

Basic custom plugin development in C++

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <Window xmlns="https://mui.dewesoft.com/schema/1.1">
3   <CaptionPanel Title="Latch criteria settings">
4     <Grid PaddingLeft="5">
5       <Grid.ColumnDefinitions>
6         <ColumnDefinition Width="140"/>
7       </Grid.ColumnDefinitions>
8       <Grid.RowDefinitions>
9         <RowDefinition Height="20"/>
10        <RowDefinition Height="20"/>
11        <RowDefinition Height="10"/>
12        <RowDefinition Height="20"/>
13        <RowDefinition Height="20"/>
14      </Grid.RowDefinitions>
15    </Grid>
16  </CaptionPanel>
17 </Window>
```

Introduction to Basic custom plugin development in C++

Do you want to be able to extend Dewesoft on your own? With the help of C++ Plugin you can create your own plugins and integrate them into any Dewesoft. Since C++ Plugin uses Dewesoft internals, it can do anything that Dewesoft can. You can take arbitrarily many input channels, process the data using modern C++, output the results into output channels, change Dewesoft settings, and much much more!

C++ Plugin also allows you to create your very own user interface which is, together with your C++ code, compiled into an external library and automatically recognised and loaded by Dewesoft. This is why your plugin can be easily exported and imported for use on other computers.

All the examples of C++ Plugins we will create and the setups we will use in this tutorial are available on Dewesoft's webpage under [Support > Downloads > Developers > C++ Plugin](#). Note that you have to be logged in to access the C++ Plugin section.

How to Install the Dewesoft plugin template?

In order to start using C++ Plugin, you must have Visual Studio IDE installed on your system. Some of the reasons we have chosen Visual Studio are its functionalities, powerful developer tooling, like IntelliSense code completion and debugging, fast code editor, easy modification/customization and many more.

Dewesoft plugin for Visual Studio development can be found on the Dewesoft webpage under [Support -> Downloads -> Developers -> C++ Plugin](#). Note that you have to be logged in to access the C++ Plugin section. After downloading, just double-click the VSIX file and the installer will guide you through the installation process.

DewesoftX

Dewesoft previous releases

Manuals & Brochures

Plugins

Drivers

Developers

Other

C++ Script

Unsubscribed

C++ Plugin

Unsubscribed

C++ Plugin Examples

A set of C++ plugin examples.

1,43 MB | 10.07.2020

Download CppPluginExamples.zip

C++ Plugin Headers

Standalone C++ headers only (no wizard).

03.08.2018

Download MUI3.zip

Visual Studio 2017/2019 Development Tool

Visual Studio 2017/2019 installer for Dewesoft plugin types in C++.

38,68 MB | 20.09.2022

Download DewesoftX_Extensions_2019.vsix

Visual Studio 2022 Development Tool

Visual Studio 2022 installer for Dewesoft plugin types in C++.

39,19 MB | 20.09.2022









Download DewesoftX_Extensions.vsix

Once VSIX plugin is downloaded and installed you will be able to create the Dewesoft C++ plugin using the *New project* window and selecting the *DewesoftX C++ Plugin*.

2

Create a new project

Recent project templates

-  Windows Forms App (.NET Framework) C#
-  DEWESoftX C++ plugin
-  WPF App (.NET Framework) C#
-  WPF App (.NET Core) C#
-  WPF Custom Control Library (.NET Core) C#
-  Dynamic-Link Library (DLL) C++
-  Console App C++
-  Console App (.NET Framework) C#

dewesoft

Clear all






C#

All platforms

All project types

No exact matches found

Other results based on your search

-  DEWESoftX C++ Marker plugin
Creates an example marker DEWESoftX plugin.
-  DEWESoftX C++ Processing plugin
Creates an example processing DEWESoftX plugin.
-  DEWESoftX C++ Visual Control plugin
Creates an example Visual Control DEWESoftX plugin.
-  DEWESoftX C++ Export plugin
Creates an example export DEWESoftX plugin.
-  DEWESoftX C++ plugin
Creates an example DEWESoftX plugin.

Next

Example: Latch math

To get a better understanding of how to work with C++ Plugin we will implement a latch math module. Latch math outputs the value of an input channel when some other input channel crosses some predefined criteria. An example use case of this module could be to monitor your car engine RPM as you pass 100 km/h: we just set the first input channel to engine RPM, the second input channel to the channel with our car's current speed, and set the criteria to 100 km/h.

Given this, at the end of the example, we should end up with a working module that allows you to select two input channels, one to look for latch criteria points and one from which the outputted values will be read, and produce an output channel with these values.

Signals for testing our module

Before we begin creating our C++ Plugin, let's first create the two input signals we will use to test it. Open Dewesoft, click on the *New setup* button and then click *Math*.

The first signal will be the one we will use to look for criteria limit crossing. For our testing purposes, this signal will be a sine with a frequency of 1 Hz. We can create it by clicking the *Formula* button next to the *Add math* button and add sine function from the *Signal* tab. Let's name it "sine(1)". In the formula area write the following line:

```
sine(1)
```

The second signal will be the one from which we read the values to output at criteria limit. This signal will be the current time. We add it by clicking the *Formula* button next to the *Add math* button and add time function from the *Signal* tab, naming it "time".

```
time
```

Don't forget to save the setup!

Example: New C++ Plugin

Now we go back to Visual Studio. To create a new C++ Plugin we click the *Project* button in File tab > New > Project. We select the Dewesoft X Plugin Template as our template and fill in the name of our project. After clicking the *Ok* button a wizard window will appear to guide us through the creation of the plugin.

Since your plugin will be integrated inside Dewesoft, it needs to know Dewesoft's location. We can define the location by ourselves and the installer will create DEWESOFT_EXE_X86 (for 32-bit Dewesoft) and DEWESOFT_EXE_X64 (for 64-bit Dewesoft) system variables for us. In that case, Visual Studio will have to be restarted so it will update its used environmental variables state.

Dewesoft MUI3 Plugin

Dewesoft X executable location
Location is used for automatic setup of output directory and debugger settings

☐ From environment variables
☒ Custom location

x86 C:\DXEProjects\DewesoftX\Dewesoft\Bin\DEWESoft.exe Browse
x64 C:\DXEProjects\DewesoftX\Dewesoft\Bin64\DEWESoft.exe Browse

☒ Enable this plugin in current Dewesoft project file

< Back Next Cancel

Once those system variables are set, installer will automatically find those fields for you and you will not have to specify the location for further usages.

Dewesoft MUI3 Plugin

Dewesoft X executable location
Location is used for automatic setup of output directory and debugger settings

☒ From environment variables
☐ Custom location

x86 C:\DXEProjects\DewesoftX\Dewesoft\Bin Browse
x64 C:\DXEProjects\DewesoftX\Dewesoft\Bin64 Browse

☒ Enable this plugin in current Dewesoft project file

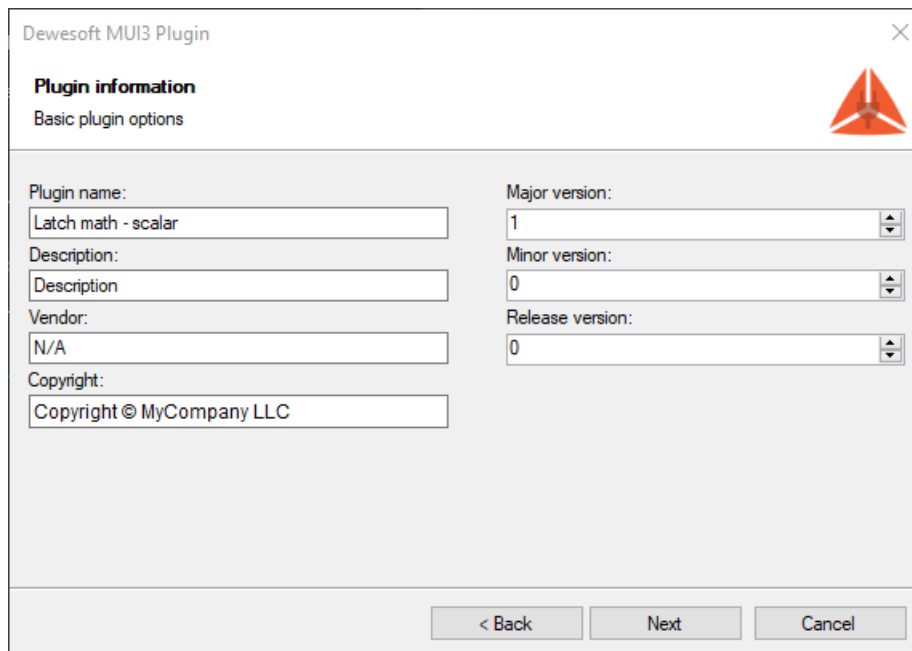
< Back Next Cancel

If we want to set those variables ourselves, we can do this using System properties window (it can be found pressing

Windows key and searching for Edit the system environment variables), and under advanced tab clicking the Environment variables.

If you only have the 64-bit (or 32-bit) version of Dewesoft on your computer, you will only be able to create 64-bit (or 32-bit) plugins.

After clicking the Next button the following window appears which is used to set Plugin information such as plugin name, its ownership, and version.

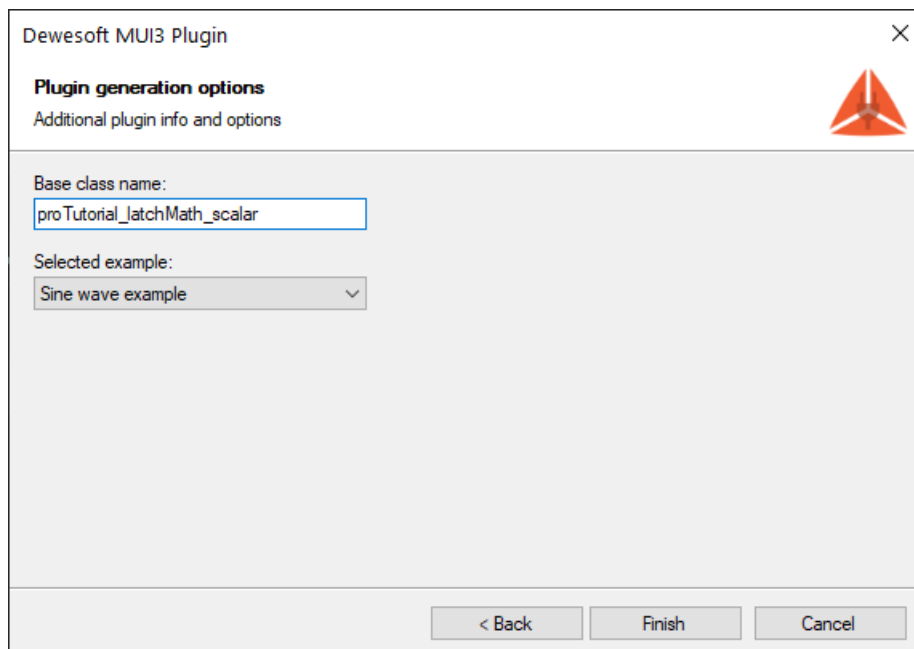


The image shows a Windows-style dialog box titled "Dewesoft MUI3 Plugin". It has a close button (X) in the top right corner and a Dewesoft logo (a red triangle with a white 'D') in the top right. The dialog is divided into two main sections. The top section is titled "Plugin information" and contains the subtitle "Basic plugin options". The bottom section contains several input fields and spinners. On the left side, there are four text input fields: "Plugin name:" (containing "Latch math - scalar"), "Description:" (containing "Description"), "Vendor:" (containing "N/A"), and "Copyright:" (containing "Copyright © MyCompany LLC"). On the right side, there are three spinner boxes: "Major version:" (set to 1), "Minor version:" (set to 0), and "Release version:" (set to 0). At the bottom of the dialog, there are three buttons: "< Back", "Next", and "Cancel".

- **Plugin name** - The name that will be seen in Dewesoft.
- **Description** - Short description of your plugin.
- **Vendor** - Company that created the plugin.
- **Copyright** - Owner of the plugin.
- **Major version** - Sets the initial major version. The value should change when a breaking change occurs (it's incompatible with previous versions).
- **Minor version** - Sets the initial minor version. The value should change when new features and bug fixes are added without breaking compatibility.
- **Release version** - Sets the initial release version. The value should change if the new changes contain only bugfixes.

All fields are optional except for the Plugin name, and they can all be modified later from the code.

After clicking the Next button a final window appears. This window is used to set your Base class name. It is used as a prefix for class and project name. When the Base class name is set, we can click the Finish button and the wizard will generate the plugin template based on your choices.



Below the base class name, you can see a dropdown menu, where you can select the example which will be generated for you. There are two options, one is a sine wave example, which will set everything up for outputting a sine wave values to the output channel. The other option is Empty example, which will remove all example code from the project. We recommend using this if you already have some knowledge about Dewesoft plugin development.

The *Project* name is the name of the file created by the Visual Studio.

The *Plugin* name is the name of the plugin as seen in Dewesoft.

The *Base class* name is the name of the plugin inside of Visual Studio and has to be a valid C++ name.

When a new C++ Plugin project is created, the wizard will create the basic files and project structure needed for development. In the picture below you can see the structure of a project in a tree view with collapsed items. In our case, *ProTutorial* refers to text, which was used as the Base class name.

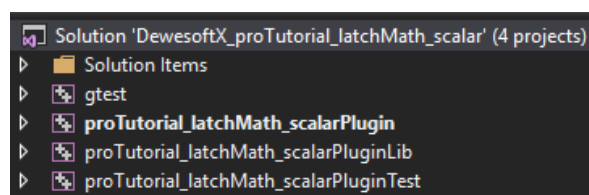
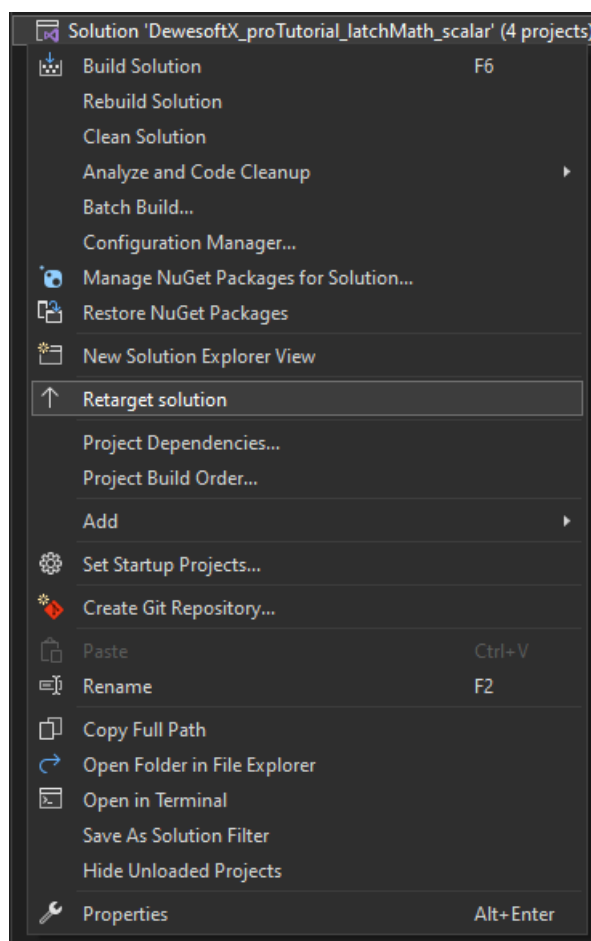


Image 7: New C++ Plugin project is created

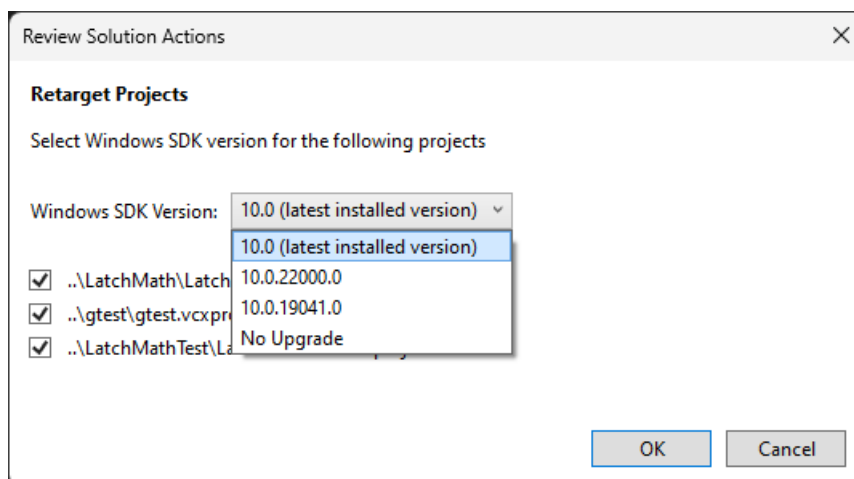
- *proTutorial_latchMath_scalar* - Used for communication with Dewesoft through its DCOM interface and creating the Plugin UI.
- *proTutorial_latchMath_scalarLib* - Your main plugin logic should be defined here. In Example I, we will write plugin logic inside the *Plugin* project to simplify things.
- *proTutorial_latchMath_scalarTest* - Contains test cases that will be run during unit testing.
- *gtest* - Simple library, required by ProTutorialPluginTest for unit testing your plugin. This project should not be modified.

When the solution is built for the first time, we recommend rescanning it (to clear cache). If not, some false positive errors might appear and auto-complete might not work. You can do this by clicking on the Project tab and choosing the Rescan solution from the drop-down list.

Once the plugin is created and you stumble upon Error MSB8036: The Windows SDK version `sdk_version` was not found. You should retarget your solution to use the one you have installed. You can do this by right clicking on your solution node in Solution explorer and selecting Retarget solution.



Once the Retarget solution window is opened, you can select "Latest installed version" item.



When our project is successfully generated, we will be able to extend Dewesoft. But before implementing the logic behind our plugin, let's take a look at how our plugin is integrated into Dewesoft. In order to do that, we have to start our program using the shortcut `F5` or pressing the *Start* button in the center of Visual Studio's main toolbar.

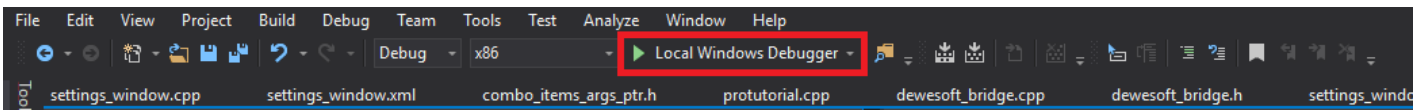


Image 8: Press the start button in the Visual Studio's main toolbar

After Dewesoft loads, our plugin can be accessed in Dewesofts main toolbar in *Measure* mode under *Ch. setup* -> *ProTutorial*. As we can see, it already contains some example elements which were automatically added to the user interface.

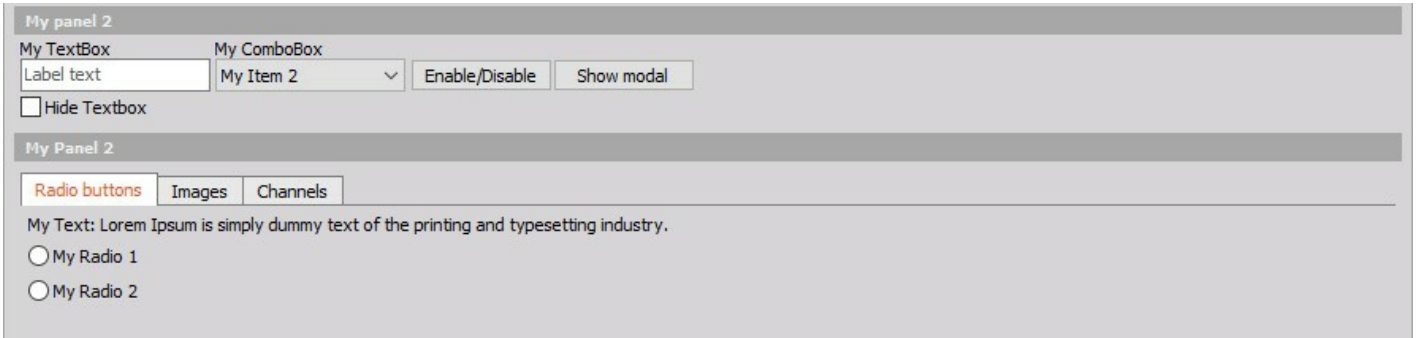


Image 9: Elements of newly added plugin


The user interface shown above is defined inside the `setup_window.xml` file. There are more user interfaces in the project (Setup, Settings), each is found in a different location inside Dewesoft and has a different purpose.

* Setup

- Found in the Dewesoft main toolbar.
- Your main plugin user interface should be defined here.

* Settings

- Found in *Options > Settings > Extensions > Plugin*.
- General settings, variables, properties, and paths, which are set once and rarely changed should be defined here.

Files associated with them can be found in Visual Studios *Solution explorer* by expanding *ProTutorialPlugin > UI* folder. If your plugin is not visible in Dewesoft you must manually add it. Click the Options icon in the top right corner and from the drop-down list choose Settings. Go to the Extensions tab, click the  button and find your plugin. Click on it and then click the Enable button on the right.

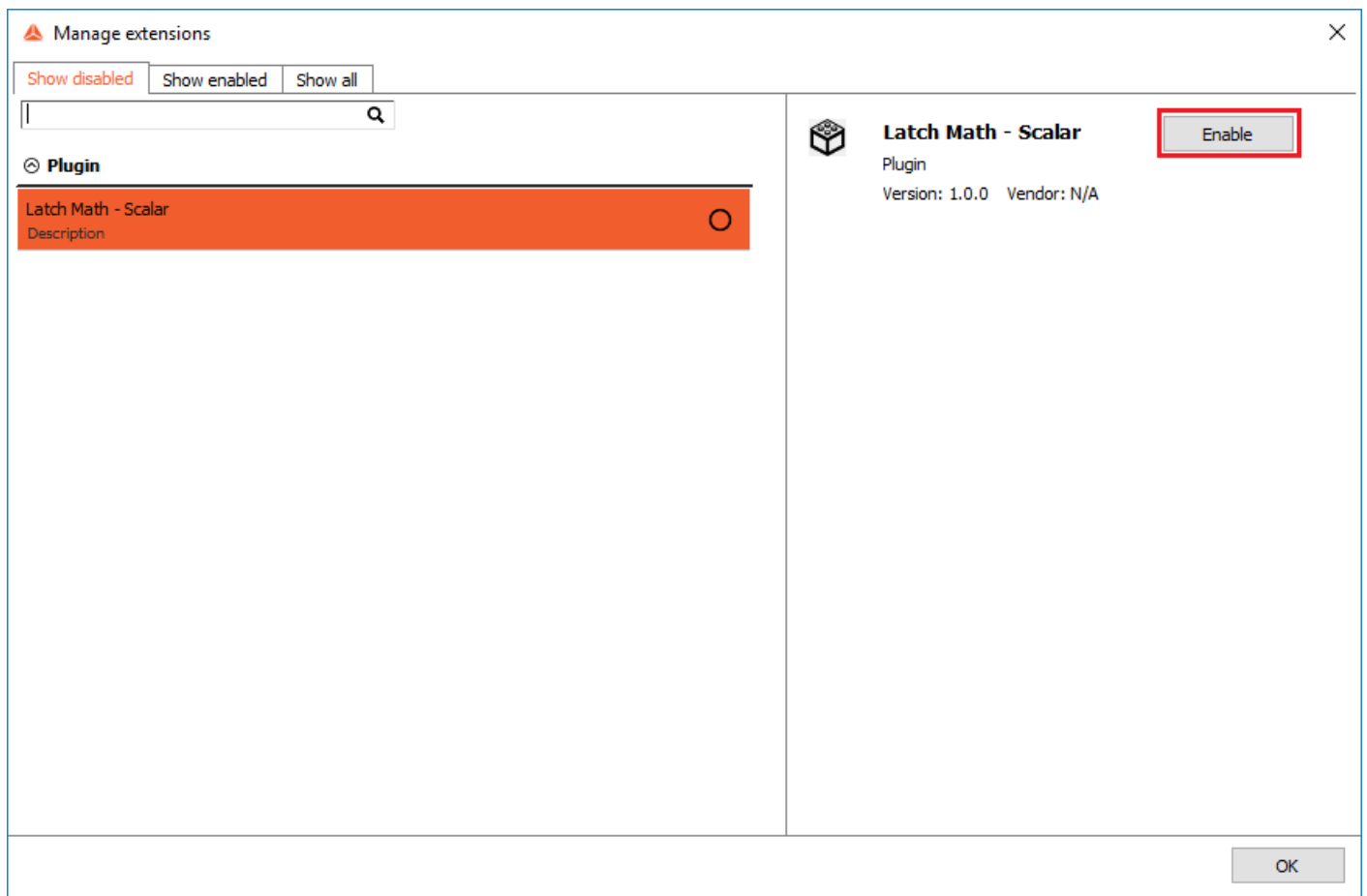


Image 10: In case the plugin is not visible in Dewesoft X, enable the Latch Math - Scalar under Settings -> Extensions

Example I: Creating custom UI

We are now ready to start creating the plugin for Latch math. We will start by taking a look at the already existing UI for latch math that is integrated into Dewesoft. It can be found in *Measure mode* -> *Math* -> *Add math* -> *Latch value math*. Keep in mind that this is a basic tutorial so we will keep things simple. Our goal is to recreate the UI on the right side of the picture below. We will not use the channel preview (bottom left corner), we will only allow the user to pick one *Input channel* and one *Criteria channel* and the *Output value* will be set to *Actual* by default, so we will not allow the user to change it. We will also add *Radio buttons* where the user will choose if we are looking for the latch condition at rising or falling edge.

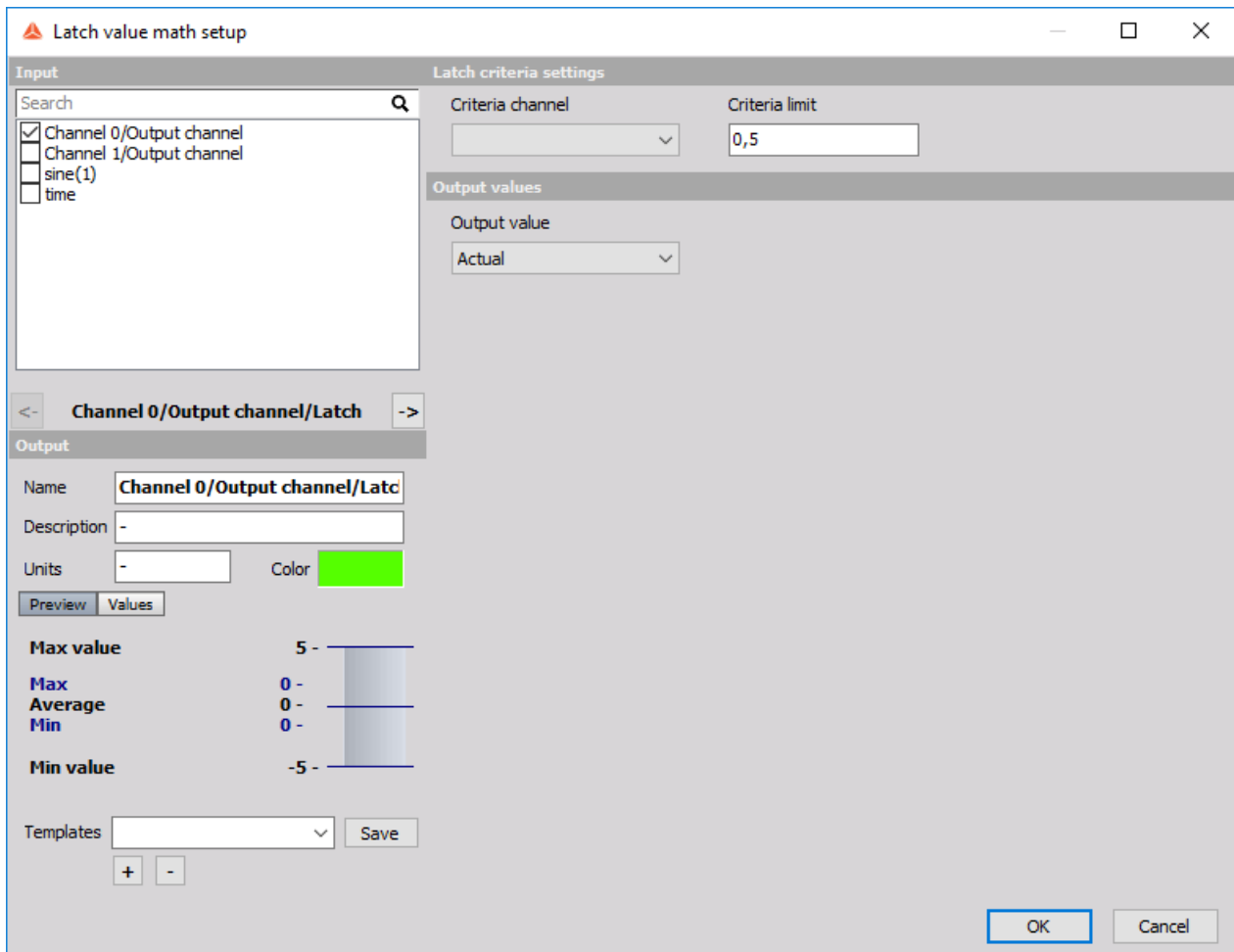


Image 11: Add Latch value math in the Measure mode

Therefore, we will need two **ComboBoxes** (control, which contains a drop-down menu where only one item can be selected), both displaying channel names. One will be used for selecting the *Input channel* and the other will be used for selecting the *Criteria channel*. We will also need a **TextBox** (control, which displays a user-editable text) so the user will be able to enter the *Criteria limit* and a **Label** (control, which displays read-only text) that will be added to the **TextBox** so it will be more descriptive. Finally, we will need two **RadioButtons** (control, whose *Checked* property can be changed by clicking on it) for determining whether we are calculating *rising* or *falling* edge.

We will visually group settings for *Input channel* and *Criteria channel* using **CaptionPanel** (layout control with a title) and a **Grid** (a type of layout, which arranges its child controls in an arbitrary number of rows and columns that can be spanned).

We are now ready to create our custom user interface for latch math. The user interface is defined in the [setup_window.xml](#) file by using simple, XAML-like syntax. The XML code for our UI will be presented in smaller pieces so it will be easier to explain and understand. For each code snippet, there will be a picture added so you can see what the code creates. Your user interface can be seen inside Dewesoft when the plugin is run.

```
<?xml version="1.0" encoding="utf-8"?>
<Window xmlns="https://mui.dewesoft.com/schema/1.1">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width ="170"/>
      <ColumnDefinition Width ="100%"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="100%"/>
    </Grid.RowDefinitions>
  </Grid>
</Window>
```

This part of the code contains a Grid that will split your UI into two columns. The first column will have a width of 170 pixels and the second column will take the remaining window width (e.g. if the window width is 500 pixels the second column will have a width of 330px). If the window is resized, the width of the first column will remain fixed but the width of the second column will be adapted to fit the window. As you can see the grid also contains a row, which takes the entire window height.

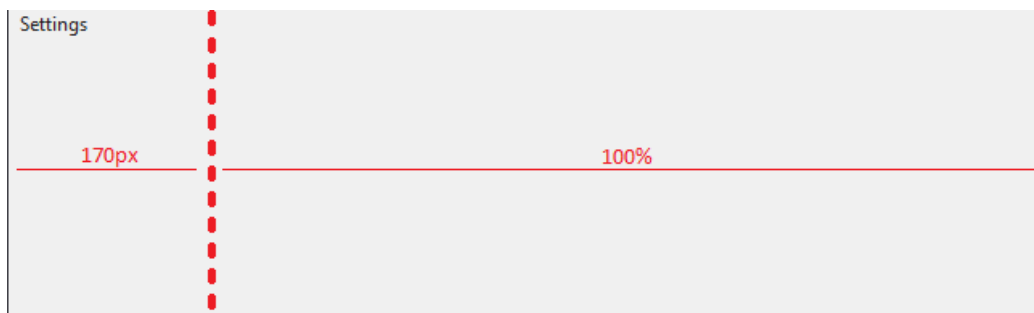


Image 12: User Interface will be split into two columns of different widths

When using a Grid, it is important to set Grid.ColumnDefinitions and Grid.RowDefinitions, even if there is only one column or row.

```

<CaptionPanel Grid.Column="0" Title="Input">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="20"/>
      <RowDefinition Height="100%"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100%"/>
    </Grid.ColumnDefinitions>

    <Label Text="Input channel" Grid.Row="0" />
    <ComboBox MarginRight="20" Grid.Row="1" Name="inputChannelName" />
  </Grid>
</CaptionPanel>

```

With the code above we will add CaptionPanel to the first column of our main Grid. This is done by adding `Grid.Column="0"` to the CaptionPanel. The Grid inside the CaptionPanel contains two rows, one for the Label and one for the ComboBox.

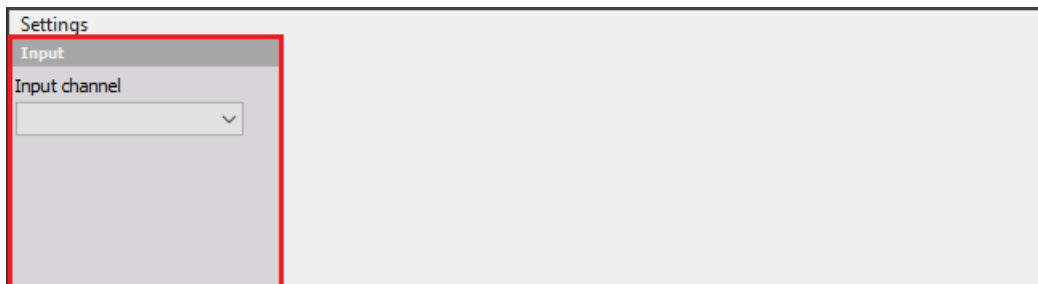


Image 13: Grid inside the CaptionPanel contains two rows: Label and the ComboBox row

Note that we have added *Name* properties to components. Names are required to access components from the C++ code so they have to be unique.

```

<CaptionPanel Grid.Column="1" Title="Latch criteria settings">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="140"/>
      <ColumnDefinition Width="30"/>
      <ColumnDefinition Width="120"/>
      <ColumnDefinition Width="100%"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="20"/>
      <RowDefinition Height="20"/>
      <RowDefinition Height="10"/>
      <RowDefinition Height="20"/>
      <RowDefinition Height="20"/>
      <RowDefinition Height="100%"/>
    </Grid.RowDefinitions>

    <Label Grid.Column="0" Text="Criteria channel"/>
    <ComboBox Grid.Column="0" Grid.Row="1" Name="criteriaChannelName" />
    <Label Grid.Column="2" Text="Criteria limit" />
    <TextBox Grid.Column="2" Grid.Row="1" Name="latchCriteria" Text="0"></TextBox>
    <RadioButton Grid.Column="0" Grid.Row="3" Name="risingEdge" Label="Rising edge"
    IsChecked="True" />
    <RadioButton Grid.Column="0" Grid.Row="4" Name="fallingEdge" Label="Falling edge" />
  </Grid>
</CaptionPanel>
</Grid>
</Window>

```

And lastly, we add the code for filling out the second column of our main Grid. This one is a bit more complex since elements need to be aligned and visually grouped (items that are connected should be closer. e.g. Criteria limit caption Label and TextBox). That is why we have created another Grid element with six rows and four columns, where our components can be inserted.

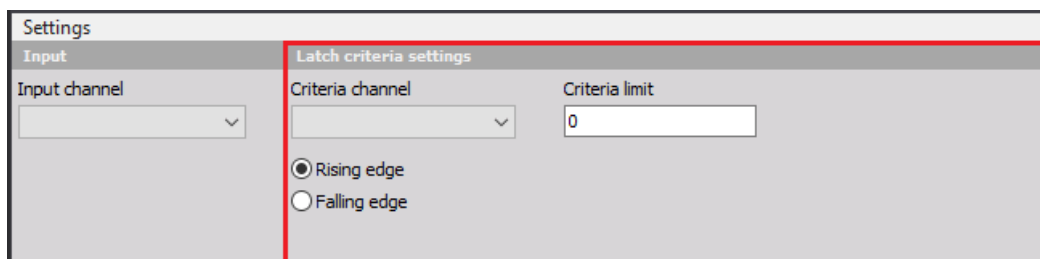


Image 14: Add the code for filling out the second column of the main Grid

So far we have tested our user interface inside Dewesoft by running our plugin. When editing the UI it is not necessary to run the plugin every time we want to see the changes we made. We can use MUI Designer, which live previews your design. It can be found in *Visual Studio > View > Other windows > MUI Designer (Dewesoft)*.

In the picture below you can see our final user interface in MUI Designer. The preview is automatically refreshed when the code is changed and if we run the plugin in Dewesoft the UI will look and behave exactly like shown in the MUI Designer.

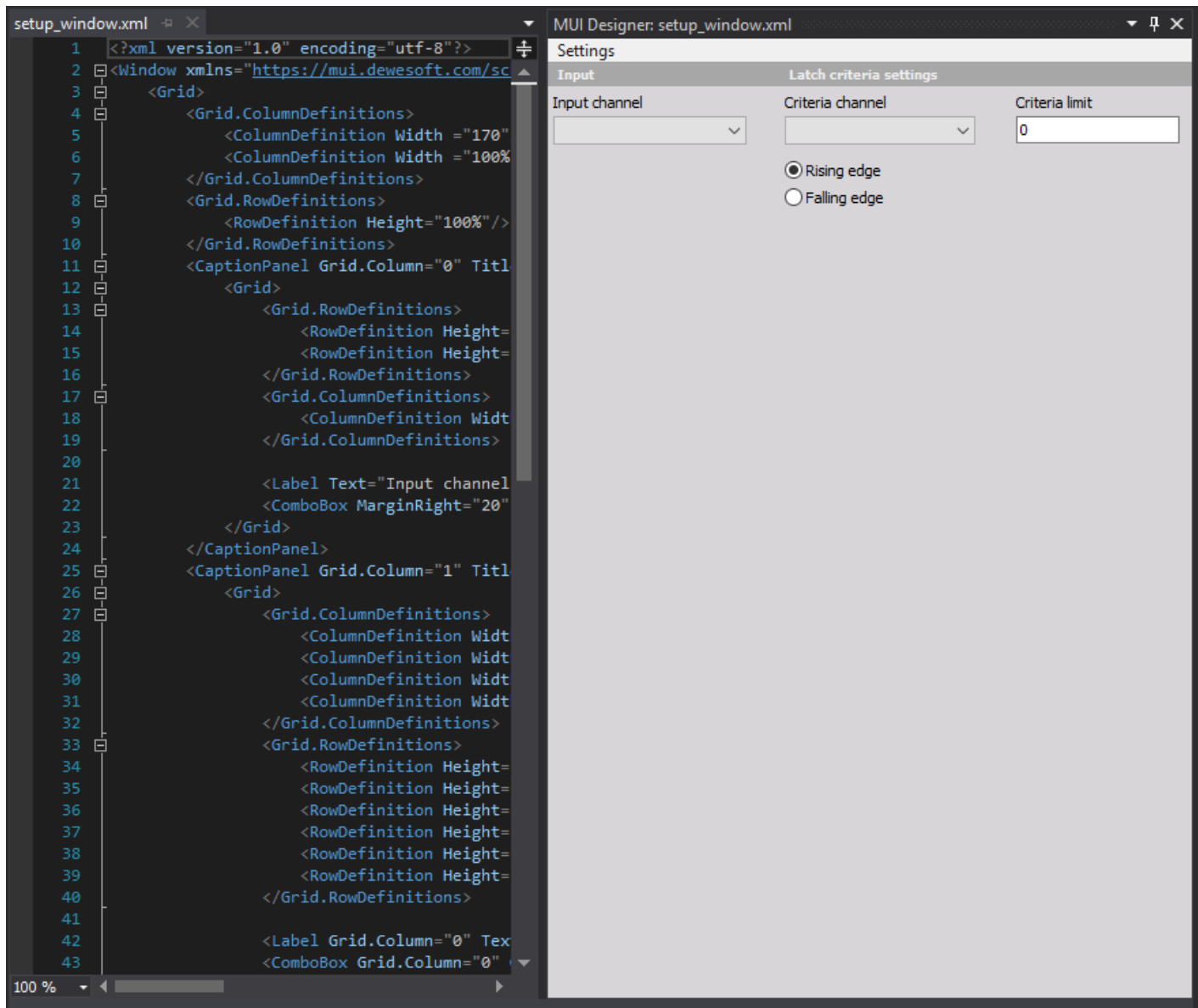


Image 15: Use MUI Designer to live preview your designs

Adding XML components is not only possible by manually typing them out; we can use the MUI Snippets window, which contains all available controls, to automate the typing. It can be found in *Visual Studio > View > Other Windows > MUI Snippet (Dewesoft)*. Adding code is extremely simple: all we have to do is place the caret where we wish the text to be inserted and double-click the desired control.

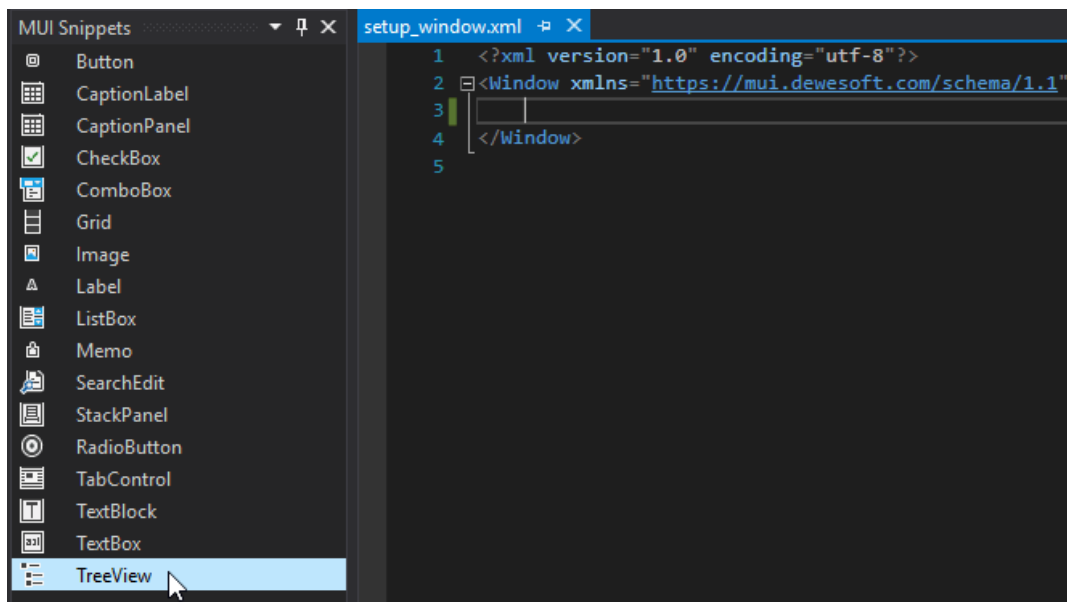


Image 16: Add XML components using the MUI Snippets window

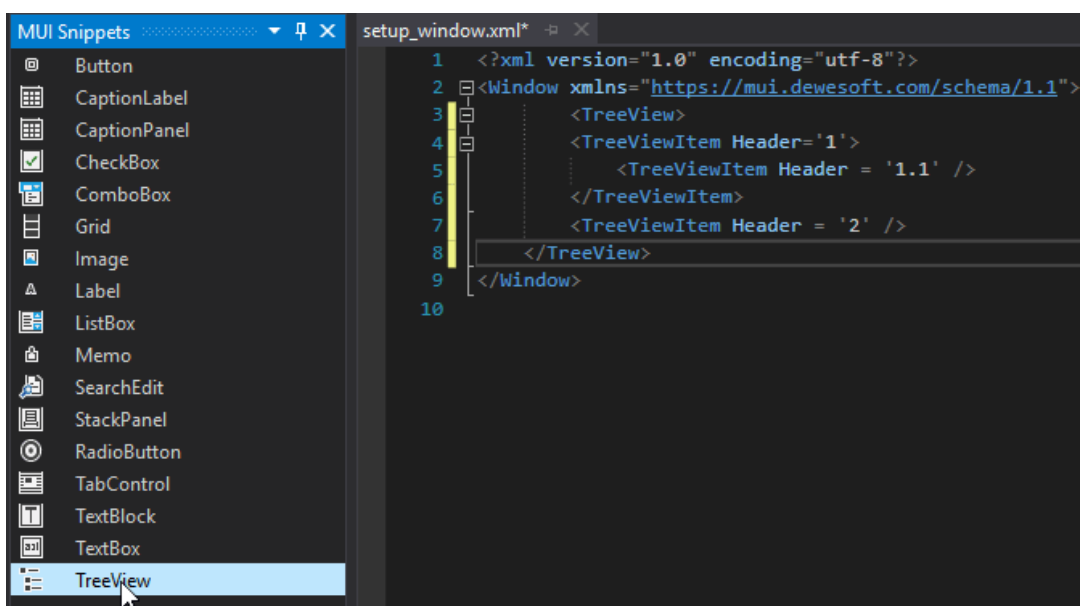


Image 17: Add XML components using the MUI Snippets window

Example I: Accessing the UI component using C++

We have now created the user interface for our plugin but it is not really doing anything at the moment. We still have to add some functionality to the UI components.

It is important to keep in mind that C++ uses header files (you can recognize them by the `.h` extension) in addition to source files. Header files are designed to provide information about your class and are used for declaration of variables and methods, while their initialization is done in the source files with `.cpp` extension. Before writing our own code, we will first remove the sample code as it is not needed. Our `setup_window.h` should look like this:

```
#pragma once
#include <mui/ds_window.h>
#include <mui/controls.h>
#include <mui/layout.h>

class DewesoftBridge;

class SetupWindow : public Dewesoft::MUI::DSWindow
{
public:
    SetupWindow(Dewesoft::MUI::WindowPtr ui, DewesoftBridge& bridge);

private:
    DewesoftBridge& bridge;
};
```

and `setup_window.cpp` should look like this:

```
#include "StdAfx.h"
#include "setup_window.h"
#include "dewesoft_bridge.h"
#include <thread>
#include <chrono>
#include <regex>

using namespace Dewesoft::MUI;
using namespace Dewesoft::RT::Core;

SetupWindow::SetupWindow(WindowPtr ui, DewesoftBridge& bridge)
    : DSWindow(ui, "ui/setup_window.xml")
    , bridge(bridge)
{
}
```

Our new components will be declared in `setup_window.h` file. We usually keep components in the *private* section of our class. We then forward all the information from the controls and their events to the *bridge* that will handle all the communication with Dewesoft.

```
private:
    DewesoftBridge& bridge;

    Dewesoft::MUI::ComboBox criteriaChannelCBox;
    Dewesoft::MUI::ComboBox inputChannelCBox;
    Dewesoft::MUI::TextBox latchCriteriaTBox;
    Dewesoft::MUI::RadioButton risingEdgeRButton;
    Dewesoft::MUI::RadioButton fallingEdgeRButton;
```

Members are later initialized in `setup_window.cpp` file where we connect them with UI component by specifying the component name. In our case this is done in the class constructor by using the `Connect` method.

```
SetupWindow::SetupWindow(WindowPtr ui, DewesoftBridge& bridge)
: DSWindow(ui, "ui/setup_window.xml")
, bridge(bridge)
{
    criteriaChannelCBox = ComboBox::Connect(ui, "criteriaChannelName");
    inputChannelCBox = ComboBox::Connect(ui, "inputChannelName");
    latchCriteriaTBox = TextBox::Connect(ui, "latchCriteria");
    risingEdgeRButton = RadioButton::Connect(ui, "risingEdge");
    fallingEdgeRButton = RadioButton::Connect(ui, "fallingEdge");
}
```

After successfully connecting the UI components we now need to get the available channels from Dewesoft and list their names in ComboBoxes so the user can see and choose them. To make our calculation easier, our input channels will be *synchronous* (time difference between two sequential samples is always the same). We first have to get all the available channels from Dewesoft and filter out all but the synchronous ones.

In order to do so, we have to access Dewesoft internals, which can be done inside the *DewesoftBridge* (you can find it in the *dewesoft_bridge.h* and *dewesoft_bridge.cpp*). The channel information is stored in the bridge member variable `app` which represents the Dewesoft DCOM interface. We create a getter function called `getSyncChannels()` that will read the channels from `app`, filter them and return only the synchronous ones.

```

// dewesoft_bridge.h
// this method needs to be public
std::vector<IChannelPtr> getSyncChannels();

// dewesoft_bridge.cpp
std::vector<IChannelPtr> DewesoftBridge::getSyncChannels()
{
    std::vector<IChannelPtr> allSyncChannels;

    app->Data->BuildChannelList();
    IChannelListPtr channels = app->Data->UsedChannels;

    for (int i = 0; i < channels->Count; i++)
        if (!channels->GetItem(i)->Async)
            allSyncChannels.push_back(channels->GetItem(i));

    return allSyncChannels;
}

```

Once `getSyncChannels()` function is written, we are ready to insert channel names into CheckBoxes. We will do this in a constructor in the file `setup_window.cpp` below the initialization of the controls. The `criteriaChannelCBox` and `inputChannelCBox` refer to CheckBoxes in which channel names will be inserted. To insert channel names to ComboBox, we will create a function called `void addChannelsToCBox()`, which should be public because we will need it later inside of the *bridge*.

```

// setup_window.h
void addChannelsToCBox();

```

```

// setup_window.cpp
SetupWindow::SetupWindow(WindowPtr ui, DewesoftBridge& bridge)
: DSWindow(ui, "ui/setup_window.xml")
, bridge(bridge)
{
    // member initialization
    // ...
    addChannelsToCBox();
}

void SetupWindow::addChannelsToCBox()
{
    criteriaChannelCBox.clear();
    inputChannelCBox.clear();
    std::vector<IChannelPtr> allChannels = bridge.getSyncChannels();

    for (int i = 0; i < allChannels.size(); i++)
    {
        std::string channelName = allChannels[i]->Name;
        criteriaChannelCBox.addItem(channelName);
        inputChannelCBox.addItem(channelName);
    }
}

```

If we now start Dewesoft, channel names should appear when we click on ComboBoxes.



Image 18: Channel names are now visible in ComboBoxes

Example I: Handling events

We have now initialized all UI components we will need in our example, but these components don't do anything yet. To bring them to life we need to introduce events to our code (e.g. Click, CheckedChanged, ...). To create an event all you have to do is provide a method with the same signature as the control's event handler and then bind it to the control.

A method signature is its return type, calling convention, and argument order and their types. Event handlers usually have the signature of `void(ComponentType& sender, EventArgsType& args)`.

Event handlers are declared in the header files. Here you can see the declaration of event handlers that will be executed when an event is triggered. This part of the code should be placed inside the private section in [setup_window.h](#) file.

```
private
// ...
void onCriteriaChannelChanged(Dewesoft::MUI::ComboBox& cBox, Dewesoft::MUI::EventArgs& args);
void onInputChannelChanged(Dewesoft::MUI::ComboBox& cBox, Dewesoft::MUI::EventArgs& args);
void onEditTextChanged(Dewesoft::MUI::TextBox& editBox, Dewesoft::MUI::EventArgs& args);
void onRadioGroupChanged(Dewesoft::MUI::RadioButton& radioButton, Dewesoft::MUI::EventArgs&
args);
}
```

Because a component can have multiple event handlers bound to the same event, we use `+=` operator for adding and `-=` operator for removing event handlers. This can only be done after the variable is connected. In our case, we bind events in the constructor immediately after connecting the variables. The code for how to bind events to our controls in [setup_window.cpp](#) can be seen below.

```
SetupWindow::SetupWindow(WindowPtr ui, DewesoftBridge& bridge)
: DSWindow(ui, "ui/setup_window.xml")
, bridge(bridge)
{
// member initialisation
// adding channel names to ComboBoxes
// ...
criteriaChannelCBox.OnChange += event(&SetupWindow::onCriteriaChannelChanged);
inputChannelCBox.OnChange += event(&SetupWindow::onInputChannelChanged);
latchCriteriaTBox.OnTextChanged += event(&SetupWindow::onEditTextChanged);
risingEdgeRButton.OnCheckedChanged += event(&SetupWindow::onRadioGroupChanged);
fallingEdgeRButton.OnCheckedChanged += event(&SetupWindow::onRadioGroupChanged);
}
```

Event handlers are defined in [setup_window.cpp](#) file. The following pictures will show which event handler is called when a certain action is performed.

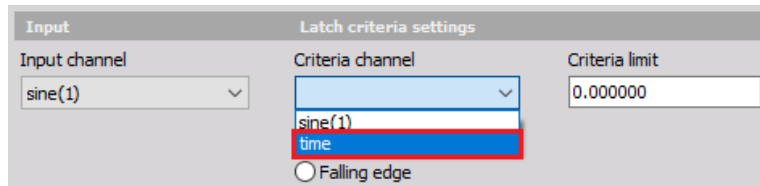


Image 19: Part of code is triggered as you click on any item inside
ComboBox containing Criteria channel names

This part of the code is triggered if we click on an item (that is not already selected) inside ComboBox containing Criteria channel names.

```
void SetupWindow::onCriteriaChannelChanged(Dewesoft::MUI::ComboBox& cBox,  
Dewesoft::MUI::EventArgs& args)  
{  
    bridge.setCriteriaChannelName(cBox.getSelectedItem());  
}
```

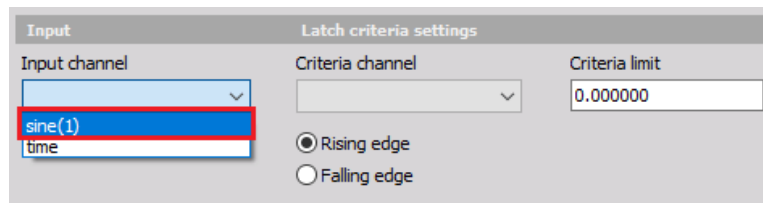


Image 20: Selecting inside ComboBox component containing Input channel names

This part of the code is triggered if we click on any item (that is not already selected) inside the ComboBox component containing input channel names.

```
void SetupWindow::onInputChannelChanged(Dewesoft::MUI::ComboBox& cBox,  
Dewesoft::MUI::EventArgs& args)  
{  
    bridge.setInputChannelName(cBox.getSelectedItem());  
}
```


Input	Latch criteria settings	
Input channel sine(1) ▾	Criteria channel time ▾	Criteria limit 0.5
	<input checked="" type="radio"/> Rising edge <input type="radio"/> Falling edge	

Image 21: If you edit text a certain part of the code is triggered

This part of the code is triggered if we edit text in component marked with the red square in the picture above.

```
void SetupWindow::onEditTextChanged(Dewesoft::MUI::TextBox& editBox, Dewesoft::MUI::EventArgs& args)
{
    bridge.setCriteriaLimit(stof(editBox.getText().toStdString()));
}
```

Input	Latch criteria settings	
Input channel sine(1) ▼	Criteria channel time ▼	Criteria limit 0.5
<input type="radio"/> Rising edge <input checked="" type="radio"/> Falling edge		

Image 22: Another part of the code is triggered as you click on a desired component

This part of the code is triggered if we click on the component marked with the red square in the picture above.

```
void SetupWindow::onRadioGroupChanged(RadioButton& radioButton, EventArgs& args)
{
    if (SetupWindow::fallingEdgeRButton.IsChecked())
        bridge.setEdgeType(FallingEdge);
    else
        bridge.setEdgeType(RisingEdge);
}
```

Example I: DewesoftBridge

The main purpose of *DewesoftBridge* is to allow your plugin to communicate with Dewesoft. It contains functions and events, that are triggered at a certain time (e.g. when measuring is started, when measuring is stopped, when setup is saved,...). The list of all function and events can be found in *dewesoft_bridge.h* file. In this section we will only mention those used in order for our plugin to work as intended, others will remain unchanged. We will use this class for implementing the logic behind the latch math.

We will first make changes to the *dewesoft_bridge.h* file where we will declare methods and variables. For our plugin to work we need variables for holding Criteria limit value, channel names and for determining whether we are calculating the rising or falling edge. We will also create *getter* and *setter* functions for these variables. All of these variables and methods are only going to be accessed inside the *DewesoftBridge*, so they should be private.

```

enum edgeTypes
{
    RisingEdge = 0,
    FallingEdge = 1
};

class DewesoftBridge
{
public:
    // ...

    /* Declarations of methods we added*/
    std::vector<IChannelPtr> getSyncChannels();

    void setCriteriaChannelName(std::string name);
    std::string getCriteriaChannelName();

    void setInputChannelName(std::string name);
    std::string getInputChannelName();

    void setCriteriaChannel(std::string channelName);
    IChannelPtr getCriteriaChannel();

    void setInputChannel(std::string channelName);
    IChannelPtr getInputChannel();

    void setCriteriaLimit(float threshold);
    float getCriteriaLimit();

    void setEdgeType(edgeTypes type);
    edgeTypes getEdgeType();

private:
    // ...

    /* Declarations of methods and variables we added */
    std::string inputChannelName = "";
    std::string criteriaChannelName = "";
    IChannelPtr inputChannel;
    IChannelPtr criteriaChannel;

    float criteriaLimit = 0;
    edgeTypes edgeType = RisingEdge;

    int64_t lastPosChecked;
};

```

Definitions of setters and getters should be written in *dewesoft_bridge.cpp*.

```

void DewesoftBridge::setCriteriaChannelName(std::string name)
{
    this->criteriaChannelName = name;
}
std::string DewesoftBridge::getCriteriaChannelName()
{
    return this->criteriaChannelName;
}

void DewesoftBridge::setInputChannelName(std::string name)
{
    this->inputChannelName = name;
}
std::string DewesoftBridge::getInputChannelName()
{
    return this->inputChannelName;
}

void DewesoftBridge::setCriteriaChannel(std::string channelName)
{
    criteriaChannel = app->Data->FindChannel(channelName.c_str());
}
IChannelPtr DewesoftBridge::getCriteriaChannel()
{
    return this->criteriaChannel;
}

void DewesoftBridge::setInputChannel(std::string channelName)
{
    this->inputChannel = app->Data->FindChannel(channelName.c_str());
}
IChannelPtr DewesoftBridge::getInputChannel()
{
    return this->inputChannel;
}

void DewesoftBridge::setCriteriaLimit(float threshold)
{
    this->criteriaLimit = threshold;
}
float DewesoftBridge::getCriteriaLimit()
{
    return this->criteriaLimit;
}

void DewesoftBridge::setEdgeType(edgeTypes type)
{
    this->edgeType = type;
}
edgeTypes DewesoftBridge::getEdgeType()
{
    return this->edgeType;
}

```

In the rest of this section we mention methods which were modified so our plugin works as it should.

DewesoftBridge::mountChannels()

The `void DewesoftBridge::mountChannels()` method is called when your plugin or setup is first loaded. This method gives you the option to create your own output channels inside Dewesoft. In our example, the output channel should be of an asynchronous type because we do not know exactly when a new sample will be added. We will set basic channel properties like channel name, base type (synchronous, asynchronous type) and underlying data type.

```
void DewesoftBridge::mountChannels()
{
    const std::vector<int> chIndexVector = {1};

    outputChannel = pluginGroup->MountChannelEx(pluginGuid, long(chIndexVector.size()),
    fromStdVec(chIndexVector));

    outputChannel->SetDataType(long(ChannelDataType::Single));
    outputChannel->Name = "Latch";
    outputChannel->Unit_ = "";
    outputChannel->SetAsync(true);
    outputChannel->ExpectedAsyncRate = app->Data->GetSampleRateEx();
    outputChannel->Used = true;
}
```

Dewesoft::onStartData()

This method gets called before the start of each measurement. Here we should initialize all variables used during our measurement.

```
STDMETHODIMP DewesoftBridge::onStartData()
{
    setInputChannel(inputChannelName.c_str());
    setCriteriaChannel(criteriaChannelName.c_str());
    lastPosChecked = 0;

    return S_OK;
}
```

DewesoftBridge::onGetData()

This method is called repeatedly during Measure mode. Here we can safely read from and write to the channels. We should *not* perform any time-consuming tasks here. The method itself is marked as `STDMETHODIMP`, this is required for Dewesoft to call this function over the DCOM interface. Keep in mind that the method gets a block of samples and not an individual sample every time it gets called, so if we want to access every sample in the channel we need to create a loop that will

perform this action.

```
STDMETHODIMP DewesoftBridge::onGetData()
{
    if (!inputChannel || !criteriaChannel)
        return S_OK;

    // calculate the size of blocks in both channels
    int blockSizeCriteriaChannel =
        (criteriaChannel->DBPos - (lastPosChecked % criteriaChannel->DBBufSize) + criteriaChannel-
        >DBBufSize) % criteriaChannel->DBBufSize;
    int blockSizeInputChannel =
        (inputChannel->DBPos - (lastPosChecked % inputChannel->DBBufSize) + inputChannel-
        >DBBufSize) % inputChannel->DBBufSize;

    int minBlockSize = min(blockSizeCriteriaChannel, blockSizeInputChannel);

    for (int i = 0; i < minBlockSize - 1; i++)
    {
        float currentSampleCriteriaChannel = criteriaChannel->DBValues[lastPosChecked %
        criteriaChannel->DBBufSize];
        float nextSampleCriteriaChannel = criteriaChannel->DBValues[(lastPosChecked + 1) %
        criteriaChannel->DBBufSize];

        // check if the two samples from Criteria channel are on different sides of Latch criteria
        bool crossedRisingEdgeCriteria = currentSampleCriteriaChannel <= criteriaLimit &&
        nextSampleCriteriaChannel >= criteriaLimit;
        bool crossedFallingEdgeCriteria = currentSampleCriteriaChannel >= criteriaLimit &&
        nextSampleCriteriaChannel <= criteriaLimit;

        // if user set the type of edge to rising edge and the Criteria channel crossed it
        // or user set the type of edge to falling edge and the Criteria channel crossed it
        if ((crossedFallingEdgeCriteria && edgeType == FallingEdge) || (crossedRisingEdgeCriteria &&
        edgeType == RisingEdge))
        {
            // add the value of the next sample from Input channel to the Latched channel
            float value = inputChannel->DBValues[(lastPosChecked + 1) % inputChannel->DBBufSize];
            outputChannel->AddAsyncSingleSample(value, (lastPosChecked + 1) / app->Data-
            >SampleRateEx);
        }
        lastPosChecked++;
    }
    return S_OK;
}
```

Writing to output channels and reading from input channels is only valid inside ::onGetData() procedure! Attempting to do so from any other function might result in your plugin crashing!

Example I: Output result

We are now ready to test our plugin. We can do this by running your plugin by pressing F5 on your keyboard and going to *Ch. setup* tab, then click the *Latch math - Scalar* button in the main toolbar in Dewesoft. A window like this should appear.

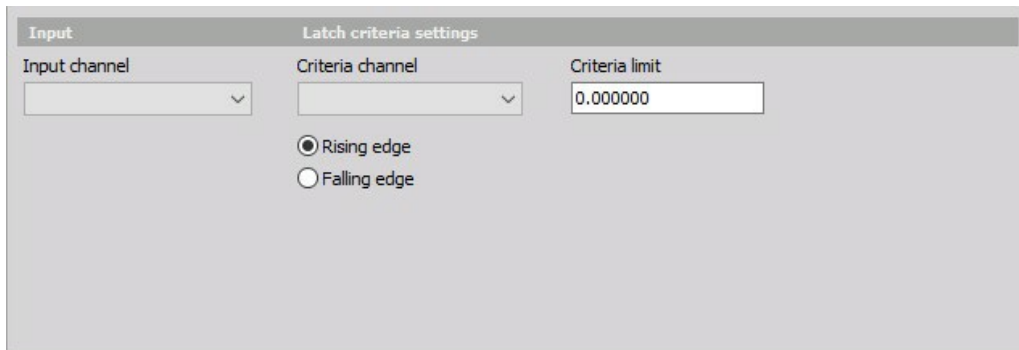


Image 23: Latch math - scalar window

Remember the setup we created and saved in Dewesoft before? We are going to use it here. We go to the Setup files tab and double-click on the setup we want to load. The setup we loaded should include the two signals we created earlier and a blank setup *Latch Math - Scalar* setup window.

We assign the signals to the Input and Criteria channel, set the Criteria limit and decide whether we are looking for a latch conditions at rising or falling edge. Our setup should now look like this:

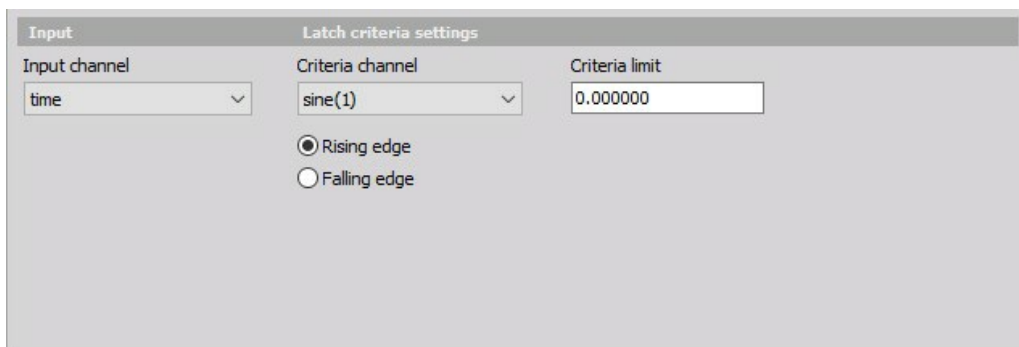


Image 24: Assign the signals to the Input and Criteria channel and set the Criteria limit

Now we can go to the *Measure* tab, where we can see a visual representation of the *Latch*.

Note that to get the exact picture below we turned off the *Interpolate asynchronous channels* option in the *Drawing option* of the recorder setup, to better demonstrate that our values are "latched".

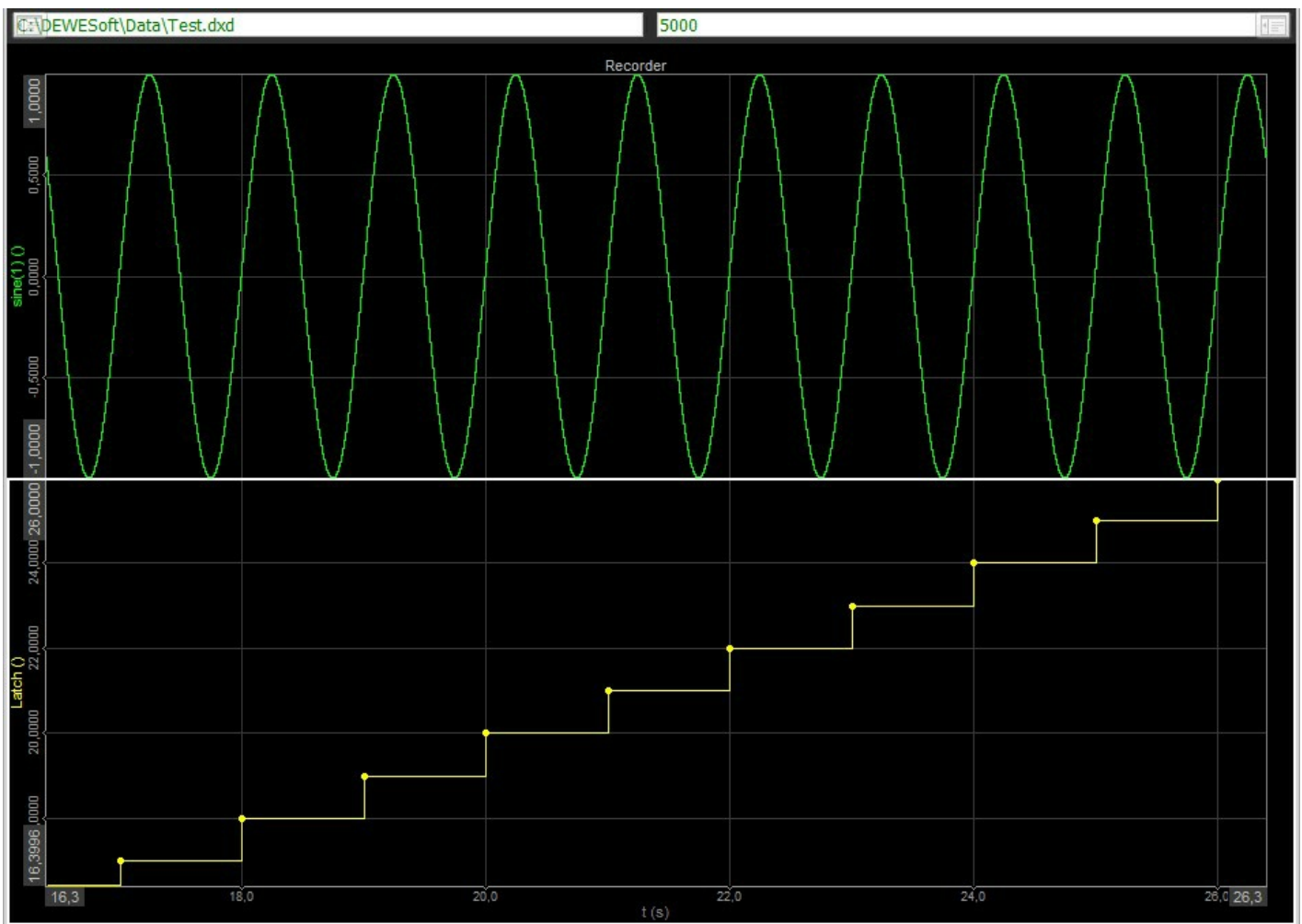


Image 25: Go to Measure mode and you can see a visual representation of the Latch. For better demonstration, turn off the Interpolate asynchronous channel

In the picture above we can see that every time the sine signal passes 0 at the rising edge threshold, the output channel outputs the current value of the time signal. The outputted value represents the time at which the sine signal passed 0.

Now change the edge type to falling edge and the criteria limit to 0,5. The results will now look like this:

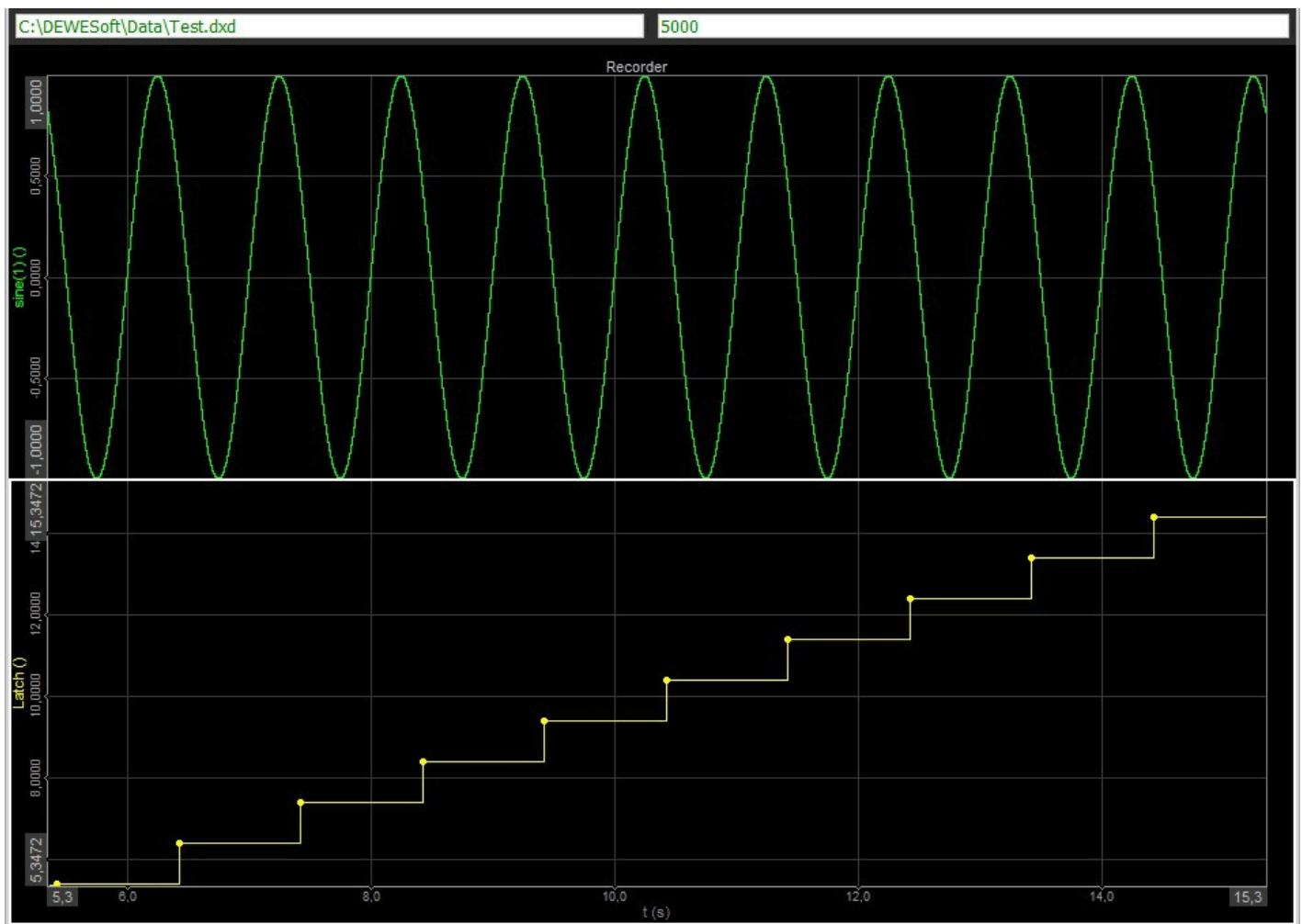


Image 26: Falling edge type and the criteria limit set to 0.5

Example II: Vector latch math explanation

Our Latch math example can only output scalar sample values. But what if we wanted to latch a vector channel at some condition? We can not just assign this channel as our input because our plugin does not know how to behave in the case of a vector input. This problem can also be solved using C++ Plugin and we will do this by making some changes to our current plugin.

To understand which changes we need, let's first make a slight detour and explain the different types of channels in Dewesoft.

Channel types

We can think of a channel in Dewesoft as a structure holding our signals. Channels can be split in two ways: the first split is based on the value type of the channel, and the second one on its time base.

Based on their value type, channels can be split into:

- scalar, vector, and matrix channels; where each of these could be
- real, or complex channels.

This means that if you have e.g. a real vector channel all the samples in the channel are vectors (of same dimensions) with real values.

Based on their time base, channels can be split into:

- synchronous;
- asynchronous; and
- single value.

Synchronous channels have equidistant time between consecutive samples. The time difference is defined by the acquisition sample rate and channel's sample rate divider. Asynchronous channels, on the other hand, don't have this restriction: the time between two consecutive samples can be arbitrary. Finally, single value channels don't care about time at all, they only contain one single sample per entire measurement with no timestamp - meaning each new sample in the single value channel simply overwrites the current value.

In Dewesoft only scalar channels can have a synchronous time base.

For a more visual example, the following image shows a sine curve in channels with different time bases.

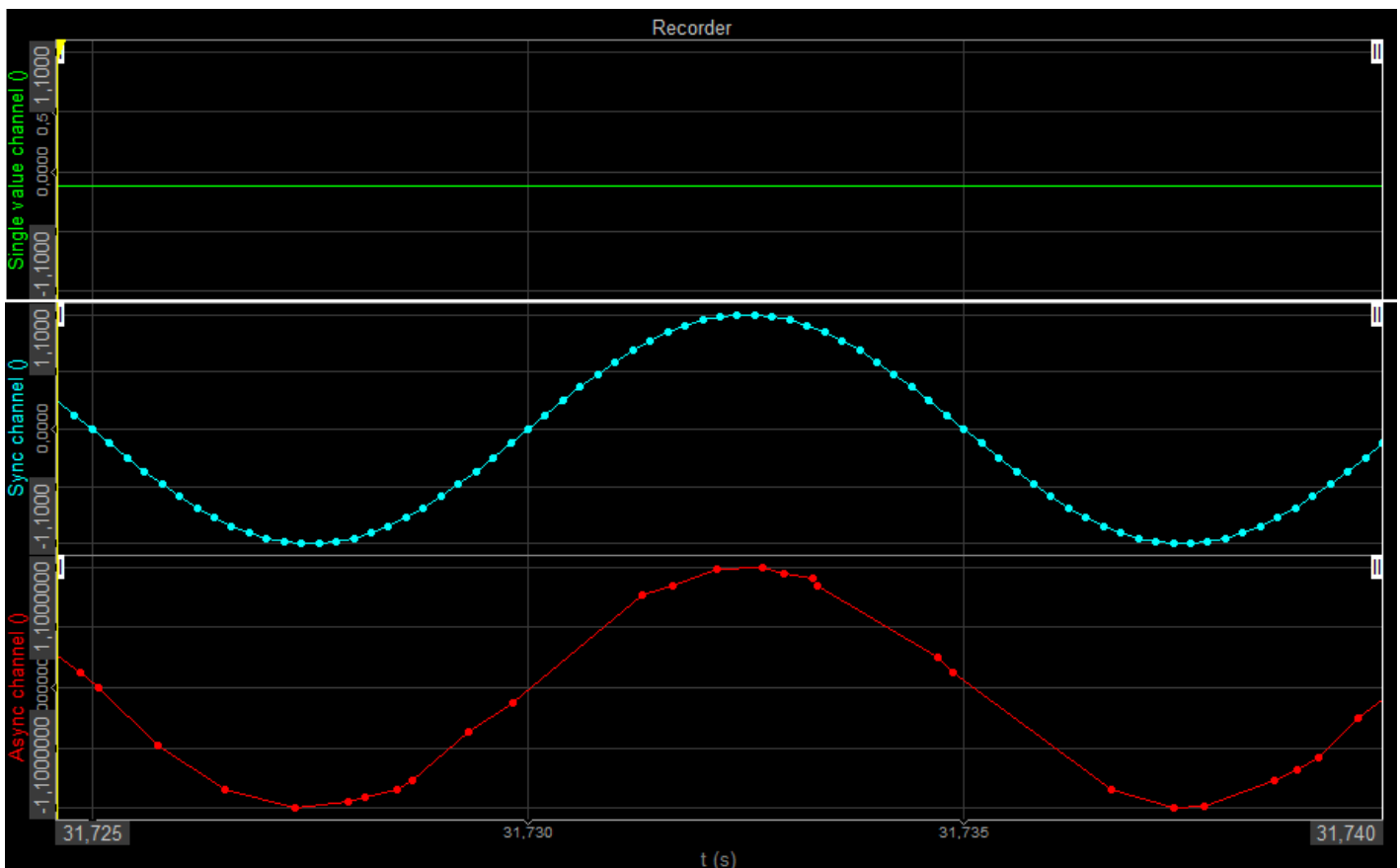


Image 27: Sine curve in channels with different time bases

Synchronous and asynchronous channels use a circular buffer. This means that when the direct buffer is full, the newest samples will override the oldest. Asynchronous channels also have an additional circular buffer used for storing timestamps.

Signals for testing our module

With these terms explained, let's go back to our problem. Like we created scalar channels at the very start of the training, let's now create some dummy vector channel for testing our modified module. One math module in Dewesoft that has vectors as output is Fourier transform, so let's use that. We add a new Fourier transform setup from the *Add math* drop-down list. We change the *Resolution* in *Calculation parameters* section to 256 lines (meaning the output vectors will have 256 elements) and click *Ok*.

Fourier transform setup

Input	Output
Search	<input type="checkbox"/> Complex <input type="checkbox"/> Overall RMS
<input checked="" type="checkbox"/> AI 1 <input type="checkbox"/> sine(1) <input type="checkbox"/> time	<input checked="" type="checkbox"/> Amplitude
	Calculation type
	<input checked="" type="radio"/> Block history <input type="checkbox"/> Average <input type="checkbox"/> History size (3D graph)
	<input type="radio"/> Overall (Averaged) 1 blocks 20 samples
	<input type="checkbox"/> Stop after 4 blocks
	Calculation parameters
	Window
	Blackman
	Resolution
	Lines 256 (lines = 256, df = 1,0 Hz, duration = 1,0 s)
	Amplitude type
	Peak
	DC cutoff
	None Hz
	Overlap
	0 %
	Weighting
	Lin

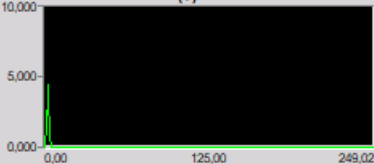
AI 1/AmpIFFT

Name: AI 1/AmpIFFT

Description: -

Units: V Color: [Red]

Preview Values X axis



Templates [Dropdown] Save

+ -

OK Cancel

Image 28: Add new Fourier transform setup from Add math drop-down list and change parameters for calculation and resolution

From Example I to Example II

Now that we have created a working plugin, we are ready to move to more complex concepts. We will use the project, which was created in *Example I* and modify it to support vector channel. *Example II* will be more complex, so we will have to modify things, which were simplified in *Example I*. We have already mentioned that all plugin logic, which does not need Dewesoft internals, should be defined inside the *proTutorial_LatchMath_vectorPluginLib* project, although we did not take that into account in *Example I*. So moving non-Dewesoft logic from the *DewesoftBridge* to the *proTutorial_LatchMath_vectorPluginLib* will be the first thing we will do.

DewesoftBridge is used for communicating with Dewesoft internals, that is why it should only contain functions which are meant to do that. This is why we will create a function for checking if we crossed the Criteria limit in the *proTutorial_LatchMath_vectorPluginLib*. In the *protutorial_latchmath_vector.h* file we declare the method:

```
class proTutorial_LatchMath_vector
{
public
    bool checkCrossedEdgeCriteria(float currentSampleCriteriaChannel, float nextSampleCriteriaChannel,
float criteriaLimit, int edgeType);
};
```

And in the *protutorial_latchmath_vector.cpp* file we define it.

```
bool proTutorial_LatchMath_vector::checkCrossedEdgeCriteria(float currentSampleCriteriaChannel,
float nextSampleCriteriaChannel,
float criteriaLimit,
int edgeType)
{
    bool crossedRisingEdgeCriteria = currentSampleCriteriaChannel <= criteriaLimit &&
nextSampleCriteriaChannel >= criteriaLimit;
    bool crossedFallingEdgeCriteria = currentSampleCriteriaChannel >= criteriaLimit &&
nextSampleCriteriaChannel <= criteriaLimit;

    if ((crossedFallingEdgeCriteria && edgeType == 1) || (crossedRisingEdgeCriteria && edgeType == 0))
        return true;
    else
        return false;
}
```

Now, we have to change the method *DewesoftBridge::onGetData()* in *dewesoft_bridge.cpp* file so it will use this new method.

```

STDMETHODIMP DewesoftBridge::onGetData()
{
    if (!inputChannel || !criteriaChannel)
        return S_OK;

    int blockSizeCriteriaChannel =
        (criteriaChannel->DBPos - (lastPosChecked % criteriaChannel->DBBufSize) + criteriaChannel-
>DBBufSize) % criteriaChannel->DBBufSize;
    int blockSizeInputChannel =
        (inputChannel->DBPos - (lastPosChecked % inputChannel->DBBufSize) + inputChannel-
>DBBufSize) % inputChannel->DBBufSize;

    int minBlockSize = min(blockSizeCriteriaChannel, blockSizeInputChannel);

    for (int i = 0; i < minBlockSize - 1; i++)
    {
        float currentSampleCriteriaChannel = criteriaChannel->DBValues[(lastPosChecked %
criteriaChannel->DBBufSize)];
        float nextSampleCriteriaChannel = criteriaChannel->DBValues[(lastPosChecked + 1) %
criteriaChannel->DBBufSize];

        bool crossedEdgeCriteria =
protutorial_latchmath_vector.checkCrossedEdgeCriteria(currentSampleCriteriaChannel,
                                                         nextSampleCriteriaChannel,
                                                         criteriaLimit,
                                                         edgeType);

        if (crossedEdgeCriteria)
        {
            float value = inputChannel->DBValues[(lastPosChecked + 1) % inputChannel->DBBufSize];
            outputChannel->AddAsyncSingleSample(value, (lastPosChecked + 1) / app->Data-
>SampleRateEx);
        }
        lastPosChecked++;
    }
    return S_OK;
}

```

Because vector channels are always of asynchronous type, we also have to change the `getSyncChannels()` method to `getAsyncChannels()` which will return all asynchronous channels. Do not forget to change the name of the method in `dewesoft_bridge.h` file and in `setup_window.cpp` where we call this method.

```

std::vector<IChannelPtr> DewesoftBridge::getAsyncChannels()
{
    std::vector<IChannelPtr> allChannels;

    app->Data->BuildChannelList();
    IChannelListPtr channels = app->Data->UsedChannels;

    for (int i = 0; i < channels->Count; i++)
        if (channels->GetItem(i) != outputChannel && channels->GetItem(i)->Async)
            allChannels.push_back(channels->GetItem(i));

    return allChannels;
}

```

Since our Input channel will now be a vector channel, we also have to make changes to `addChannelsToCBox()` method. We will change this method so the `inputChannelCBox` will only accept vector inputs and the `criteriaChannelCBox` will only accept scalar inputs.

```

void SetupWindow::addChannelsToCBox()
{
    criteriaChannelCBox.clear();
    inputChannelCBox.clear();

    std::vector<IChannelPtr> allChannels = bridge.getAsyncChannels();
    for (int i = 0; i < allChannels.size(); i++)
    {
        std::string channelName = allChannels[i]->Name;
        if (allChannels[i]->ArrayChannel)
            inputChannelCBox.addItem(channelName);
        else
            criteriaChannelCBox.addItem(channelName);
    }
}

```


Example II: Saving and loading settings

Let's now start modifying our project. The reason for removing the non-Dewesoft logic from `dewesoft_bridge.cpp` file will be seen soon, but not just yet.

It may be annoying for you to select channels, enter criteria limit and determine rising or falling edge every time you start your plugin in Dewesoft. There is no need to do that anymore because C++ Plugin allows you to store and save your plugin settings in an XML file. In our case, we should save information about Criteria limit, Input channel name and Criteria channel name. To save enum settings we will change it to an integer and save it using `WriteInteger` function.

Storing is done with methods that contain *Write* prefix (e.g. `WriteInteger`) and reading is done with methods that contain *Read* prefix (e.g. `ReadInteger`). We will use class member `doc` which is automatically set. Parameters for Write and Read methods depend on data type which should be saved, but we can generalize it for every data type.

- The first parameter is the pointer to the node which Dewesoft uses. *This parameter is the same for every type.*
- The second parameter is the name of XML element under which your setting is saved. *This parameter should be unique for every setting.*
- The third parameter is the actual value to be stored.
- The fourth parameter is the default value to be stored if the actual value is not yet initialized.

We will save plugin settings when setup is saved and load them when a new setup is loaded.

Storing our settings inside `dewesoft_bridge.cpp` file is done in `onSaveSetup` as seen in the code below.

```
void DewesoftBridge::onSaveSetup(const Setup& setup)
{
    auto node = setup.getNode();
    node->write("criteriaLimit_vector", getCriteriaLimit(), 0);
    node->write("inputChannelName_vector", getInputChannelName().c_str(), "");
    node->write("criteriaChannelName_vector", getCriteriaChannelName().c_str(), "");
    node->write("edgeType_vector", getEdgeType(), 0);
}
```

Loading our settings inside `dewesoft_bridge.cpp` file is done as seen in the code below. We need to set the default values in Read methods because `onLoadSetup` method is also called when a new plugin is created and XML elements (in `node` variable) do not even exist yet. They are first added when the plugin is saved. Therefore names for our *Input* and *Criteria channel* will be "" until the user chooses a channel name in ComboBox.

```

void DewesoftBridge::onLoadSetup(const Setup& setup)
{
    const auto node = setup.getNode();

    float criteriaValue = 0;
    node->read("criteriaLimit_vector", criteriaValue, 0);
    setCriteriaLimit(criteriaValue);

    node->read("inputChannelName_vector", inputChannelName, "");
    node->read("criteriaChannelName_vector", criteriaChannelName, "");

    long edgeTypeVectorIndex;
    node->read("edgeType_vector", edgeTypeVectorIndex, 0);
    edgeType = edgeTypeVectorIndex == 0 ? RisingEdge : FallingEdge;

    mountChannels();
}

```

We will also create a public method which will be used to fill input fields in *SetupWindow* class.

```

//setup_window.h
void setSavedValues();

```

```

// setup_window.cpp

void SetupWindow::setSavedValues()
{
    inputChannelCBox.setSelectedIndex(inputChannelCBox.indexOf(bridge.getInputChannelName()));

    criteriaChannelCBox.setSelectedIndex(criteriaChannelCBox.indexOf(bridge.getCriteriaChannelName()));

    latchCriteriaTBox.setText(std::to_string(bridge.getCriteriaLimit()));

    edgeTypes savedEdgeType = bridge.getEdgeType();
    if (savedEdgeType == RisingEdge)
        risingEdgeRButton->setChecked(risingEdgeRButton.isEnabled());
    else
        fallingEdgeRButton->setChecked(fallingEdgeRButton.isEnabled());
}

```

Input fields should also be filled every time when the plugin user interface is entered so the user will not have to set channel names and *Criteria limit* again.

```
void DewesoftBridge::onSetupEnter()
{
    setupWindow->addChannelsToCBox();
    setupWindow->setSavedValues();
}
```

When Dewesoft is now opened and saved setup is loaded, saved values will load automatically.

Example II: Vector latch math

Now that we know what *Example II* is about, we are ready to start with the main code: the code for handling vector channels.

In order to create our *Output channel* to support vectors, we will have to modify its mounting inside Dewesoft. It will accept an *array* containing values of the type `Single`. That is why we have to set the channel's dimension (property `DimCount`) to 1. If we would need our *Output channel* to support matrix samples, we would have set `DimCount` property to 2. We also set the size of the vector we want to output. In this method, we will set it to 1 and we will change this property later to fit the size of the input vectors.

```
void DewesoftBridge::mountChannels()
{
    const std::vector<int> chIndexVector = {1};
    outputChannel = pluginGroup->MountChannelEx(pluginGuid, long(chIndexVector.size()),
    fromStdVec(chIndexVector));

    outputChannel->SetDataType(long(ChannelDataType::Single));
    outputChannel->Name = "Latch";
    outputChannel->Unit_ = "";
    outputChannel->SetAsync(true);
    outputChannel->ExpectedAsyncRate = 5;
    outputChannel->Used = true;
    outputChannel->ArrayChannel = true;

    outputChannel->ArrayInfo->DimCount = 1;
    outputChannel->ArrayInfo->DimSizes[0] = 1;
    outputChannel->ArrayInfo->Init();
}
```

Every time we make changes to the dimension of the array channel by setting the `DimCount` or `DimSizes` properties, we need to call the `Init()` method.

There is one thing we need to keep an eye on: `ExpectedAsyncRate` property of our output channel. This warrants a slight detour:

Expected async rate per second

If our module contains **asynchronous output channels**, we have to set their expected rate per second. You can think of this as "approximately how many samples will I be adding to this channel per second". We can change the value of this setting in `DewesoftBridge::mountChannels()` by modifying the channel's `ExpectedAsyncRate`. This setting is required because we need to help Dewesoft figure out much memory it needs to reserve for our channel. While we can calculate this value in any way we want, it can be useful if we know the rate of our output channel is somehow going to be connected to the rate of some other input channel, in which case we can simply set the `outputChannel->ExpectedAsyncRate` to `inputChannel->ExpectedAsyncRate`.

We can set `ExpectedAsyncRate` to a completely arbitrary value, but if the `ExpectedAsyncRate` is set too high Dewesoft will reserve too much memory and if it is set too low we might lose some important data. We don't need to set `ExpectedAsyncRate` to the exact value, but we need to specify it to within an order of magnitude.

Note that since our output channel is now a vector channel, Dewesoft will be reserving `ArrayInfo->ArraySize` times as much memory as it would if the channel were a simple scalar channel; so we have to be extra careful to not put in a number that is too high.

Method `DewesoftBridge::onEstablishConnections()` is called when the acquisition is started (this occurs when *ch.setup* is entered). In this method, we will retrieve the Input and Criteria channels by their names. If names are not yet set (this will happen every time when no channel had been set in ComboBoxes containing channel names), we do not set any channels.

```
STDMETHODIMP DewesoftBridge::onEstablishConnections()
{
    setInputChannel(inputChannelName.c_str());
    setCriteriaChannel(criteriaChannelName.c_str());

    return S_OK;
}
```

As mentioned before, we still need to set the size of the vectors we want to output. This needs to be done before Dewesoft reserves space to run our plugin, so we have to add an event that gets triggered before this happens. This event is called `evPreInitiate` and we need to add it to *plugin_impl.cpp* file, where we also define the method that gets triggered when this event happens. The method needs to be declared in the *plugin_impl.h* file.

```
// plugin_impl.h
//IPlugin4
STDMETHODIMP eventBeforeReserveMemory();

// plugin_impl.cpp
STDMETHODIMP Plugin::raw_OnEvent(enum EventIDs eventID, VARIANT inParam, VARIANT* outParam)
{
    // ...
    switch (eventID)
    {
        // ...
        case evPreInitiate:
            returnValue = eventBeforeReserveMemory();
            break;
        // ...
    }
    return returnValue;
}

STDMETHODIMP Plugin::eventBeforeReserveMemory()
{
    return bridge.onBeforeReserveMemory();
}
```

As we see in the code above the method that gets triggered when `evPreInitiate` happens triggers a method defined in `DewesoftBridge`. We will add the declaration and definition of this method to the bridge.

```

// dewesoft_bridge.h
STDMETHODIMP onBeforeReserveMemory();

// dewesoft_bridge.cpp
STDMETHODIMP DewesoftBridge::onBeforeReserveMemory()
{
    if (!inputChannel)
        return S_OK;

    outputChannel->ArrayInfo->DimCount = 1;
    outputChannel->ArrayInfo->DimSizes[0] = inputChannel->ArraySize;
    outputChannel->ArrayInfo->Init();
    outputChannel->ArrayInfo->AxisDef[0]->Name = inputChannel->ArrayInfo->AxisDef[0]->Name;
    outputChannel->ArrayInfo->AxisDef[0]->_Unit = inputChannel->ArrayInfo->AxisDef[0]->_Unit;

    outputChannel->ArrayInfo->AxisDef[0]->AxisType = atFloatLinearFunc;
    outputChannel->ArrayInfo->AxisDef[0]->StartValue = inputChannel->ArrayInfo->AxisDef[0]-
>StartValue;
    outputChannel->ArrayInfo->AxisDef[0]->StepValue = inputChannel->ArrayInfo->AxisDef[0]-
>StepValue;

    return S_OK;
}

```

We have now successfully mounted an output channel to Dewesoft, set its axis values, and updated the *Output channel* to support outputting vector values from the *Input channel*. All there is left to do is to actually output vector values to the *Output channel*.

This will be done inside `onGetData()` method. For precautionary reasons, we also check if the *Input channel* or *Criteria Channel* is *nullptr*, otherwise, the *Input channel* could be uninitialized when new setup was created. To insert data to the *Output channel* we will use the method `AddAsyncData(...)` which allows insertion of a vector to the *Output channel*.

```

STDMETHODIMP DewesoftBridge::onGetData()
{
    if (!inputChannel || !criteriaChannel)
        return S_OK;

    if (inputChannel->DBDataSize == 0 || criteriaChannel->DBDataSize == 0)
        return S_OK;

    int blockSizeCriteriaChannel =
        (criteriaChannel->DBPos - (lastPosChecked % criteriaChannel->DBBufSize) + criteriaChannel-
        >DBBufSize) % criteriaChannel->DBBufSize;

    for (int i = 0; i < blockSizeCriteriaChannel - 1; i++)
    {
        float currentSampleCriteriaChannel = criteriaChannel->DBValues[lastPosChecked %
        criteriaChannel->DBBufSize];
        float nextSampleCriteriaChannel = criteriaChannel->DBValues[(lastPosChecked + 1) %
        criteriaChannel->DBBufSize];

        bool crossedEdgeCriteria =
        protutorial_latchmath_vector.checkCrossedEdgeCriteria(currentSampleCriteriaChannel,
                                                                nextSampleCriteriaChannel,
                                                                criteriaLimit,
                                                                edgeType);

        if (crossedEdgeCriteria)
        {
            double time = criteriaChannel->DBTimeStamp[(lastPosChecked + 1) % criteriaChannel->DBBufSize];

            int posInputChannel = (inputChannel->DBPos - 1);

            // reading vector from channel
            std::vector<float> results;
            for (int j = 0; j < outputChannel->ArraySize; j++)
            {
                float value = inputChannel->DBValues[posInputChannel * inputChannel->ArraySize + j];
                results.push_back(value);
            }

            if (outputChannel)
                outputChannel->AddAsyncData(fromStdVec(results), time);
        }
        lastPosChecked++;
    }
    return S_OK;
}

```


Example II: Debugger

Because C++ Plugin uses Visual Studio IDE, it supports really efficient debugging. It helps you find semantic errors, see variable values in real-time, set breakpoints, add watches on variables to see values changing, and much more.

If we now run our plugin by pressing the F5 on the keyboard, load the setup we created for testing the plugin and assign the channels as seen in image 29.

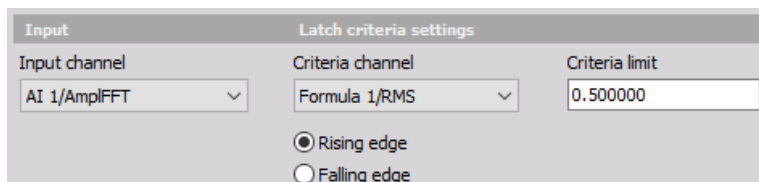


Image 29: Run plugin by pressing on F5 and assign the channels

Then enter the *Measure* mode in Dewesoft we quickly get an error message.

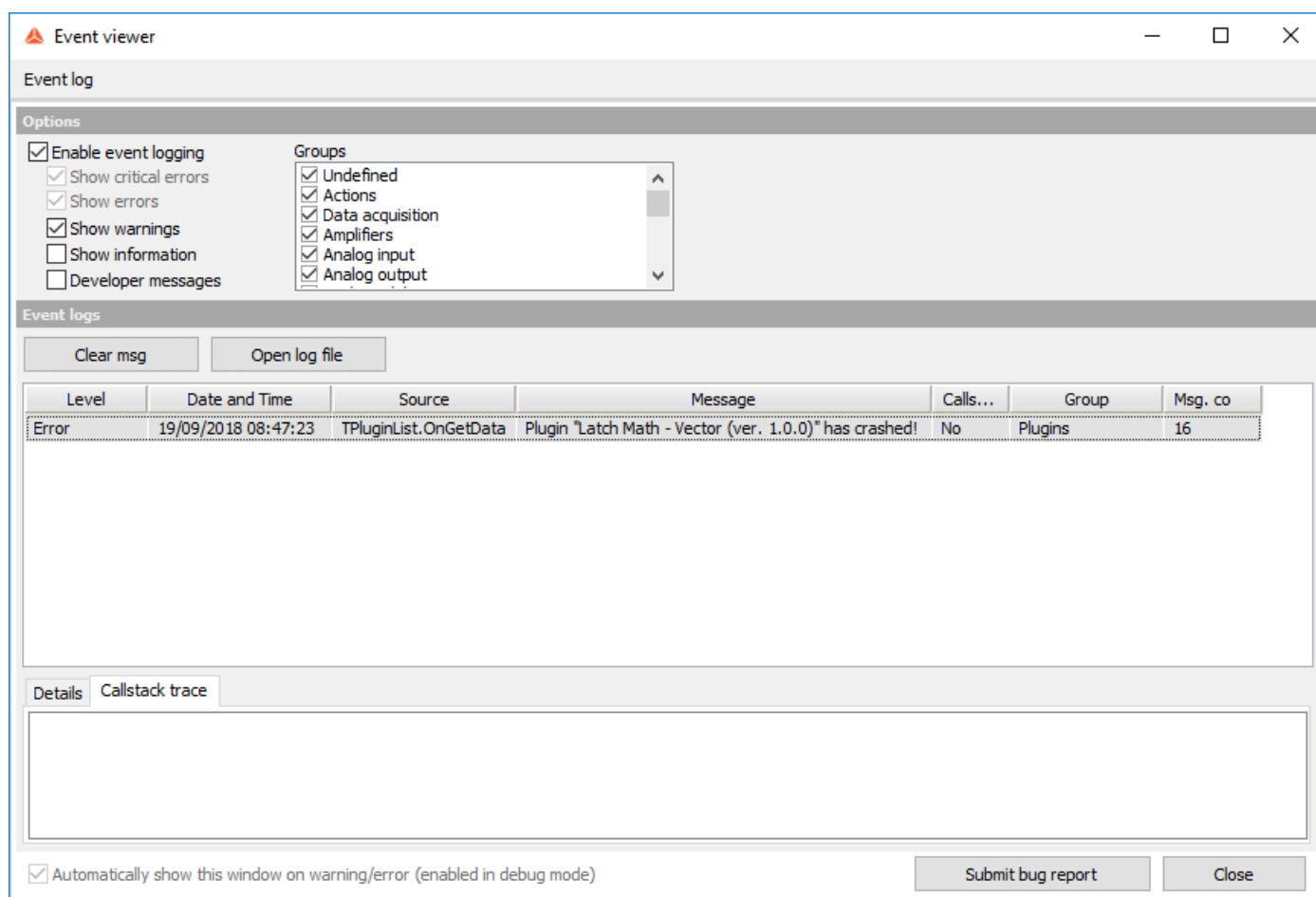


Image 30: Error message that occurred when entering the Measure mode

We can see that the error is thrown in the `OnGetData()` method. We will now set a breakpoint inside of this method, start our plugin with F5 keystroke again and go to *Measure* tab. When the program execution reaches our breakpoint we will be

able to look at every line of code to find out, which one is causing our plugin to crash. To move to the next line use the F10 keystroke (this will step over function calls).

We will try to find the error using Visual Studio Debugger.

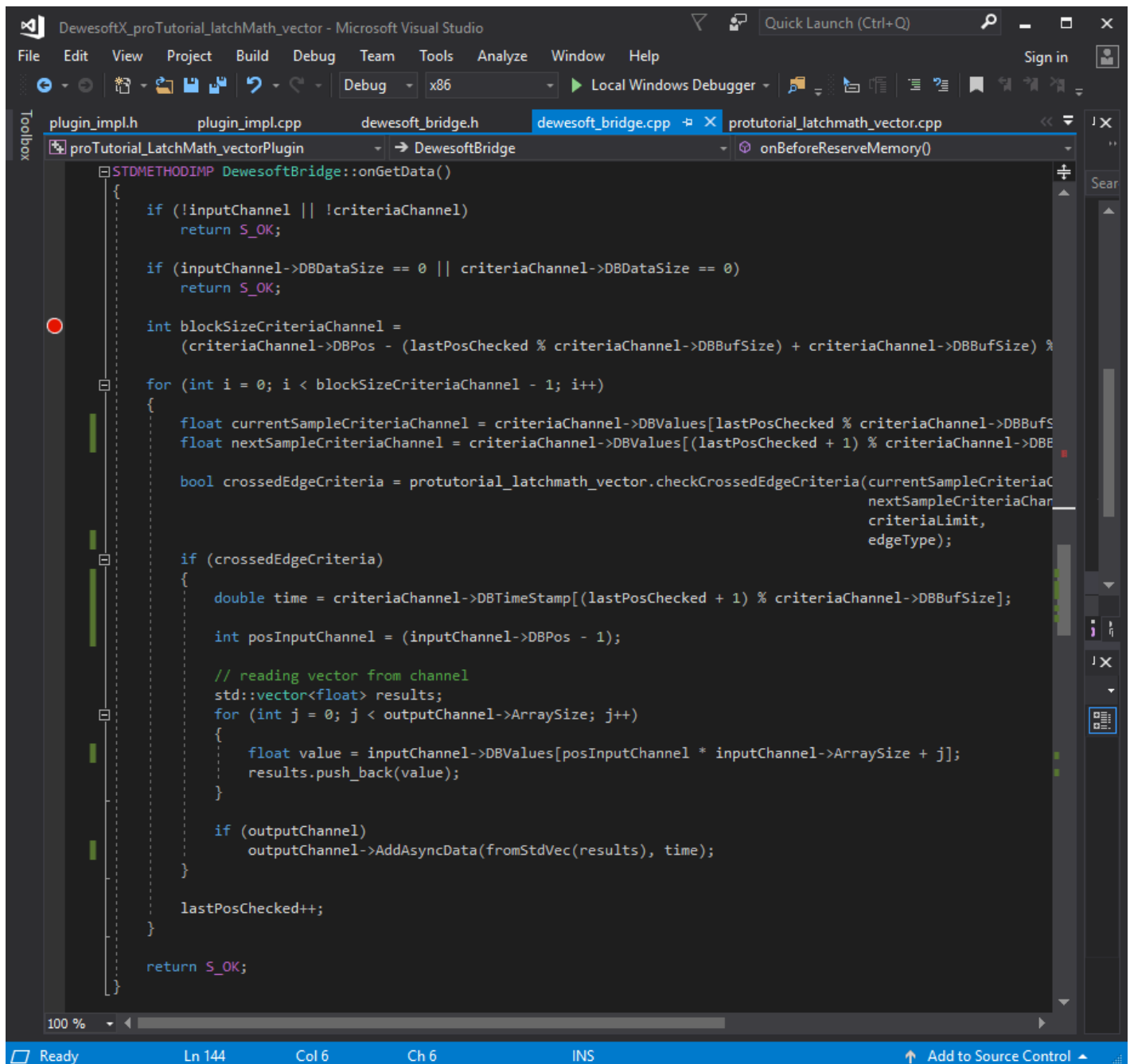


Image 31: Find an error using the Visual Studio program

After some iterations, we can see that `posInputChannel` has a value of -1 when the last position of the Input channel is 0. This happens because the data in channels is stored in circular buffers and when we try to access the channel values using the `inputChannel->DBValues[...]` we get an error.

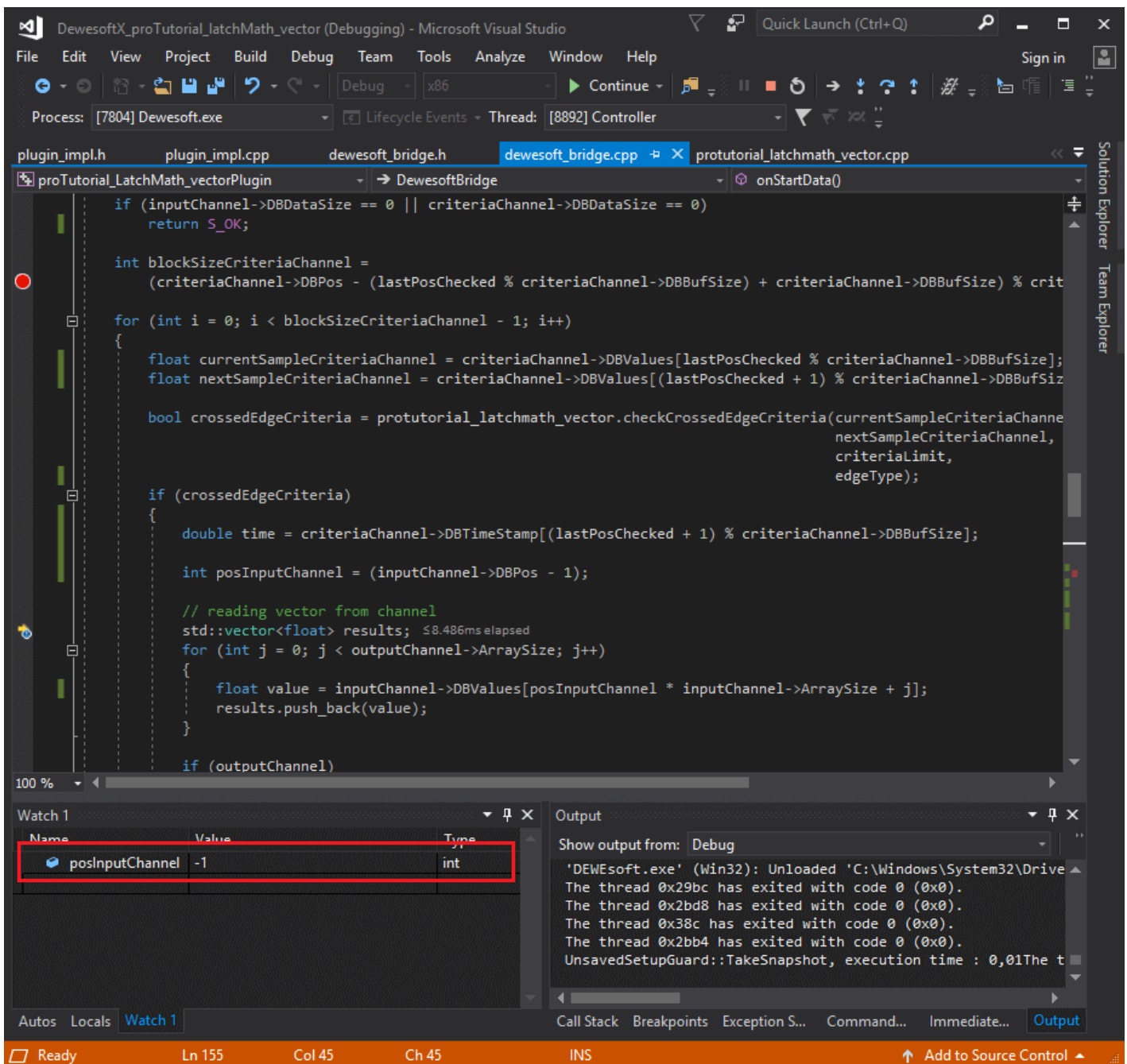


Image 32: `posInputChannel` has the value of -1 when the last position of the Input channel is 0, which causes an error

Debugging will not always be that easy. In this case, *Call stack* (the window in the bottom-right corner) is something to keep an eye on. It will show you how methods were called and allow you to step up the stack by clicking the call stack line to see the actual call.

Now we know what is the source of the problem, so we are able to fix it. We change the `posInputChannels` to first check if the position of the last value received in the Input channels is 0, and if it is, the value we want to output is not -1 but it is the last element of the buffer.

```
int posInputChannel = inputChannel->DBPos != 0 ? inputChannel->DBPos - 1 : inputChannel->DBBufSize - 1;
```

Example II: Unit testing

One of the important stages of plugin development is testing. Luckily, C++ Plugin allows you to run automated unit tests, which will shorten your testing time. Unit testing is a software testing method that tests your plugin by running predefined test cases and checking if the result is correct.

Unit testing is one of the main reasons for separating the Dewesoft logic (defined inside *DewesoftBridge*) and non-Dewesoft logic (defined inside *proTutorial_LatchMath_vectorPluginLib*).

We can now test each segment without having to start Dewesoft. In order to perform unit tests, you have to set *proTutorial_LatchMath_vectorPluginTest* as your startUp project (right clicking it and clicking on *set as startUp project*). We will get rid of sample code so we will only test functions which we need. Every test is written inside *protutorial_latchmath_vector_item_test.cpp* file.

We will create two simple unit tests, to test our method `checkCrossedEdgeCriteria(...)`. If the returned value is the same as the one set, the unit test will output "PASSED".

Here you can see two examples of a unit test (both should successfully pass).

```

TEST_F(proTutorial_LatchMath_vectorItemTest, CheckCriteriaLimitRisingEdge)
{
    proTutorial_LatchMath_vector protutorial_latchmath_vector;

    float currentSampleCriteriaChannel = 0.49;
    float nextSampleCriteriaChannel = 0.51;
    float criteriaLimit = 0.5;
    int edgeType = 0; // RisingEdge

    bool crossedEdgeCriteria =
protutorial_latchmath_vector.checkCrossedEdgeCriteria(currentSampleCriteriaChannel,
                                                         nextSampleCriteriaChannel,
                                                         criteriaLimit,
                                                         edgeType);

    ASSERT_TRUE(crossedEdgeCriteria);
}

TEST_F(proTutorial_LatchMath_vectorItemTest, CheckCriteriaLimitFallingEdge)
{
    proTutorial_LatchMath_vector protutorial_latchmath_vector;

    float currentSampleCriteriaChannel = -0.12;
    float nextSampleCriteriaChannel = 0.01;
    float criteriaLimit = 0;
    int edgeType = 1; // FallingEdge

    bool crossedEdgeCriteria =
protutorial_latchmath_vector.checkCrossedEdgeCriteria(currentSampleCriteriaChannel,
                                                         nextSampleCriteriaChannel,
                                                         criteriaLimit,
                                                         edgeType);

    ASSERT_FALSE(crossedEdgeCriteria);
}

```

To see the result of unit tests, we have to start your project. The result is outputted in the command prompt, which terminates after the testing is completed. In order for the command prompt to stay visible, we should place a breakpoint right before `return res;` line of code in `main.cpp` file. In the picture below you can see an example of setting a breakpoint.

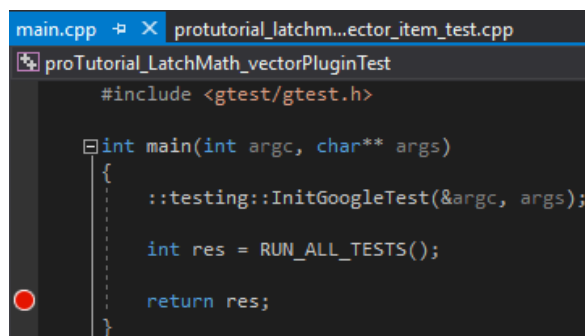
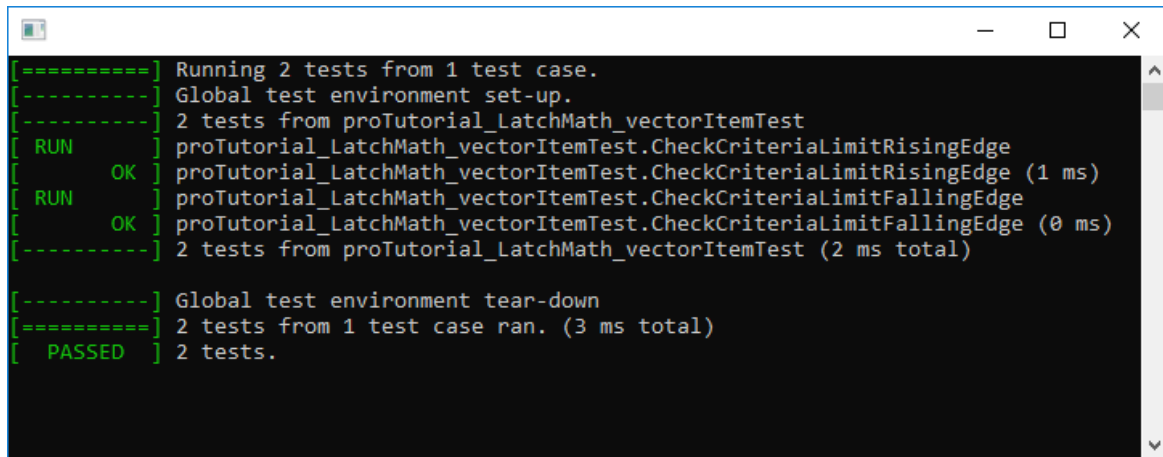


Image 33: Place a breakpoint right before 'return res;'

The output in the command prompt should look like that.



```
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from proTutorial_LatchMath_vectorItemTest
[ RUN      ] proTutorial_LatchMath_vectorItemTest.CheckCriteriaLimitRisingEdge
[      OK   ] proTutorial_LatchMath_vectorItemTest.CheckCriteriaLimitRisingEdge (1 ms)
[ RUN      ] proTutorial_LatchMath_vectorItemTest.CheckCriteriaLimitFallingEdge
[      OK   ] proTutorial_LatchMath_vectorItemTest.CheckCriteriaLimitFallingEdge (0 ms)
[-----] 2 tests from proTutorial_LatchMath_vectorItemTest (2 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 2 tests.
```

Image 34: Command prompt output

Example II: Output

If you now start your plugin and set your Input channel to *AI 1/AmplFFT* and Criteria channel to *sine(1)*, as seen in the image 35.

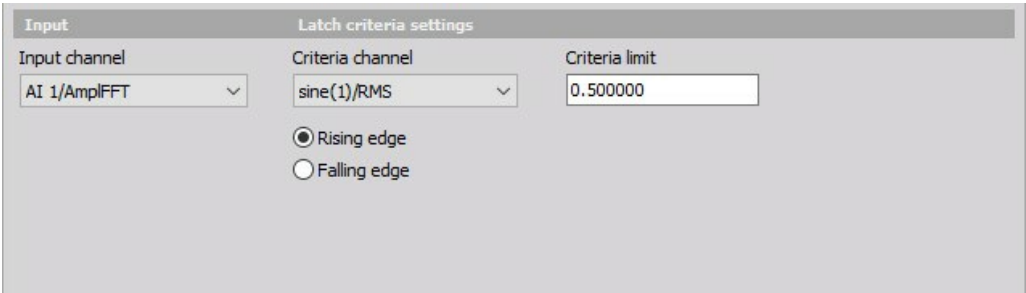


Image 35: Set your Input channel to AI 1/AmplFFT and Criteria channel to sine(1)

You will be able to see the outputted vectors on a 3D graph. A new vector is outputted every time the *sine(1)* signal passes the value 0.5 on the rising edge.

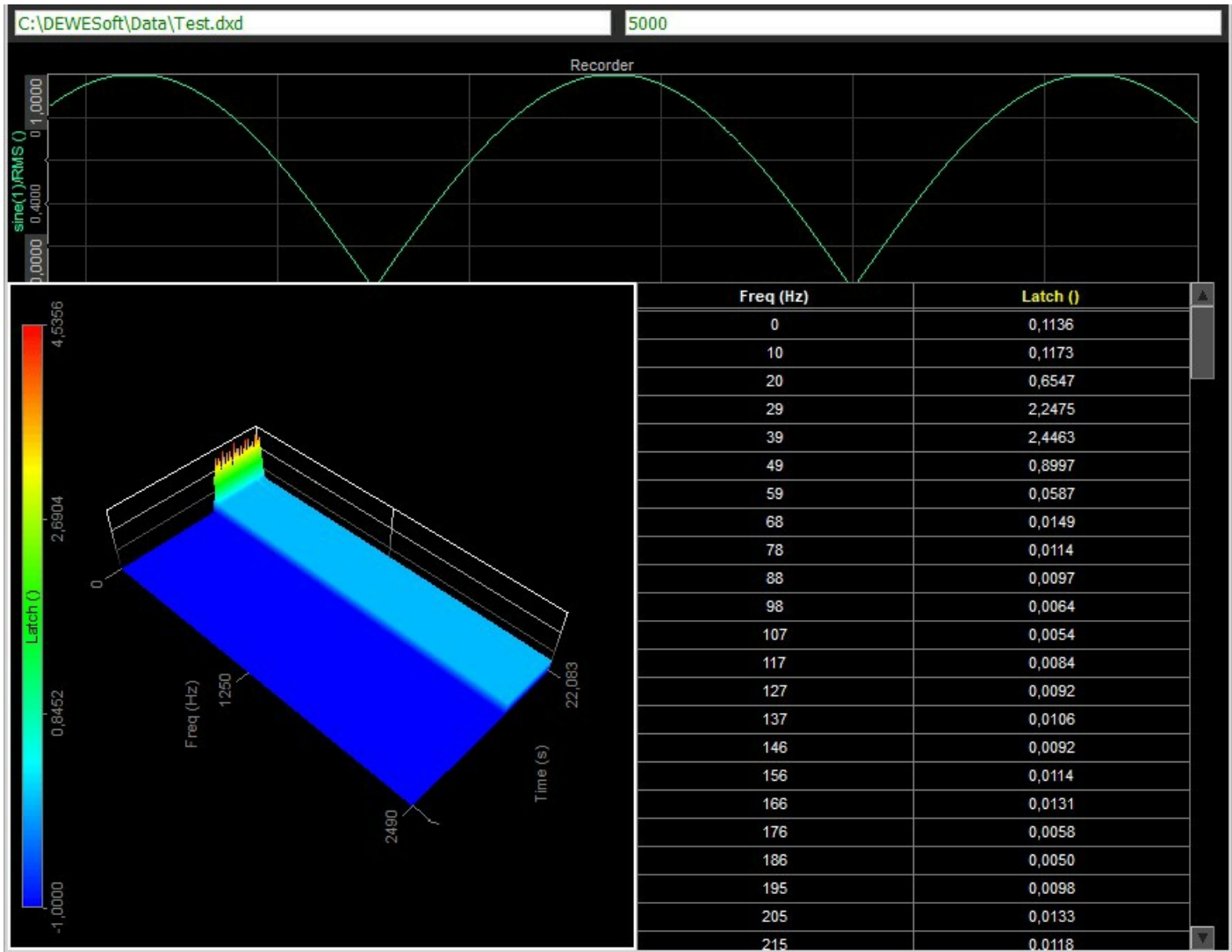


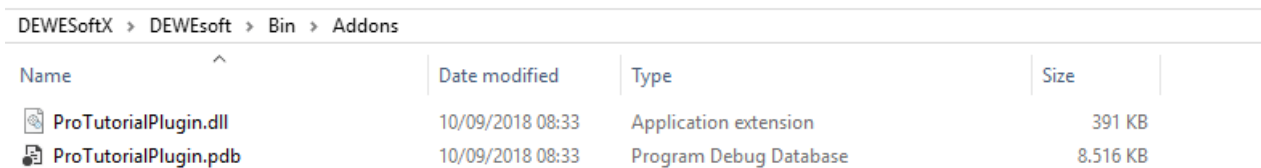
Image 36: Outputted vectors shown on a 3D graph

As you can see, whenever $\text{sine}(1)$ crosses the *Criteria limit*, we output the last vector sample from our input channel *AI1/AmplFFT* to our *Output channel*.

How to Import/Export the C++ plugin?

In this Pro Training, we have created a pretty useful plugin, which inserts samples into our *Output channel*. We might want to use it in other setups or on other computers. C++ Plugin packs your plugin into an external library, which can be inserted into any Dewesoft around the world.

Your C++ Plugin is found inside a file with .dll extension (it contains instructions that Dewesoft can call upon to do certain things, based on the purpose of your plugin). To export it, you need to locate the .dll file first. It can be found inside *DEWESoftX\DEWESoft\Bin\Addons* folder.





DEWESoftX > DEWESoft > Bin > Addons				
Name	Date modified	Type	Size	
 ProTutorialPlugin.dll	10/09/2018 08:33	Application extension	391 KB	
 ProTutorialPlugin.pdb	10/09/2018 08:33	Program Debug Database	8.516 KB	

Image 37: Locate the .dll file in the Addons folder of the installation

To import your plugin you have to copy and paste file with .dll extension into any Dewesoft that requires your plugin. You need to paste it inside *Addons* folder so Dewesoft will be able to automatically recognize and load it.

Your C++ Plugin also creates a file with the .pdb extension, which contains instructions for your debugger. It is not necessary to export it with your .dll file in order for your plugin to work, but in case the imported plugin will be debugged, copying it together with a .dll file is a good idea.

Comparison with the other ways of extending Dewesoft

At this point, you might have a pretty good grasp on how to use C++ Script. But C++ Script is just one of many ways of extending Dewesoft to suit your needs, and it might be slightly confusing to try and figure out if it actually is the best solution for your task. So in this section, we briefly compare different approaches and list a couple of pros and cons which can hopefully help you pick the right tool.

Just a quick reminder: Dewesoft is a big software. It is always worth trying to figure out if Dewesoft can already do whatever you need out of the box, because if it can, you will waste very little of your time, and will have full support from Dewesoft team if anything doesn't work as expected.

Extention	Description
Formula	If you want to manipulate channels in a simple way, the Formula module is usually the best one to start experimenting with. Because of its ease of use, it can serve as a great starting point for quick prototyping, and it is usually good enough for most typical problems (signal generation, simple manipulation of data in channels, etc.).
C++ Script	During its development, we mainly envisioned C++ Script as a tool to create custom math modules that you could export and use just like standard Dewesoft modules. C++ Script is probably a good second step after your approach with Formula modules gets too complicated, too cluttered, or, in the worst case, you cannot figure out how to solve the problem with them.
Plugins	If you want to develop anything other than math modules, or if you tried creating a module with C++ Script and it proved to not be fast or powerful enough, or if you want to create a completely custom GUI for your module, Plugins are the right way to go. With Dewesoft Plugins you get access to entire Dewesoft from your code, including direct access to buffers behind channels, making Plugins incredibly fast compared to C++ Script.
Sequencer / DCOM	Sequencer and DCOM are slightly different than the other 3 approaches mentioned in this section. Regardless, they serve a very useful purpose and deserve to be mentioned here: they are used to automate a person clicking on different parts of the Dewesoft UI. The difference among them is that with Sequencer you can create sequences by dragging and dropping graphical blocks (requiring little to no experience with programming) while with DCOM you need to use a programming language. Sequencer is easier to use, but you get much more control with DCOM.

Extention	PROS	CONS
Formula	<ul style="list-style-type: none">- The most intuitive of all the approaches, very simple to use.- Integrated fully into Dewesoft meaning no set up required to get running.	<ul style="list-style-type: none">- Input channels are fixed in the formula, making reusability a lot of work.- While it supports combining arbitrarily many input channels, it always produces just one output channel.- Poor support for non-scalar channels.
C++ script	<ul style="list-style-type: none">- Dewesoft setups look much nicer as you (usually) only need one C++ Script to solve a problem that would require a bunch of Formula modules- Reusability and generality of your module: you can hide the code from the end-user and only expose the <i>Published setup</i> tab.- It can work with an arbitrary amount of input and output channels.	<ul style="list-style-type: none">- Requires familiarity with at least basics of programming in C++.- Difficult to test and debug.

Plugins	<ul style="list-style-type: none"> - Much easier to write nice code with proper unit tests. - Full control over the creation of GUI, access to Dewesoft internals, and blazing fast. - It can be used to create custom export formats, custom visual controls, add support for additional acquisition devices, ... - Made to work with Visual Studio, giving you access to a great debugger, code completion, and other static analysis tools. 	<ul style="list-style-type: none"> - Requires Visual Studio. - Much harder to learn to use than C++ Script.
Sequencer / DCOM	<ul style="list-style-type: none"> - It can be used to create an automated sequence of events in Dewesoft. - Creator of the sequence can hide the details from the end-user, exposing only a simple user interface to control Dewesoft. 	/