the value of a cursor position or the contents of waveform.

Three types of user data sources can be created:

- Numerical data
- String data
- Waveform data

In order to create a user data source you will need to add a reference to the Perception.UserVariables.

To add the Perception UserVariables reference

1. Go to **Project > Add Reference...**
2. In the dialog that comes up select **Browse...**
3. Navigate to the Perception folder. Typically **C:\Program Files\HBM\Perception**.
4. Select the file **Perception.UserVariables.dll** and click **OK**.

Now the reference is added.

Add a using statement as follows:

```
using Perception.UserVariables;
```

7.3.1 User numerical data source

The user numerical data source requires two parameters to be created:

- Path name
- User name

Once created it has multiple properties, e.g.:

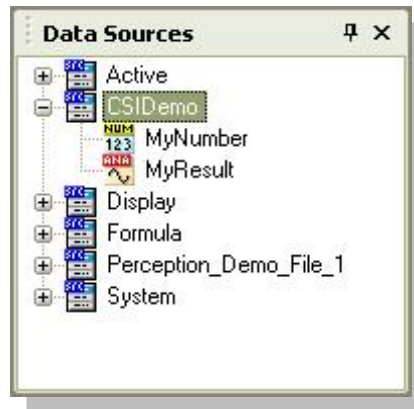
- Numerical value
- Units string

To create a user numerical data source see the following code:

```
NumericalDataSource MyNum;
UserDataSources MySources = UserDataSources.Instance;
MyNum = MySources.CreateNumber("CSIDemo.MyNumber", "CSI Numerical");
MyNum.Value = 10;
MyNum.Units = "Volt";
```

This code will create a user numerical data source and the required collection.

You can find the your newly created variable in the data sources navigator.



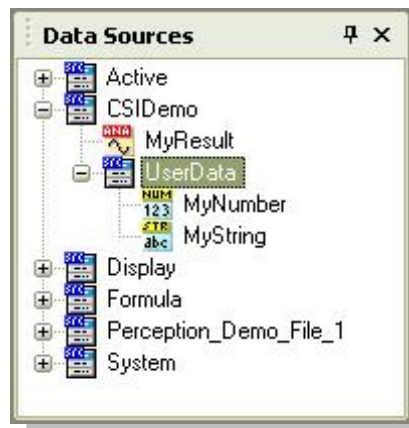
The variable has been created with the name MyNumber in the branch CSIDemo.

7.3.2 User string data source

Very much the same as we created a user numerical variable, we can create a user string variable. Assuming we already have created an instance of the UserDataSources collection:

```
StringDataSource MyString;
MyString = MySources.CreateString("CSIDemo.UserData.MyString", "CSI
String");
MyString.Value = "Hello World";
```

Here we also have expanded the navigator tree.



When you modify the variable name string from the user numerical value also gives you the above result.

7.3.3 User waveform data source

More complex is the user waveform data source.

The main parameters include:

- Name
- Both horizontal and vertical units
- Display range
- The use of blocks: a waveform is composed of blocks, each block with his own start time, X-Step value (sample rate) and number of samples.

Have a look at the following code, assuming we already have created an instance of the UserDataSources collection:

```
WaveformDataSource MyWaveData;
WaveformDataBlock WfBlock;

// fill a buffer with a double sinewave
// array is zero-based
double[] Buffer = new double[720];
double angle;
for (int i = 1; i <= 720; i++)
{
    angle = i * (Math.PI / 180);
    Buffer[i-1] = (double) (5 * Math.Sin(angle));
}

// create the waveform
MyWaveData = MySources.CreateWaveForm("CSIDemo.UserData.MyWave",
    "SineWave");

// set display range and units
MyWaveData.DisplayRangeFrom = 8;
MyWaveData.DisplayRangeTo = -8;
MyWaveData.YUnits = "Volt";
MyWaveData.XUnits = "s";

// add the data block and fill with buffer data
// data starts at t=0 and uses an XStep of 0.001 s
WfBlock = MyWaveData.AddDataBlock(0, 0.001);
WfBlock.WriteWaveform(0, Buffer);
```

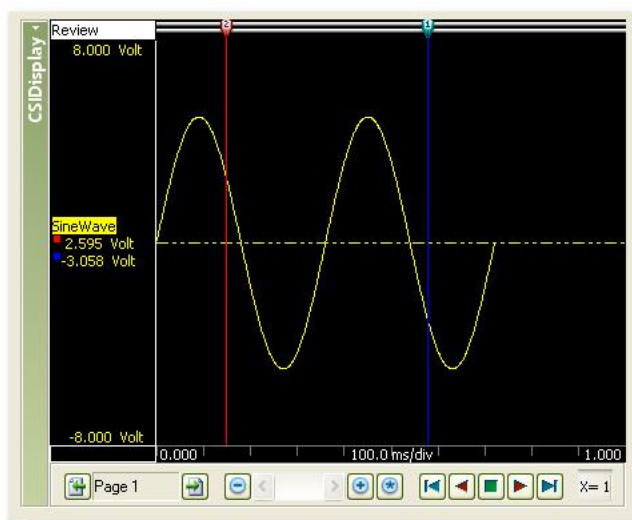
Start with the creation of a waveform data source and data block. For this example we fill an array with a dual sine wave, ranging from - 5 to + 5.

The waveform is added to the pool and the display range and units are set.

Then the block is added to the waveform, starting at t=0 and with a delta-t of 0.001 seconds between the samples, i.e. A sample rate of 1 kHz.

The block itself is filled with the buffer contents and an offset within the block of 0 samples. You can add (append) more data afterwards by using a different offset.

If you run this example you can drag the newly created waveform into a display to get the following result:



Here you see the waveform with the user name, the corresponding X- and Y-units and values.

Note: That at this point the user data source is still a user data source and not an `IDataSrc` as described earlier. E.g. the waveform is based on blocks and not on segments.

To convert a user data source into a general purpose data source use the following code:

```
IDataSrc NewWaveform = MyWave as IDataSrc;
m_DataManager.PoolEntries["CSIDemo.NewWaveform"].DataSource = NewWaveform;
m_DataManager.PoolEntries["CSIDemo.NewWaveform"].EntryType =
PoolEntryType.PoolEntryType_Waveform;
```

The `NewWaveform` is an alias for the `MyWave`, but behaves and can be treated as a 'regular' `IDataSrc` waveform pool entry.

7.4 The example

In the following example we will learn how to:

- position the measurement cursors automatically
- use the cursor positions to cut out a piece of the active trace and copy this data
- do some operations on the copied data
- put the result back in a new trace / data source

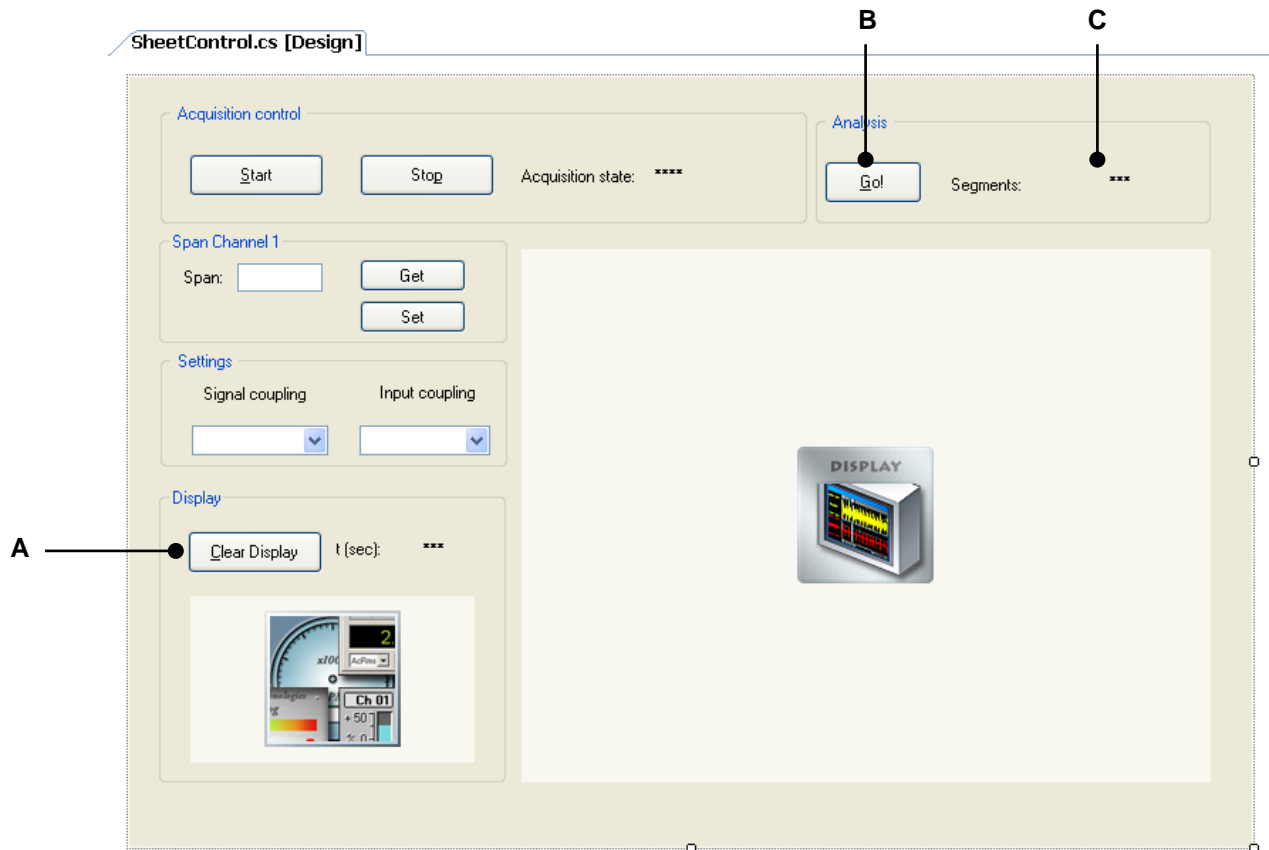
We will use a number of techniques discussed earlier in this chapter.

7.4.1 User interface

The user interface is based on our previous examples. The following modifications will be made:

- The Fill Display command button will be used to as a Clear Display command button: from now on we will use the navigators to fill the display. Hit the Clear Display button to empty the display.
- A new area is created with a command button that will start our "analysis". A text label is temporarily used to show the number of segments we are working on.

Below you see the layout.



- A. **Clear display** command button "FillCmd"
- B. **Go!** analysis command button "AnalysisCmd"
- C. Output label to show number of segments "SegmentsCnt"

7.4.2 The code - Getting started

We will start with some coding that needs to be done before we can actually start.

Make sure you have added the reference to the Perception.UserVariables as discussed earlier.

Our part of the using directives region should look as follows:

```

using Perception.Sheets;
using Perception;
using Perception.ILO.Engine;
using DataSrcManager;
using RecordingInterface;
using Perception.UserVariables;

```

The region MyMembers should look like this:

```

protected CtrlAcquisitionSystem m_MyDemoSystem = null;
protected CtrlGroup m_GroupAll = null;
protected CtrlChannel m_Channel = null;
protected DataManager m_DataManager = null;
protected PoolEntry m_ActiveCursor = null;
protected IDataSrc m_MySource = null;

```

The FillCmd_Click routine is now used only to clear the display and should be stripped down to:

```

private void FillCmd_Click(object sender, EventArgs e)
{
    // clear the display
    DisplayHelper.ClearDisplay(display1.pDisplay);
}

```

In this code we have used the DisplayHelper class, this class can be found in the Perception.UserKeys reference file. It contains some useful functions related to the Perception display.

Now we can start with the code behind the AnalysisCmd command button.

7.4.3 *Manipulate measurement cursor position*

In this example we will manually position a cursor on a point of interest. When the command button is pressed a region will be created by the two measurement cursors that occupies 40% of the horizontal display range: 20% before and 20% after the initial position of the active cursor.

To do so create the following code:

```
691 private void AnalysisCmd_Click(object sender, EventArgs e)
692 {
693     // start with defining the area of interest:
694     // +/- 20% of display range around active cursor
695
696     // get active cursor time position
697     double dActTime =
698     display1.pDisplay.TimeDisplay.Cursors.ActiveItem.time;
699
700     // centre display around this time
701     display1.pDisplay.TimeDisplay.CtlLayout.TimeController.CentralTime =
702     dActTime;
703
704     // fetch start and endtime = range
705     double dStartTime =
706     display1.pDisplay.TimeDisplay.CtlLayout.TimeController.StartTime;
707     double dEndTime =
708     display1.pDisplay.TimeDisplay.CtlLayout.TimeController.EndTime;
709     double dRange = dEndTime - dStartTime;
710
711     // set interval to +/- 20% of range around active cursor time
712     double dStep = 0.2 * dRange;
713
714     // set cursors
715     dStartTime = dActTime - dStep;
716     dEndTime = dActTime + dStep;
717     display1.pDisplay.TimeDisplay.Cursors[1].time = dStartTime;
718     display1.pDisplay.TimeDisplay.Cursors[2].time = dEndTime;
719 }
}
```

The code itself is very straightforward. You only need to know where to find the required information.

- **697**: get the time position of the active cursor.
- **700**: centre the display around this point. Actually the time position is moved to the centre of the display. By doing so we will not end up somewhere outside the display.
- **703 - 705**: calculate the total time span of the display.
- **708**: set the required measurement interval to $\pm 20\%$ of display range.
- **711 - 714**: set the measurement cursors to the calculated position.

To verify this code you do not need any data: just grab a cursor, position it anywhere in your display and hit the command button.

When this code runs well we can continue:

```

716 // locate data source of active trace
717 string sActDataSource =
       display1.pDisplay.TimeDisplay.ActiveTrace.TraceProp.DataSourceName;
718 if (sActDataSource == null)
719     return;
720
721 // use only analog waveforms for the this example
722 m_MySource = m_DataManager.PoolEntries[sActDataSource].DataSource;
723 if (m_MySource.DataType !=
       DataSourceDataType.DataSourceDataType_AnalogWaveform)
724     return;
  
```

Here we start with finding the data source of the active trace. This data source name can be found as a trace property from the active trace. Use the returned string as pool entry.

Verify if the data source type (not the pool entry) is an analogue waveform. When all is OK continue:

```

726 // initialize variables
727 object Result;
728 IDataSegments Segments;
729
730 // fetch data
731 m_MySource.Data(dStartTime, dEndTime, out Result);
732
733 // fetch segments within this data
734 Segments = Result as IDataSegments;
735 SegmentsCnt.Text = Segments.Count.ToString();
736
737 // no segments
738 if (Segments.Count == 0)
739     return;
  
```

- **727 - 728:** we need a **Result** object that is used to return the segment information from the data source. This information is placed in **Segments**.
- **731:** we fetch the data from the data source. Actually we fetch the segment information. Start and end time are the values we calculated earlier.
- **734 -735:** place the segment information in Segments and put the number of segments as text in our label.
- **738 -739:** when no segments are available we quit, otherwise we continue.

Typically when we request data from a continuous recording, a single segment will be returned. When we request data from a swept/transient recording, multiple segments can be returned.

In our example we will make a copy of the active trace between the two cursor positions and create a new waveform from this. We will also display the result.

Create the (user) waveform:

```

741 // create waveform in the user data sources
742 WaveformDataBlock Block = null;
743 WaveformDataSource MyWave = null;
744 UserDataSources MySources = UserDataSources.Instance;
745 MySources.RemoveDataSource(MySources["CSIDemo.MyResult"]);
746 MyWave = MySources.CreateWaveForm("CSIDemo.MyResult", "My Waveform");

```

- **742 -743:** initialize required variables.
- **744:** create an instance of the user data sources collection.
- **745:** for the purpose of this example: delete the waveform if it exists.
- **746:** create the waveform

Once created, we need to initialize the various properties of the waveform. In this situation we can copy various information from the original data source.

```

748 // set initial parameters
749 MyWave.XUnits = m_MySource.XUnit;
750 MyWave.YUnits = m_MySource.YUnit;
751 MyWave.DisplayRangeFrom = Segments[1].DisplayFrom;
752 MyWave.DisplayRangeTo = Segments[1].DisplayTo;

```

We copy the X- and Y-units from the data source, the display range is copied from the first segment. Usually the display range is the same for all segments. If you are not sure you will have to loop through all the segments to find the maximum display range.

We can now copy the data. We loop through all segments. For each segment we will add a block to the waveform and fill it with the relevant parameters and data.

```

754 // loop through all returned segments
755 for (int i = 1; i <= Segments.Count; i++)
756 {
757     int dSampleCount = Segments[i].NumberOfSamples;
758     Segments[i].Waveform(
759         DataSourceResultType.DataSourceResultType_Double64, 1,
760         dSampleCount, 1,
761         out Result);
762     double dXStep = Segments[i].SampleInterval;
763     Block = MyWave.AddDataBlock(Segments[i].StartTime, dXStep);
764     Block.WriteWaveform(0, (double[])Result);
765 }

```

- **755:** start the loop for all segments
- **758:** fetch the data: the data becomes available in Result. We request floating point data, starting at the first sample, all samples without reduction.
- **759:** get the sample interval.

- **761**: add the data block with the correct start time and sample interval.
- **762**: fill the data block with the data starting at the first position.

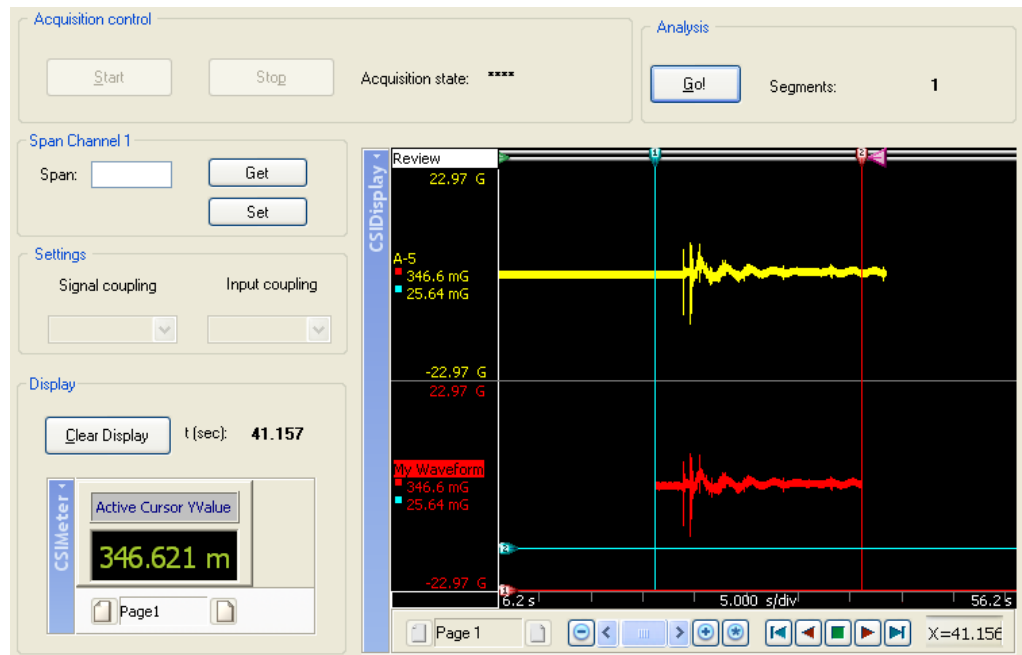
Note that in **line 761** we use the start time of the segment. We do this because this is the exact time of the first available sample.

Once we have added all the available information we can display the data:

```
765 display1.pDisplay.TimeDisplay.CtlLayout.Pages.ActivePage.Panes.  
    AddPane().Activate();  
766 string[] aPoolEntry = new string[] { "CSIDemo.MyResult" };  
767 display1.pDisplay.AddDataSources(aPoolEntry);
```

To do so: add a pane, activate it and add our new waveform data as a trace to the active pane.

As a test: clear the display and drag waveform data into the display. Click the command button.



Clear the display and use some other data. See what happens when an event trace is the active trace. Try to find a trace with multiple segments.

7.5 Do some basic math

At the same time we can already perform some basic manipulation on the data. As a very straightforward example we will create an additional waveform that is a straight line. Its DC value is the average value (mean) of the waveform interval we have just extracted.

To do so add the additional code on the appropriate places to create a second waveform:

```
// initialize also for calculated average value
WaveformDataBlock BlockAve = null;
WaveformDataSource MyWaveAve = null;

...

// create waveform also for calculated average value
MySources.RemoveDataSource(MySources["CSIDemo.MyResultAve"]);
MyWaveAve = MySources.CreateWaveForm("CSIDemo.MyResultAve",
    "My Average Waveform");

...

// set initial parameters also for average waveform
MyWaveAve.XUnits = m_MySource.XUnit;
MyWaveAve.YUnits = m_MySource.YUnit;
MyWaveAve.DisplayRangeFrom = Segments[1].DisplayFrom;
MyWaveAve.DisplayRangeTo = Segments[1].DisplayTo;
```

To find the average value you must add all sample values from all segments and divide this result by the total number of sample of all segments.

We should do this before the data is copied into our new waveform.

The following section does this calculation.


```

// loop through all segments to find out the average value
double dAverage = 0;
int iTotalCnt = 0;

// for all segments
for (int j = 1; j <= Segments.Count; j++)
{
    // fetch the data
    int dCnt = Segments[j].NumberOfSamples;
    Segments[j].Waveform(
        DataSourceResultType.DataSourceResultType_Double64,
        1, dCnt, 1, out Result);
    double[] Buffer = (double[])Result;

    // sum all sample values
    for (int m = 0; m < Buffer.Length; m++)
    {
        dAverage = dAverage + Buffer[m];
    }

    // sum the total number of samples
    iTotalCnt = iTotalCnt + dCnt;
}

// calculate the average
dAverage = dAverage / iTotalCnt;

```

Within the loop that does the copying of the original waveform section into our new waveform we also create the second waveform.

```

// create a block to waveform with average value
BlockAve = MyWaveAve.AddDataBlock(Segments[i].StartTime, dxStep);
double[] ResultAve = new double[dSampleCount];
for (int k = 1; k <= dSampleCount; k++)
{
    ResultAve[k - 1] = Convert.ToSingle(dAverage);
}
BlockAve.WriteWaveform(0, ResultAve);

```

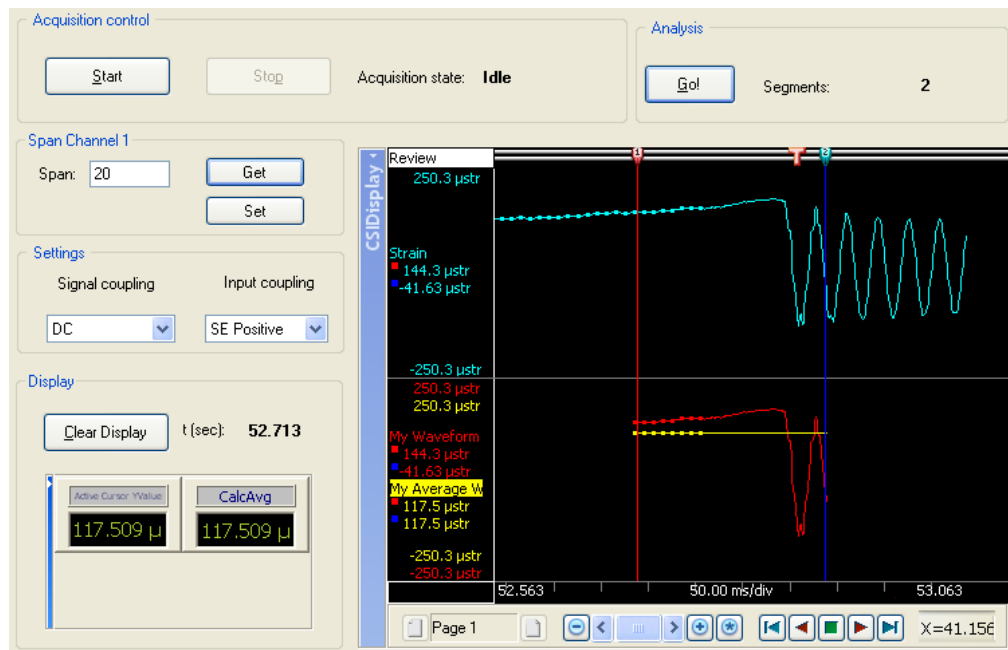
And after we have added the trace to the display we will also add the second trace to the display, overlaid on the first trace with a different color.

```

// add also - overlaid - the average value
aPoolEntry = new string[] { "CSIDemo.MyResultAve" };
display1.pDisplay.AddDataSources(aPoolEntry);
display1.pDisplay.TimeDisplay.CtlLayout.ActiveTrace.TraceProp.PrimaryColor
= 0x00FFFF; // yellow

```

When you run the code the result could look like this:



Here you see a situation in which there are two segments, the left-most with a lower sample rate.

For verification a formula in the formula database was created with the same calculation:

CalcAvg = @Mean(CSIDemo.MyResultAve)

This result is shown in a second meter that was dragged into the meter area we already had.

7.6 Using Perception waveform calculators

Perception has some built in calculators which are accessible via **CSI**. The calculators work on a single waveform, they can work on the complete waveform from beginning to end or you can define the interval for calculation.

The following calculators are available:

- **WavMinMax** - Finds Min, Max, MinPos and MaxPos
- **WavFastMinMax** - Finds only Min and Max
- **WavAreaEnergy** - Finds Area and Energy under curve
- **WavPeriodandCounter** - Finds Period, Frequency and number of Cycles
- **WavPulse** - Finds pulse characteristics like Rise Time, Fall Time and Pulse Width
- **WavStatistics** - Finds Mean, RMS and Sigma
- **WavHistogram** – Returns histogram array
- **WavFindLevelCrossing** - Finds the position of a level crossing
- **WavFindLocalExtreme** - Finds the local extremes

For our example we will calculate the maximum and minimum values and their positions. You can have a look in the HBM – Perception Interfaces help file to get detailed information on the **WavMinMax** class.

The screenshot displays the HBM - Perception Interfaces help file for the **WavMinMax Class**. The interface includes a navigation pane on the left with a tree view of namespaces and classes. The main content area shows the following details:

- Declaration Syntax:**

```
C# Visual Basic Visual C++
public class WavMinMax : WavBase
```
- Members:**

All Members	Constructors	Methods	Properties
<input checked="" type="checkbox"/> Public	<input checked="" type="checkbox"/> Instance	<input checked="" type="checkbox"/> Declared	<input checked="" type="checkbox"/> Inherited
<input checked="" type="checkbox"/> Protected	<input checked="" type="checkbox"/> Static		

Icon Member	Description
WavMinMax(DataSrc, Double, Double)	Constructor of the min/max calculator.
End	End position of time interval used for the calculations.
Max	Maximum value of the waveform between start and end time.
MaxPos	Position of the maximum value of the waveform between start and end time.
Min	Minimum value of the waveform between start and end time.
MinPos	Position of the minimum value of the waveform between start and end time.
Result(Double, Double, Double, Double)	Returns the result of the min/max calculator.
Start	Start position of time interval used for the calculations.
- Remarks:** This calculator returns the maximum and minimum value and their positions between 'Start' and 'End' time. When only the maximum or minimum value is wanted then use the **WavFastMinMax** calculator.
- Examples:**

```
C# Copy
WavMinMax Calculator = new WavMinMax(1ActiveTrace, double.MinValue, double.MaxValue);
WavCalcResult CalcResult = Calculator.Exec();

if (CalcResult != WavCalcResult.OK)
{
    MessageBox.Show(this, "Can not calculate Max/Min\r\n\r\n" + Calculator.ErrorCode, "CSI Display Demo");
    return;
}

double dMin; double dMax; double dMinPos; double dMaxPos;
Calculator.Result(out dMin, out dMax, out dMinPos, out dMaxPos);
```
- Inheritance Hierarchy:**

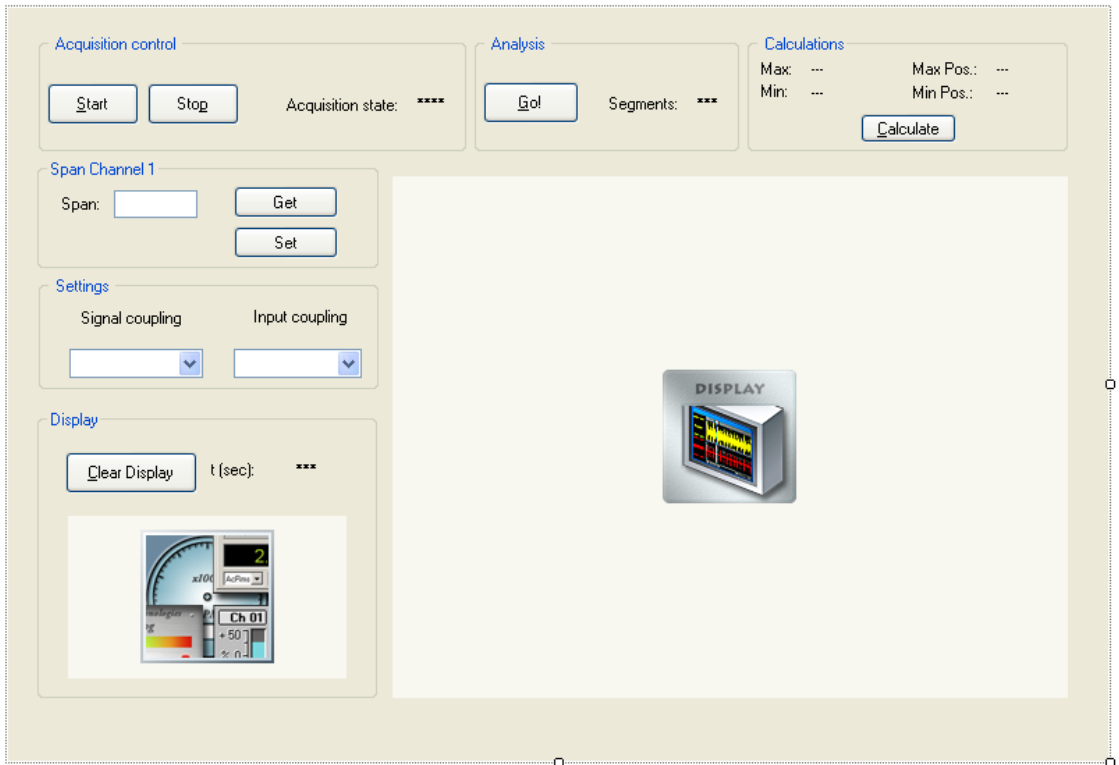
```
Object
└─ WavBase
   └─ WavMinMax
```
- See Also:** WavFastMinMax

Send comments on this topic to [HBM - CSI Support](#)
 Copyright © 2009, HBM
 Assembly: Perception.WavCalc (Module: Perception.WavCalc) Version: 6.0.9064.1215

After the maximum, minimum, maximum position and minimum position are found the vertical and horizontal cursors will be set. We also label the two points with two Perception trace markers.

Extend the last example with a groupbox called **Calculations** see picture below.

SheetControl.cs [Design]*



In order to create a calculator you will need to add a reference to the **Perception.WavCalc.dll**

Also add a reference to the **Perception.Common.dll** because this dll contains the conversion routines.

Add the following code behind the button **Calculate** event click:

```

private void btnCalc_Click(object sender, EventArgs e)
{
    // Get Active Trace
    DTrace myTrace = display1.pDisplay.TimeDisplay.ActiveTrace;
    if (myTrace == null)
    {
        PerceptionMessageBox.Show(this, "No active trace found",
            "CSI Display Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    // Get datasource from active trace
    IDataSrc iActiveTrace = myTrace.TraceProp.DataSrc;
    if (iActiveTrace == null)
    {
        PerceptionMessageBox.Show(this, "Active trace has no datasource",
            "CSI Display Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    // Create a new Min Max calculator
    WavMinMax Calculator = new WavMinMax(iActiveTrace,
        double.MinValue, double.MaxValue);
    // Execute the calculator
    WavCalcResult CalcResult = Calculator.Exec();
    // Check if result is OK
    if (CalcResult != WavCalcResult.OK)
    {
        PerceptionMessageBox.Show(this, "Can not calculate Max/Min\r\n\r\n" +
            Calculator.ErrorCode, "CSI Display Demo", MessageBoxButtons.OK,
            MessageBoxIcon.Error);
        return;
    }

    // Get the results from the calculator
    double dMin; double dMax; double dMinPos; double dMaxPos;
    Calculator.Result(out dMin, out dMax, out dMinPos, out dMaxPos);

    // Show the results using Perception conversion functions
    lblMax.Text = Conversion.ConvertDoubleToString(dMax, 4,
        iActiveTrace.YUnit) + iActiveTrace.YUnit;
    lblMin.Text = Conversion.ConvertDoubleToString(dMin, 4,
        iActiveTrace.YUnit) + iActiveTrace.YUnit;

    lblMaxPos.Text = Conversion.ConvertDoubleToString(dMaxPos, 4,
        iActiveTrace.YUnit) + iActiveTrace.XUnit;
    lblMinPos.Text = Conversion.ConvertDoubleToString(dMinPos, 4,
        iActiveTrace.YUnit) + iActiveTrace.XUnit;

    // Position the vertical cursors at the maximum and minimum position
    display1.pDisplay.TimeDisplay.Cursors[1].time = dMaxPos;
    display1.pDisplay.TimeDisplay.Cursors[2].time = dMinPos;

    // Set the Horizontal cursors to the maximum and minimum levels.
    display1.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors.Visible = true;

    // Get the high and low display range values from the active trace
    double dHigh, dLow;
    myTrace.TraceProp.GetRange(out dHigh, out dLow);
    // Use these values to get the relative levels via a linear
    //conversion
    double dLevel1 = (dMax - dLow) / (dHigh - dLow);
    double dLevel2 = (dMin - dLow) / (dHigh - dLow);

    // We also need to know the position of the active pane.
    double dTop = 1;
    double dBottom = 0;
    myTrace.Pane.GetPositions(ref dTop, ref dBottom);
    dLevel1 = (dTop - dBottom) * dLevel1 + dBottom;
    dLevel2 = (dTop - dBottom) * dLevel2 + dBottom;
  }
}

```

```
// Now we can set the relative level values
display1.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors[1].Location =
    dLevel1;
display1.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors[2].Location =
    dLevel2;

// Set a display marker for the maximum value.

// First clear all existing markers
ClearAllMarkers();

// Create a new display marker to indicate the maximum position
DisplayMarker myTraceMarkerMax = myTrace.DisplayMarkers.AddMarker(
    DisplayModeType.DisplayModeType_Review);
myTraceMarkerMax.DrawingType =
    MarkerDrawingType.MarkerDrawingType_TraceMarker;
myTraceMarkerMax.MarkerStartSourceX = dMaxPos;
myTraceMarkerMax.MarkerStartSourceY = dMax;
myTraceMarkerMax.UnformattedText = "Max";
myTraceMarkerMax.SetLabelPosition(DisplayModeType.DisplayModeType_Review,
    LabelAlignment.LabelAlignment_Left, -0.1, -0.1);

// Create a new display marker to indicate the minimum position
DisplayMarker myTraceMarkerMin = myTrace.DisplayMarkers.AddMarker(
    DisplayModeType.DisplayModeType_Review);
myTraceMarkerMin.DrawingType =
    MarkerDrawingType.MarkerDrawingType_TraceMarker;
myTraceMarkerMin.MarkerStartSourceX = dMinPos;
myTraceMarkerMin.MarkerStartSourceY = dMin;
myTraceMarkerMin.UnformattedText = "Min";
myTraceMarkerMin.SetLabelPosition(DisplayModeType.DisplayModeType_Review,
    LabelAlignment.LabelAlignment_Left, 0.1, 0.1);
}
```

The following code is used to clear all display markers.

```
// Clears all existing markers
private void ClearAllMarkers()
{
    // get the number of display pages
    int NumberOfPages = display1.pDisplay.TimeDisplay.CtlLayout.Pages.Count;
    // Remove the markers per page
    for (int i = 1; i <= NumberOfPages; i++)
    {
        DPage myPage = display1.pDisplay.TimeDisplay.CtlLayout.Pages[i];
        // Get the display markers collection from the page
        DisplayMarkers myDisplayMarkers = myPage.DisplayMarkers;

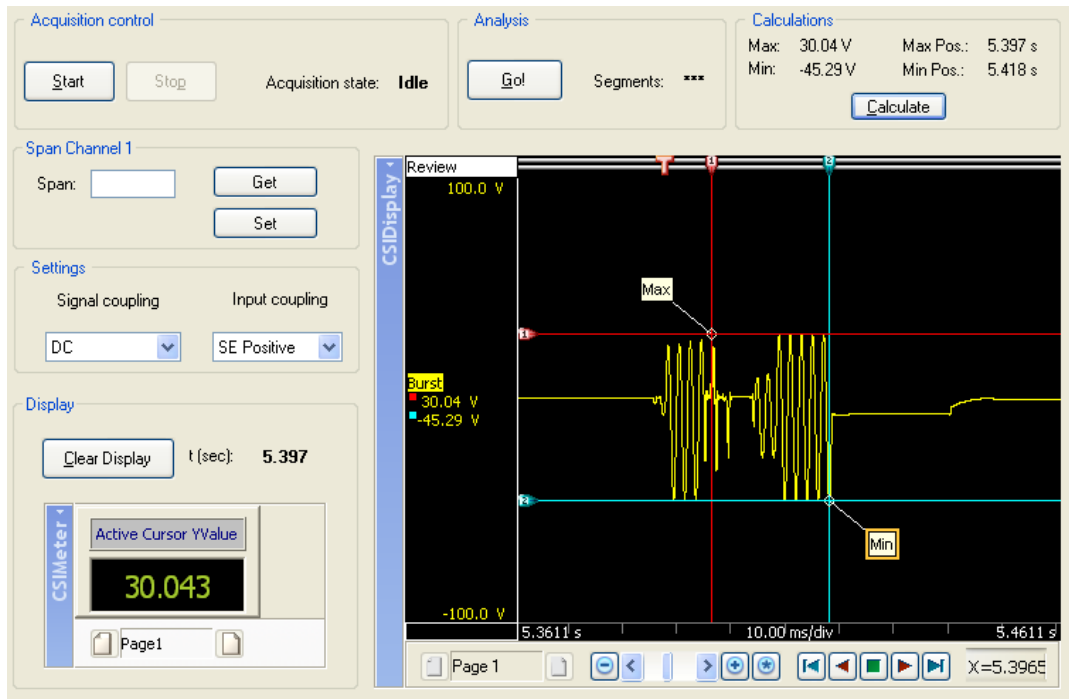
        // Keep on deleting the markers until the collection is empty
        while (myDisplayMarkers.Count > 0)
        {
            DisplayMarker myPageDisplayMarker = myDisplayMarkers[1];
            myPageDisplayMarker.Delete();
        }

        // Now we will delete the display markers connected to the individual
        // traces. Therefore we have to loop through the panes
        for (int n = 1; n <= myPage.Panes.Count; n++)
        {
            DPane myPane = myPage.Panes[n];

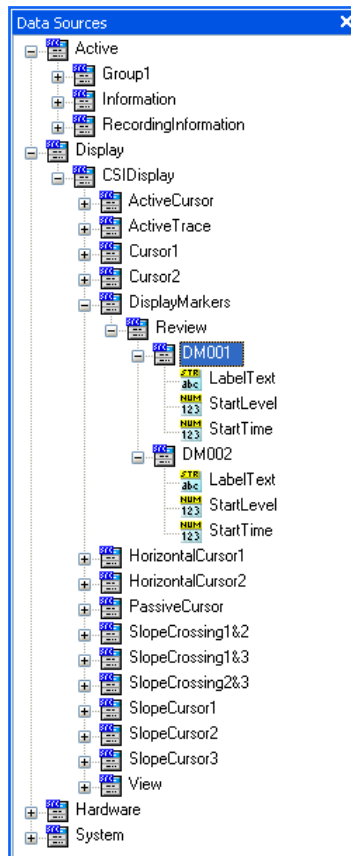
            // Go through all the traces from this pane
            for (int m = 1; m <= myPane.Traces.Count; m++)
            {
                DTrace myTrace = myPane.Traces[m];
                // Get the display markers collection from this trace
                DisplayMarkers myTraceDisplayMarkers = myTrace.DisplayMarkers;

                // Keep on deleting the markers until the collection is empty
                while (myTraceDisplayMarkers.Count > 0)
                {
                    DisplayMarker myTraceDisplayMarker =
                        myTraceDisplayMarkers[1];
                    myTraceDisplayMarker.Delete();
                }
            }
        }
    }
}
```

If you compile and run the code then drag and drop a trace into the display you can click the **Calculate** button. The vertical and horizontal cursors will be set, two trace markers will be added and the numerical data is displayed into the calculation groupbox.



The two display markers are also added automatically to the pool of **Data Sources**, you can find them with the **Data Sources** navigator as show in the picture below.



8 Data Analysis - Part Two

Within Perception the advanced analysis option allows you to define your own formulas so as to compute results in seconds rather than hours.

A "formula database" is used to enter math expressions like **CH1*CH2** or **MAX@CH1** to compute results immediately when new data arrives. The formula results can be re-used in other formulas to get even more advanced answers.

With Perception CSI you have access to the formula database. The formula database is a powerful tool to quickly evaluate complex expressions in your application, simply by inserting a formula into the database, and evaluating the function.

In addition you can also create your own functions, functions that are not already part of the formula database.

In this chapter we will discuss both implementations.

8.1 Formula database as calculator

It is not difficult to use the formula database in your code to perform calculations. Simply add a reference to the formula database functions in your project and add the required formula "as text" to the database. Perception will do the math and update for you.

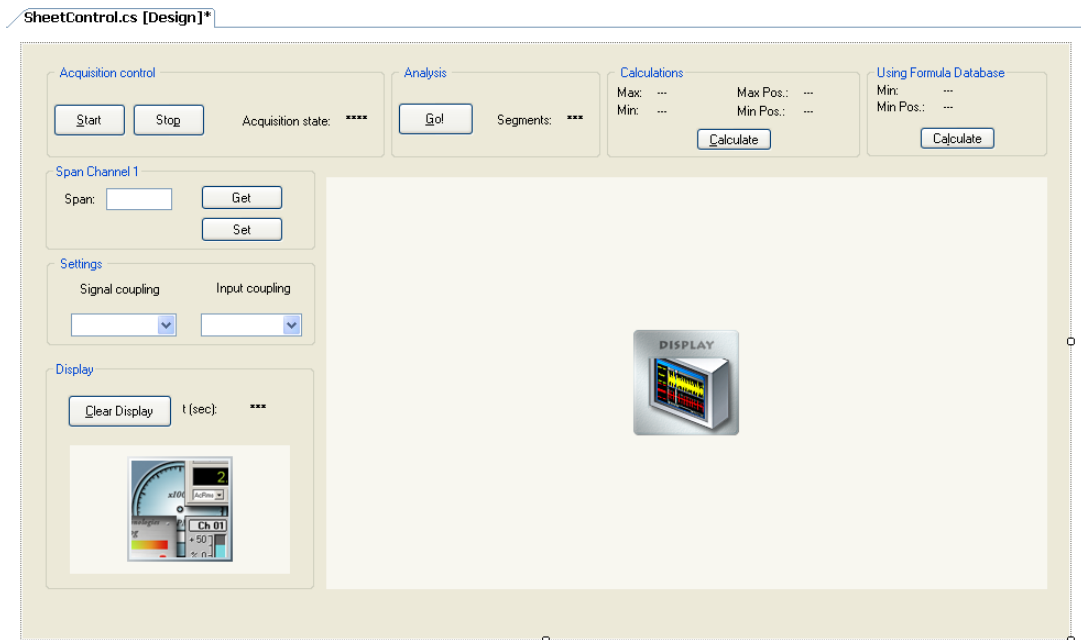
We will continue with our example from the previous chapter and do the following:

- add the formula database reference to our project
- add a formula to calculate the minimum of the waveform segment we created
- add a formula that returns the time position of this minimum
- position the cursor on this point

Please note that when the Advanced Analysis option is not installed in Perception, the code will 'install' correctly, but no results will become available from the formula database.

8.1.1 User interface

The only thing we want to do is to display the minimum value. You can do this either by dragging the result into the meter area (or create an additional meter in the user interface) or use new text labels. In this example we will go with this last option. We will add a new group box called "*Using Formula Database*" see next picture.



8.1.2 The code

Start with adding the correct reference to the project.

To add the Perception Formula Database reference

- 1 Go to **Project > Add Reference...**
- 2 In the dialog that comes up select **Browse...**
- 3 Navigate to the Perception folder. Typically **C:\Program Files\HBM\Perception**.
- 4 Select the file **Perception.FormulaDatabase.dll** and click **OK**.

Now the reference is added.

Add a using statement as follows:

```
using Perception.FormulaDatabase;
```

The following code has to be added in the new button event click

```
1082 private void btnCalcUsingForDB_Click(object sender, EventArgs e)
1083 {
1084     // Get Active Trace
1085     DTrace myTrace = display1.pDisplay.TimeDisplay.ActiveTrace;
1086     if (myTrace == null)
1087     {
1088         PerceptionMessageBox.Show(this, "No active trace found",
1089             "CSI Display Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
1090         return;
1091     }
1092     // Get datasource from active trace
1093     IDataSrc iActiveTrace = myTrace.TraceProp.DataSrc;
1094     if (iActiveTrace == null)
1095     {
1096         PerceptionMessageBox.Show(this, "Active trace has no datasource",
```

```

1097         "CSI Display Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
1098     return;
1099 }
1100
1101 // Get Data source name from trace
1102 // e.g.: "Active.Group1.Recorder_B.Burst"
1103 string cDSName = myTrace.TraceProp.DataSourceName;
1104
1105 // create required formulas in formula database
1106
1107 // create instance of formula database
1108 FormulaDB ForDB = FormulaDB.Instance;
1109
1110 // check if formula already exist
1111 if (ForDB.Formulas["CSI:MinVal"] == null)
1112 {
1113     // search for an empty line
1114     foreach (Formula frml in ForDB.Formulas)
1115     {
1116         if (frml.IsEmpty())
1117         {
1118             // Define Name, Formula and Units
1119             // (See columns formula database sheet)
1120             frml.Name = "CSI:MinVal";
1121             // Expression can be: "@Min(Active.Group1.Recorder_B.Burst)"
1122             frml.Expression = string.Format("@Min({0})", cDSName);
1123             ForDB.Formulas[frml.LineNumber + 1].Name = "CSI:MinPos";
1124             // Expression can be: "@MinPos(Active.Group1.Recorder_B.Burst)"
1125             ForDB.Formulas[frml.LineNumber + 1].Expression =
1126                 string.Format("@MinPos({0})", cDSName);
1127
1128             // fetch units from source
1129             m_MySource = m_DataManager.PoolEntries[cDSName].DataSource;
1130             frml.Units = iActiveTrace.YUnit;
1131             ForDB.Formulas[frml.LineNumber + 1].Units = iActiveTrace.XUnit;
1132             break;
1133         }
1134     }
1135 }
1136

```

- **1108:** Create the single instance of the formula database.
- **1111:** check if formula with that name already exists. Here we perform a very simple test. In reality this should be better.
- **1114 - 1116:** we scan through the complete formula database in search of the first empty line.
- **1118 - 1122:** enter the formula name and the formula itself. Here we use the loop variable.
- **1123 - 1123:** next formula. Here we use the exact line number, which is the variable's line number + 1
- **1139 - 1131:** enter the units. These are the same units as from the original data source.

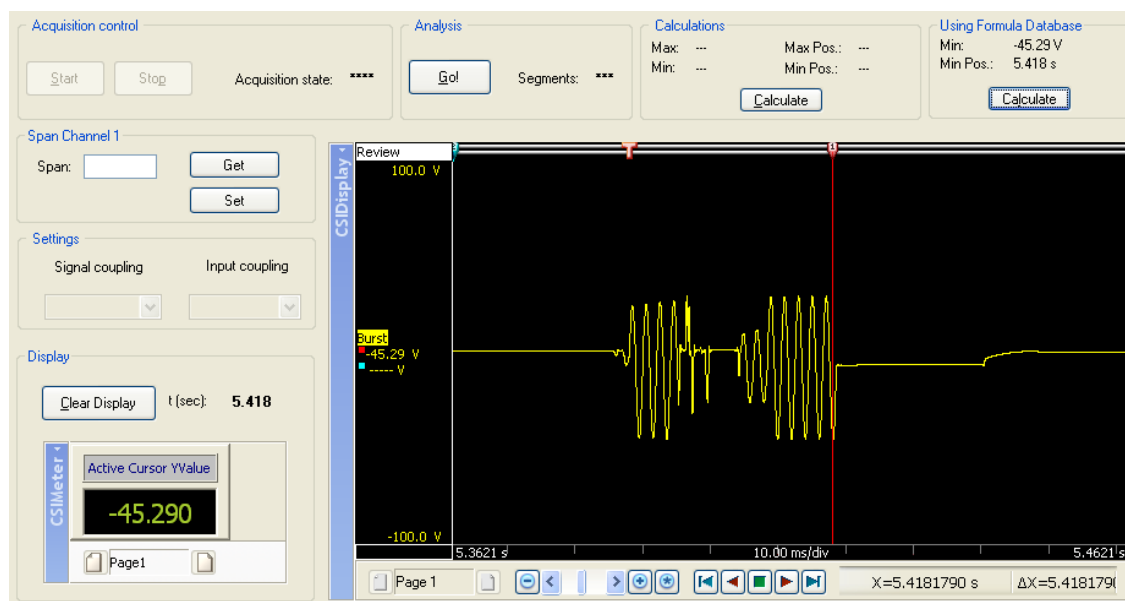
At this point the correct formulas are created. We now use these calculated values that are in the data pool. After use we clean it up. When you do not remove them, they will show up in the Data Sources Navigator and the formulas sheet.

```
1137 // display formatted value
1138 m_MySource =
        m_DataManager.PoolEntries["Formula.CSI:MinVal"].DataSource;
1139 double dMinVal = (double)m_MySource.Value;
1140 lblMinForDB.Text = Conversion.ConvertDoubleToString(dMinVal, 4,
1141     iActiveTrace.YUnit) + iActiveTrace.YUnit;
1142
1143 // place cursor
1144 m_MySource =
        m_DataManager.PoolEntries["Formula.CSI:MinPos"].DataSource;
1145 if (m_MySource == null) return;
1146 double dMinPos = (double)m_MySource.Value;
1147 display1.pDisplay.TimeDisplay.Cursors[1].time = dMinPos;
1148 lblMinPosForDB.Text = Conversion.ConvertDoubleToString(dMinPos, 4,
1149     iActiveTrace.XUnit) + iActiveTrace.XUnit;
1150
1151 // once done: clean up the mess
1152 ForDB.Formulas["CSI:MinVal"].Clear();
1153 ForDB.Formulas["CSI:MinPos"].Clear();
1154 }
```

- 1137 - 1139: fetch value and cast it to a double.
- 1140: use the **ConvertDoubleToString()** function to display the minimum value.
- 1144 - 1149: fetch position of value and place cursor on it.
- 1152 - 1153: remove the formulas.

Of course the formulas can be as extensive as required. Also you may leave them in the formula database when required.

If you have compiled and run your code the following information can be shown after a calculate button click:



8.2 Make your own functions

In the previous section we have seen how you can use the formula database to create 'volatile' calculations. In this section we will discuss how you can create 'non-volatile' functions that become an integral part of the formula database.

You can create such a function in your CSI sheet. It then becomes part of the Perception software when this sheet is loaded.

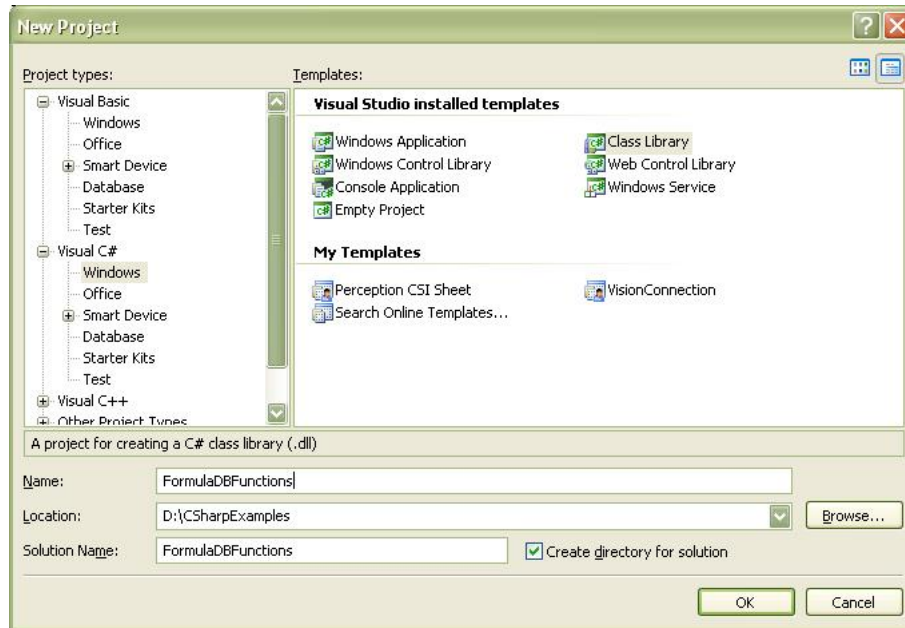
There is also a second - more general - option. You can build your function in a Windows Class Library, this dll has to be saved into the Perception subfolder **Functions**. When Perception starts and it finds such a "dll" it will load this code and - assuming the advanced analysis option as well as the CSI option are installed - add the function to the formula database. I.e. you will have access to your very own built function any time you start Perception.

In this section we will explain the second option: create an external file that comprises your own functions that are always available whenever you run Perception.

As an example we will create a function that we have done already once: create a waveform that is a straight line. Its DC value is the average value (mean) of a selected waveform or part of the waveform.

8.2.1 Create and initialize the external class library

To create the class library in Visual Studio: select **File > New > Project...**



In the dialog that comes up select a Visual C# Windows project. Select as template the Class Library. Give the project a name and click OK.

Add the required references:

- 1 Go to **Project > Add Reference...**
- 2 In the dialog that comes up select **Browse...**
- 3 Navigate to the Perception folder. Typically: **C:\Program Files\HBM\Perception**.
- 4 Multi-select the files **Perception.FormulaDatabase.dll**, **Perception FormulaDatabase.FunctionSupport.dll**, **Perception.Interops.dll** and click **OK**.

Now the references are added.

Add the following using clauses:

```

| using Perception.FormulaDatabase.Functions;
| using Perception.FormulaDatabase.FunctionSupport;
| using RecordingInterface;

```

Now we can start with the initial code.

8.3 The function information

When you want to create your own functions to be used in Perception, you first need to create a "function information" module for that function. This module is implemented as a class which implements the **IFunctionInfo** interface.

The formula database engine searches for classes implementing the **IFunctionInfo** and creates instances of these classes to get the relevant information.

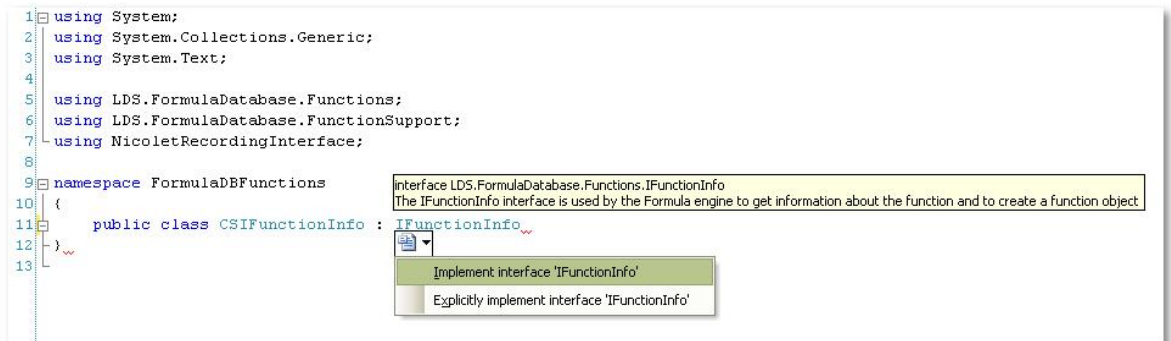
When a function is used in a formula, the IFunctionInfo interface is used to create an instance of that function.

To create the IFunctionInfo interface proceed as follows:

Type the line of code:

```
public class CSIFunctionInfo : IFunctionInfo
```

Now you should see below the "I" from IFunctionInfo a small rectangle. Click on the rectangle:



and select Implement Interface 'IFunctionInfo'.

A complete code block will be implemented:

```

public class CSIFunctionInfo : IFunctionInfo
{
    #region IFunctionInfo Members

    public string Category
    {
        get { throw new Exception(
            "The method or operation is not implemented."); }
    }

    ...

    public nicDataManager.PoolEntryType[] ParameterTypes
    {
        get { throw new Exception(
            "The method or operation is not implemented."); }
    }

    #endregion
}
    
```

We must now fill all these 'entries' with the correct information.

8.3.1 Category

This entry is used by the function wizard (when implemented in your version of the software) to place the function within a specific category.

```
public string Category
{
    get
    {
        return "CSI Functions";
    }
}
```

8.3.2 CreateFunction

This entry creates the new function. For this example we use the internal name "CreateWaveformFromAverage".

```
public IFunction CreateFunction()
{
    // for testing purposes as long as the function
    // is not implemented: return null;
    return new CreateWaveformFromAverage();
}
```

As long as the actual function is not yet implemented we return "null".

8.3.3 Description

A literal description of the function.

```
public string Description
{
    get
    {
        return "Creates a DC waveform with value of average of the
        waveform to process (InWave).";
    }
}
```

8.3.4 Example

Provides an example on how to use the function.

```
public string Example
{
    get
    {
        return "@CSI_MeanWave(Active.Group1.Recorder1.Ch_A2)";
    }
}
```


8.3.5 *MinimumParameterCount*

This value indicates the minimum number of parameters required by the function. Must be at least one (1).

```

public int MinimumParameterCount
{
    get
    {
        return 1;
    }
}

```

8.3.6 *Name*

The name of the function exactly as it will appear in the function list without the "@" character in front.

```

public string Name
{
    get
    {
        return "CSI_MeanWave";
    }
}

```

8.3.7 *Parameters*

Each parameter has a name, type and description. The name and description are strings, the type is a PoolEntryType. Multiple parameters can be required. Therefore these values are all returned as arrays.

8.3.8 *ParameterDescriptions*

These are literal descriptions of each parameter. For the purpose of our example only one parameter, and therefore one description, is required.

```

public string[] ParameterDescriptions
{
    get
    {
        string[] ParamDescriptions = {"The waveform to process"};
        return ParamDescriptions;
    }
}

```

8.3.9 *ParameterNames*

These are the names of each parameter. For the purpose of our example only one parameter, and therefore one name, is required.

```

public string[] ParameterNames
{
    get
    {

```

```
    string[] ParamNames = {"InWave"};
    return ParamNames;
}
}
```

8.3.10 ParameterTypes

These are the parameter types as they are known in the data pool: the data pool entries.

```
public nicDataManager.PoolEntryType[] ParameterTypes
{
    get
    {
        nicDataManager.PoolEntryType[] PTypes =
            {nicDataManager.PoolEntryType.PoolEntryType_Waveform};
        return PTypes;
    }
}
```

8.4 Getting it all to work

At this point we should be able to test the IFunctionInfo interface.

The most important part is to create this dll and put it in the correct folder.

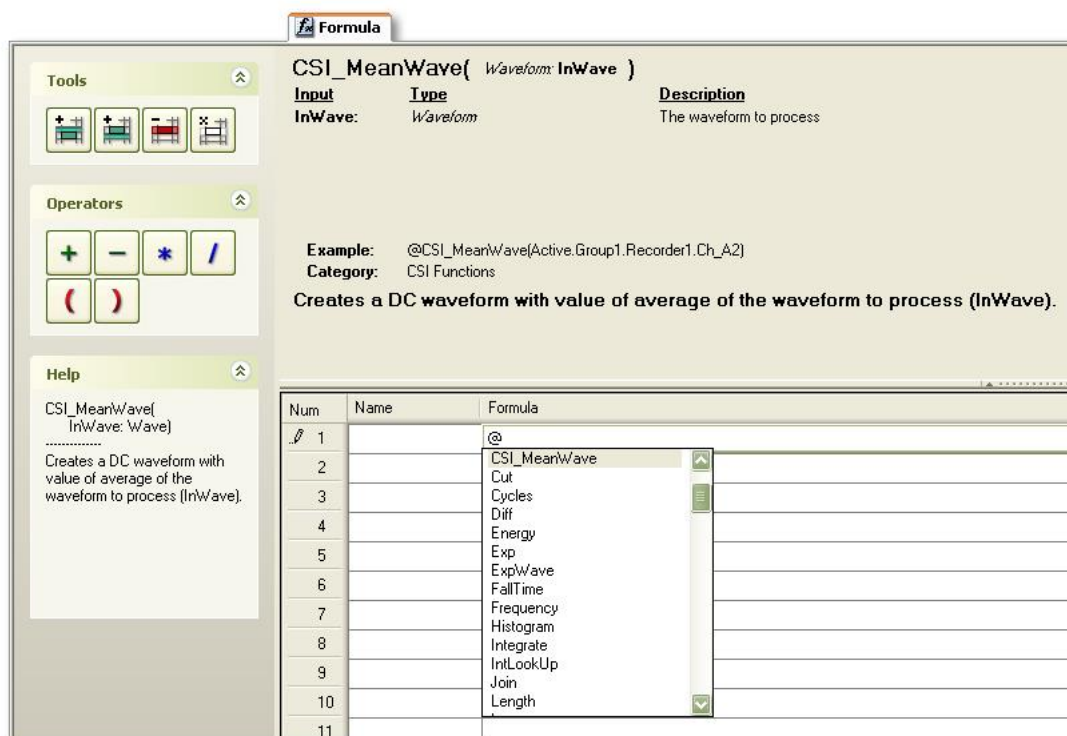
The file must be placed in the Perception program folder, typically **C:\Program Files\HBM\Perception\Functions**.

To get this done we need to modify the properties of the project.

- 1 Go to **Project > <projectname> Properties...**
- 2 In the **Build** tab locate the **Output** section and set the **Output path** to the Perception program path, typically **C:\Program Files\HBM\Perception\Functions**.

Run the code and when Perception starts go to the Formula sheet.

When you type an "@" on a formula command line, the drop-down list should list the new function and when you select that function the provided information must be available.



Have a close look to the output and modify some parameters in your code to see the various results.

8.5 Implement the function

To actually implement the function there are two options:

- The comprehensive method
- The intelligent method

Both methods have their advantages. The intelligent method uses code that has already been defined and tested in Perception. This will cover 90% of the applications. For very specific

requirements you might need to use the comprehensive method. We will explore the intelligent method, but before we start give an idea of the comprehensive method.

8.5.1 Using the comprehensive method

Once we have defined the IFunctionInfo we need to implement the IFunction function that is returned by the CreateFunction entry of the IFunctionInfo.

The IFunction requires three methods:

- **Init** Initialize: an array of datasource interfaces is passed in and a resulting datasource should be returned.
- **Reset** The method that is called by the calculation engine to reset the function when a DataChanged or DataSourceChanged event is processed.
- **OnDataAdded** Tells the calculation engine which type of event should be fired when a DataAdded event is being processed.

As a general example have a look at the following code:

```
class ExampleFunction : IFunction, IDataSrc
{
    protected IDataSrc[] m_Arguments;

    public void Init(IDataSrc[] Parameters, out IDataSrc Result)
    {
        m_Arguments = Parameters;
        Result = this;
    }

    public PoolEventType OnDataAdded(double StartTime,
                                     double EndTime)
    {
        return PoolEventType.DataChanged;
    }

    public void Reset()
    {
    }

    // IDataSrc implementation
}
```

The difficult part starts at the last line: **IDataSrc** implementation. Here you will need to implement the complete IDataSource interface for your routine. This includes Data, DataType, GetUTCTime, GetValueAtTime, Name, Properties, Status, etc. To see a complete list follow the procedure described to quickly implement the IFunctionInfo.

As mentioned earlier this method may be useful for very special situations where the intelligent method fails, or when you only need to perform a very basic calculation that returns a single value.

For the purpose of this manual we will continue with the intelligent method.

8.5.2 Using the intelligent method

The intelligent method is based on **inheritance** and the **override** method that provides a new implementation of a member inherited from a base class. We will build the function code based on well-tested, existing classes from Perception.

Without going into details we start by using the **SampleBySampleOrNumericalFunction** class. When you start typing your entry for the class, Visual Studio will assist you in a correct inheritance of the base class. Your first result should look like this:

```
public class MyExample : SampleBySampleOrNumericalFunction
{
    protected override SampleBySampleDataSegment CreateWrapperSegment
    (IDataSegment OriginalSegment)
    {
        throw new Exception(
            "The method or operation is not implemented.");
    }

    protected override object ProcessValue(object Value)
    {
        throw new Exception(
            "The method or operation is not implemented.");
    }
}
```

As you can see this inheritance requires two method overrides:

- CreateWrapperSegment
- ProcessValue

The last one is the easiest one for us. **ProcessValue** is a method that is called when 'somebody' needs to have a single value processed using our function. E.g. when the result of our function is displayed, a cursor movement would request a single value at the cursor position. It therefore passes the value of the original data to this method and expects a result back. This is true since we perform a 'sample-by-sample' function.

A more detailed example: assume our function divides a waveform by 10 and puts the result back in a result: result waveform = (original waveform) / 10. Each sample of the result is one-tenth of the value of the original value.

What happens when a measurement cursor is between two samples? It (the cursor engine) will do a calculation of the Y-value based on linear interpolation between the two samples and pass that value to our function. Our function does its trick (divides the value by 10) and returns the result. E.g. a sample with value 10 and a sample with value 12. The cursor is exactly in the middle: it passes 11 to our function and the return value should be 1.1, which is exactly between 1.0 and 1.2.

In our programming example it is even simpler: we always return the calculated average.

The **CreateWrapperSegment** is a little bit more complicated to understand. Basically what happens is the following: the **SampleBySampleOrNumerical** class takes care of the complete handling of a waveform and therefore also supports the segments as we know them from the **IDataSrc** interface. Since we are performing analysis on a waveform on a sample by sample basis we need to process each original segment individually and return a processed segment. That is where the **CreateWrapperSegment** comes in.

The **CreateWrapperSegment** provides an original segment and should return a processed segment. These segments are of type **SampleBySampleDataSegment**. Therefore the override must:

- Get the average value of the complete waveform
- Create a new segment based on the calculation

To calculate the average value of the complete waveform is not difficult as we will demonstrate later in this section.

Processing the segment will be done in a separate class based on the `SampleBySampleDataSegment`. When we start defining this class the initial result will look like this:

```
public class MyWrapper : SampleBySampleDataSegment
{
    public override void ProcessSamples(double[] fSamples)
    {
        throw new Exception(
            "The method or operation is not implemented.");
    }
}
```

The only required override is the `ProcessSamples`, i.e. we must process all samples which is a valid requirement.

Summarized we need to perform the following main actions:

- create a class based on the `SampleBySampleOr Numerical` class
- create a class based on the `SampleBySampleDataSegment`
- find a way to calculate the average value of the original data source

If we assume that we have our average value we can start with coding the `SegmentWrapper` class for our purpose.

```
92 class AverageWrapperSegment : SampleBySampleDataSegment
93 {
94     // this class uses the "SampleBySampleDataSegment"
95     // as base class. Creates a waveform segment that is a DC value
96     // DC = a calculated average value
97
98     // internal member
99     double m_DeAverageValue;
100
101     // constructor, requires a data segment, the average value
102     // and uses base class for "difficult" stuff
103     public AverageWrapperSegment(IDataSegment OriginalSegment,
104     double DeAverageValue) : base(OriginalSegment)
105     {
106         // copy average value to internal
107         m_DeAverageValue = DeAverageValue;
108     }
109
110     public override void ProcessSamples(double[] fSamples)
111     {
112         // override the process samples method.
113         // This one gets a segment of samples in a double array
114         for (int nSample = 0; nSample < fSamples.Length; nSample++)
115         {
116             // replace each sample with the average value
117             fSamples[nSample] = m_DeAverageValue;
118         }
119     }
120 }
```

- **98 - 99**: create a local variable for storage of the average value
- **101 - 103**: the class expects as input the original data segment and the calculated average value. Uses the OriginalSegment class as base.
- **104 - 107**: in the constructor we copy the average value to our internal variable
- **109 - 118**: the ProcessSamples gives us an array of floating point samples that we may use to create our own data. We do this by simply replacing each sample by the average value.

At this point we can start implementing our formula database function "CSI_MeanWave". The function internally is CreateWaveformFromAverage.

```

131 public class CreateWaveformFromAverage :
      SampleBySampleOrNumericalFunction
132 {
133     // this class uses the "SampleBySampleOrNumericalFunction"
134     // it creates a complete waveform that is a DC value
135     // The DC value = a calculated average value
136
137     double m_Average = double.NaN;
138
139     // override the CreateWrapperSegment
140     // it creates an interim data segment, calculates the average
141     // and passes it to the interim data segment
142     protected override SampleBySampleDataSegment CreateWrapperSegment
      (IDataSegment OriginalSegment)
143     {
144         CalculateAverageOfSource();
145         return new AverageWrapperSegment(OriginalSegment, m_Average);
146     }
147
148     private void CalculateAverageOfSource()
149     {
150         if (!double.IsNaN(m_Average))
151             return;
152         IDataSrc myDS = m_Parameters[0];
153         // calculate average here, for testing now = 1
154         m_Average = 1;
155     }
156
157     protected override object ProcessValue(object Value)
158     {
159         // override the ProcessValue method and perform calculation
160         CalculateAverageOfSource();
161         return m_Average;
162     }
163 }

```

- **137**: define the internal variable for the average value and initialize as Not a Number (NaN)
- **142 - 146**: here is the override of the CreateWrapperSegment method
- **144**: a call to calculate the average value
- **145**: return the result of the segment calculation. This is a SampleBySampleDataSegment

created by a new `AverageWrapperSegment` object. Our `AverageWrapperSegment` uses the `OriginalSegment` and the average value as input.

- **148 - 155:** calculate the average value
- **150 - 151:** if our average value is a number, then we're done
- **152 - 154:** for testing purposes so far: fetch the original data source, do nothing and give the average value a temporary value, e.g. 1
- **157 - 162:** implement the `ProcessValue` override as discussed earlier.

We are now ready for another test. We can start using the function. The result is again a straight line, now with a value of "1" and the length is equal to the length of the waveform we use as input.

When you move a cursor, the reading should also be available through the `ProcessValue` implementation (try another value as proof of concept!).

Depending on your data source you might run into trouble with this particular example: when you have a data source that has all values below "1" you will not see the straight line on your display. Therefore you must 'set' the correct display range. This is also true for other calculations. E.g. when you multiply a source with 10, the scaling must also be set to 10 times higher.

As we have seen already in a previous chapter segments provide display range information. When we go to the definition of `SampleBySampleDataSegment` we will find nothing that relates to a display range. However, we see that it is based on `DataSegmentWrapper`. Go to that definition and you will see an override possibility for the display range. (use the Go To Definition function of Visual Studio to see the definition or meta data).

A basic implementation could now be - in the `AverageWrapperSegment` class:

```
public override void DisplayRange( out double DisplayFrom,
    out double DisplayTo )
{
    base.DisplayRange(out DisplayFrom, out DisplayTo);
    DisplayFrom = base.DisplayFrom;
    DisplayTo = base.DisplayTo;
}
```

First we fetch the `DisplayFrom` and `DisplayTo` values from our base segment (the original segment). After this we copy these values onto our own `DisplayFrom` and `DisplayTo` properties of the newly created segment.

For the purpose of our example we could implement the following code:

```
public override void DisplayRange( out double DisplayFrom,
    out double DisplayTo)
{
    base.DisplayRange(out DisplayFrom, out DisplayTo);
    if (DisplayFrom < m_DeAverageValue)
        DisplayFrom = 2 * m_DeAverageValue;
    if (DisplayTo > m_DeAverageValue)
        DisplayTo = -2 * m_DeAverageValue;
}
```

For the calculation of the average value we can use (part of) the code we already have implemented.


```

188 private void CalculateAverageOfSource()
189 {
190     if (!double.IsNaN(m_Average))
191         return;
192
193     // m_Parameters array is part of the inherited Functionbase
194     // m_Parameters[0] is the data source
195     IDataSrc myDS = m_Parameters[0];
196
197     // calculate average
198
199     // use only analog waveforms for the this example
200     if (myDS.DataType != DataSourceDataType.DataSourceDataType_
201         AnalogWaveform)
148         return;
149
202     // initialize variables
203     object Result;
204     IDataSegments Segments;
205
206     // fetch data
207     myDS.Data(myDS.Sweeps.StartTime, myDS.Sweeps.EndTime,
208         out Result);
209
210     // fetch segments within this data
211     Segments = Result as IDataSegments;
212
213     // no segments
214     if (Segments.Count == 0)
215         return;
216
217     // loop through all segments to find out the average value
218     m_Average = 0;
219     int iTotCnt = 0;
220
221     // for all segments
222     for (int j = 1; j <= Segments.Count; j++)
223     {
224         // fetch the data
225         int dCnt = Segments[j].NumberOfSamples;
226         Segments[j].Waveform(DataSourceResultType.
227             DataSourceResultType_Double64, 1, dCnt, 1, out Result);
228         double[] Buffer = (double[])Result;
229
230         // sum all sample values
231         for (int m = 0; m < Buffer.Length; m++)
232         {
233             m_Average = m_Average + Buffer[m];
234         }
235
236         // sum the total number of samples
237         iTotCnt = iTotCnt + dCnt;
238     }
239
240     // calculate the average
241     m_Average = m_Average / iTotCnt;
242 }

```

For a detailed comment refer to the initial code. Apart from some naming differences, the main parts of interest are:

- **150 - 151**: if our average value is a number, then we're done
- **155**: fetch the data source. Our **CreateWaveformFromAverage** class is derived from the **SampleBySampleOrNumericalFunction** class that is derived from the **FirstArgumentFunction** which is derived from the base class **FunctionBase**. Within this base class there is a **m_Parameters[]** array that comprises all passed data sources. Since we only have one parameter we need to address the first one which is **m_Parameters[0]**.
- **168**: fetch all data. For this we need to know the start time and end time of the complete waveform. This information is available through the **Sweeps** property of the data source.

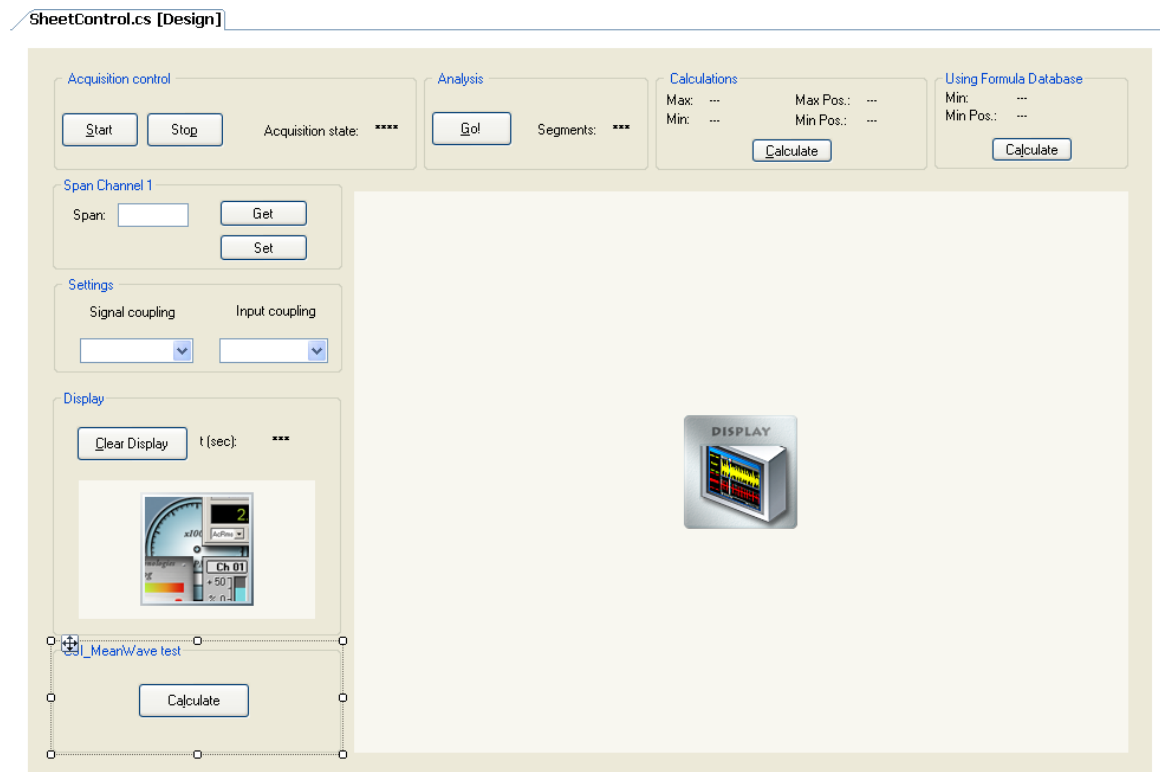
Now we are done with our function implementation.

You could test it by creating a formula in the formula database and compare it with the initial method:

```
test = @CSI_MeanWave(CSIDemo.MyResult)
```

Compare **test** with **MyResultAve**. They should be identical.

The last step would be to verify the new formula by extending our example sheet. To do this we add a new group box with a single button see picture below;



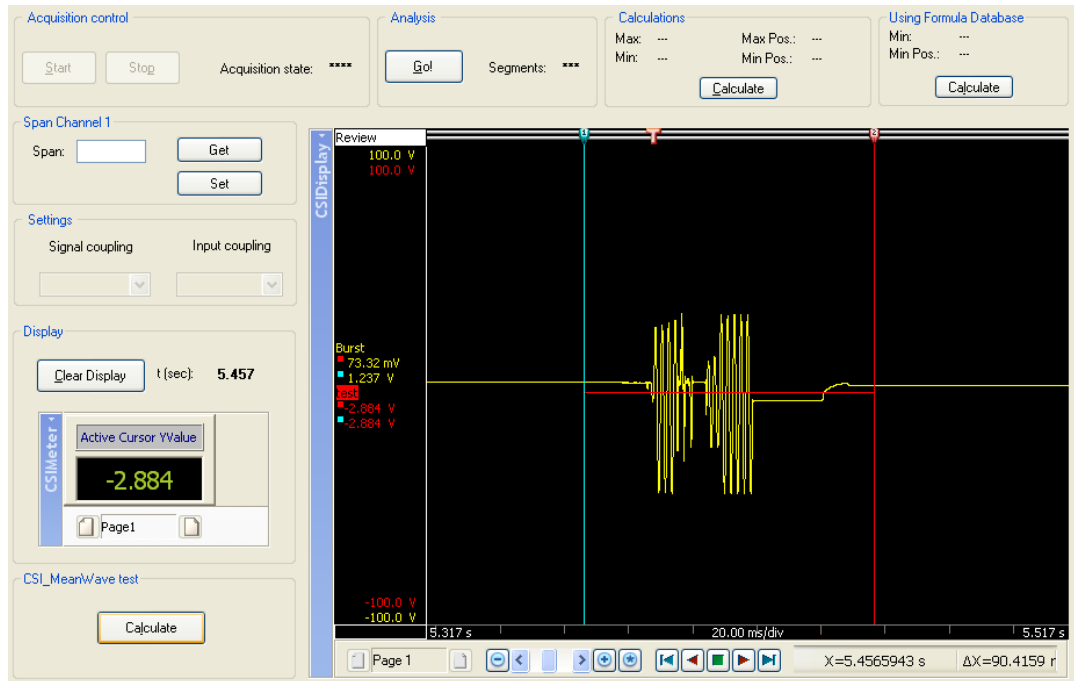
The code behind the button click looks like:

```
1149 private void btnCreateMeanWave_Click(object sender, EventArgs e)
1150 {
1151     // Get Active Trace
1152     DTrace myTrace = display1.pDisplay.TimeDisplay.ActiveTrace;
1153     if (myTrace == null)
1154     {
```

```

1155     PerceptionMessageBox.Show(this, "No active trace found",
1156         "CSI Display Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
1157     return;
1158 }
1159 // Get datasource from active trace
1160 IDataSrc iActiveTrace = myTrace.TraceProp.DataSrc;
1161 if (iActiveTrace == null)
1162 {
1163     PerceptionMessageBox.Show(this, "Active trace has no datasource",
1164         "CSI Display Demo", MessageBoxButtons.OK, MessageBoxIcon.Error);
1165     return;
1166 }
1167 // Get Data source name from trace e.g.:
1168 // "Active.Group1.Recorder_B.Burst"
1169 string cDSName = myTrace.TraceProp.DataSourceName;
1170
1171 // create required formulas in formula database
1172
1173 // create instance of formula database
1174 FormulaDB ForDB = FormulaDB.Instance;
1175
1176 // check if formula already exist
1177 if (ForDB.Formulas["poc"] == null)
1178 {
1179     // search for an empty line
1180     foreach (Formula frml in ForDB.Formulas)
1181     {
1182         if (frml.IsEmpty())
1183         {
1184             // Define Name, Formula and Units (See columns formula
1185             // database sheet)
1186             frml.Name = "poc";
1187             frml.Expression = string.Format("@Cut({0};
1188                 Display.CSIDisplay.Cursor1.XPosition;
1189                 Display.CSIDisplay.Cursor2.XPosition)", cDSName);
1190             ForDB.Formulas[frml.LineNumber + 1].Name = "test";
1191             ForDB.Formulas[frml.LineNumber + 1].Expression =
1192                 "@CSI_MeanWave (Formula.poc)";
1193
1194             // fetch units from source
1195             ForDB.Formulas[frml.LineNumber + 1].Units =
1196                 iActiveTrace.YUnit;
1197             break;
1198         }
1199     }
1200 }
1201 // Add the trace to the display
1202 DTrace TestTrace;
1203 display1.pDisplay.TimeDisplay.CtlLayout.ActivePane.Traces.AddDataSource (
1204     out TestTrace, "Formula.test");
1205 }
    
```

If you compile and run the code you will see the same average trace as before. See picture below.



If you look into the formula sheet then you will see the following formulas:

Num	Name	Formula	Units
1	poc	@Cut(Active.Group1.Recorder_B.Burst; Display.CSIDisplay.Cursor1.XPosition; Display.CSIDisplay.Cursor2.XPosition)	
2	test	@CSI_MeanWave(Formula.poc)	V
3			

Note: since we are using the cursor position in the formulas, moving the cursors will rebuild the cut out segment. When we would have used the actual numbers the result would have been fixed.

In addition: although the cut out segment updates, the average value will not since we use a test:

```

if (!double.IsNaN(m_Average))
    return;
    
```

I.e. when the average value is already calculated we quit. We could omit this test, but then we run into trouble with our multi-tasking, multi-threading software. To overcome this we need a few more lines of code.

In the CreateWaveformFromAverage we need to override the Reset method. This method is called whenever the function needs to 'restart', e.g. when new data arrives, or data is modified. In this method we must reset our average value:

```

m_Average = double.NaN;
    
```

However, when doing so, we may not be interrupted. Therefore:

```

public override void Reset()
{
    lock (this)
    {
        m_Average = double.NaN;
    }
}

```

Also we do not want interruptions when we do the actual calculations. So:

```

protected override SampleBySampleDataSegment CreateWrapperSegment
(IDataSegment OriginalSegment)
{
    lock (this)
    {
        CalculateAverageOfSource();
        return new AverageWrapperSegment(OriginalSegment, m_Average);
    }
}

```

By "locking" pieces of code we are sure that no other thread can interrupt our work while making sensitive calculations.

8.6 Summary

In these last two chapters we have seen a variety of functionality required to implement your own analysis:

- The datamanager as central point of information on all (types of) data.
- Fetch and use existing waveforms, numerical values and strings.
- Create your own waveforms, numerical values and strings and perform some basic math while creating them.
- Measurement cursors manipulation.
- Use the formula database as a waveform calculator.
- Create your own functions to be used within the formula database.

One important choice you will need to make when doing your own analysis: do you want to create new results that become available as "non-volatile" data or are "volatile" formula database interim results acceptable?

The Perception CSI offers both: the choice is yours. And you can always combine both options.

9 Automation

With everything said and done so far only two issues remain to be discussed: reporting and automation. Since reporting by itself is something that cannot be controlled from within the CSI we will discuss in this chapter some automation issues and include command to print out a predefined report.

As an example we will discuss a typical type of application: set the acquisition parameters, start acquisition. After the acquisition do some analysis and print a report.

9.1 Example: post-acquisition analysis and reporting

Our example is based on a real application where an object is tested by dropping a weight on it. Forces on the three axis are measured and the resultant force is calculated. The value of interest is the time the resultant force is above a certain level.

9.1.1 Procedure

The signals of X-, Y- and Z-transducers are each fed to a separate input channel. The recording is made with a sample rate of 10000 samples per second.

The recording has two sweeps: the first one is used as calibration data and triggered just before the actual impact is recorded in the second sweep.

The required calculations can be done in the formula database:

- for each channel subtract the mean value of the first sweep from the second sweep to compensate for any offset,
- calculate the result as the square root of the sum of squares of the three channels,
- measure the time above two predefined levels.

In our example we will also position the cursors on the points of interest.

We will start with this example from scratch. To minimize the work involved by an operator and to reduce the chance on errors we will do as much as possible in our code. We also want to make sure that test information (e.g. a Device Under Test serial number) has been entered before the actual print out is made.

We assume that we are working in a fixed environment with a single recorder with four channels with predefined names.

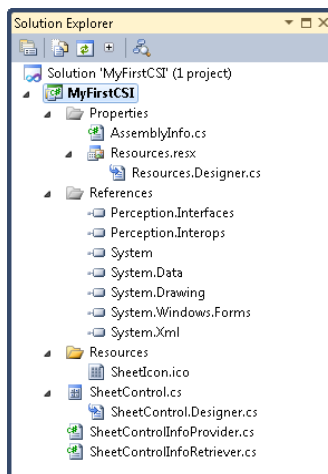
9.1.2 Before you begin

Before you begin create a new project as described earlier. For your convenience here is the procedure with some modifications.

To start you new project proceed as follows:

- 1 Start your Microsoft Visual Studio, and select **File > New > Project**.
- 2 In the dialog that comes up select a **Visual C# Windows** project.
- 3 Select the **Perception CSI Sheet** template
- 4 Enter a name and location for this project and click **OK**

The Solution Explorer will now include the following:



A reference to the Perception.Interfaces

- C# code for the SheetControl

This code is sufficient to create a sheet in Perception. Before we can actually build it we need to add some more information to the project itself.

Optionally give the sheet a name and icon other than default:

- 1 Go to **Project > Project Name Properties**
- 2 Go to **Resources > Strings** and modify the text from IDS_USERNAME into your sheet name
- 3 Go to **Resources > Icons** and select **Add Resource > Add Existing File**. Browse to and select your own icon file.
- 4 Remove the default icon
- 5 Rename your icon into "SheetIcon"

Mandatory:

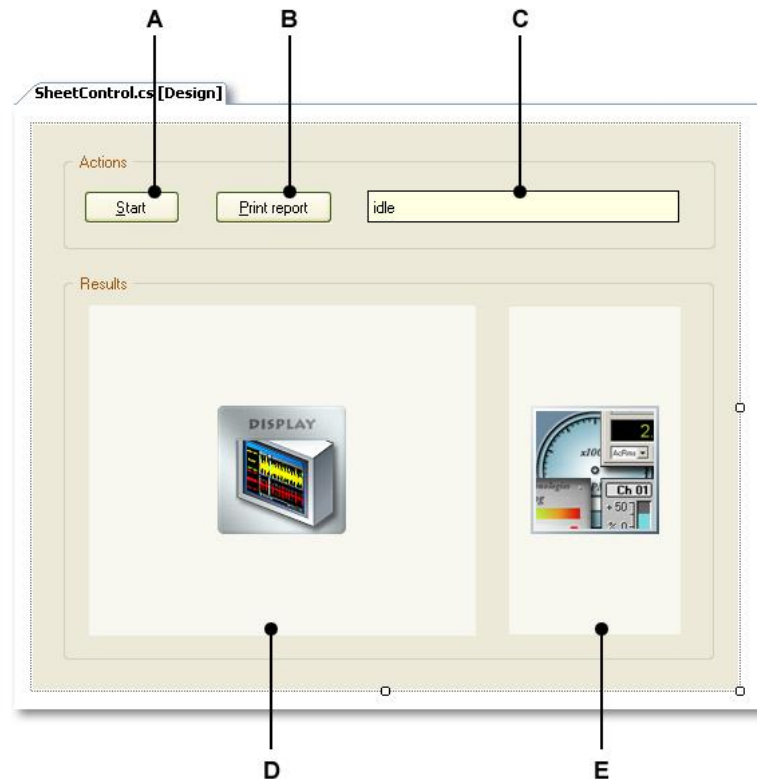
- 1 Go to **Project > Project Name Properties**
- 2 Go to **Build > Output > Output path**
- 3 Verify the output path: **C:\Program Files\HBM\Perception\Sheets**
- 4 Go to **Build > Configuration** and select **Release** or **Active (debug)**. When you want to debug include in **Debug > Start Action** the **Start external program: C:\Program Files\HBM\Perception\Perception.exe**.
- 5 In the main menu select **Build > Build Solution**

When all is OK, no error messages are generated. Ignore warnings for the time being.

When error messages are generated verify all of the above steps. Also make sure you have the latest version of the template and the latest version of Perception.

9.1.3 User interface

For the user interface we need a Start button, a Print Report button (could be done also automatically), a display that shows the three compensated signals as well as the resultant and two meters, one for each level. In addition a status field can be used for various messages. Look in the Data Visualization section how you can add a Display and Meter to the sheet.



- A. Start button **StartCmd**
- B. Print Report button **ReportCmd**
- C. Status messages text label **StatusArea**
- D. Perception display component **ResultDisplay** (Perception.CSI.Support.PerceptionDisplay)
- E. Perception meter component **ResultMeters** (Perception.CSI.Support.PerceptionMeter)

The Perception components should be available in the Toolbox of the Designer. If not so, reload them.

To add the Perception components

- 1 Go to the SheetControl Design layout.
- 2 In the Toolbox select one of the tab headers and do a right mouse click.
- 3 In the context menu that comes up select **Add Tab** and give it a relevant name like "*Perception Components*".

- 4 With this tab selected do a right mouse click.
- 5 In the context menu that come up select **Choose Items ...**
- 6 In the dialog that comes up select **Browse...**
- 7 Navigate to the Perception folder. Typically: **C:\Program Files\HBM\Perception.**
- 8 Select the file **Perception.Components.dll** and click **Open.**
- 9 Click **OK** in the Choose Toolbox Items dialog.

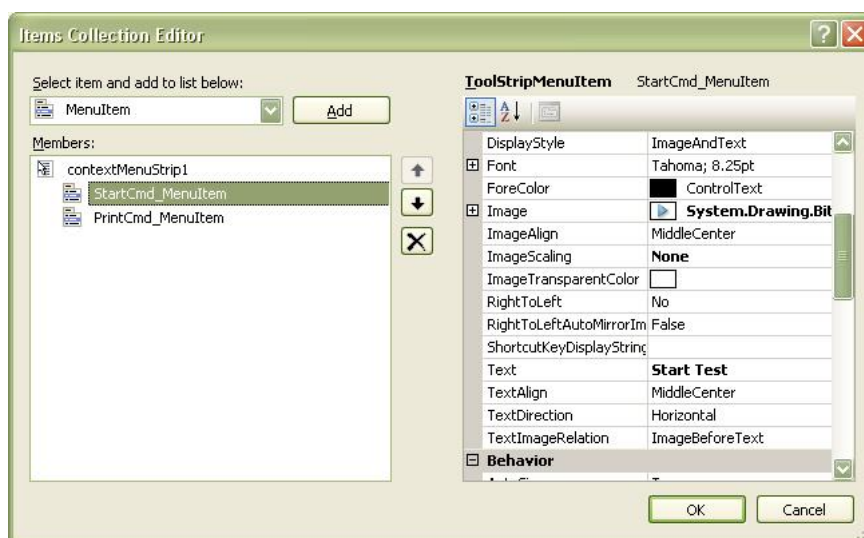
Now the components are added to the toolbox.

Since we are still doing the user interface we can also add at this point the menu entries and the toolbar.

To add items to the dynamic menu you must create a context menu strip in the sheet user interface. This menu is not actually deployed, but used as a piece of menu transferred to the dynamic menu.

To add items to the dynamic menu

- 1 In the programming environment add a tool strip menu to your user interface: **Toolbox > Menus & Toolbars > ContextMenuStrip**
- 2 Use **Edit Items ...** to add/modify menu items



- 3 Modify the GetDynamicmenu method that is located in the Method region of the template (you can copy this from your previous project):

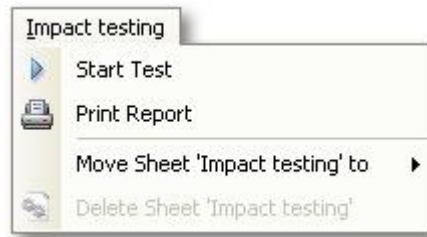
```

public ToolStripItem[] GetDynamicMenu()
{
    ContextMenuStrip strip = contextMenuStrip1;
    if (this.IsDisposed || this.Disposing || strip == null)
        return null;
    ToolStripItem[] Result = new ToolStripItem[strip.Items.Count];
    strip.Items.CopyTo(Result, 0);
    return Result;
}
    
```

Here we create a toolstrip item called **aMyItems** based on the size of the designed

contextMenuStrip. After this we copy the contents of the contextMenuStrip to our toolStrip and return this toolStrip.

4 Run the program



You will see that the new commands are added.

In very much the same way as we created a dynamic menu, you can also create a toolbar related to your sheet. Instead of creating a context menu strip, you now create a toolStrip and modify its contents. Once created you can use the following code:

```
public ToolStripItem[] GetDynamicToolBar()
{
    ToolStrip strip = toolStrip1;
    if (this.IsDisposed || this.Disposing || strip == null)
        return null;
    ToolStripItem[] Result = new ToolStripItem[strip.Items.Count];
    strip.Items.CopyTo(Result, 0);
    return Result;
}
```

Make sure you have set the visibility to false.

9.1.4 The code

At this point the user interface is ready and we can start implementing the code. We start with the acquisition control stuff. As described in one of the earlier chapters acquisition control will be event driven.

9.1.5 Acquisition control

Acquisition control is 'hosted' by the **Engine**. Since we will be using the Engine multiple times we will add a Using Directive at the beginning of the source code where already other directives are placed. Add this directive below the other CSI directives:

```
using Perception.Sheets;
using Perception;
using Engine;
```

Since we want to control an acquisition system we will need to create an acquisition system object:

In the Members region add a region below the ISheet region as follows:

```
#region -> MyMembers

protected CtrlAcquisitionSystem m_MyDemoSystem = null;
protected CtrlGroup m_GroupAll = null;

#endregion
```

We now have a member that is a control of an acquisition system and a control group.

Do some initialization and create the **eventhandler** that reacts on changes of the acquisition state. This will be done in the Initialization method as follows:

```

public InitializeState Initialize(IProgram iProgram)
{
    if (iProgram.UserMode == UserMode.Review)
    {
        this.m_InitializeState = InitializeState.NotAllowed;
    }
    else
    {
        try
        {
            this.m_iProgram = iProgram;
            RegisterComponents();
            this.m_MyDemoSystem = CtrlAcquisitionSystemFactory.Create();
            this.m_GroupAll = m_MyDemoSystem.Groups.GroupAll;
            HookToGroupAll();
            this.m_InitializeState = InitializeState.Succeeded;
        }
        catch
        {
            this.m_InitializeState = InitializeState.NotAllowed;
            PerceptionMessageBox.Show("Could not initialize sheet",
                "CSI: Catch", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    return this.m_InitializeState;
}

```

The Hook and Unhook to group all functions look like:

```

private void HookToGroupAll()
{
    UnHookFromGroupAll();
    if (m_GroupAll != null)
        m_GroupAll.AcquisitionStateChanged +=
            GroupAllAcquisitionStateChanged;
}

private void UnHookFromGroupAll()
{
    if (m_GroupAll != null)
    {
        try
        {
            m_GroupAll.AcquisitionStateChanged -=
                GroupAllAcquisitionStateChanged;
        }
        catch
        {
        }
    }
}

```

Tip: you can copy various pieces of code from the previous project(s).

Now start with the event handler to include the basic code for the start button behaviour and the status text:

```

void GroupAllAcquisitionStateChanged(object sender, int Running, int
OneShot, int Stopping, int Paused, int Idle)
{
    this.InvokeOnUI(() => DoGroupAllAcquisitionStateChanged(sender, Running,
        OneShot, Stopping, Paused, Idle));
}

void GroupAllAcquisitionStateChanged(object sender, int Running, int
OneShot, int Stopping, int Paused, int Idle)
{
    if (Running > 0)
    {
        StatusArea.Text = "Acquisition: running";
        StartCmd.Enabled = false;
        StartCmd_MenuItem.Enabled = false;
        StartCmd_ToolStripItem.Enabled = false;
    }
    else if (OneShot > 0)
    {
        StatusArea.Text = "Acquisition: single shot";
        StartCmd.Enabled = false;
        StartCmd_MenuItem.Enabled = false;
        StartCmd_ToolStripItem.Enabled = false;
    }
    else if (Stopping > 0)
    {
        StatusArea.Text = "Acquisition: stopping";
        StartCmd.Enabled = false;
        StartCmd_MenuItem.Enabled = false;
        StartCmd_ToolStripItem.Enabled = false;
    }
    else if (Paused > 0)
    {
        StatusArea.Text = "Acquisition: pause";
        StartCmd.Enabled = true;
        StartCmd_MenuItem.Enabled = true;
        StartCmd_ToolStripItem.Enabled = true;
    }
    else if (Idle > 0)
    {
        StatusArea.Text = "Acquisition: idle";
        StartCmd.Enabled = true;
        StartCmd_MenuItem.Enabled = true;
        StartCmd_ToolStripItem.Enabled = true;
    }
    else
    {
        StatusArea.Text = "*****";
        StartCmd.Enabled = false;
        StartCmd_MenuItem.Enabled = false;
        StartCmd_ToolStripItem.Enabled = false;
    }
    // Fire the ToolItemsUpdated event to update the toolbar
    if (this.ToolItemsUpdated != null)
        this.ToolItemsUpdated(this, new EventArgs());
}

```

And add some initialization in the SheetControl_Load:

```

private void SheetControl_Load(object sender, EventArgs e)
{
    StatusArea.Text = "****";
    StartCmd.Enabled = false;
    StartCmd_MenuItem.Enabled = false;
    StartCmd_ToolStripItem.Enabled = false;
    ResultMeters.UserName = "No Values";
}

```

Do not forget to Unhook from the group all in the **SheetDisposed()** procedure:

```

private void SheetDisposed(object sender, EventArgs e)
{
    if (IsDisposed) return;
    if (m_bDisposed) return;
    try
    {
        UnRegisterComponents();
        UnHookFromGroupAll();
        m_GroupAll = null;
        m_MyDemoSystem = null;
    }
    catch
    {
    }
    m_bDisposed = true;
}

```

To incorporate the display in our report we must add it to the list of available components in Perception. We will do this in the RegisterComponents() procedure:

```

private void RegisterComponents()
{
    if (m_iPorogram == null) return;
    {
        if (ResultDisplay != null)
        {
            if (ResultDisplay.pDisplay != null)
            {
                m_iProgram.ComponentManager.Add(ResultDisplay);
            }
        }
    }
}

```

When the sheet is removed then call the UnRegisterComponents() procedure

```

private void UnRegisterComponents()
{
    if (m_iPorogram == null) return;
    {
        if (ResultDisplay != null)
        {
            if (ResultDisplay.pDisplay != null)
            {
                m_iProgram.ComponentManager.Remove(ResultDisplay);
            }
        }
    }
}

```

Try this to see if everything works as expected: use the acquisition commands from the acquisition control palette in Perception to scroll through the various acquisition states.

At this point we can implement some more acquisition control:

- when a recorder is added we check if it is 'our' recorder and copy the object
- when the start command button is clicked we set acquisition parameters and start recording.

Add a recorder member to the user member section:

```
#region -> MyMembers
private CtrlAcquisitionSystem m_MyDemoSystem = null;
private CtrlGroup m_GroupAll = null;
private CtrlRecorder m_TheRecorder = null;
#endregion
```

Add the following hooking code the HookToGroupAll procedure:

```
m_GroupAll.RecorderAdded += GroupAllRecorderAdded;
```

And the unhook code in the procedure UnHookFromGroupAll::

```
m_GroupAll.RecorderAdded -= GroupAllRecorderAdded;
```

For the event handler itself start with something like this:

```
void GroupAllRecorderAdded(object sender, CtrlRecorder Recorder)
{
    this.InvokeOnUI(() => DoGroupAllRecorderAdded(sender, Recorder));
}

void DoGroupAllRecorderAdded(object sender, CtrlRecorder Recorder)
{
    // is this the recorder we want ?
    if (Recorder.Name != "Recorder A")
        return;

    m_TheRecorder = Recorder;

    // this should be > 0, but you never know
    if (m_TheRecorder.Channels.Count == 0)
    {
        // error handling here
    }
}
```

An event is generated for each recorder added. E.g. when a system is connected with four acquisition cards (recorders), this event is fired is four times, each time with the recorder object that is added.

We test here for 'our' recorder and copy the object to our local recorder object.

In the **StartCmd_Click** routine we can do now something like:

```
private void StartCmd_Click(object sender, EventArgs e)
{
    // set all required acquisition parameters - example
    StatusArea.Text = "Setting test parameters";
}
```

```

if (m_TheRecorder != null)
{
    m_TheRecorder.Group.SweepCountEnabled = true;
    m_TheRecorder.Group.SweepCount = 2;
    m_TheRecorder.Group.SweepLength = 4000;
    m_TheRecorder.Group.TriggerPosition = 50;
    m_TheRecorder.Group.HighSamplingFrequency = 10000;
}
// start acquisition
m_TheRecorder.Group.Run();
}

```

We start by displaying a status message. This message will be cleared automatically when acquisition actually starts.

Some parameters are set as an example. When done, the acquisition is started.

There also other places and methods to set the acquisition parameters.

Please note the following: in our example we set the various parameters without checking if these are valid settings. E.g. valid sample rates are typically 5000 and 10000 samples per second. If we set the sample rate to 8000 samples per second it may either be clipped to a valid value by the firmware in the acquisition system when the value is entered, or when the acquisition starts.

Fundamentally the only correct option is to interrogate the system's capabilities. This is especially true:

- When we do not know what system will be connected / used
- When we want to support future differences in hardware

You also have to note that some capabilities can change over time. We have seen one example with the signal coupling. Not all signal coupling capabilities are available when an acquisition is active.

Also the sample rate capabilities can change while a system is connected. By default a decimal sequence is used, e.g. 100, 125, 200, 250. However, when you switch the central timebase to binary (power of two), the available sample rates change into 102.4, 128, 204.8, 256.

Therefore it is also wise to implement a "capabilities changed" event handler when you expect this kind of behaviour.

We will keep it as it is for our example.

9.1.6 *Print control*

The second button we need to implement is the print report command button. For this button we could define the following behaviour:

- The Print Report button is enabled when:
 - an acquisition is finished that was started through 'our' Start command and
 - the relevant experiment information is entered in the data pool
- The Print Report button initiates a print command for the loaded report

We will start with implementing a synchronization mechanism through a simple 'flag' called `m_MyTest`.

Define and initialize this flag in the **MyMembers** region:

```
|| private bool m_MyTest = false;
```

Reset this flag in the **SheetControl_Load** routine where we also disable the Print button:

```
|| PrintCmd.Enabled = false;
|| PrintCmd_MenuItem.Enabled = false;
|| PrintCmd_ToolStripItem.Enabled = false;
||
|| m_MyTest = false;
```

When we start an acquisition through our Start button we set the flag to true, but disable the Print command:

```
|| // start acquisition
|| m_TheRecorder.Group.Run();
||
|| PrintCmd.Enabled = false;
|| PrintCmd_MenuItem.Enabled = false;
|| PrintCmd_ToolStripItem.Enabled = false;
||
|| m_MyTest = true;
|| m_Retry = false;
|| ResultDisplay.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors.Visible = false;
```

The rest of the synchronization is done in the acquisition state changed event handler. For every change in acquisition state the print button is disabled, unless the state is idle with `m_MyTest` true:

```
|| // for any acquisition state change disable print
|| // unless correctly finished
|| PrintCmd.Enabled = false;
|| PrintCmd_MenuItem.Enabled = false;
|| PrintCmd_ToolStripItem.Enabled = false;
||
|| if (Running > 0)
|| {
||     ...
|| }
|| else if (Idle > 0)
|| {
||     if (m_MyTest == true)
||     {
||         StatusArea.Text = "Test completed";
||         PrintCmd.Enabled = true;
||         PrintCmd_MenuItem.Enabled = true;
||         PrintCmd_ToolStripItem.Enabled = true;
||
||         m_MyTest = false;
||     }
||     else
```



```

    {
        StatusArea.Text = "Acquisition: idle";
    }
    StartCmd.Enabled = true;
    StartCmd_MenuItem.Enabled = true;
    StartCmd_ToolStripItem.Enabled = true;
}

```

For the actual implementation of the Print Report command we need to add a reference to the **Perception.CSI.Support.dll**. Add this reference as usual.

Once done add a "using" clause in the Using directives region:

```

| using Perception.CSI.Support.Sheets;

```

This will add additional support for sheets.

The **PrintCmd_Click** routine now becomes very simple:

```

| private void PrintCmd_Click(object sender, EventArgs e)
| {
|     Reporting.Print();
| }

```

What's on the report we print? Typically some standard text, company logo, table with results and data from the display.

To incorporate the display in our report we must add it to the list of available components in Perception.

To do this, add the following two lines to the **SheetControl_Load** routine:

```

| ResultDisplay.UserName = "Impact Result";
| m_iProgram.ComponentManager.Add(ResultDisplay);

```

First make sure that the display has a unique name, then add the display object/component to the component manager.

Now the display is selectable from within the Report sheet.

To complete the user interface of our automation we still need to verify if user information is present before we print.

In general the Information Sheet is used to enter information into the system. For our example we will be satisfied with a single variable called DUTSerial, the serial number of the device Under Test, entered as a string.

To be able to investigate a pool entry we must perform the following steps as we have done before.

Add a using statement:

```

| using DataSrcManager;

```

Add a member in the MyMembers region:

```
|| protected DataManager m_ThisDataManager = null;
```

Initialize in the constructor of the SheetControl:

```
public SheetControl()
{
    InitializeComponent();

    if (m_ThisDataManager == null)
    {
        m_ThisDataManager = new DataManager();
    }
}
```

Now we can test. The test will be performed in the AcquisitionStateChanged where we already did some additional programming in the "idle" section.

```
else if (Idle > 0)
{
    if (m_MyTest == true)
    {
        // correctly finished, but serial number?
        if (m_ThisDataManager.PoolEntries[
            "Active.Information.DUTSerial"].DataSource == null)
        {
            StatusArea.Text =
                "Please create serial number entry DUTSerial";
            m_Retry = true;
        }
        else
        {
            // variable exists
            string sDUT = m_ThisDataManager.PoolEntries[
                "Active.Information.DUTSerial"].DataSource.Value.ToString();

            // some example testing, make your own
            if (sDUT.Length < 8)
            {
                StatusArea.Text = "Please enter a correct serial number";
                m_Retry = true;
            }
            else
            {
                StatusArea.Text = "Test completed for device: " + sDUT;
                PrintCmd.Enabled = true;
                PrintCmd_MenuItem.Enabled = true;
                PrintCmd_ToolStripItem.Enabled = true;
            }
        }
    }
    m_MyTest = false;
}
else
{
    StatusArea.Text = "Acquisition: idle";
}
StartCmd.Enabled = true;
StartCmd_MenuItem.Enabled = true;
```

```

    StartCmd_ToolStripItem.Enabled = true;
  }

```

For the time being forget about the `m_Retry` flag. What happens is:

Test 1: if `m_MyTest` is not true we have an 'illegal' situation, enable the Start buttons only and place the text "Acquisition : idle".

Test 2: if it is a legal situation test if there is a serial number. If not, display a message and raise a flag.

Test 3: there is a serial number. When not correct, display a message and raise a flag.

When all tests are passed correctly, a message is displayed including the serial number and the Print report command button is enabled.

What should we do when no serial number or an incorrect serial number was available? The experiment itself may be a success and doing it all over is not an option.

In both situations we need to go to the Info sheet in Perception and resolve this issue. But then we also need to have a means to 'inform' our sheet that the value has been added or modified and the Print command may be enabled after all.

For this we use a hook to an event: when the data source in question has been changed, an event will be fired and an event handler is used to cope with the new situation.

We have done this already a number of times, so it should be not to difficult.

Add some members to the **MyMembers** region. This region now should like this:

```

#region -> MyMembers

protected CtrlAcquisitionSystem m_MyDemoSystem = null;
protected CtrlGroup m_GroupAll = null;
protected CtrlRecorder m_TheRecorder = null;
protected DataManager m_ThisDataManager = null;

protected bool m_MyTest = false;
protected bool m_Retry = false;
protected PoolEntry m_DUTSerialPEntry = null;

#endregion

```

The flag `m_Retry` is used to synchronise retry options. The `m_DUTSerialPEntry` is used to make a sticky pool entry that we use for the event handler.

In the initialize routine add support for the `DUTSerialPEntry`:

```

// support pool entry
m_DUTSerialPEntry =
    m_ThisDataManager.PoolEntries["Active.Information.DUTSerial"];
HookToPoolEntries();

```

The hooking and unhooking procedures for the pool entries look like:

```

private void HookToPoolEntries()
{
    UnHookFromPoolEntries();
    if (m_DUTSerialPEntry != null)
    {
        m_DUTSerialPEntry.DataSourceChanged +=

```

```

        DUTSerialPEntree_DataSourceChanged;
        m_DUTSerialPEntree.DataChanged += DUTSerialPEntree_DataChanged;
    }
}

private void UnHookFromPoolEntries()
{
    if (m_DUTSerialPEntree != null)
    {
        try
        {
            m_DUTSerialPEntree.DataSourceChanged -=
                DUTSerialPEntree_DataSourceChanged;
            m_DUTSerialPEntree.DataChanged -= DUTSerialPEntree_DataChanged;
        }
        catch
        {
        }
    }
}
}

```

First we initialize the member. Then we create two event handlers: one for the "data source changed" and one for the "data changed". The first one will be fired when the variable is created, the second one if data changes.

The event handlers themselves are very straightforward.

```

void DUTSerialPEntree_DataSourceChanged()
{
    this.InvokeOnUI(() => DoDUTSerialPEntree_DataSourceChanged());
}

void DoDUTSerialPEntree_DataSourceChanged()
{
    // the pool entry for the DUTSerial has been changed
    // was there a request that we should handle?
    if (m_Retry == false)
        return;

    // yes, do the standard test
    // correctly finished, but serial number?
    if ((m_DUTSerialPEntree.DataSource == null) ||
        (m_DUTSerialPEntree.DataSource.Value == null))
    {
        StatusArea.Text =
            "Please create serial number entry DUTSerial";
        m_Retry = true;
    }
    else
    {
        // variable exists
        string sDUT = m_DUTSerialPEntree.DataSource.Value.ToString();

        // some example testing, make your own
        if (sDUT.Length < 8)
        {
            StatusArea.Text = "Please enter a correct serial number";
            m_Retry = true;
        }
        else
        {
            StatusArea.Text = "Test completed for device: " + sDUT;
            PrintCmd.Enabled = true;
        }
    }
}

```

```

        PrintCmd_MenuItem.Enabled = true;
        PrintCmd_ToolStripItem.Enabled = true;
        m_Retry = false;
    }
}
m_MyTest = false;
}

```

When the data source is changed do the complete test as usual, only when the m_Retry flag is set.

The "data changed" event is even simpler since we now already know that the data source itself exists.

```

void DUTSerialPEEntry_DataChanged()
{
    this.InvokeOnUI(() => DoDUTSerialPEEntry_DataChanged());
}

void DoDUTSerialPEEntry_DataChanged()
{
    // the data in the pool entry DUTSerial has been changed
    // this value is always transferred to our status
    // we know the entry exists, so only verify value
    string sDUT = m_DUTSerialPEEntry.DataSource.Value.ToString();

    // some example testing, make your own
    if (sDUT.Length < 8)
    {
        StatusArea.Text = "Please enter a correct serial number";
        m_Retry = true;
    }
    else
    {
        StatusArea.Text = "Test completed for device: " + sDUT;
        PrintCmd.Enabled = true;
        PrintCmd_MenuItem.Enabled = true;
        PrintCmd_ToolStripItem.Enabled = true;
        m_Retry = false;
    }
    m_MyTest = false;
}

```

We're almost done. One last issue: when you switch from one sheet to another the StatusArea is not updated.

Go to the UIState properties and modify the "set" as follows:

```

set
{
    this.m_UIState = value;
    if (this.m_UIState == UIState.Active)
    {
        StatusArea.Refresh();
    }
}

```

When the sheet becomes active, the StatusArea will be refreshed.

To test start your project in debug mode and Perception. Go to your sheet and connect real hardware or the simulator. Press the start command button and initiate two triggers. The acquisition will stop and the top of your sheet will look like this:

Now go to the information sheet and add a line. On that line add a string Field. The corresponding properties dialog will come up. Enter the required information, but make sure that you enter less than 8 characters in the default value text box:

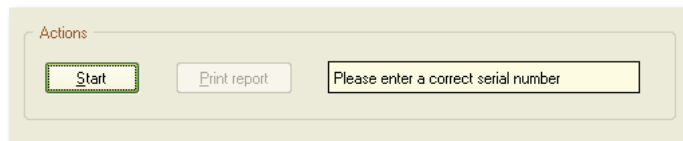


The dialog box titled "Properties of Line 3 - String" contains the following fields and options:

- Text:** Serial number: [Text box]
- Variable:** DUTSerial [Text box]
- Units:** [Text box]
- Default value:** 1234 [Text box]
- Field description:** Enter the serial number of the device under test [Text box]
- Required
- Read-only
- Persistent
- Limits:**
 - Check limits
 - Minimum length: 0 [Spin box]
 - Maximum length: 1000 [Spin box]

Buttons: OK, Cancel

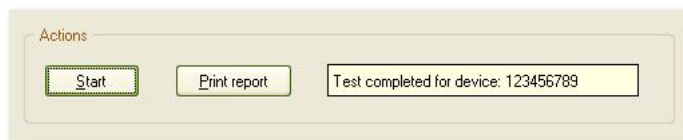
Go to the Impact Testing sheet. You will now see:



The Actions panel shows:

-
-
- Message box: Please enter a correct serial number

Return to the information sheet and enter a serial number of at least 8 digits and return to the Impact Testing sheet:



The Actions panel shows:

-
-
- Message box: Test completed for device: 123456789

The test is now completed and you can print your report.

So far for the sequence of the automation. In the next section we will deal with the calculations.

To make sure you clean up the event hooking and other housekeeping activities we recommend you to look at the following **SheetDisposed()** code:

```

#region Disposing
/// <summary>
/// Procedure to do your cleanup, this function is called from within the
/// Dispose() method.
/// </summary>
private void SheetDisposed(object sender, EventArgs e)
{
    if (IsDisposed) return;
    if (m_bDisposed) return;

    try
    {
        UnHookFromPoolEntries();
        UnRegisterComponents();
        m_DUTSerialPEntry = null;
        UnHookFromGroupAll();
        m_GroupAll = null;
        m_T1PEntry = null;
        m_T2PEntry = null;
        m_deltaT80 = null;
        m_MyDemoSystem = null;
        m_TheRecorder = null;
        m_ThisDataManager = null;
    }
    catch
    {
    }
    m_bDisposed = true;
}
#endregion

```

9.1.7 Calculations

The calculations themselves will be done by using the formula database functions. These functions provide all the power we need. Should you need extra functions you can design and use them as described earlier. The end result within the formula database will look like this:

9			
10	X	@Sweep(Active.Group1.Recorder_A.Ch_A1; 2) - @Mean(@Sweep(Active.Group1.Recorder_A.Ch_A1; 1))	
11	Y	@Sweep(Active.Group1.Recorder_A.Ch_A2; 2) - @Mean(@Sweep(Active.Group1.Recorder_A.Ch_A2; 1))	
12	Z	@Sweep(Active.Group1.Recorder_A.Ch_A3; 2) - @Mean(@Sweep(Active.Group1.Recorder_A.Ch_A3; 1))	
13			
14	Resultant	@Sqrt(@Pow(Formula.X; 2) + @Pow(Formula.Y; 2) + @Pow(Formula.Z; 2))	
15			
16	ResultMax	@Max(Formula.Resultant)	
17	Ref80	0.8 * Formula.ResultMax	
18	Ref60	0.6 * Formula.ResultMax	
19			
20	T1	@NextLwCross(Formula.Resultant; 0; Formula.Ref80; 1)	
21	T2	@NextLwCross(Formula.Resultant; Formula.T1; Formula.Ref80; -1)	
22	deltaT_80	Formula.T2 - Formula.T1	\$
23	T3	@NextLwCross(Formula.Resultant; 0; Formula.Ref60; 1)	
24	T4	@NextLwCross(Formula.Resultant; Formula.T3; Formula.Ref60; -1)	
25	deltaT_60	Formula.T4 - Formula.T3	\$
* 26			

We will "load" the functions from within our code.

Lines 10, 11 and 12 are used to correct for any offset that might be available.

In **line 14** the resultant is calculated as the square root of the sum of squares.

Lines 16 through 18 are defined here to calculate two levels: 60% and 80% of the maximum value.

In **line 20** the first crossing of the resultant with the 80% level is searched, starting from the beginning. The crossing should go in the positive direction.

In **line 21** the second level crossing is searched, starting at the position of the first crossing and should go in the negative direction.

Line 22 calculates the difference.

In **lines 23 through 25** the same is done for the 60% level crossing.

In our code we need to implement this. We can do it on several locations. For this example we will do it when our recorder is connected.

The code itself is very straightforward. Before you start make sure you have added a reference to the Perception.FormulaDatabase.dll in the project references.

Also make sure when entering the formula text that you use a semicolon as separator, not a comma.

```
// create required formulas in formula database
// create instance of formula database
FormulaDB ForDB = FormulaDB.Instance;

// add formulas at fixed location
// compensate X, Y and Z with mean of first sweep
```



```

ForDB.Formulas[10].Name = "X";
ForDB.Formulas[11].Name = "Y";
ForDB.Formulas[12].Name = "Z";

ForDB.Formulas[10].Expression = "@Sweep(Active.Group1.Recorder_A.Ch_A1; 2)
- @Mean(@Sweep(Active.Group1.Recorder_A.Ch_A1; 1))";
ForDB.Formulas[11].Expression = "@Sweep(Active.Group1.Recorder_A.Ch_A2; 2)
- @Mean(@Sweep(Active.Group1.Recorder_A.Ch_A2; 1))";
ForDB.Formulas[12].Expression = "@Sweep(Active.Group1.Recorder_A.Ch_A3; 2)
- @Mean(@Sweep(Active.Group1.Recorder_A.Ch_A3; 1))";

// calculate the result
ForDB.Formulas[14].Name = "Resultant";
ForDB.Formulas[14].Expression = "@Sqrt(@Pow(Formula.X; 2) +
@Pow(Formula.Y; 2) + @Pow(Formula.Z; 2))";

// for the time being use these values as the reference values
ForDB.Formulas[16].Name = "ResultMax";
ForDB.Formulas[16].Expression = "@Max(Formula.Resultant)";

ForDB.Formulas[17].Name = "Ref80";
ForDB.Formulas[17].Expression = "0.8 * Formula.ResultMax";

ForDB.Formulas[18].Name = "Ref60";
ForDB.Formulas[18].Expression = "0.6 * Formula.ResultMax";

// calculate time parameters
ForDB.Formulas[20].Name = "T1";
ForDB.Formulas[20].Expression = "@NextLvlCross(Formula.Resultant; 0;
Formula.Ref80; 1)";

ForDB.Formulas[21].Name = "T2";
ForDB.Formulas[21].Expression = "@NextLvlCross(Formula.Resultant;
Formula.T1; Formula.Ref80; -1)";

ForDB.Formulas[22].Name = "deltaT_80";
ForDB.Formulas[22].Expression = "Formula.T2 - Formula.T1";
ForDB.Formulas[22].Units = "s";

ForDB.Formulas[23].Name = "T3";
ForDB.Formulas[23].Expression =
"@NextLvlCross(Formula.Resultant; 0; Formula.Ref60; 1)";

ForDB.Formulas[24].Name = "T4";
ForDB.Formulas[24].Expression = "@NextLvlCross(Formula.Resultant;
Formula.T3; Formula.Ref60; -1)";

ForDB.Formulas[25].Name = "deltaT_60";
ForDB.Formulas[25].Expression = "Formula.T4 - Formula.T3";
ForDB.Formulas[25].Units = "s";
    
```

We can now connect the calculated waveforms to the display. First we will clear the display.

```

// clear display
for (int j = ResultDisplay.TimeDisplay.CtlLayout.Pages.Count;
     j > 0; j--)
{
    ResultDisplay.pDisplay.TimeDisplay.CtlLayout.Pages[j].Delete();
}
    
```

```

}
ResultDisplay.pDisplay.TimeDisplay.AddPage().Activate();

```

Connect the results:

```

// connect these waveforms to the display

ResultDisplay.pDisplay.TimeDisplay.CtlLayout.Pages.
    ActivePage.Panes[1].Activate();
string[] aPoolEntry = new string[] { "Formula.X" };
ResultDisplay.pDisplay.AddDataSources(aPoolEntry);
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.ActiveTrace.
    TraceProp.PrimaryColor = 0x0000FF; // red
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.Pages.ActivePage.
    Panes.AddPane().Activate();
aPoolEntry[0] = "Formula.Y";
ResultDisplay.pDisplay.AddDataSources(aPoolEntry);
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.ActiveTrace.
    TraceProp.PrimaryColor = 0xFFFFFF; // white
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.Pages.ActivePage.
    Panes.AddPane().Activate();
aPoolEntry[0] = "Formula.Z";
ResultDisplay.pDisplay.AddDataSources(aPoolEntry);
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.ActiveTrace.
    TraceProp.PrimaryColor = 0xFFFF00; // blue
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.Pages.ActivePage.
    Panes.AddPane().Activate();
aPoolEntry[0] = "Formula.Resultant";
ResultDisplay.pDisplay.AddDataSources(aPoolEntry);
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.ActiveTrace.
    TraceProp.PrimaryColor = 0x00FFFF; // yellow

```

Set some display properties:

```

// set display
ResultDisplay.ReviewType =
    TimeView.ReviewModeType.ReviewModeType_Recording;
// set pane height
double[] PaneArray = new double[4] { 0.2, 0.2, 0.2, 0.4 };
object PaneParams = PaneArray;
ResultDisplay.pDisplay.TimeDisplay.CtlLayout.ActivePage.Panes.SetPaneHeights
(
    ref PaneParams);

```

First we set the review type, then we modify the pane height. To modify the pane height you must create an array that sets the height of all panes. The pane height ranges from 0 (0%) to 1 (100%) of total display size.

Then an object is created from that array and passed to the SetPaneHeights method by reference.

We are done now with our calculations. However, there is more that we want to do:

- Set the two measurement cursors on the points of interest
- Set the two horizontal cursors on the 60% and 80% levels as reference

We can set the cursors when correct data is available. To verify this we need to create event handlers for the T1 and T2 variables. Follow the standard procedure to create these event handlers.

Create the members:

```
protected PoolEntry m_T1PEntry = null;
protected PoolEntry m_T2PEntry = null;
protected PoolEntry m_deltaT80 = null;
```

Create the event handlers:

```
m_T1PEntry = m_ThisDataManager.PoolEntries["Formula.T1"];
m_T1PEntry.DataChanged += new
    _IPoolEntryEvents_DataChangedEventHandler(m_T1PEntry_DataChanged);
m_T2PEntry = m_ThisDataManager.PoolEntries["Formula.T2"];
m_T2PEntry.DataChanged += new
    _IPoolEntryEvents_DataChangedEventHandler(m_T2PEntry_DataChanged);
m_deltaT80 = m_ThisDataManager.PoolEntries["Formula.deltaT_80"];
m_deltaT80.DataChanged += new
    _IPoolEntryEvents_DataChangedEventHandler(m_deltaT80_DataChanged);
```

The code of the event handler for T and T2 are basically the same. Therefore only the T1 will be described.

In the routine the following actions will be done:

- Verify if there is a valid value.
- If so, position the cursor.
- When also the other cursor position is valid, expand the time view to show more of the point of interest.
- Set the Y-range for optimal expansion
- Position both horizontal cursors

```
328 void T1PEntry_DataChanged()
329 {
330     this.InvokeOnUI(() => DoT1PEntry_DataChanged());
331 }
332
333 void DoT1PEntry_DataChanged()
334 {
335     if (m_T1PEntry == null)
336         return;
337
338     double dTP1Time = (double) m_T1PEntry.DataSource.Value;
339     double dTP2Time = (double) m_T2PEntry.DataSource.Value;
340
341     if (double.IsNaN(dTP1Time))
342         return;
343
344     // position cursor
345     ResultDisplay.pDisplay.TimeDisplay.Cursors[1].time = dTP1Time;
346
347     // if also TP2 is true: move visible window
```

```

347     if (double.IsNaN(dTP2Time))
348         return;
349
350     double delta = dTP2Time - dTP1Time;
351     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.TimeController.CentralTime =
352         dTP1Time + (delta / 2.0);
353     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.TimeController.StartTime =
354         dTP1Time - (delta / 3.0);
355     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.TimeController.EndTime =
356         dTP2Time + (delta / 3.0);
357
358     double dMaxVal = (double) m_ThisDataManager.PoolEntries
359         ["Formula.ResultMax"].DataSource.Value * 1.2;
360     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.ActiveTrace.TraceProp.
361         SetRangeFromTo(dMaxVal, 0.0);
362
363     double dHorCurPos = (double) m_ThisDataManager.
364         PoolEntries["Formula.Ref80"].DataSource.Value * (0.4 / dMaxVal);
365     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors.Visible
366         = true;
367     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors[1].Location
368         = dHorCurPos;
369     ResultDisplay.pDisplay.TimeDisplay.CtlLayout.HorizontalCursors[2].Location
370         = dHorCurPos * 0.75;
371 }

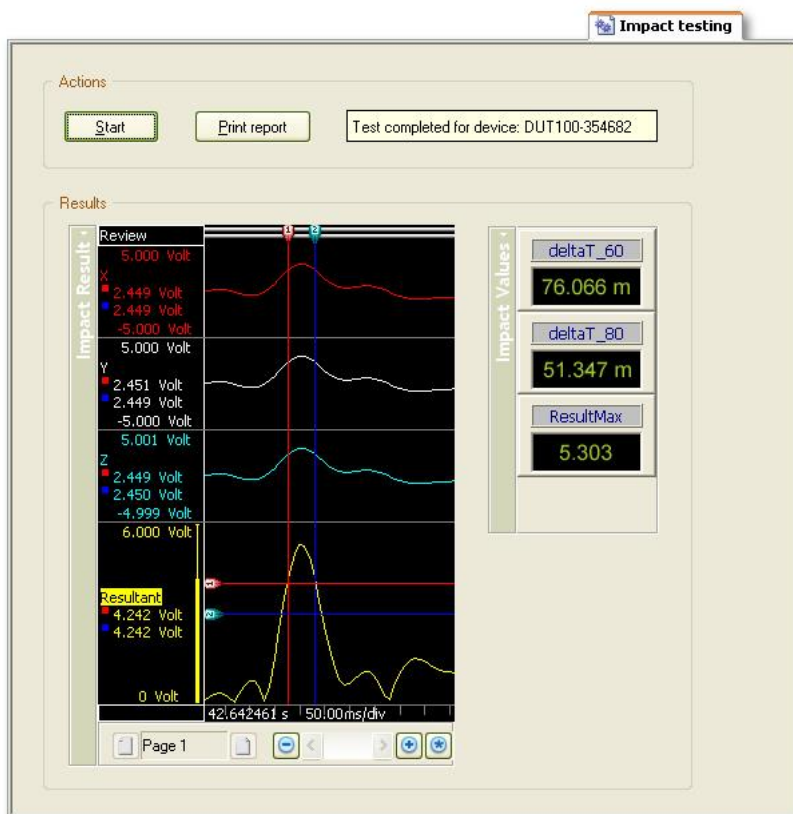
```

- **330 - 334:** standard for these event handlers
- **334 - 335:** check if entry is valid
- **337 - 338:** fetch values
- **340 - 341:** check if value is valid
- **343 - 344:** position the cursor
- **346 - 348:** check if other value is also valid
- **351 - 353:** modify and position time window. Center between the two cursors and make window approximately 1.6 times difference between cursors.
- **355 - 356:** set Y-range of trace from 0 to 1.2 times maximum value
- **358 - 361:** position the horizontal cursors. The pane size is 0.4 times the display size and has a range of 0 - dMaxval. The first horizontal cursor is at 80% (0.8) of the maximum of the resultant. The position of the horizontal cursor is in range 0 (0% = bottom) to 1 (100% = top) of total display size.

Therefore the position is Y-value in display range times pane height:
 $(\text{Ref80} / \text{dMaxVal}) * 0.4$

The second cursor is $\text{Ref60} = \text{Ref80} * 75\%$

To test all this you will need to improvise a little bit. You could use a microphone connected to three channels and tap on it to simulate an impact. For this you might need to modify sample rate, sweep length and filter settings. Also you might need to modify in software the test levels and display positioning for the best results.



In the above image you see an actual recording made as described with a microphone.

9.2 Points of consideration

In this example we have used the formula database for our calculations. Unfortunately the formula database updates the results each time new data arrives and a result is requested, e.g. by a display. Therefore our application will respond very slowly. After each sweep the complete calculations must be done. To overcome this problem we could plan to add the formulas to the database after the recording has been made. Since this application uses only two sweeps with fixed sample rate, i.e. there are no segments, the calculations could also be done internally in the code to speed things up. Numerous schemes exist.

In addition it would also be a possibility to create a user waveform from the Resultant and save it with the complete experiment. With multiple results you could do statistical analysis.

So far we did not discuss the meters. Currently the meter support in CSI is limited therefore automation is not easy. You could drag the results after the test into the meters.

Or you can add the sources when the results are valid. For this you need again an event handler, e.g. hooked to the `deltaT80_changed`. Example:

```

void DeltaT80_DataChanged()
{
    this.InvokeOnUI(() => DoDeltaT80_DataChanged());
}

void DoDeltaT80_DataChanged()
{
    if (m_deltaT80 == null)
        return;

    double ddelta = (double)m_deltaT80.DataSource.Value;
  
```

```
if (double.IsNaN(ddelta))
    return;

// set meter name and connect meter to the correct pool entry
if (ResultMeters.UserName == "Impact Values")
    return;

ResultMeters.UserName = "Impact Values";
string[] sPoolEntry = new string[] { "Formula.deltaT_60",
    "Formula.deltaT_80", "Formula.ResultMax" };
ResultMeters.pMeter.LDSMeter.AddDataSources(sPoolEntry);
}
```

As mentioned earlier this would slow down the overall test progress when the meters are connected and displayed. Removing the meters automatically, however, is not possible. You will need to this manually.

10 User-key script action

Perception comes with many pre-defined script actions which can be used by defining the functionality of a user-key. However there might be situations where you want to add new functionality behind a user key. This can be done via CSI. You can program a new script action. This paragraph will show you how this can be done.

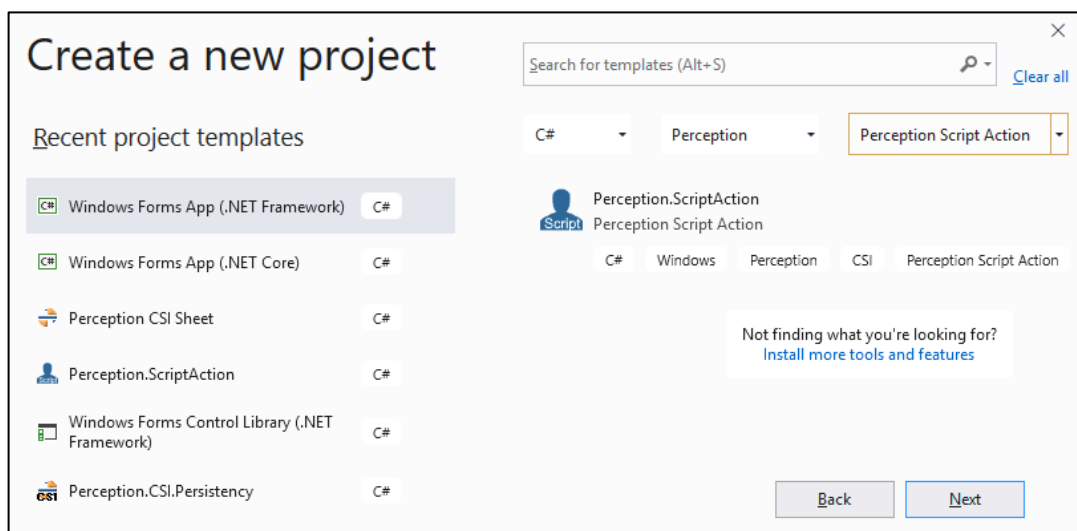
10.1 Perception.ScriptAction

Just like for the CSI user-sheet a **Perception.ScriptAction** template is available to make it easy to create your own user-key script action.

To verify this you should start your Microsoft Visual Studio and create a new project. The selection criteria should be:

- c#
- Perception
- Perception Script Action

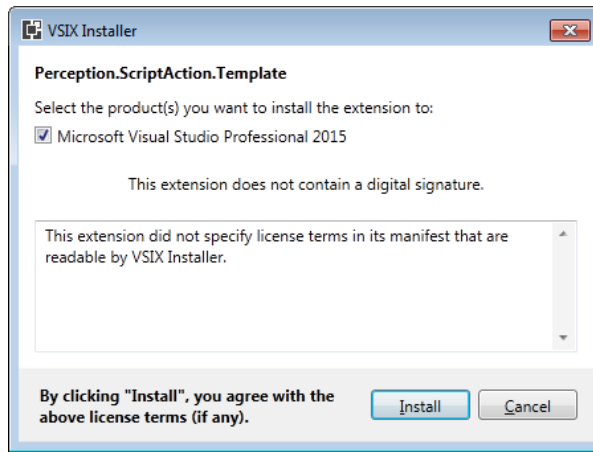
The screen should now look like:



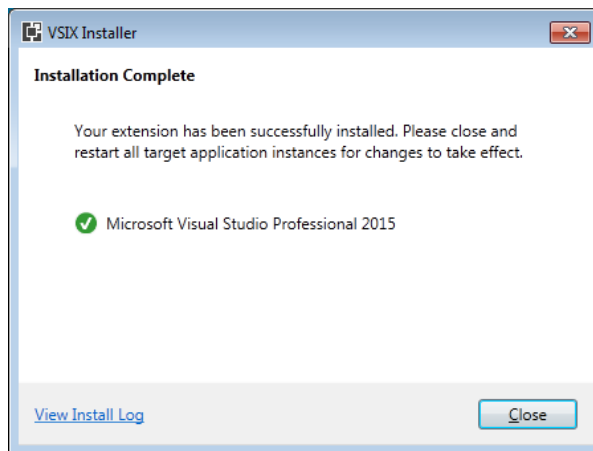
In the Template section, the **Perception.ScriptAction** should be available. If not so, proceed as described below.

10.1.1 To load the Perception CSI template

1. Locate the file named **Perception.ScriptAction.Template.vsix**
2. Double-click this file. The VSIX installer will be launched



3. Click **Install** to install

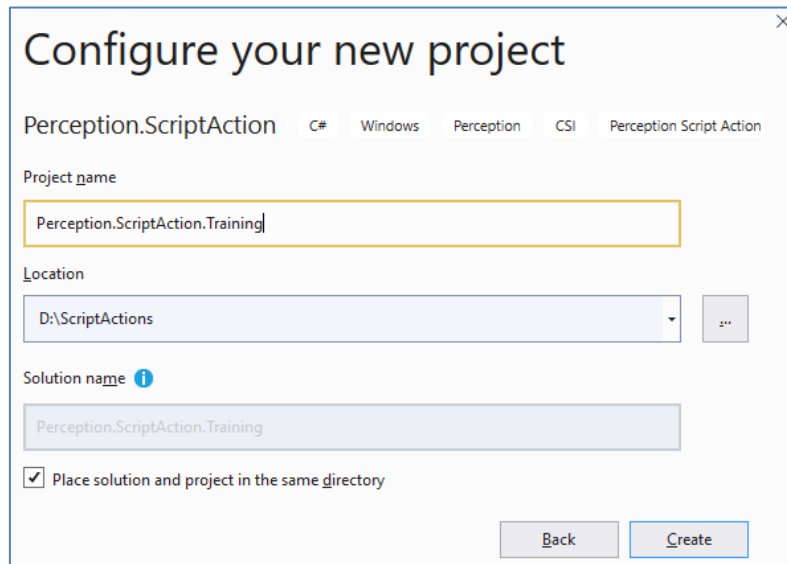


4. Click Close

10.2 Your first User key Script action

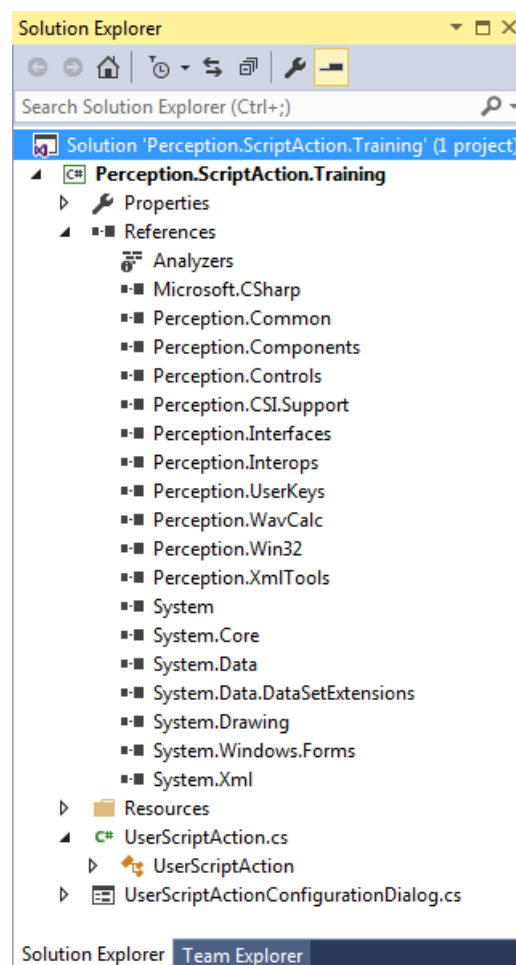
You now should be able to create, compile and run your first user-key script action. To do so proceed as follows:

1. Start your Microsoft Visual Studio and select File > New > Project.
2. In the dialog that comes up select a Visual C# Perception project.
3. Select the **Perception.ScriptAction** template



4. Enter a name **Perception.ScriptAction.Training** and location for this project
5. Click OK

The Solution Explorer will now include the following:



- References to Perception interfaces

- C# code for the script action

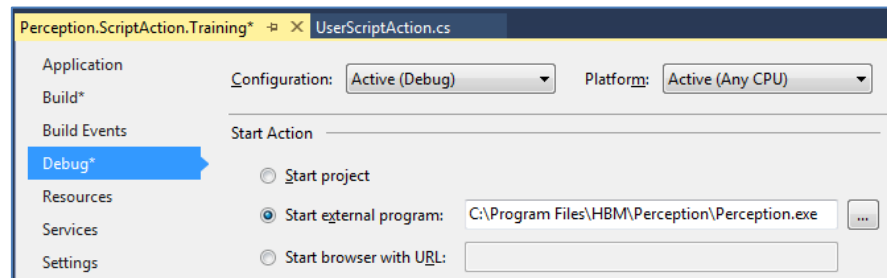
This code is enough to create a script action.

- Compile the generated code without any modification
- A dll called **Perception.ScriptAction.Training.dll** is created and saved in the folder

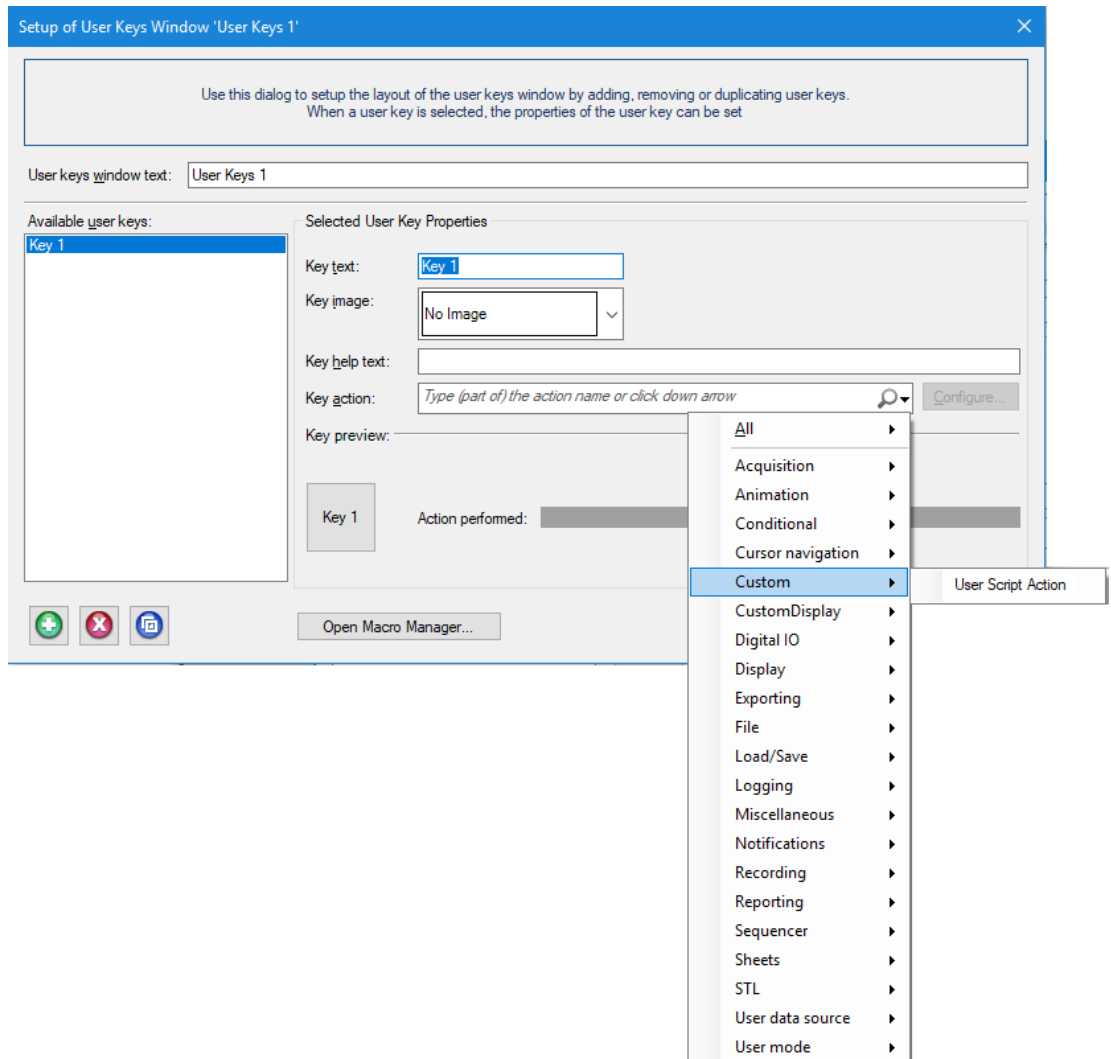
C:\Program Files\HBM\Perception\ScriptActions

Note: This is an example location; however Perception will look during start-up in the **..HBM\Perception** folder and all its subfolders if it can find dll's which implement the **IScriptActionInfo** and **IScriptAction** interfaces. All the dll's found will then be loaded. Continue reading to get more information on those interfaces.

- Check if Perception.exe is used to debug the script dll:



- Run Perception, create a new user key and link the new script action to this user key



- Test and debug the new script action.

Now we will go into the details of the generated code.

A file called **UserScriptAction.cs** has been generated. This file contains the class called **UserScriptAction**. This class implements the interfaces **IScriptAction**, **IScriptActionInfo** and **IConfigurable**.

10.2.1 *IScriptActionInfo*

This interface gives information about the script action:

```
public interface IScriptActionInfo
{
    string Category { get; }
    string Text { get; }
    string HelpText { get; }
    Image Image { get; }
}
```

Where:

- **Category:** The name of the category to which the script belongs.
- **Text:** The name which is shown in the scrip selection and in the *Action* field

- **HelpText:** The text shown in the *Description* field
- **Image:** The default used picture

10.2.2 *IScriptAction*

This interface is used by Perception to actually run the action after for example a user button click:

```
public interface IScriptAction : ISerializable
{
    ScriptActionResult RunAction(IScriptContext context);
}
```

The RunAction parameter context has implemented the **IScriptContext** interface:

```
public interface IScriptContext
{
    IWin32Window OwnerWindow { get; }

    ScriptActionResult NotifyUser(IScriptAction action, string message,
        ScriptActionResult actionResult);
}
```

Where:

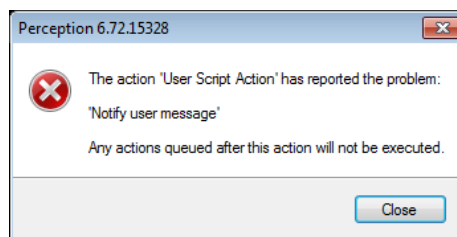
OwnerWindow: Window handle to be used when you want to show your own dialogs.

When the window handle is null then the action script is executed from the Perception automation and not from a user key click, in those cases we advise that the script action do not stop on modal dialogs, because there might not be an operator to close such a dialog

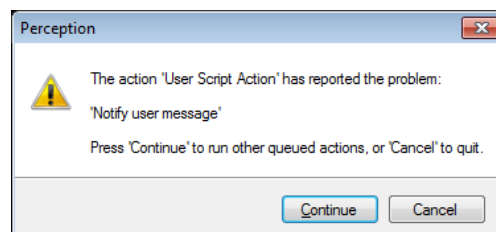
NotifyUser: Method to be used to interact with Perception

Examples:

- `context.NotifyUser(this, "Notify user message", ScriptActionResult.ErrorAbort);`

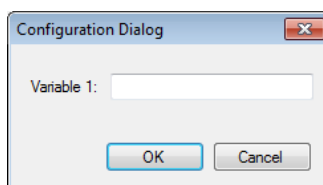


- `context.NotifyUser(this, "Notify user message", ScriptActionResult.ErrorContinue);`



10.2.3 IConfigurable

This interface is used to configure the script action



```
public DialogResult Configure(IWin32Window owner)
{
    using (UserScriptActionConfigurationDialog Dialog = new UserScriptActionConfigurationDialog())
    {
        Dialog.txtVariable1.Text = m_strVariable1;
        DialogResult result = Dialog.ShowDialog(owner);
        if (result == DialogResult.OK)
        {
            m_strVariable1 = Dialog.txtVariable1.Text;
        }
    }
    return DialogResult.OK;
}
```

10.3 Example: Create a script action for auto scaling all traces of the active display

We now will use the above framework to create a script action which can be used to auto scale all traces of the active display

- Rename the file **UserScriptAction.cs** to **TraceAutoScaleAction.cs**
- Modify the HandleError method:

```
private ScriptActionResult HandleError(IScriptContext context,
    string message, ScriptActionResult actionResult)
{
    if (context != null)
        actionResult = context.NotifyUser(this, message, actionResult);
    else
        actionResult = ScriptActionResult.ErrorAbort;
    return actionResult;
}
```

- Modify the ExecuteAction():

```
private bool ExecuteAction(IWin32Window owner, out string cError)
{
    Display activeDisplay = DisplayHelper.GetActiveDisplay();
    if (activeDisplay == null)
    {
        cError =
            "Can not perform this action because no active display isfound";
        return false;
    }

    ActionResultType aResult = DisplayHelper.DoAction(activeDisplay,
        TimeView.ActionType.ActionType_TraceAutoScale);
}
```

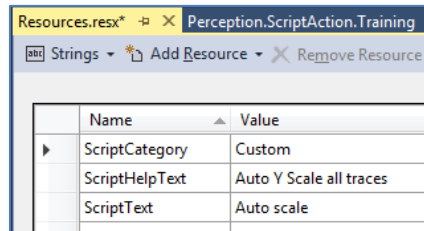
```

if (aResult == ActionResultType.ActionResultType_OK)
{
    cError = "";
    return true;
}

cError = string.Format(
    "Auto trace scale failed: Action result code: {0}" , aResult);
return false;
}

```

- Set the following resources:



Name	Value
ScriptCategory	Custom
ScriptHelpText	Auto Y Scale all traces
ScriptText	Auto scale

- Compile the project and check if the new script action works

10.4 Add option to select all or only active trace to be auto scaled

We will now modify the above script action by adding an option which is used to define if the auto scale should be done for all traces or only the active trace.

- Add new class to the project called **ScriptDisplayHelper**
- The code of this helper class looks like:

```

using TimeView;
using Perception.Components;
using TimeDisplayLib;

namespace Perception.ScriptAction.Training
{
    public static class ScriptDisplayHelper
    {
        internal delegate ActionResultType ActionTypeDelegate(
            Display aDisplay, DTrace aTrace, ActionType action);

        public static ActionResultType DoTraceAction(Display aDisplay,
            DTrace aTrace, ActionType action)
        {
            if (aDisplay.InvokeRequired)
            {
                aDisplay.BeginInvoke(new ActionTypeDelegate(DoTraceAction),
                    aDisplay, aTrace, action);
                return ActionResultType.ActionResultType_Failed;
            }

            object aOcx = aDisplay.ComInterface;
            return DoDisplayTraceAction(aTrace, aOcx as ITimeDisplay, action);
        }

        internal static ActionResultType DoDisplayTraceAction(DTrace aTrace,
            ITimeDisplay itfDisplay, ActionType action)
        {
            if (aTrace == null)
                return ActionResultType.ActionResultType_Failed;

            if (itfDisplay == null)

```

```

        return ActionResultType.ActionResultType_Failed;

        IDView itfActiveView = null;
        itfDisplay.GetView(DisplayModeType.DisplayModeType_ActiveView,
            out itfActiveView);

        if (itfActiveView == null)
            return ActionResultType.ActionResultType_Failed;

        ActionResultType result;
        itfActiveView.InvokeTraceCommand(aTrace, action, out result);

        return result;
    }
}

```

- Rename **FIELDNAME_VARIABLE_1** to **FIELDNAME_SCALE_ALL** in the TraceAutoScaleAction.cs file
- Rename **m_strVariable1** to **m_bScaleAll**
- The code should look like:

```

private const string FIELDNAME_SCALE_ALL = "ScaleAll";
private bool m_bScaleAll = true;

```

- Modify the serialization constructor to

```

public TraceAutoScaleAction(SerializationInfo info,
    StreamingContext context)
{
    m_bScaleAll = Tools.GetValue<bool>(info, FIELDNAME_SCALE_ALL, true);
}

```

- Modify the GetObjectData() method

```

public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue(FIELDNAME_SCALE_ALL, m_bScaleAll);
}

```

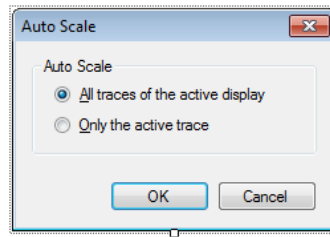
- Modify the ExecuteAction() to:

```

ActionResultType aResult;
if (m_bScaleAll)
    aResult = DisplayHelper.DoAction(activeDisplay,
        TimeView.ActionType.ActionType_TraceAutoScale);
else
    aResult = ScriptDisplayHelper.DoTraceAction(activeDisplay,
        activeDisplay.TimeDisplay.ActiveTrace,
        ActionTraceType.ActionTraceType_AutoScale);

```

- Change the **HelpText** property:
- Rename the file **UserScriptActionConfigurationDialog.cs** to **TraceAutoScaleConfigurationDialog.cs**
- Modify the configuration dialog as below:



- Set the property *Modifiers* of the group-box and the radio buttons to **Internal**. By doing this these components are available from the `TraceAutoScaleAction` class.
- Modify the `Configure` method in the `TraceAutoScaleAction` class:

```
public DialogResult Configure(IWin32Window owner)
{
    using (TraceAutoScaleConfigurationDialog Dialog = new
        TraceAutoScaleConfigurationDialog())
    {
        Dialog.radioAll.Checked = m_bScaleAll;
        Dialog.radioSingleTrace.Checked = !m_bScaleAll;

        DialogResult result = Dialog.ShowDialog(owner);

        if (result == DialogResult.OK)
        {
            m_bScaleAll = Dialog.radioAll.Checked;
        }
    }
    return DialogResult.OK;
}
```

- Compile and debug the program

11 Summary

Within this document we have tried to give you a basic understanding of the capabilities and the concepts of the Perception Custom Software Interface CSI. By itself the CSI is so extensive that it is not possible to describe all functions and features in a single document. Also a reference document is beyond the scope of the CSI.

The information provided in this manual should get you started. In addition to this manual you could follow a course provided by HBM. Also additional personal support is a possibility.

Contact HBM directly or through your distributor/agent to get more information on these topics.

12 Appendix: Multithreading

Multithreading, or free threading, refers to the ability of a program to execute multiple threads of operation simultaneously. An example of a multithreaded application might be a program that receives user input on one thread, performs a variety of complex calculations on a second thread, and updates a database on a third thread. In a single-threaded application, a user might spend idle time waiting for the calculations or database updates to finish. In a multithreaded application, these processes can proceed in the background so user time is not wasted.

Perception is a multithreading application with the user interface UI on one thread. As a result, accessing the UI from another thread is 'forbidden': you may not operate on a window from other than its creating thread.

In most of our examples we use the UI. When this use is invoked by a control that is already on the user interface, we are on the same thread. However, on various occasions we use event handlers to perform actions on the UI. Since we do not know - usually - from which thread this event is generated, we need to synchronize with the UI thread.

Synchronizing is done with `Invoke`, **`BeginInvoke`** and **`EndInvoke`**. Refer to the Microsoft documentation for full details.

The use of `Invoke` gives a safe use of multithreading in the application. The UI thread spawns a worker thread to do our operation, and the worker thread passes control back to the UI thread when the UI needs updating. In addition we need to verify if `Invoke` is required.

`BeginInvoke` is always preferred if you don't need the return of a function call because it sends the worker thread to its work immediately and avoids the possibility of deadlock.

Summary: when event handling code affects the user interface, a marshal of the event to the user interface thread is required. A common example for this is:

```
private void EventHandler(object sender, EventArgs e)
{
    if (this.InvokeRequired)
    {
        this.BeginInvoke(new EventHandler(EventHandler), sender, e);
        return;
    }

    this.SetFinished();
}
```

This is what we have seen multiple times. This code will work fine, as long as the "this" is already created and has a handle, if not the call **`this.InvokeRequired`** will throw an exception and may cause your application to exit.

There are two ways around this problem:

- add a check in your event handler code to see if the handle is already created, or
- hook to an event in the **`OnHandleCreated`** function (advanced)

- 1 Check if the handle is created in the event handler, before using the **InvokeRequired** property. Your code will look like this:

```
private void EventHandler(object sender, EventArgs e)
{
    if (!this.IsHandleCreated)
        return;

    if (this.InvokeRequired)
    {
        this.BeginInvoke(new EventHandler(EventHandler), sender, e);
        return;
    }

    this.SetFinished();
}
```

- 2 Do not hook to the event(s) until the handle is created. You can do this by hooking to an event in the **OnHandleCreated** function. Your code will look like this:

```
protected override void OnHandleCreated(EventArgs e)
{
    base.OnHandleCreated(e);

    //
    // Hook to events.
    Hook();
}
```

Create a function **Hook()** that contains all the required hooks.

For a better performance Perception supports a method called **InvokeOnUI()** which can be used to update the GUI from an event handler fired from any possible thread and replaces the above **InvokeRequired** and **BeginInvoke** code.

```
private void EventHandler(object sender, EventArgs e)
{
    this.InvokeOnUI(() => DoEventHandler(sender, e));
}

private void DoEventHandler(object sender, EventArgs e)
{
    this.SetFinished();
}
```

Head Office

HBM

Im Tiefen See 45
64293 Darmstadt
Germany
Tel: +49 6151 8030
Email: info@hbm.com

France

HBM France SAS

46 rue du Champoreux
BP76
91542 Mennecey Cedex
Tél: +33 (0)1 69 90 63 70
Fax: +33 (0) 1 69 90 63 80
Email: info@fr.hbm.com

UK

HBM United Kingdom

1 Churchill Court, 58 Station Road
North Harrow, Middlesex, HA2 7SA
Tel: +44 (0) 208 515 6100
Email: info@uk.hbm.com

USA

HBM, Inc.

19 Bartlett Street
Marlborough, MA 01752, USA
Tel : +1 (800) 578-4260
Email: info@usa.hbm.com

PR China

HBM Sales Office

Room 2912, Jing Guang Centre
Beijing, China 100020
Tel: +86 10 6597 4006
Email: hbmchina@hbm.com.cn

© Hottinger Baldwin Messtechnik GmbH. All rights reserved.
All details describe our products in general form only.
They are not to be understood as express warranty and do
not constitute any liability whatsoever.

measure and predict with confidence

