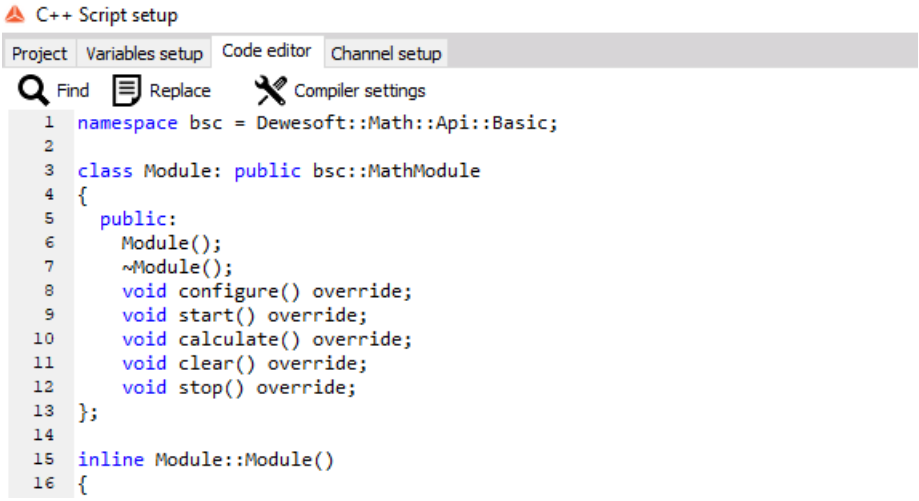
C++ Script




```
1  namespace bsc = Dewesoft::Math::Api::Basic;
2
3  class Module: public bsc::MathModule
4  {
5    public:
6      Module();
7      ~Module();
8      void configure() override;
9      void start() override;
10     void calculate() override;
11     void clear() override;
12     void stop() override;
13 };
14
15 inline Module::Module()
16 {
```

What is C++ Script?

Did you ever want to perform some calculations on a signal inside of Dewesoft but couldn't find the right module/function? Ever struggled with implementing a non-trivial algorithm and ended up with a mess of formula modules (only for it to not work because of limitations of the formula module)?

With the help of the C++ Script, **you can write your own math module** that can do exactly what you want, and can be ***imported into any setup you might need it in.*** You can take nearly arbitrarily many input channels, process the data using very simple abstractions in modern C++, and output the results into arbitrarily many output channels.

The way C++ Script works behind the scenes is it takes your C++ code and compiles it into an external library which Dewesoft can automatically load and communicate with. It then continuously fills the data from the input channels and, based on the code you wrote inside the C++ Script, outputs the processed data into output channels, taking care of the hard part of handling communication with Dewesoft for you.

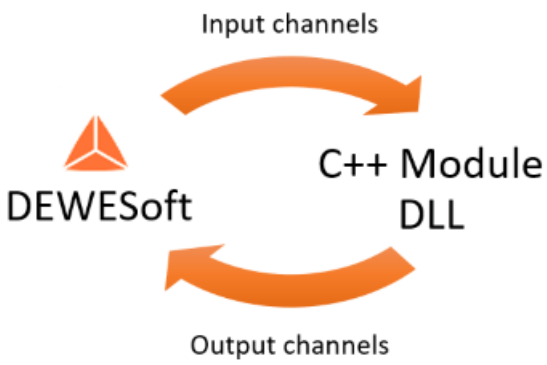


Image 1: Principle of C++ script

All the examples of C++ Scripts we will create in this tutorial and the setups we will use are available on Dewesoft's webpage under Support > Developer Downloads > Example C++ scripts (note that you have to be logged in to access the C++ Script section), so while you are highly encouraged to follow along on your own, you can refer to that if you get lost.

How to install the C++ Script?

C++ Script is available to Dewesoft users by default in Dewesoft X3 software versions SP6 and on. Since we want to write our own custom C++ Scripts, the one thing you need to do is make sure you **enable** *DSMinGW option under External Dependencies* during the installation of Dewesoft with an *online version of the installer* (offline versions don't have this option, but you can always install DSMinGW manually, as described below).

DSMinGW is a package of compilers for C++ that allows you to write and compile your own C++ Scripts in Dewesoft.

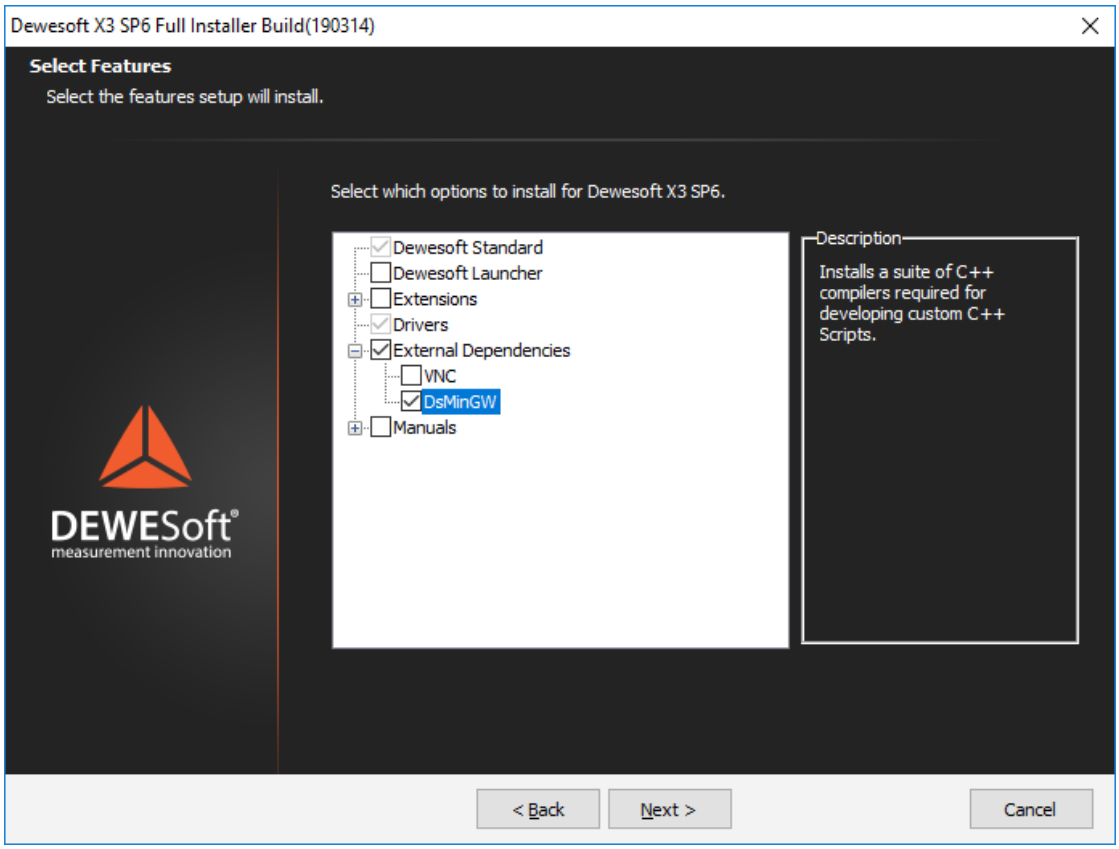


Image 2: Enabling of DsMingGW option with an online installer

If you didn't install DSMinGW during the installation of Dewesoft, you can always go to Windows' *Apps & features* (or, *Add or remove programs* in older versions of Windows), find and click on *Dewesoft*, and finally on *Modify* button. There you will be able to modify your installation and include DSMinGW.

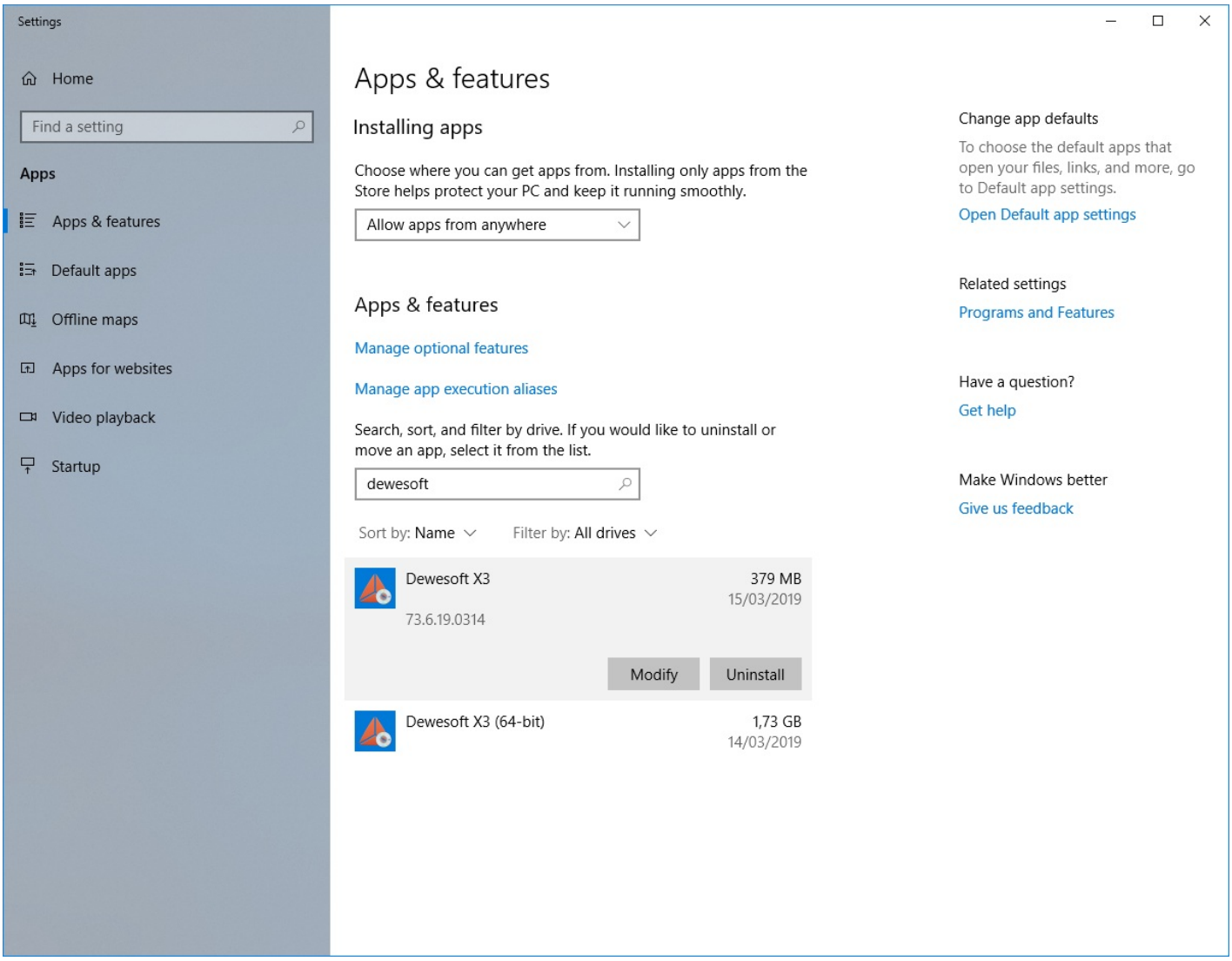


Image 3: For offline modification open Apps & Features and find Dewesoft to modify it

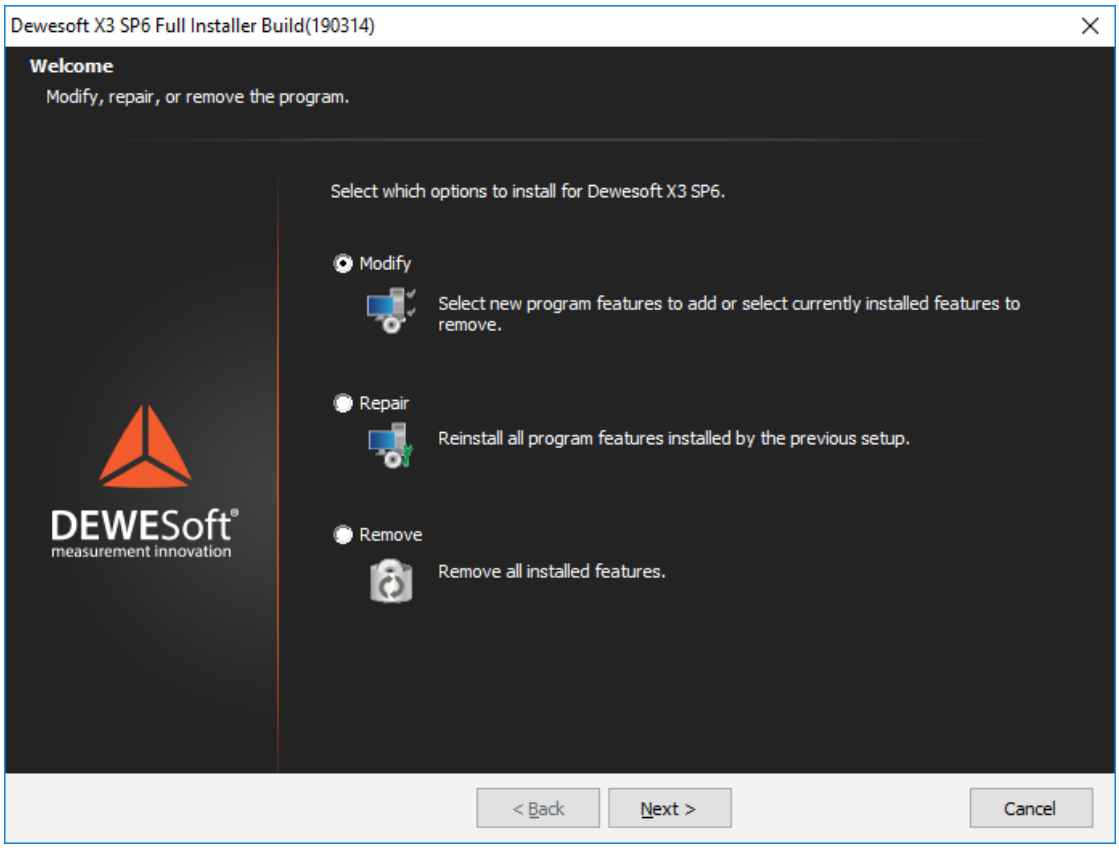


Image 4: Modify the Dewesoft installation offline

Alternatively you can also install DSMinGW manually by downloading the installer from the Dewesoft webpage under Support > Downloads > Developers > C++ Script.

Older versions of Dewesoft (before X3 SP6)

In order to write custom C++ Scripts in Dewesoft before X3 SP6, you need to manually install DSMinGW on your system. You can download the package on the Dewesoft webpage under Support > Downloads > Developers > C++ Script.

C++ Script can be found among experimental features in Dewesoft starting from X3 SP4, so you have to manually enable it. To do that, navigate to Dewesoft's *Options* and click on *Settings*, then select *Advanced* and finally *Experimental* subsections. There you will be able to find a checkbox C++ Script (Math) under *Features*; click it so that a checkmark appears next to it, click *Ok*, and restart Dewesoft.

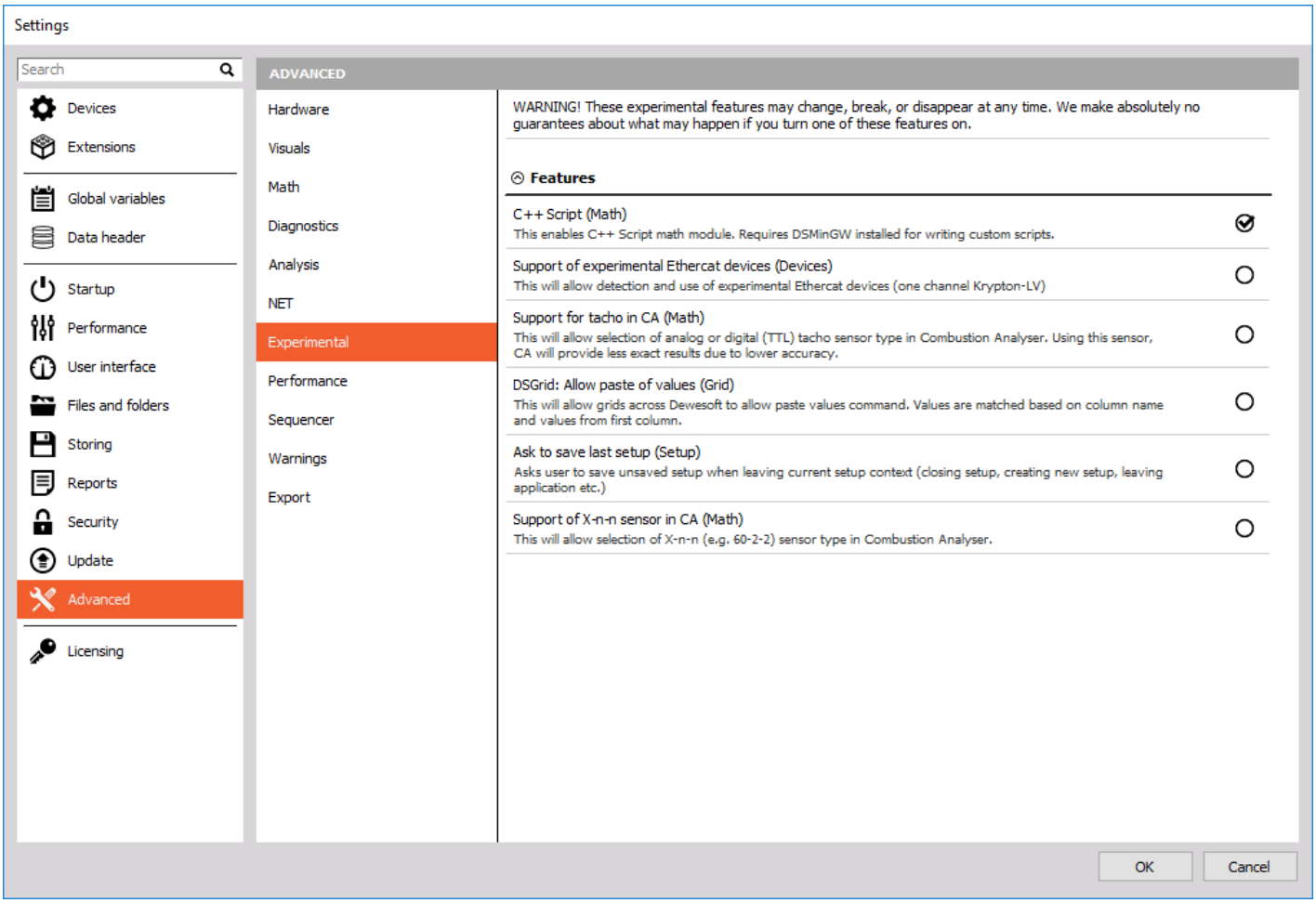


Image 5: Open settings -> Advanced -> Experimental and enable C++ Script (Math) option

C++ Example: Latch math

To get a better feel for both the use cases and the simplicity of C++ Script we will implement a latch math module. The example will serve to demonstrate a lot of different capabilities of C++ Script in a (hopefully) easily understandable way.

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++ Script, let's first create the two input signals we will use to test it.

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)".

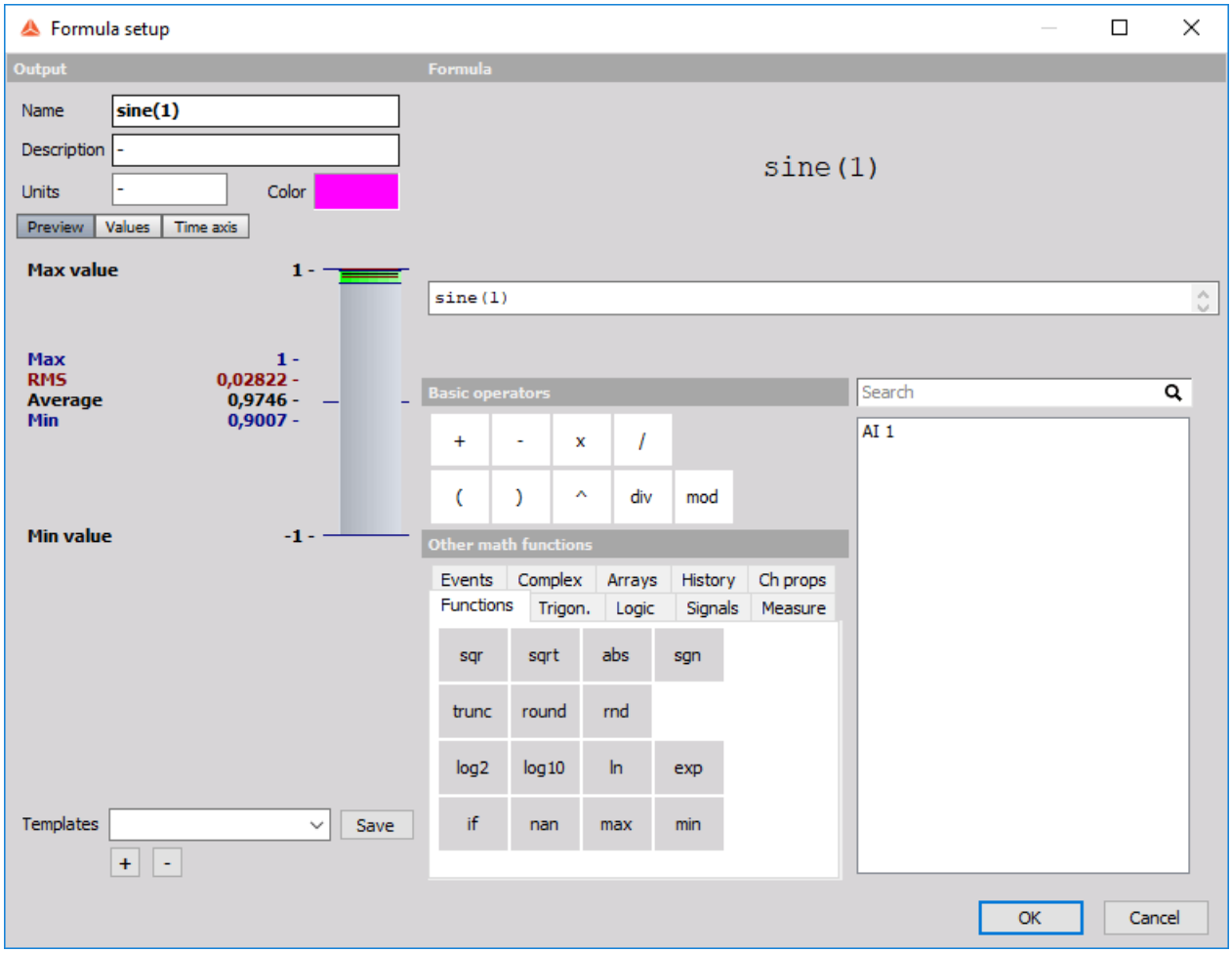


Image 6: Add formula sine(1)

The second signal will be the one from which we read the values to output at the 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".

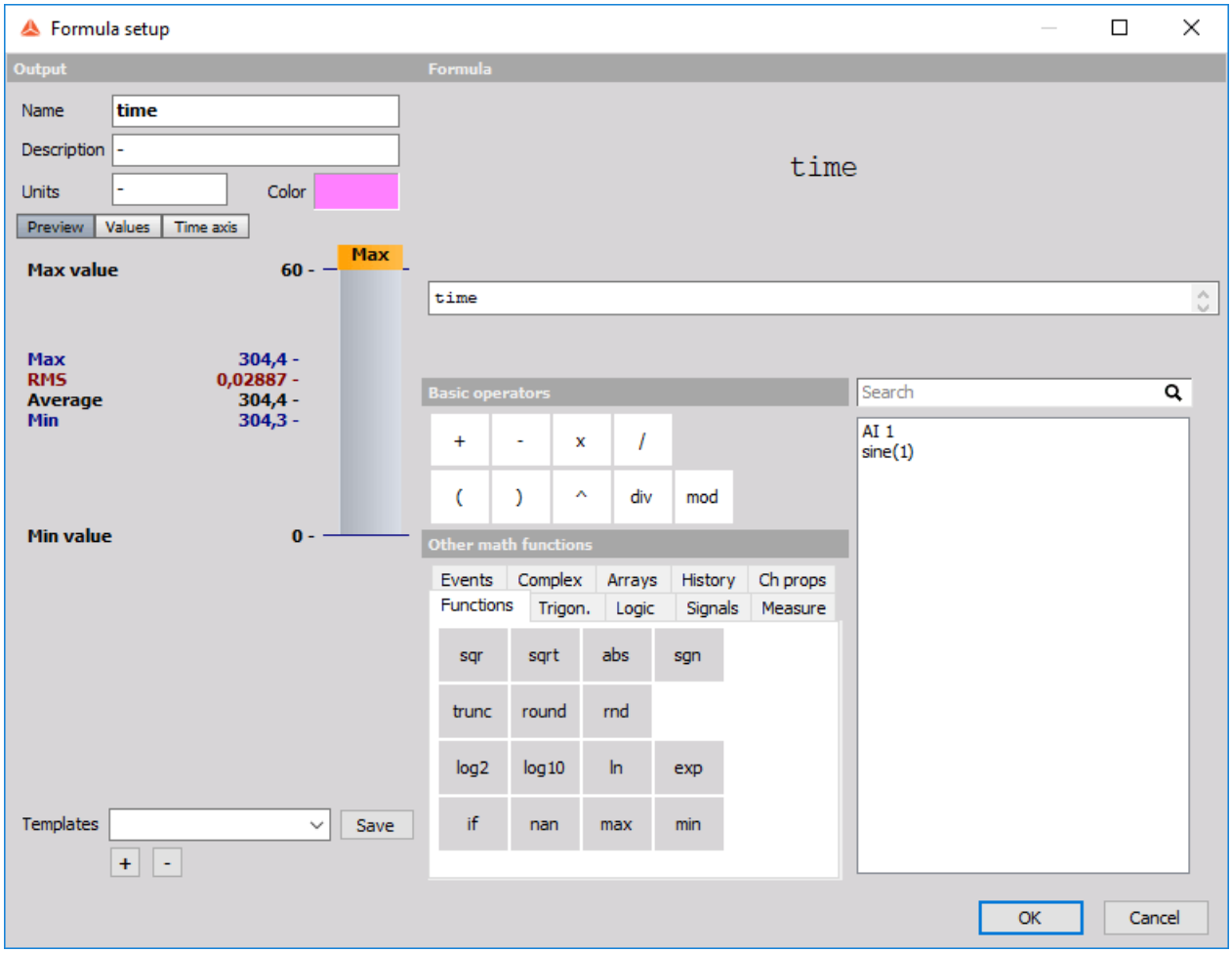


Image 7: Add 'time' formula

Sample rate

Another thing we should take care of before we begin development is to *lower* the **Dynamic acquisition rate.**

During the development of your C++ Script module **it is recommended to set the sample rate to some low value**, primarily for performance reasons. In case your script has any problems, it is much easier to recover and fix the error if it doesn't get triggered e.g. 100 000 times/second.

We do this by clicking the *Analog in* button and in the drop-down list set *Dynamic acquisition rate* to e.g. 500 Hz.

C++ Example: Latch math - Create C++ script

To create a new C++ Script we click the *Math* button and *Add Math* button which appears underneath it. In the menu that just opened up, we find the section called *Formula and scripting* and finally click on *C++ Script*.

A new C++ Script setup window will appear as it is shown on the image 8. The setup window has four tabs: Project tab, Configure tab, Code editor tab, and Published setup tab. Our example will be split into steps, corresponding to these tabs.

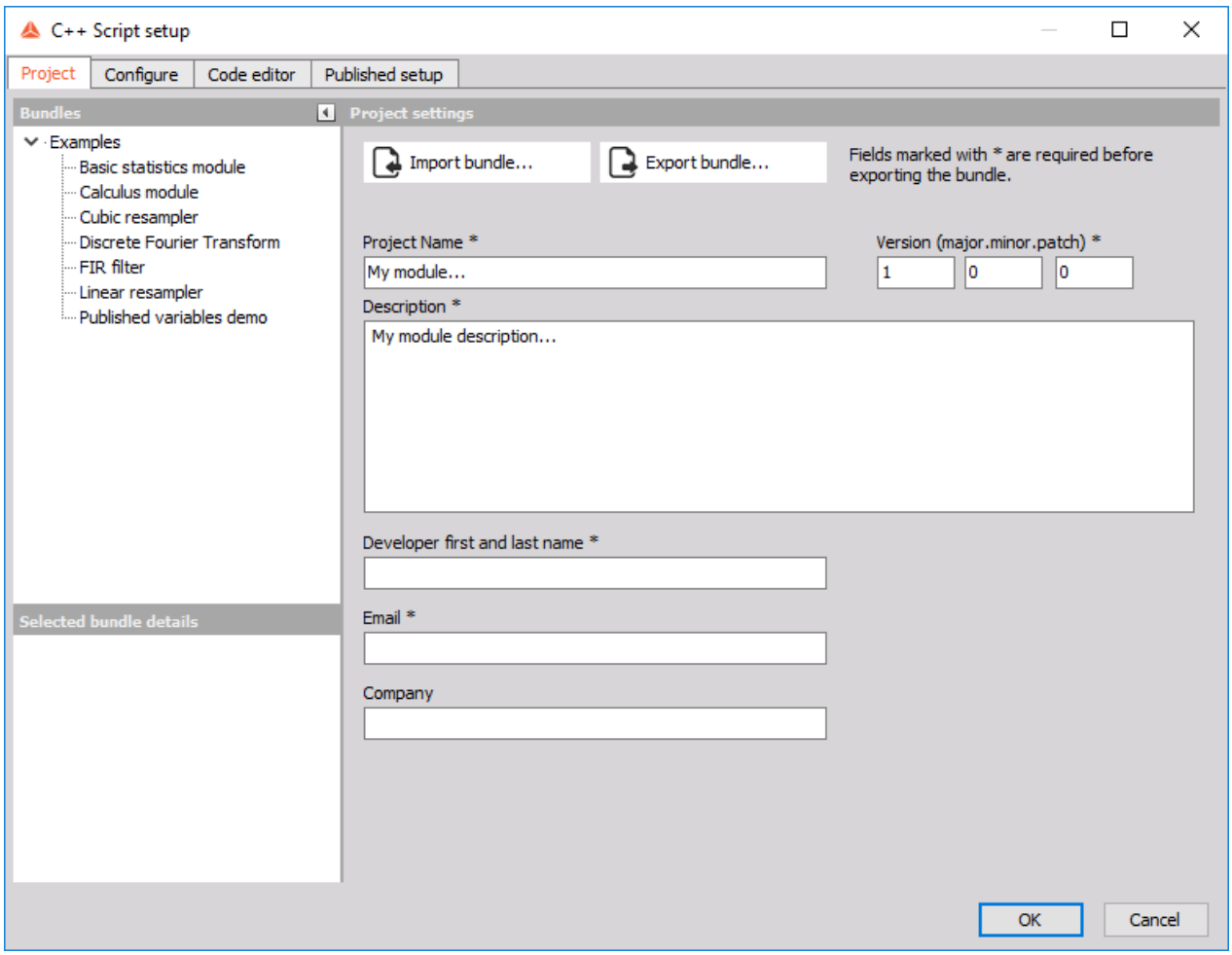


Image 8: C++ Script setup window

Published setup

But before we get started, let's click on the *Published setup* tab. We can see that a user interface not too dissimilar to other Math modules found in Dewesoft was automatically created; it has a section for input channels, empty space for output channels below that, as well as a section for defining and changing settings of our module. Since our module is currently empty, the form is also empty.

Project

In the *Project* tab, we are going to specify the module name, a brief description of what our module does, and the version of our module. We proceed by filling out the *Project name* field with **Latch math**,

Description with **Module performing latch math** and leave the *Version* fields as they are. Also fill out the *Developer first and last name*, *Email*, and *Company* text fields with your information.

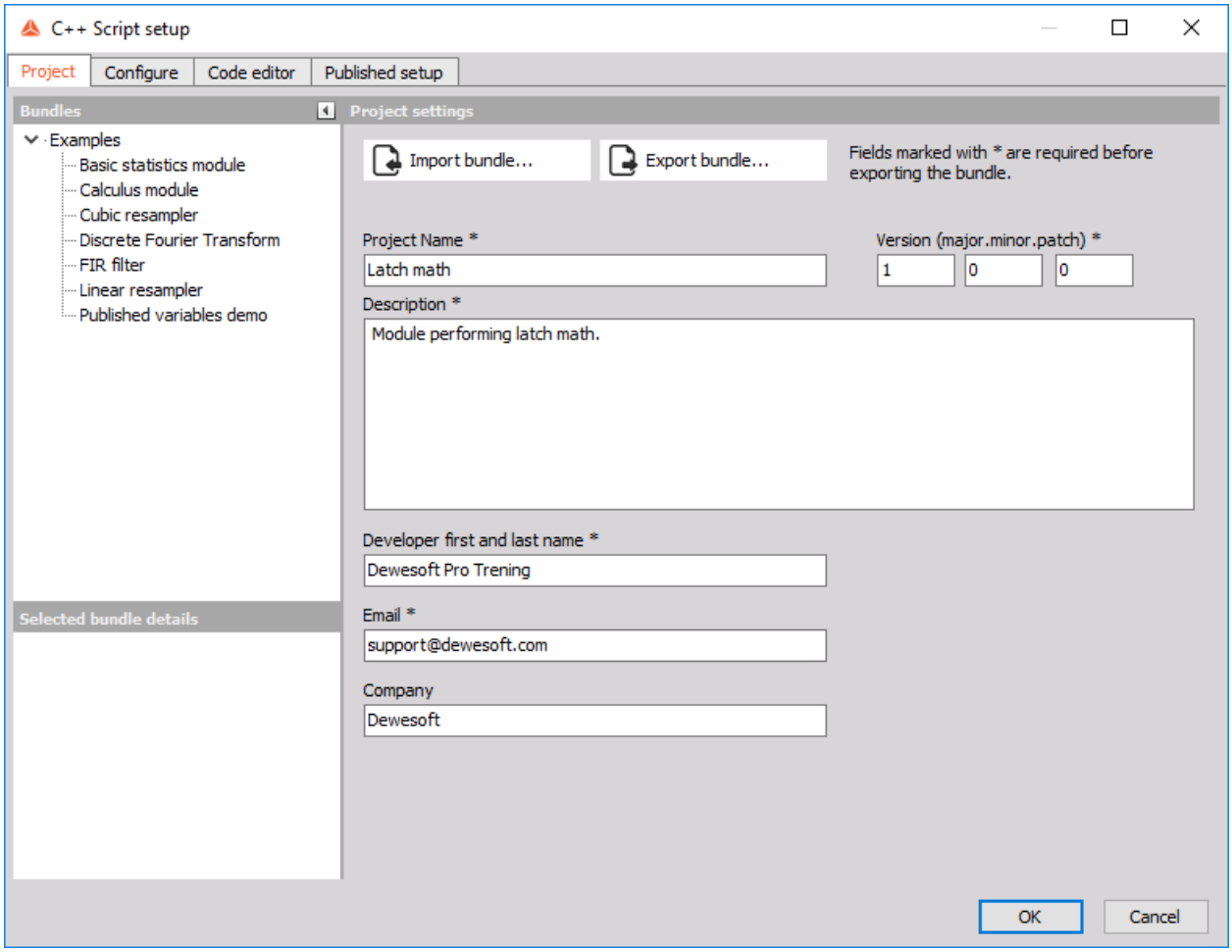


Image 9: Edit the Project tab of C++ Script

When we are done, click the *Configure* tab at the top of the setup window.

Configure

After clicking on the ***Configure*** tab the first thing we get to change is the general behavior of our module. We can choose the calculation call and define the block size for our calculations. C++ Script can work in two modes: Sample-based, where each new sample in the input channels triggers the calculation step, or Block based, where Dewesoft waits for Block size new samples from input channels before calling our module. For our latch module example, we do not change the setting, because we want our calculation to be performed for every new sample of the signal.

Let's say we want the user of our module to be able to specify 2 settings: whether he wants to latch the output on a rising or falling edge (i.e. when the car is accelerating or decelerating), and the Criteria limit to determine which value the criteria channel needs to pass before we output the value (i.e. the speed of the car we are interested in). ***Configure*** tab is exactly the place where we can define user settings like that.

Published variables

Inside the ***Configure*** tab under ***Published variables***, we can define various types of variables. The main idea behind defining the variables here instead of directly in our C++ code is that these variables are accessible from the ***Published setup*** tab and changing their values there does not require recompiling the code.

Under the ***Published variables*** tab click the button. This brings up a Variable setup form for defining our variables.

For start let's create a variable for deciding whether the latch condition is rising edge or falling edge on the signal. We do this by first filling out the Variable name section of the setup. We define the ***C++ variable name*** as "edgeType" -- this represents the name of the variable as it will later appear in the C++ code. We also define the ***Published name*** as "Type of edge", and this represents the name of the variable as it will appear in the ***Published setup*** tab.

Every time we fill in a field labeled "C++ X", we are setting the X inside the C++ code, and every time we fill in a "Published Y" field, we are changing Y in Published setup tab. Because "C++ X" fields need to be valid C++ variable names, we are not allowed to use spaces and special characters (other than "_") or start the name with a number.

Now we move on to the Variable type section of the setup window. Here we choose the ***Variable type*** as "Enumeration" from a drop-down list. After choosing "Enumeration" as our variable type a new section will appear where we input the values user can choose from; in our case we create two values "risingEdge" and "fallingEdge". To add a new value we simply click on the button. Now we can also define the ***Default value*** of our variable by choosing a value from the drop-down list. Lastly we can add a ***Published description*** for our variable. After filling out the Variable setup window click the ***Apply*** button.

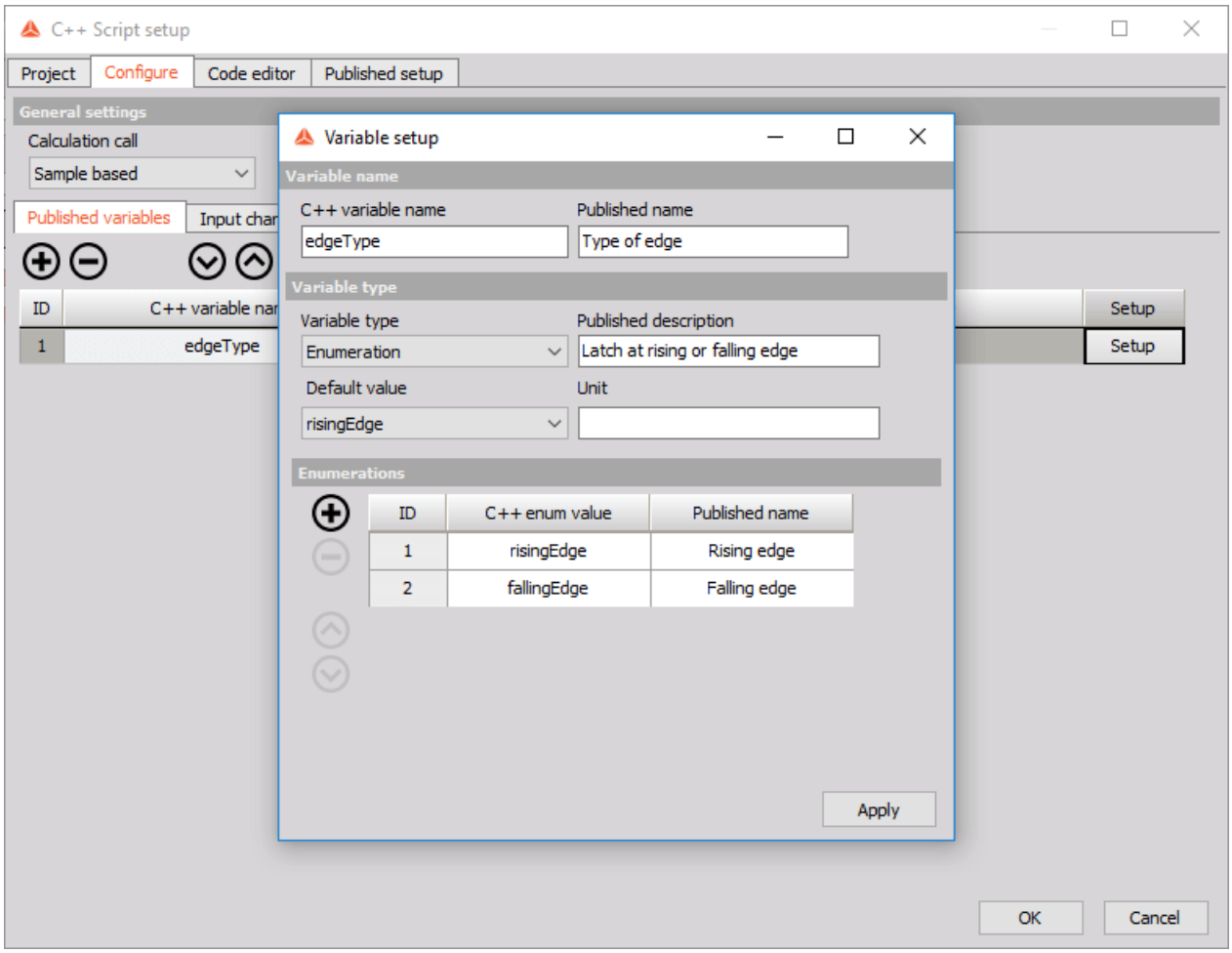


Image 10: Setup the variable with rising and falling edge

We also create another variable that our users will be able to change to control the value of the latch criteria. This variable will be of the type float and after choosing this type a Numerical Settings section appears, where we define the *minimum* and *maximum* numerical value user will be able to input for this variable in the *Published setup* tab.

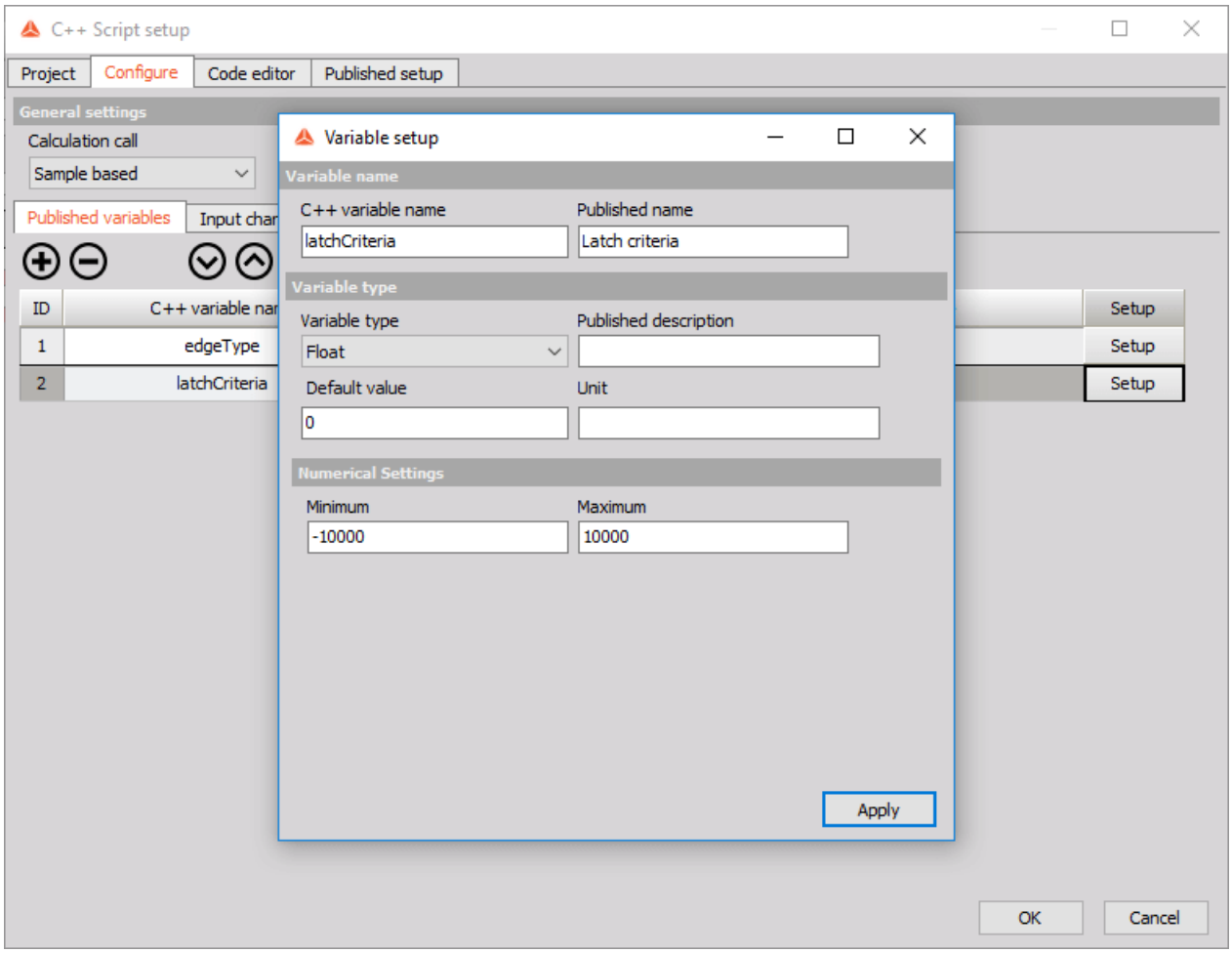


Image 11: Define minimum and maximum numerical value

The two published variables we created above can now be seen in the *Published setup* tab of the C++ Script setup window. As we can see from the picture below, C++ Script automatically created a drop-down combo box for letting the user choose the type of edge, as well as an edit field through which the user can set the value of the latch criteria.

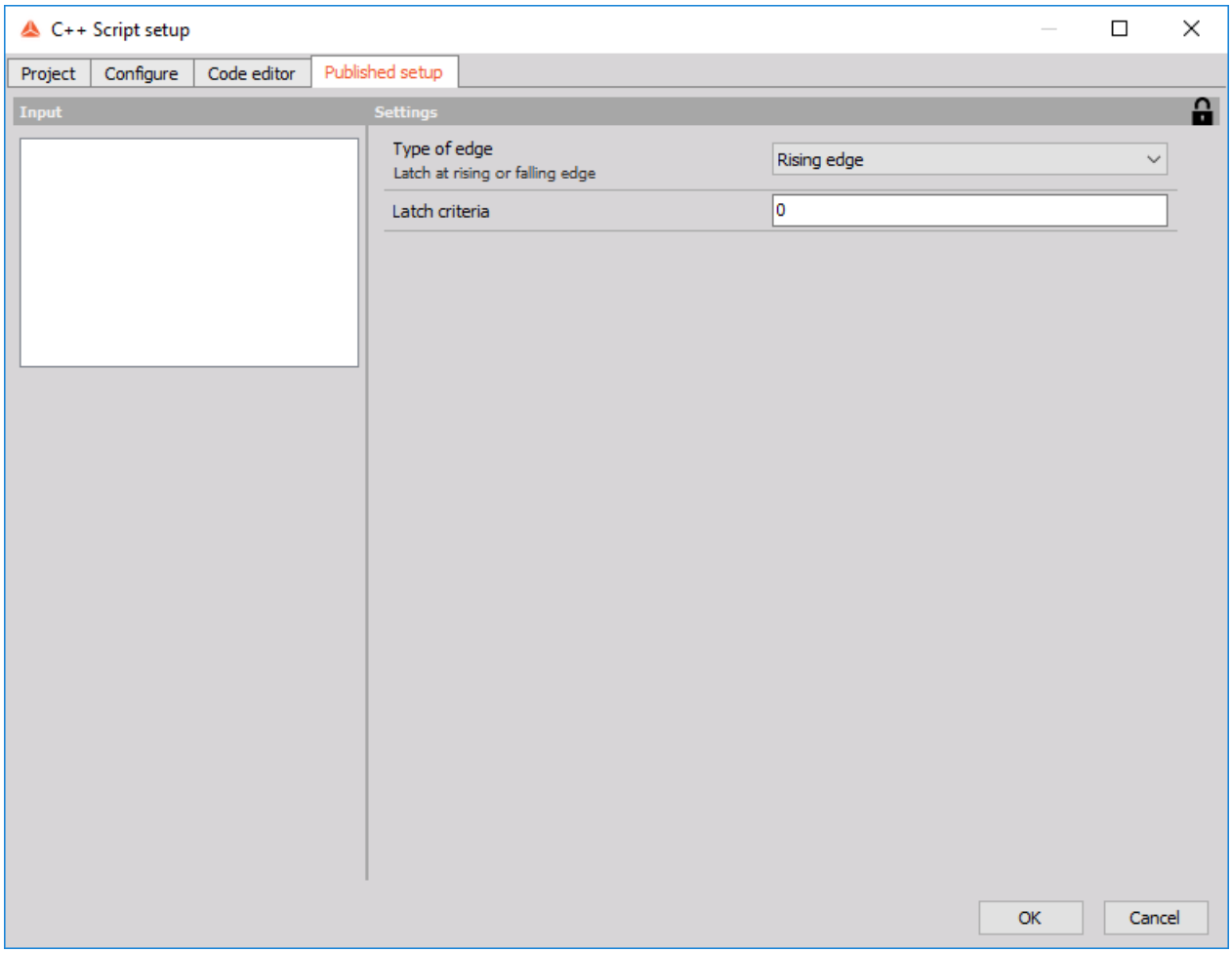


Image 12: Two published variables are now seen in the Published setup tab

Input channels

Our latch math will need two input channels, one for detecting the trigger points at latch condition and one from which we read the outputted values. So let's add them.

Under the ***Input channels*** tab click the ⊕ button. This brings up a ***Variable setup*** form for configuring your input channel. Change the ***C++ variable name*** to "criteriaChannelIn" and ***Published name*** to "Criteria channel". Leave ***Value type*** as Scalar and Real, and tick the Synchronous check-box under ***Time base***. Don't worry if you don't understand these settings yet as we will explain the different types of channels later on in this training. Clicking ***Apply*** you should find that the table now has a single row, containing "criteriaChannelIn" under the C++ variable name column and "(I) Sync Real Scalar" under Channel type.

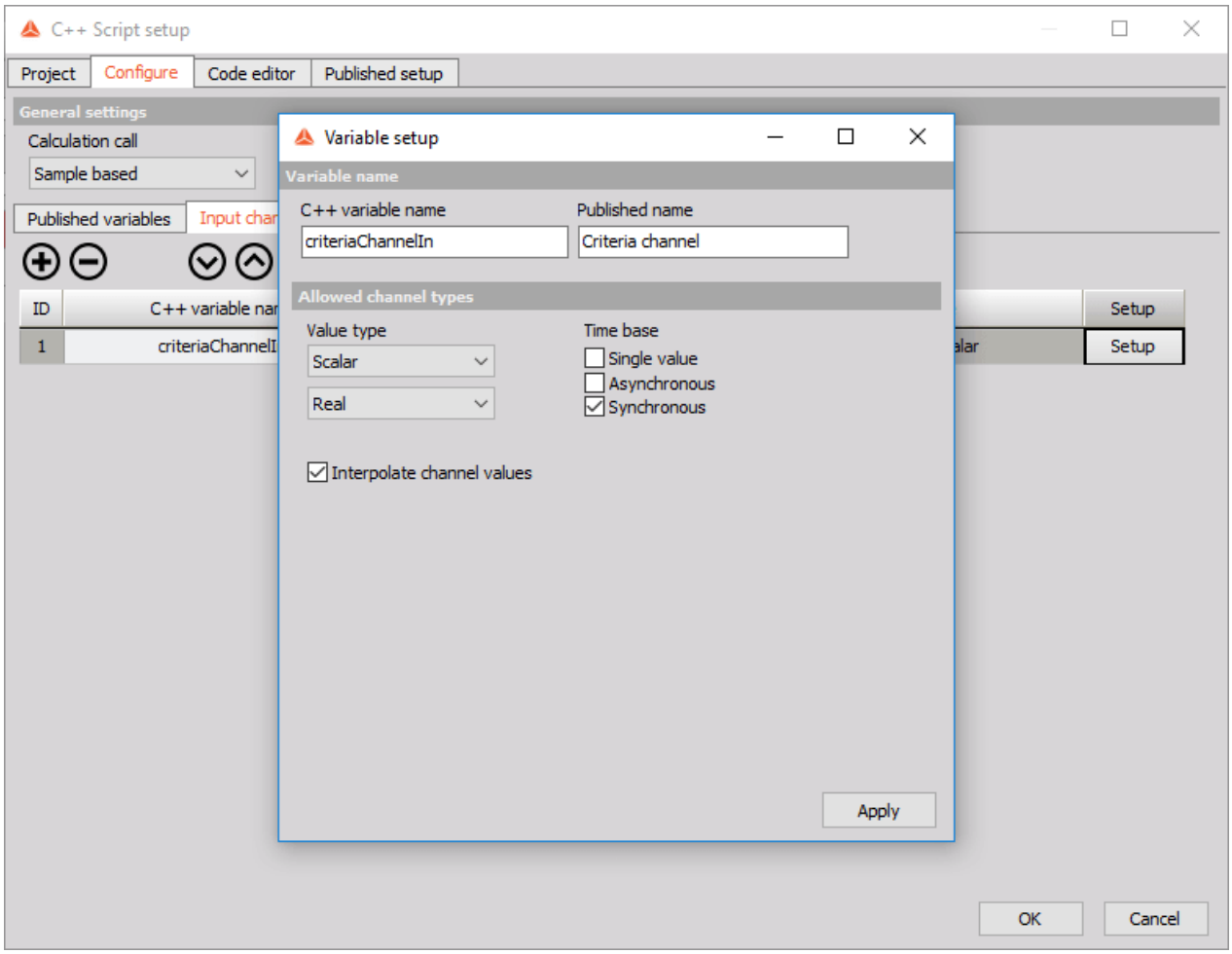


Image 13: Setup the variable as shown

Now we create another Input channel with the ***C++ variable name*** "inputChannelIn" and ***Published name*** "Input channel". Leave ***Value type*** as Scalar and Real and again tick the Synchronous check-box under ***Time base***.

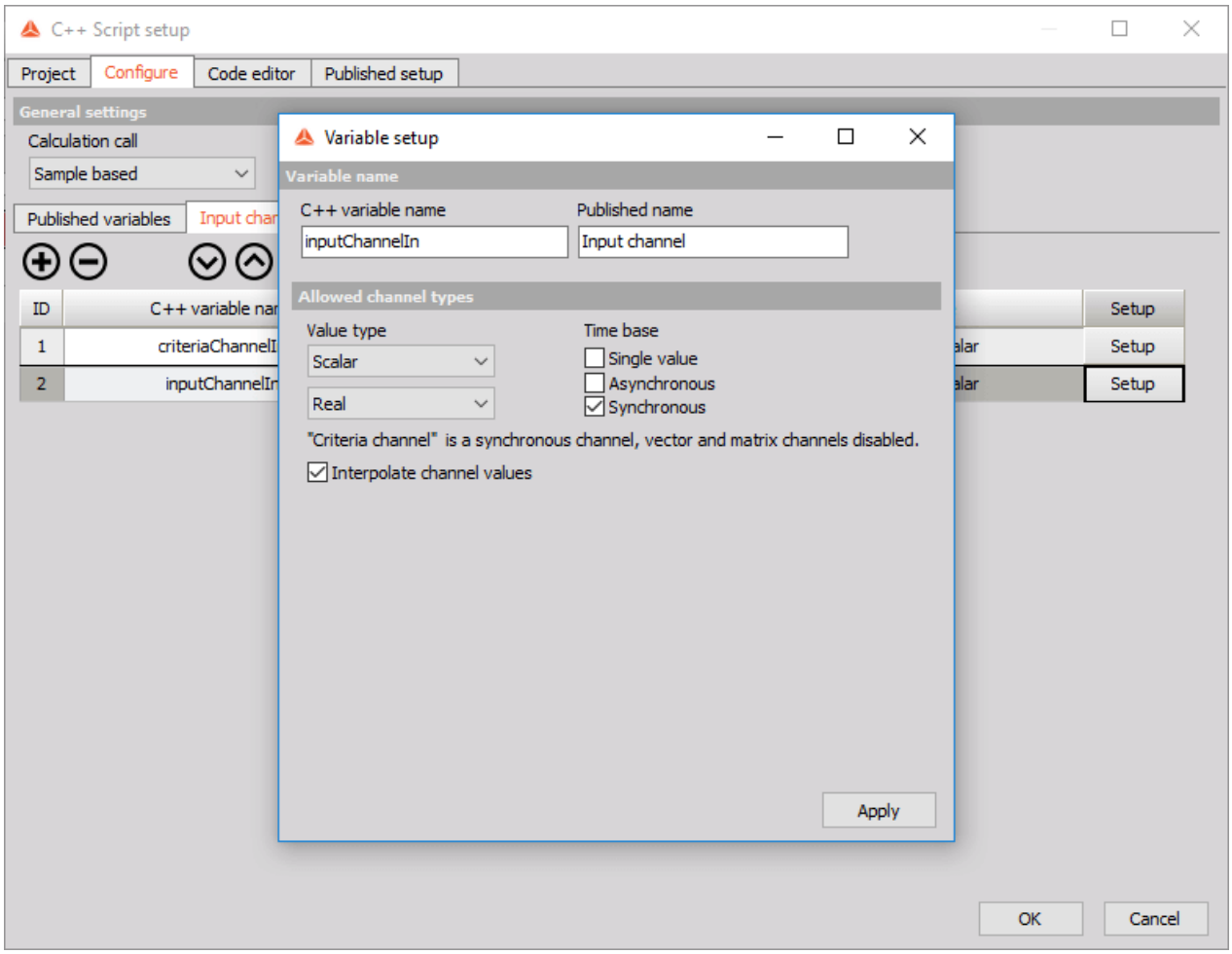


Image 14: Create another Input channel 'inputChannelIn' and set it up as shown

Clicking *Apply* you should find that the table now has two rows, containing "criteriaChannelIn" and "inputChannelIn" under the *C++ variable name* column and "(I) Sync Real Scalar" under *Channel type*. This means your module is going to have two channels of the synchronous type as input. Again, we will explain what exactly this means in one of the following sections.

Output channels

The output of our latch math module will be a channel holding the correct value of the second input signal whenever the first input signal crosses the latch criteria. So let's create it.

Under *Output channels* tab you will be able to find a **debug** channel. Ignore it for now and click the ⊕ button again. This brings up a form with settings slightly resembling the settings for input channels. Change the *C++ variable name* to "latchedChannelOut" and *Published name* to "Latch". Leave the *Value type* as Scalar and Real, change *Time base* to Asynchronous and leave the *Expected async rate per second* as 10. Don't worry too much about the meaning of Expected async rate per second setting either as it too will be explained in detail later on in the training. Clicking *Apply* creates a new row in the table with the channel we just made.

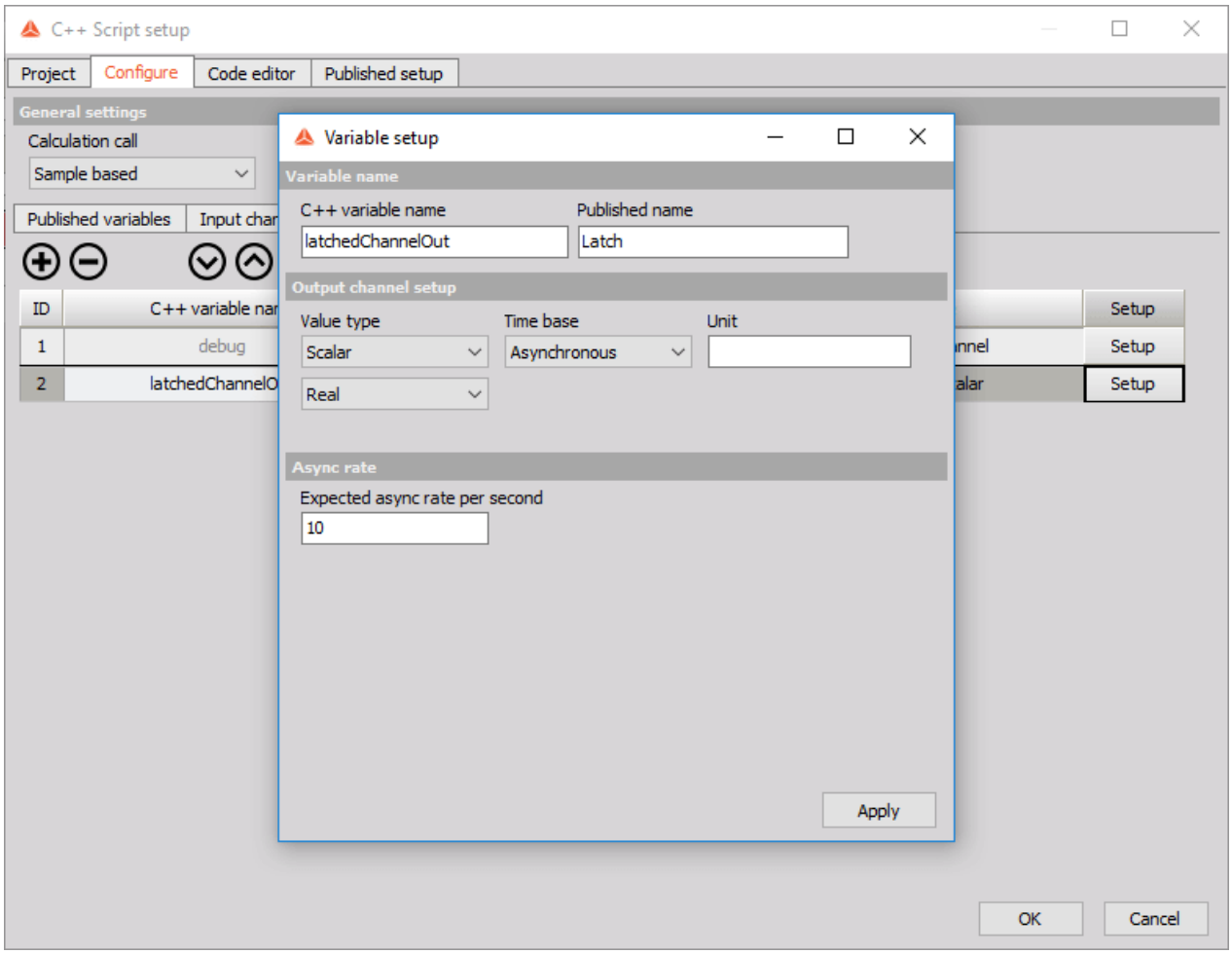


Image 15: Setup the 'latchedChannelOut' variable

In the *Published setup* tab, we can now see all the variables we created in the *Configure* tab, including the two currently unassigned input channels, and the Published variables with their default values.

C++ Example: Latch math - Code editor tab

Now comes the time to actually implement the logic of our latch module. We do this in the C*ode editor* tab.

One way our module could operate is by looking at every two consecutive samples in the criteria channel. As long as both samples are below or above the criteria limit we will do nothing. But when the first sample is below the criteria limit and the second sample is above it (in case of rising edge, swap the two for falling edge), this must mean we passed the criteria limit and we will output the value from the input channel to the output channel.

Let's try implementing this logic.

Code

The very top of the template in a fresh C++ Script module contains the following line:

```
namespace bsc = Dewesoft::Math::Api::Basic;
```

All of C++ Scripts types and some of its structures reside in the `Dewesoft::Math::Api::Basic` namespace. The line above allows us to access them using the abbreviation `bsc`, making the code more readable.

The very next thing in the code template is a class called `Module`; this is the class we will fill out with our module's logic.

C++ Script interacts with Dewesoft by letting it invoke certain methods of our implemented `Module` code (all of which come pre-populated in the C++ Scripts code editor by default) at specific predefined events. Hints about when exactly Dewesoft calls each method are provided in the comments inside the Code editor.

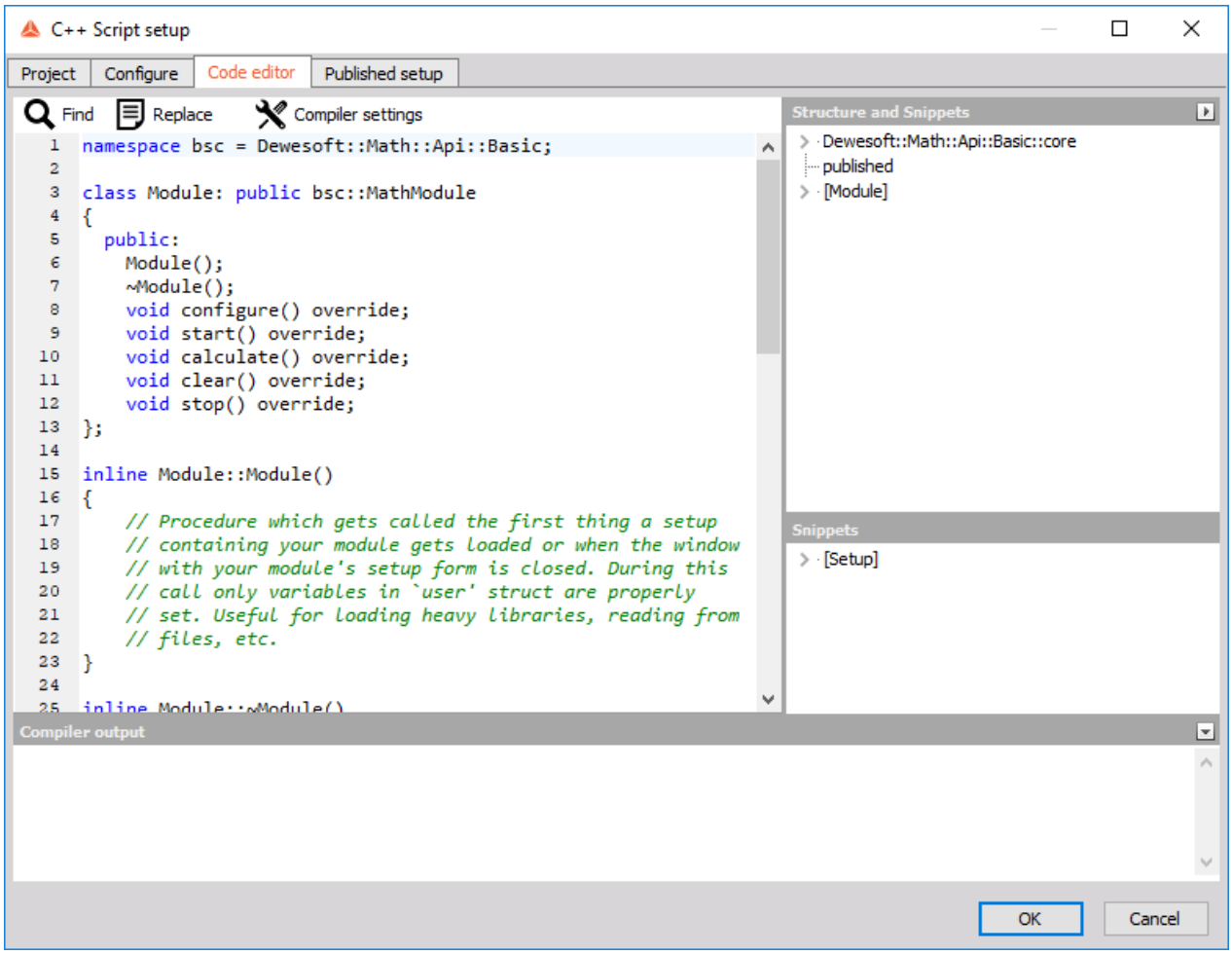


Image 16: Hints about when exactly Dewesoft calls each method are provided in the comments inside the Code editor

In our example, we will only need to fill in the `Module::start()` and `Module::calculate()` methods. Since our `Module::calculate()` method is going to be called with exactly one new sample in the input channels, we will also need a global variable for remembering the previous sample of criteria channel which we will use to check whether we crossed the latch criteria of the rising or falling edge. We define this variable in class Module so we can store it between the calls to `Module::calculate()` :

```
class Module: public bsc::MathModule
{
 public:
  Module();
  ~Module();
  void configure() override;
  void start() override;
  void calculate() override;
  void clear() override;
  void stop() override;

  bsc::Scalar prevCriteriaChannelValue = 0;
};
```

void Module::Start()

`Module::start()` function gets called at the very start of the measurement. It should be used for lightweight tasks such as initializing your variables. Here we initialize the global variable we created above.

```
inline void Module::start()
{
   prevCriteriaChannelValue = 0;
}
```

void Module::calculate()

`Module::calculate()` function gets called repeatedly during the measurement mode. Remember when we set the calculation call to "Sample based" in ***Configure*** tab? That told Dewesoft to call our `Module::calculate()` method whenever there is a new sample in input channels.

The code in the `Module::calculate()` function looks like this:

```
inline void Module::calculate()
{
   // read new sample from Criteria channel
   bsc::Scalar currCriteriaChannelValue = criteriaChannelIn.getScalar(0);

   // check if the two samples from Criteria channel are on different sides of Latch criteria
   bool crossedRisingEdgeCriteria = prevCriteriaChannelValue <= published.latchCriteria
                                    && currCriteriaChannelValue >= published.latchCriteria;

   bool crossedFallingEdgeCriteria = prevCriteriaChannelValue >= published.latchCriteria
                                    && currCriteriaChannelValue <= published.latchCriteria;

   // 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 ((published.edgeType == risingEdge && crossedRisingEdgeCriteria)
    || (published.edgeType == fallingEdge && crossedFallingEdgeCriteria))
   {
      // add the value of the current sample from Input channel to the Latched channel
      bsc::Scalar currInputChannelValue = inputChannelIn.getScalar(0);
      latchedChannelOut.addScalar(currInputChannelValue, callInfo.endBlockTime);
   }

   // in any case remember the value of current sample from the Input channel for the next calculation
call
   prevCriteriaChannelValue = currCriteriaChannelValue;
}
```

From the code above notice a couple of things:

- to access the channels from our code we use the same names as we defined them under *C++ variable name* in *Configure* tab;
- we can access and add the samples in the channels by invoking `.getScalar()` and `.addScalar()` methods;
- we can access the values of published variables from *Published setup* tab through a special `published` struct by using `published.X`; where X is again the name as defined under *C++ variable name* in *Configure* tab; and
- we used a `callInfo` structure which contains four read-only variables that get updated before each call to `Module::calculate()` with the latest values. In our case, we use the `callInfo.endBlockTime` which tells us the time in seconds of the last new sample in input channels -- which in our case, since we only receive one new sample per calculate call, is always the time that this sample arrived at.

You can click on the Structure and snippets bar on the right-hand side of the code editor to bring up the tree of all structures Dewesoft has made available for you. By expanding nodes you will be able to find `callInfo.endBlockTime`, `criteriaChannelIn`, `inputChannelIn` and `latchedChannelOut` with their respective methods for reading and writing from the channels, as well as some other variables and methods we didnÃ¢â‚¬â„¢t mention.

Elements in the tree view can also be double-clicked to populate them in the code editor.

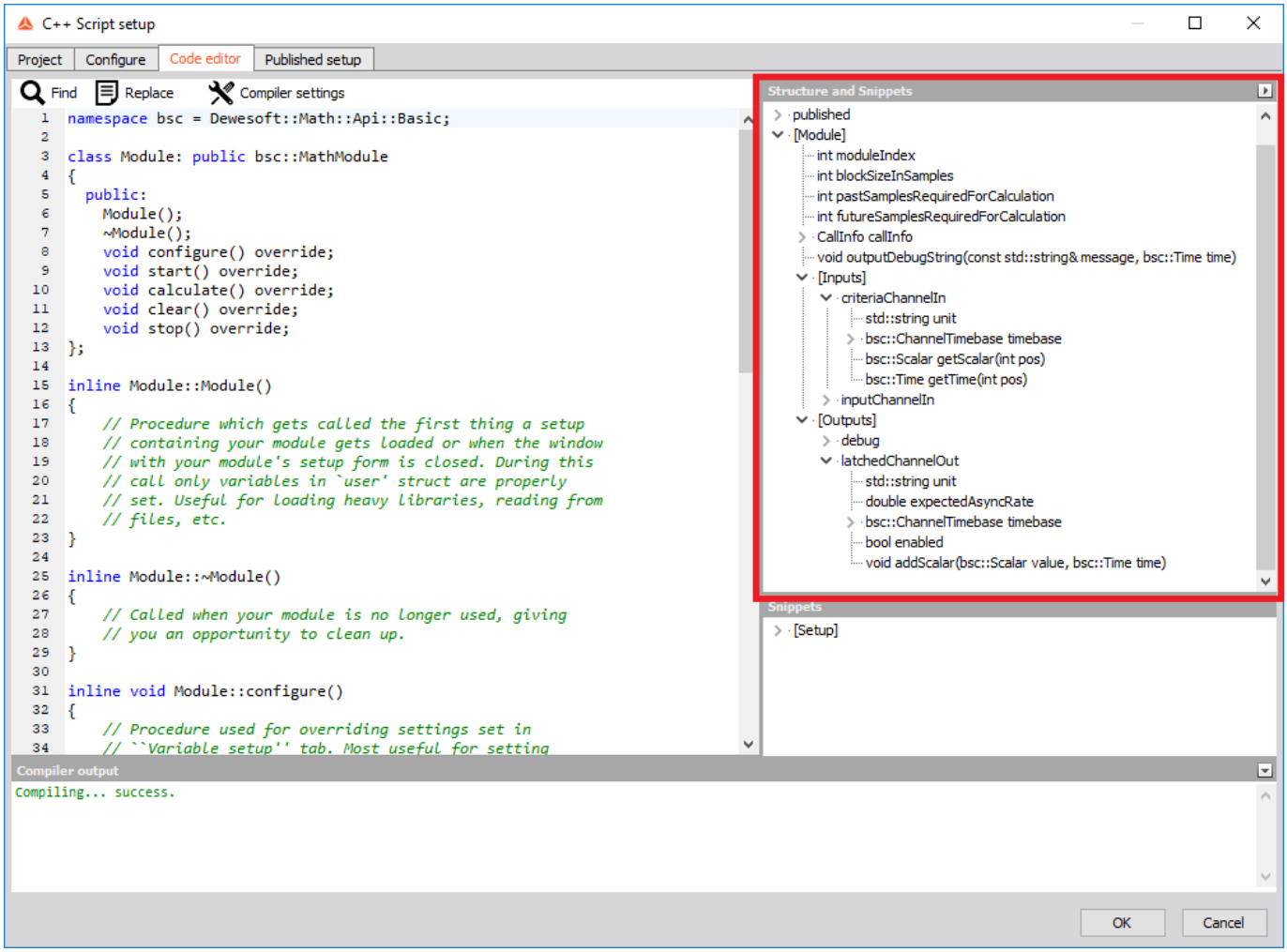


Image 17: Structure and Snippets - Elements in the tree view can also be double-clicked to populate them in the code editor

At the bottom of the *Code editor* tab we can find a panel called **Compiler output** where we see if our code compiled successfully or not. In the latter case this panel would contain the compiler errors and warnings that occurred during the compilation of our code.

Image 18: Compiler output outputs the status of the code

C++ Example: Latch math - Published setup tab

We can now assign the signals we created before to the correct input channels. In the *Input* section of the window we can see a table with our two input channels. To assign a signal to Criteria channel, click on the white cell under *1* and click on the 🔍 button. From the drop-down list choose the "sine(1)" channel, and repeat the same process for Input channel only this time choose the "time" channel from the drop-down list.

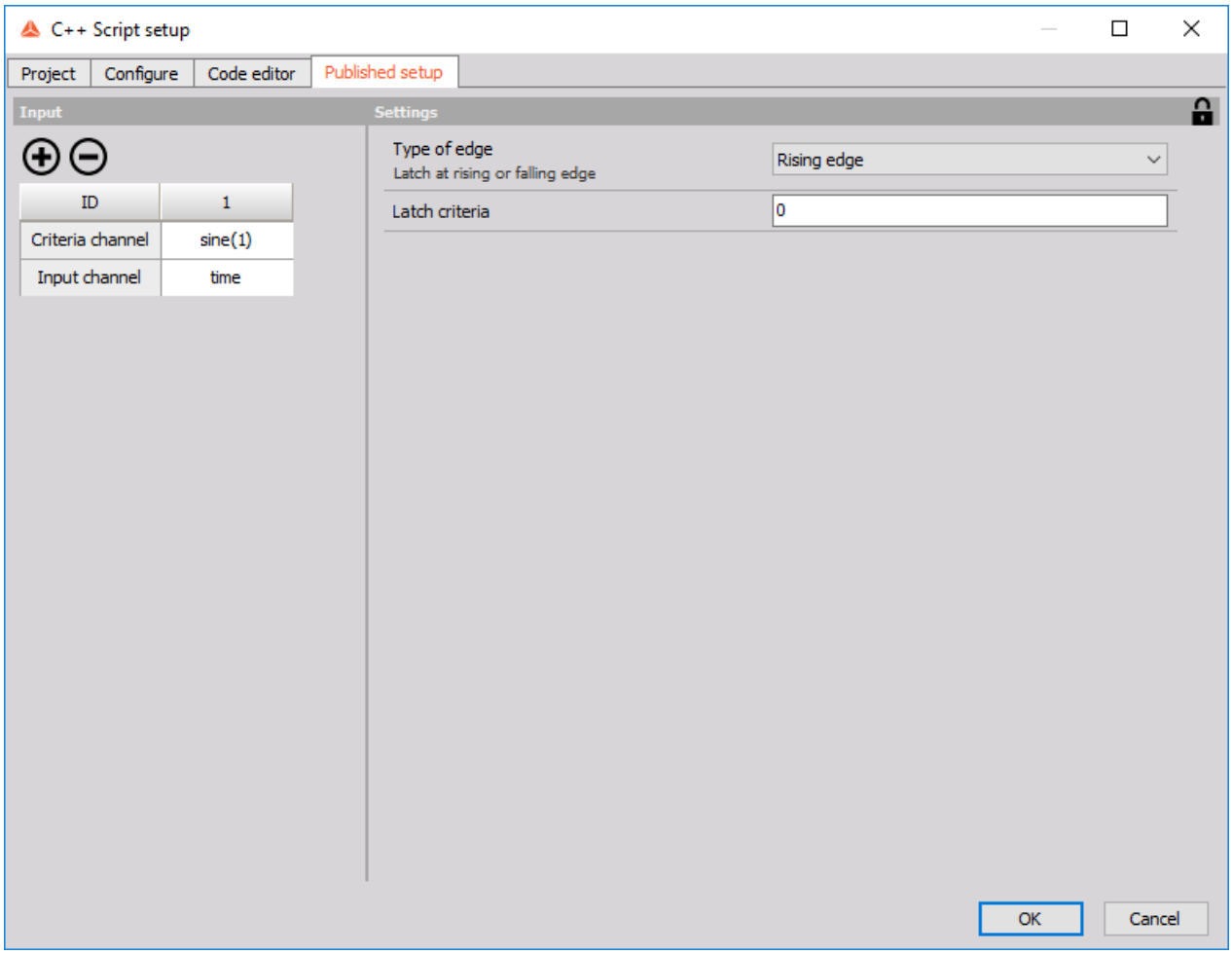
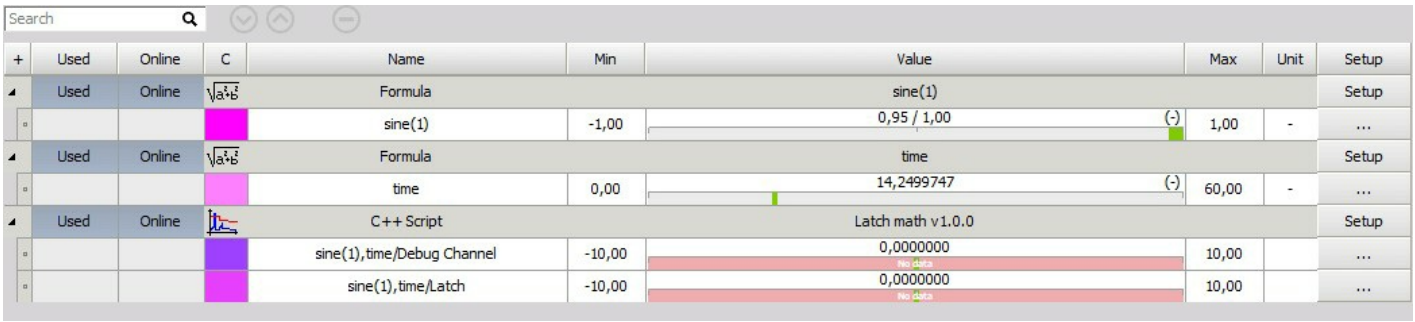


Image 19: Assign the signals to the correct input channels

Click the *OK* button at the very bottom of the setup form and do not forget to save your setup in Dewesoft.

Save your setup often during development! In case your module contains a serious bug that crashes Dewesoft, loading the old setup is the only way to restore your progress.

On the image 20 it is seen how the setup should look inside the *Math section*.



+	Used	Online	C	Name	Min	Value	Max	Unit	Setup
	Used	Online		Formula		sine(1)			Setup
				sine(1)	-1,00	0,95 / 1,00 (-)	1,00	-	...
	Used	Online		Formula		time			Setup
				time	0,00	14,2499747 (-)	60,00	-	...
	Used	Online		C++ Script		Latch math v1.0.0			Setup
				sine(1),time/Debug Channel	-10,00	0,0000000 No data	10,00		...
				sine(1),time/Latch	-10,00	0,0000000 No data	10,00		...

Image 20: Final setup in Math section

It is perfectly normal for C++ Script's output channels in setup mode to have "No data", as Dewesoft doesn't call C++ Script's `Module::calculate()` method during setup.

Initiate Dewesoft's measurement mode by clicking on the ***Measure*** tab and observe your channel with outputted values at latch condition.

Example: Latch math - Output

From the picture below 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 passes 0.

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

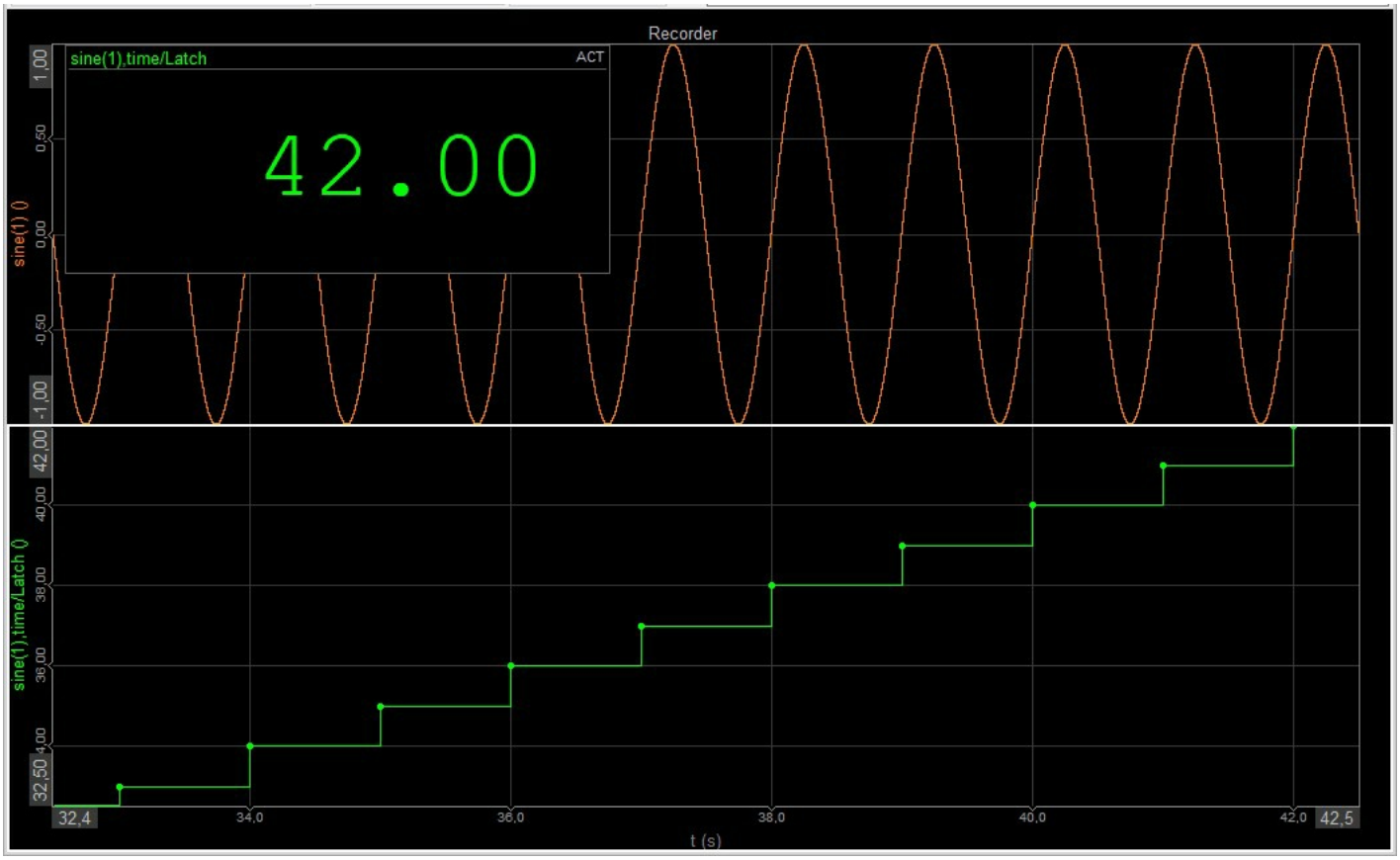


Image 21: Demonstrate 'latched' values

We can now change the variables in the ***Published setup*** tab of the C++ Script setup to look for the latch criteria at the falling edge at the value 0,5.

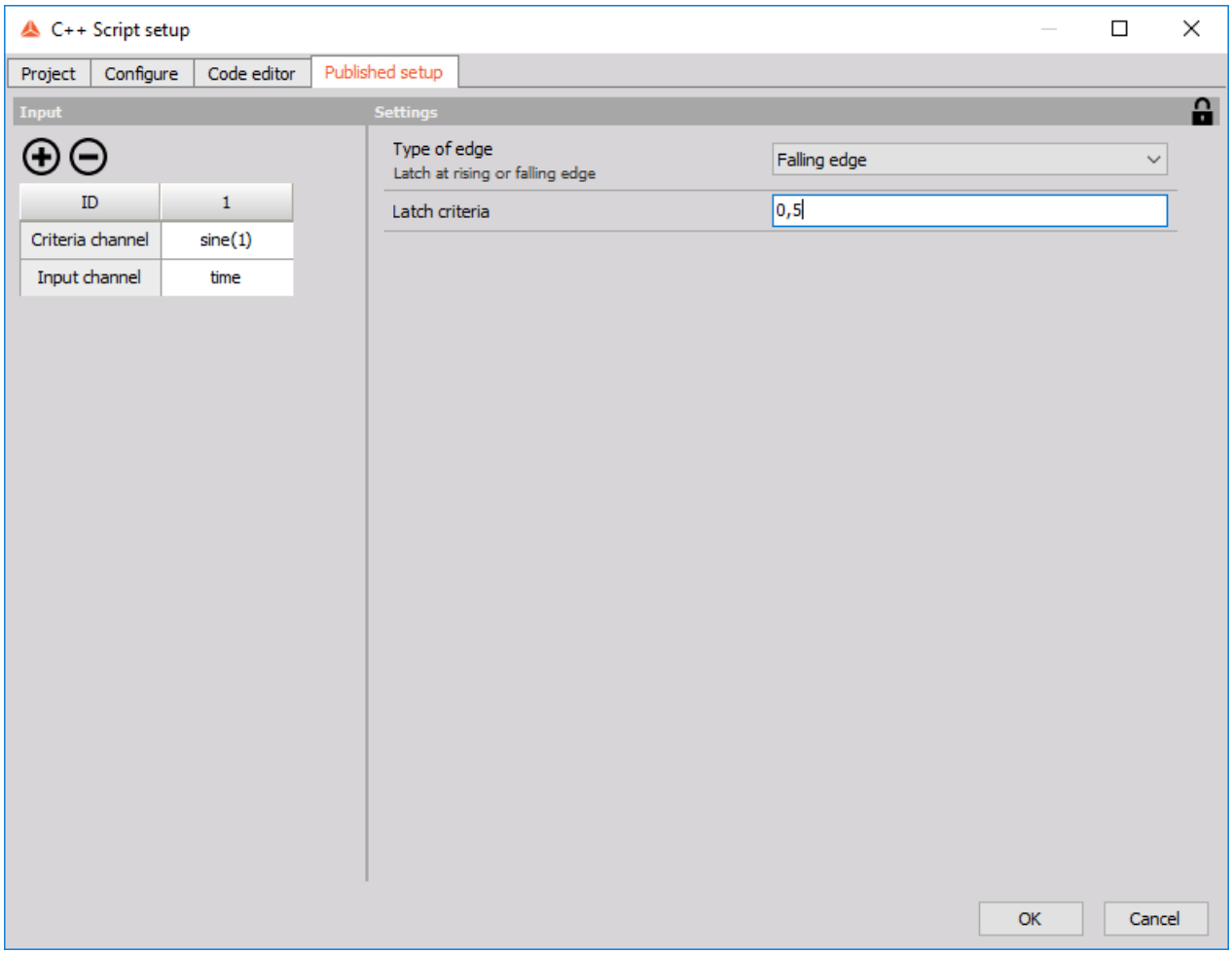


Image 22: Change the variables in the Published setup and set the latch criteria at the falling edge to the value 0,5

The closing of the module is instantaneous! The module doesn't get recompiled when we just change values in the *Published setup* tab.

The results will now look like this:

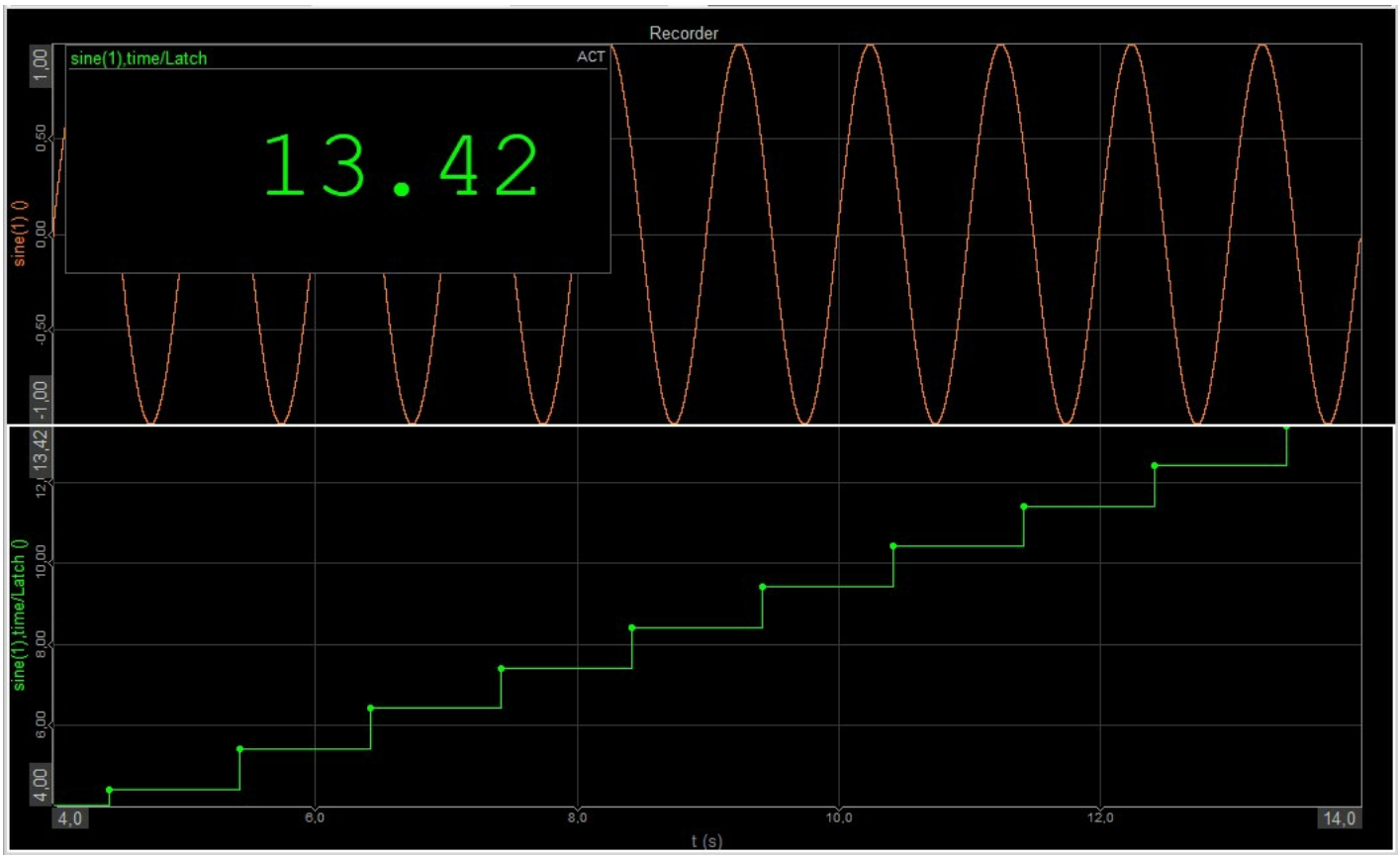


Image 23: New latch result

C++ Example: Latch math - Code simplification

The code as presented in the section before will work, but it has one part which stands out: the way we access the "previous" sample from the Criteria channel to see if our signal passed the Latch criteria. Remember that we do it by saving the current sample to a temporary variable `prevCriteriaChannelValue` we made just for this purpose at the very end of `Module::calculate()` method. But since accessing more than one sample from input channels is a pretty useful feature in general, C++ Script comes with a simpler way to do it. Instead of a special variable in our code, we can just set the predefined module variable `pastSamplesRequiredForCalculation` in `Module::configure()` method to a value greater than 0 (in our case 1, since we need just one additional sample from the past).

`Module::configure()` is a special method in which you get a chance to override any settings from *Configure* tab, as well as define a few additional ones. This can be useful if you want to change the properties depending on values set by user in *Published setup* tab. If you choose to not modify any of the values inside `Module::configure()` form, they take on the default values as set in *Configure* tab.

With setting the `pastSamplesRequiredForCalculation` to some value, our input channels contain this many additional samples per call to `Module::calculate()` from the past, and we can access them by simply calling `.getScalar(-i)` where `i` is the index of the sample we want.

This means that the code in our `Module::configure()` method should look like this:

```
inline void Module::configure()
{
   pastSamplesRequiredForCalculation = 1;
}
```

And our code in `Module::calculate()` will now look like this:

```
inline void Module::calculate()
{
   bsc::Scalar prevCriteriaChannelValue = criteriaChannelIn.getScalar(-1);
   bsc::Scalar currCriteriaChannelValue = criteriaChannelIn.getScalar(0);

   bool crossedRisingEdgeCriteria = prevCriteriaChannelValue <= published.latchCriteria
                                    && currCriteriaChannelValue >= published.latchCriteria;

   bool crossedFallingEdgeCriteria = prevCriteriaChannelValue >= published.latchCriteria
                                     && currCriteriaChannelValue <= published.latchCriteria;

   if ((published.edgeType == risingEdge  && crossedRisingEdgeCriteria)
    || (published.edgeType == fallingEdge && crossedFallingEdgeCriteria))
   {
      bsc::Scalar currInputChannelValue = inputChannelIn.getScalar(0);
      latchedChannelOut.addScalar(currInputChannelValue, callInfo.endBlockTime);
   }
}
```

We can now remove the global variable `prevCriteriaChannelValue` from the `Module::start()` and its definition from the

Module class.

If you ever need to access samples "from the future", you could similarly use `futureSamplesRequiredForCalculation`. and access the samples in channels via `.getScalar(i)` with `i` greater or equal to `callInfo.newSamplesCount`.

Another way we could approach this

After mentioning the Block based calculation call in the previous sections, you might have thought about using that functionality for our module. We could, for example, change the calculation call in *Configure* tab to Block based, set the Block size to 2, and then check each pair of new samples inside `Module::calculate()`.

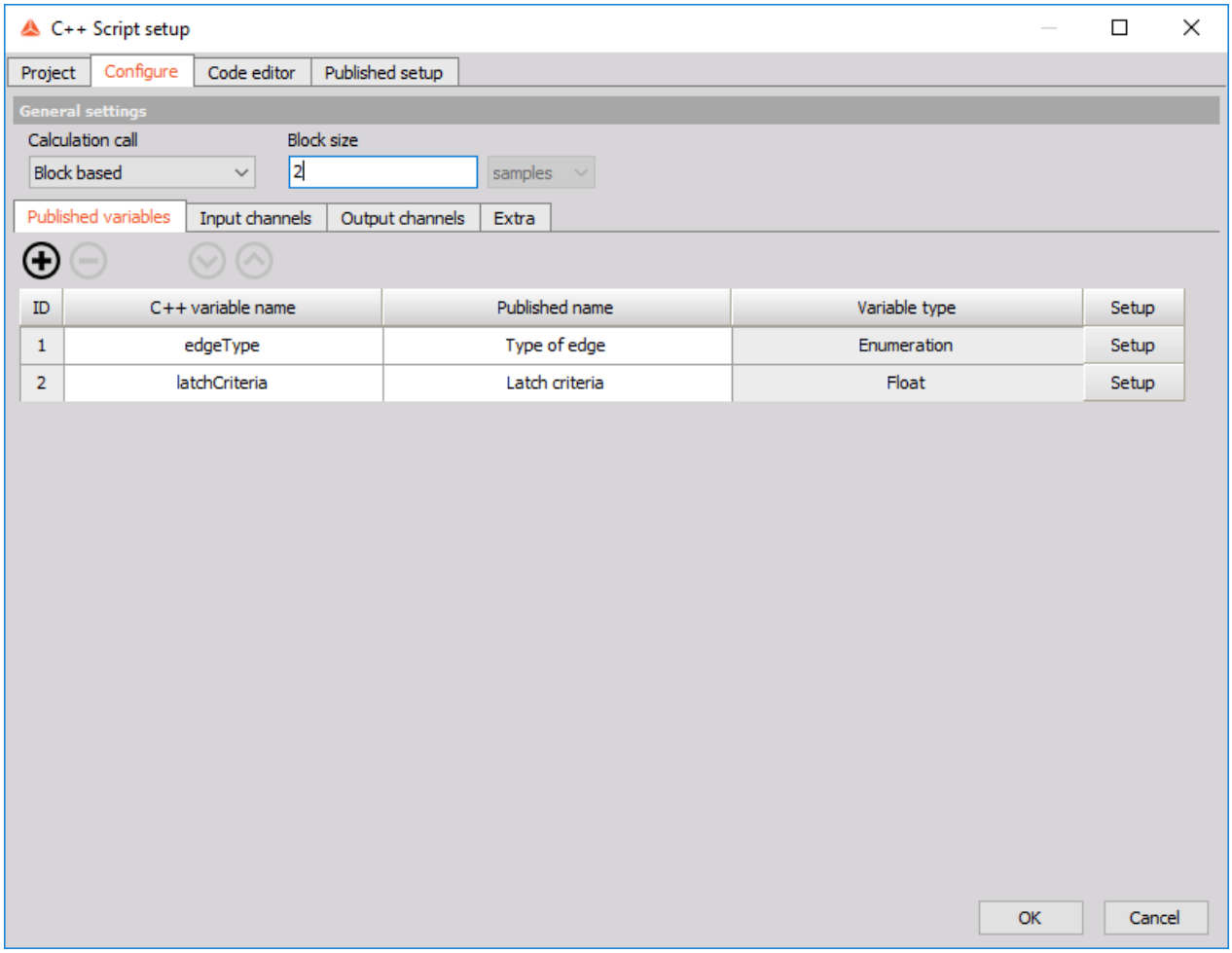


Image 24: Change the calculation call in Configure tab to Block based

We could also change the Block size from within `Module::configure()` by setting Module's `blockSizeInSamples` variable to 2. Setting it to 1 is equivalent to setting Calculation call to Sample based.

Now we change the code in the *Code editor* tab to:

```
inline void Module::calculate()
{
   bsc::Scalar prevCriteriaChannelValue = criteriaChannelIn.getScalar(0);
   bsc::Scalar currCriteriaChannelValue = criteriaChannelIn.getScalar(1);

   bool crossedRisingEdgeCriteria = prevCriteriaChannelValue <= published.latchCriteria
                                    && currCriteriaChannelValue >= published.latchCriteria;

   bool crossedFallingEdgeCriteria = prevCriteriaChannelValue >= published.latchCriteria
                                    && currCriteriaChannelValue <= published.latchCriteria;

   if ((published.edgeType == risingEdge  && crossedRisingEdgeCriteria)
    || (published.edgeType == fallingEdge && crossedFallingEdgeCriteria))
   {
      bsc::Scalar currInputChannelValue = inputChannelIn.getScalar(1);
      latchedChannelOut.addScalar(currInputChannelValue, callInfo.endBlockTime);
   }
}
```

This might seem like a good solution to achieve the same result, but if we think about it, it is not going to work correctly. We said that Dewesoft will call `Module::calculate()` method when there are exactly two **new** samples in the input channel, which is problematic in our case. Our code only compares these two new samples to each other and tries to figure out if the latch criteria was crossed.

But what happens if the latch criteria is not between the two samples in the current block but is instead between the last sample of the previous block and the first sample of the current block? Those two samples will never get compared and we won't notice we crossed the latch criteria, which means we won't get an output for this criteria point.

C++ Example II: Vector latch math

Our Latch math module can only accept scalar channels as input and can only output scalar samples.

But what if we wanted to latch a vector channel at some condition? We can not just assign this channel as our input because the module will not accept it. Lucky for us quick prototyping is one of C++ Script's strong points and we don't have to write the entire thing from scratch but just make some minor changes to our current module.

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. The table below shows the methods used to read from and write into different types of channels:

Channel type		Reading	Writing
Real	Scalar	.getScalar()	.addScalar()
	Vector	.getVector()	.addVector()
	Matrix	.getMatrix()	.addMatrix()
Complex	Scalar	.getComplexScalar()	.addComplexScalar()
	Vector	.getComplexVector()	.addComplexVector()
	Matrix	.getComplexMatrix()	.addComplexMatrix()

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 synchronous time base.

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

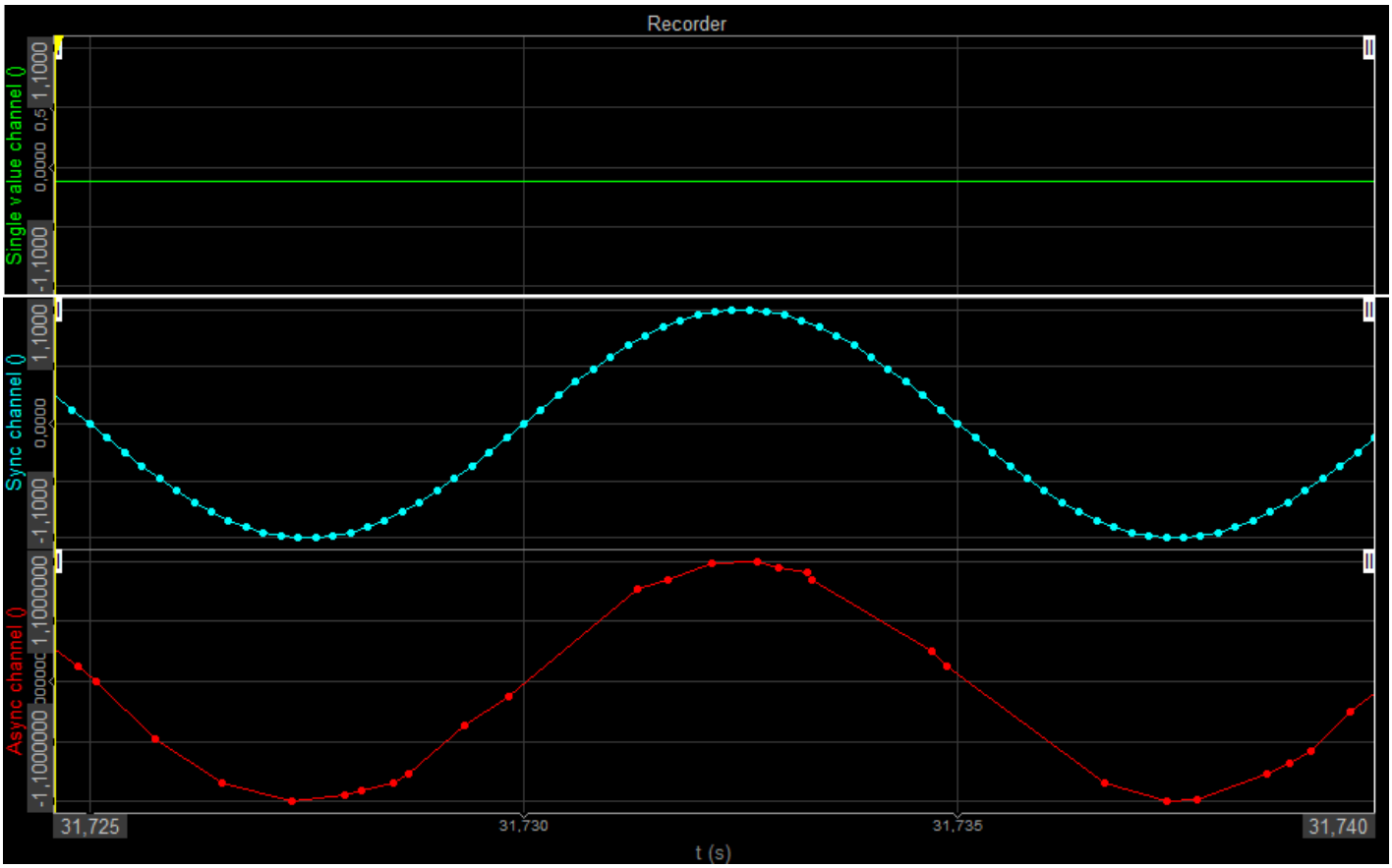


Image 25: Sine curve in channels with different time bases

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) so as to not have too much data in our C++ Script during development, and click ***Ok***.

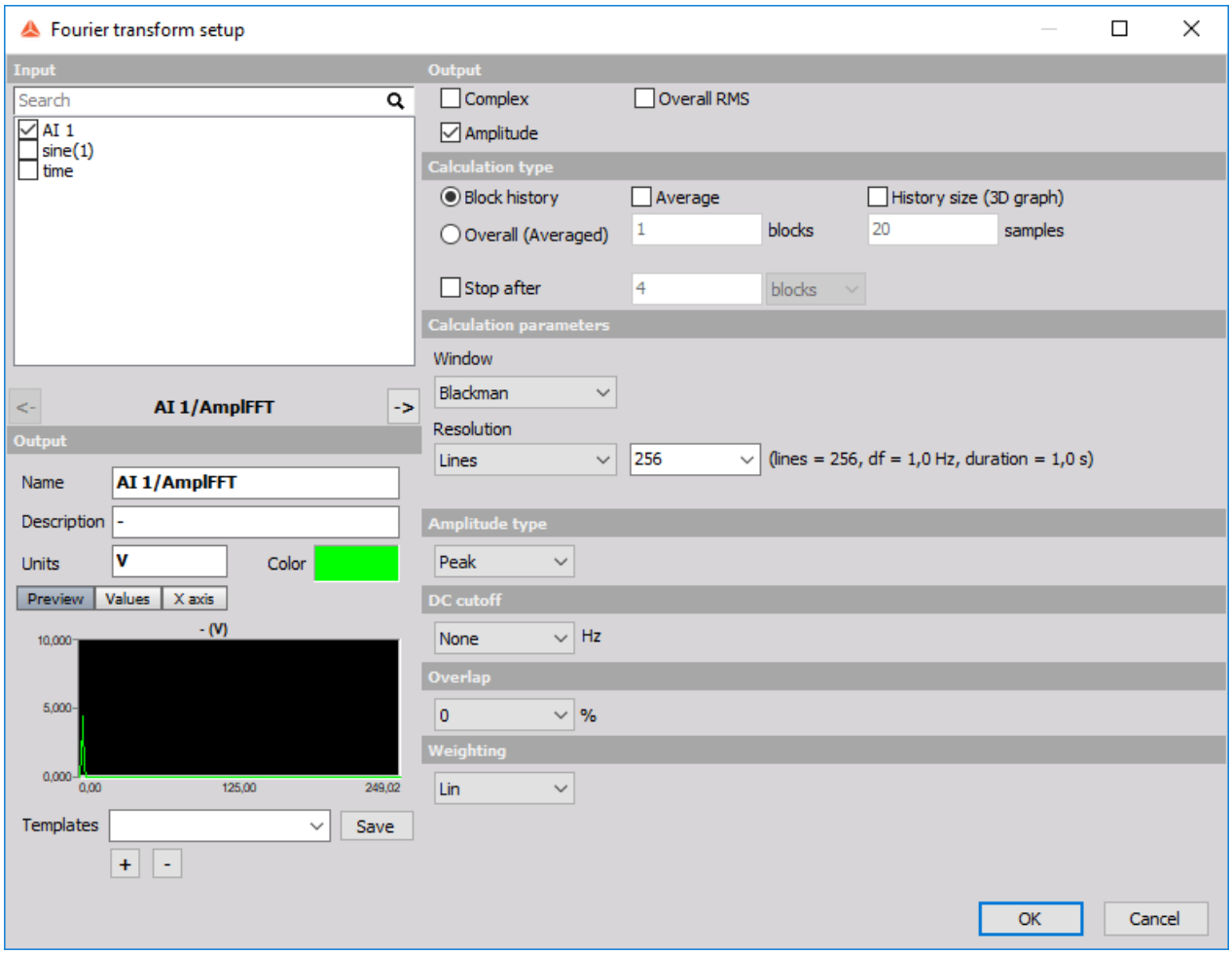


Image 26: Create a vector channel inside Fourier transform setup for testing the created module

Example II: Latch math - Changes in Configure tab

In the *Configure* tab we leave the *Published variables* as they are, and only make changes in the *Input channels* and *Output channels* tabs.

In the *Input channels* tab, we change the Time base of the Criteria channel to Asynchronous. We need to make this change because the first input channel determines the sampling rate of our entire module and if our first input channel were synchronous and the second channel a vector, we could have synchronous vector channels which Dewesoft does not support.

The first input channel's time base determines the rate at which `Module::calculate()` gets called, unless we define any of the output channels as synchronous, in which case it gets called at synchronous rate. This means that in case we have multiple input channels they will all get resampled to the same timebase. We can optionally turn resampling off for each individual input channel by unticking the checkbox inside input channel's setup form in *Configure* tab, in which case the exact last value in the channel is used.

We make this change by pressing the *Setup* button of the Criteria channel and the setup form of the channel will open. We then change the *Time base* of the Input channel to Asynchronous and Single Value and then change the *Value type* to Vector.

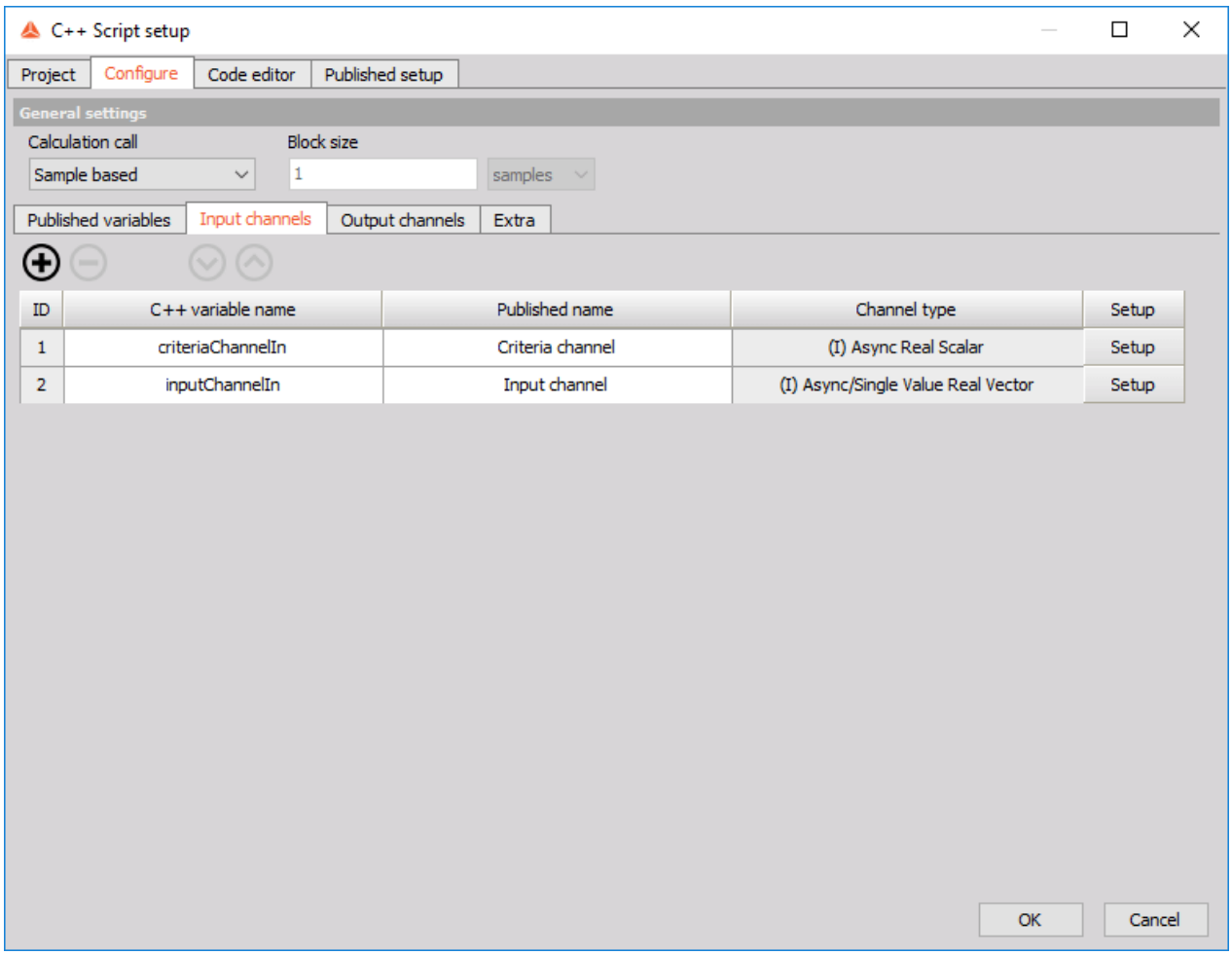


Image 27: Change the Input channels to Asynchronous and Single Value

In the *Output channels* tab we change the Value type of the Latch channel to vector and leave the vector size to the default value, as we will override this setting in the code anyway (since it depends on the size of the input vector from Input channel). Notice this setup also has one more field: Expected Async Rate. Don't worry about this yet, we will explain what this is in the next section.

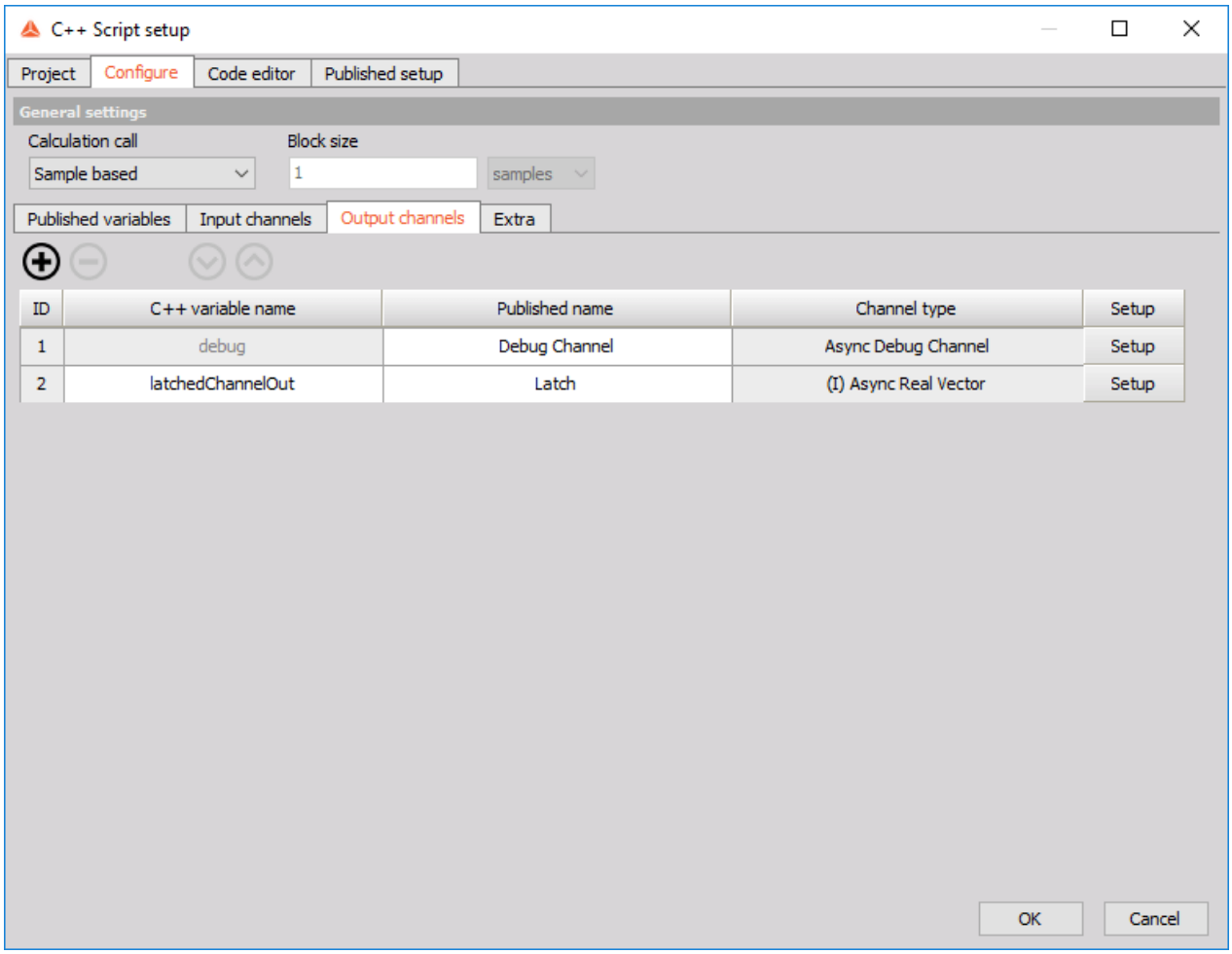


Image 28: In the Output channels tab change the Value type of the Latch channel to vector and leave the vector size to the default value

C++ Example II: Latch math - Changes in Code editor tab

Starting from the correctly "simplified" version of our code, we have to make just a few changes in the *Code editor* tab. We will make changes in `Module::configure()` and `Module::calculate()` methods.

Updated void Module::configure()

We first add code to the `Module::configure()` function. Because our output channel will now be a vector channel we have to set the latchedChannelOut channel's axis to fit the vectors we want to output. In our case we are just copying the values from input channel to output, so this vector will be of the same size as the vector in the input channel, meaning we can simply copy the dimensions over. The name and unit of the output vector will also be the same as the name and unit of the input channel so we can just copy these values as well.

```
inline void Module::configure()
{
    latchedChannelOut.expectedAsyncRate = inputChannelIn.expectedAsyncRate;
    latchedChannelOut.axes[0].values = inputChannelIn.axes[0].values;
    latchedChannelOut.axes[0].name = inputChannelIn.axes[0].name;
    latchedChannelOut.axes[0].unit = inputChannelIn.axes[0].unit;

    pastSamplesRequiredForCalculation = 1;
}
```

Since our output channel is an asynchronous output channel we also need to set one more thing: `expectedAsyncRate`.

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 "how many samples at most will I be adding to this channel per second". We can change the value of this setting in `Module::configure()` by modifying the channel's `expectedAsyncRate`. This setting is required because we need to help Dewesoft figure out how 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`, which we did in our case as seen in the code above.

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.

Updated void Module::calculate()

In the `Module::calculate()` function we have to change the `getScalar()` and `addScalar()` calls to `getVector()` and `addVector()` calls, and `bsc::Scalar` type of `currInputChannelValue` to `bsc::Vector`. We can do this very easily by clicking

the Replace button and replacing "Scalar" with "Vector" where it is necessary, using the Find next and Replace buttons.

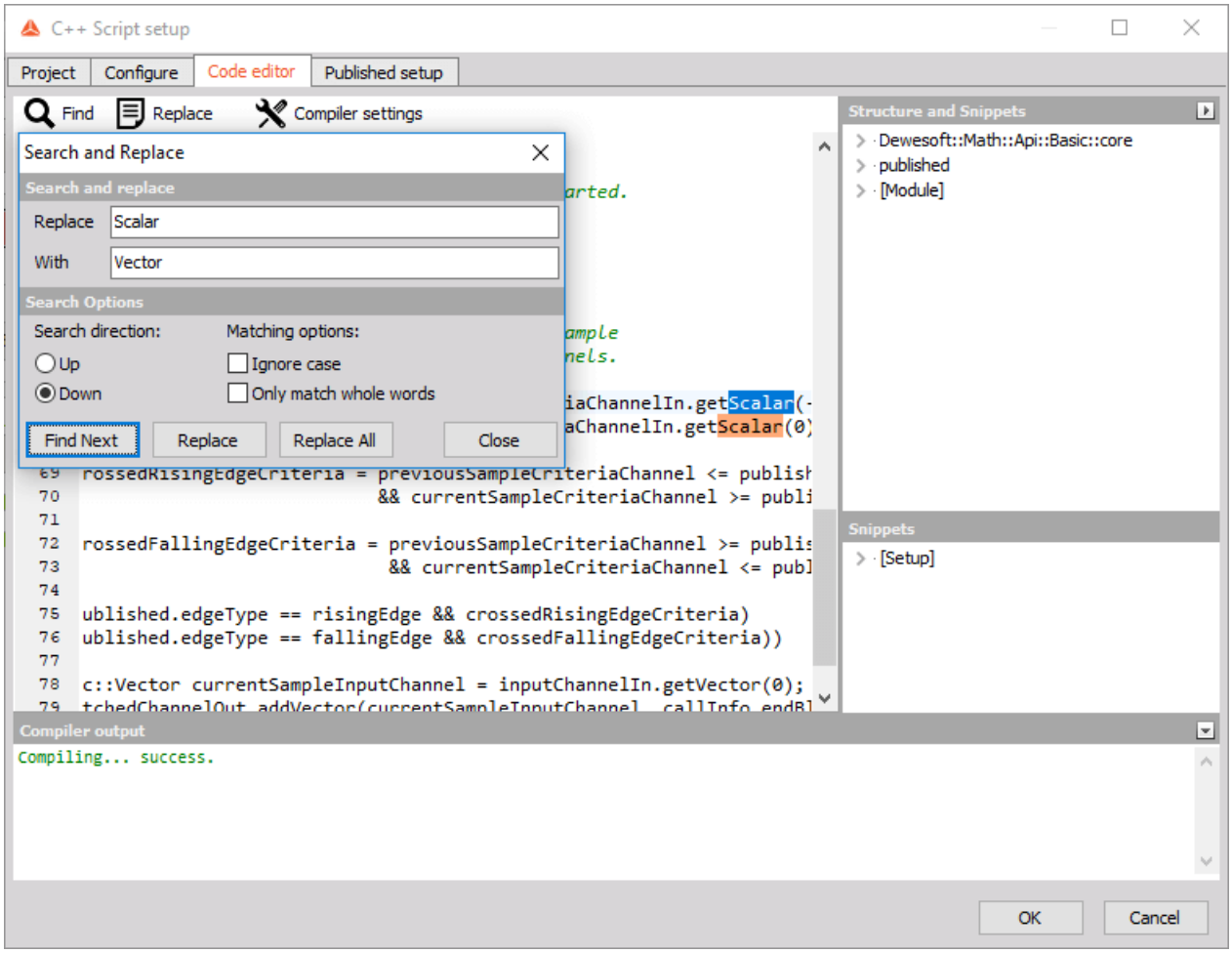


Image 29: To change calls click the Replace button

The code in the `Module::calculate()` function should now look like this:

```
inline void Module::calculate()
{
   bsc::Scalar prevCriteriaChannelValue = criteriaChannelIn.getScalar(-1);
   bsc::Scalar currCriteriaChannelValue = criteriaChannelIn.getScalar(0);

   bool crossedRisingEdgeCriteria = prevCriteriaChannelValue <= published.latchCriteria
                                    && currCriteriaChannelValue >= published.latchCriteria;

   bool crossedFallingEdgeCriteria = prevCriteriaChannelValue >= published.latchCriteria
                                     && currCriteriaChannelValue <= published.latchCriteria;

   if ((published.edgeType == risingEdge && crossedRisingEdgeCriteria)
    || (published.edgeType == fallingEdge && crossedFallingEdgeCriteria))
   {
      bsc::Vector currInputChannelValue = inputChannelIn.getVector(0);
      latchedChannelOut.addVector(currInputChannelValue, callInfo.endBlockTime);
   }
}
```

These simple changes are enough to allow our module to receive vector channels as input and to output a vector at the latch criteria.

C++ Example II: Latch math - Changes in Published setup tab

Because we made changes in the ***Conigure*** tab, the Input channels in the ***Published setup*** tab are unassigned.

But before we continue to notice that we still need one more thing to test our new C++ Script: we need an asynchronous criteria channel. To create one we can just convert a synchronous channel we used in the previous example to an asynchronous one. The simplest way to do this in Dewesoft is by using a ***Basic statistics module***. Close the C++ Script setup by clicking the ***Ok*** button at the bottom of the page and add a Basic statistic module to our setup by clicking the ***Add math*** button and under the statistics section choose ***Basic statistics***. Now we check the box before the sine(1) signal in the Input window and check the RMS box in the Output channels. We set the Calculation type to Sample based and Block based with Block size of 1 sample. We add the statistic to our setup by clicking the ***Ok*** button.

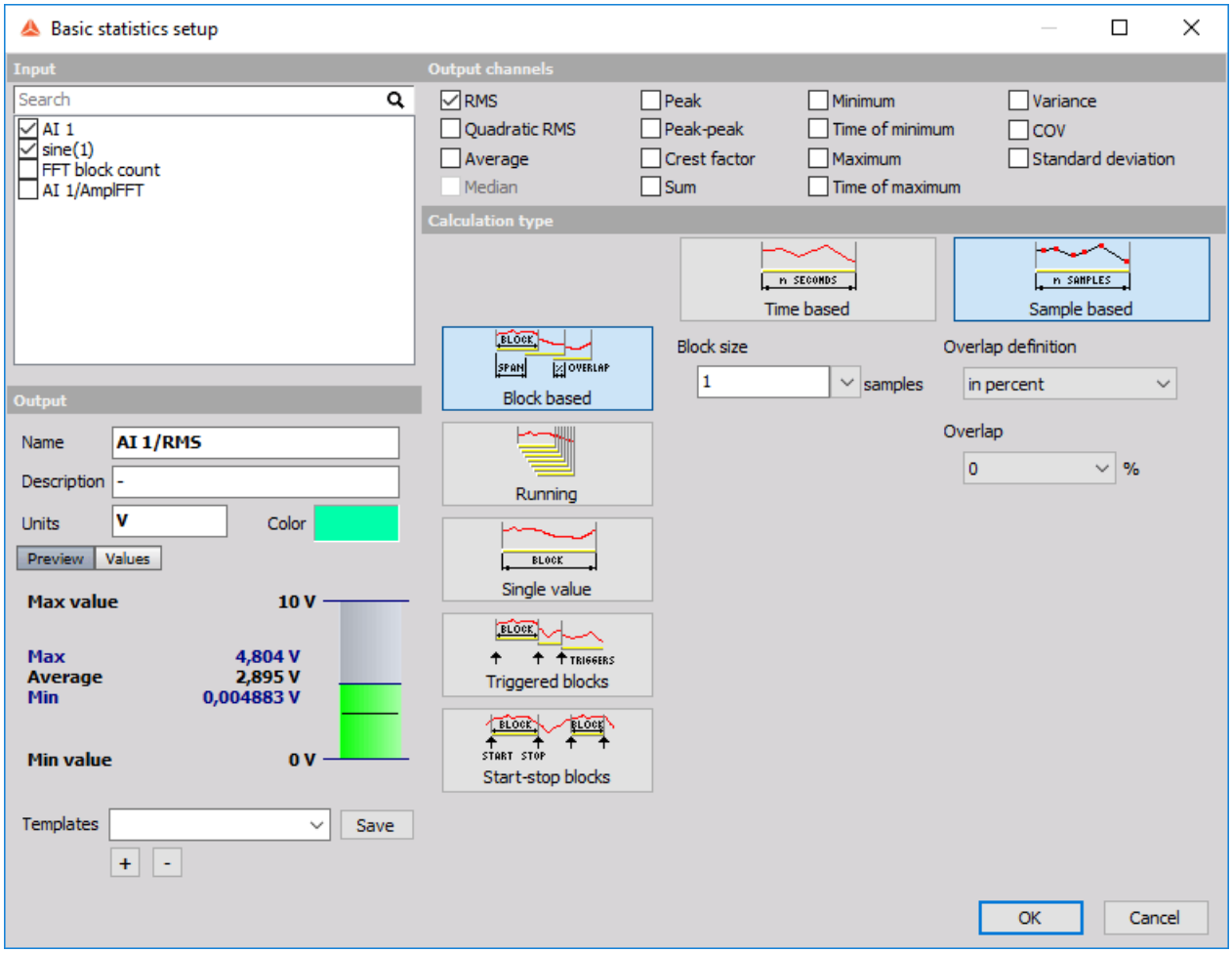
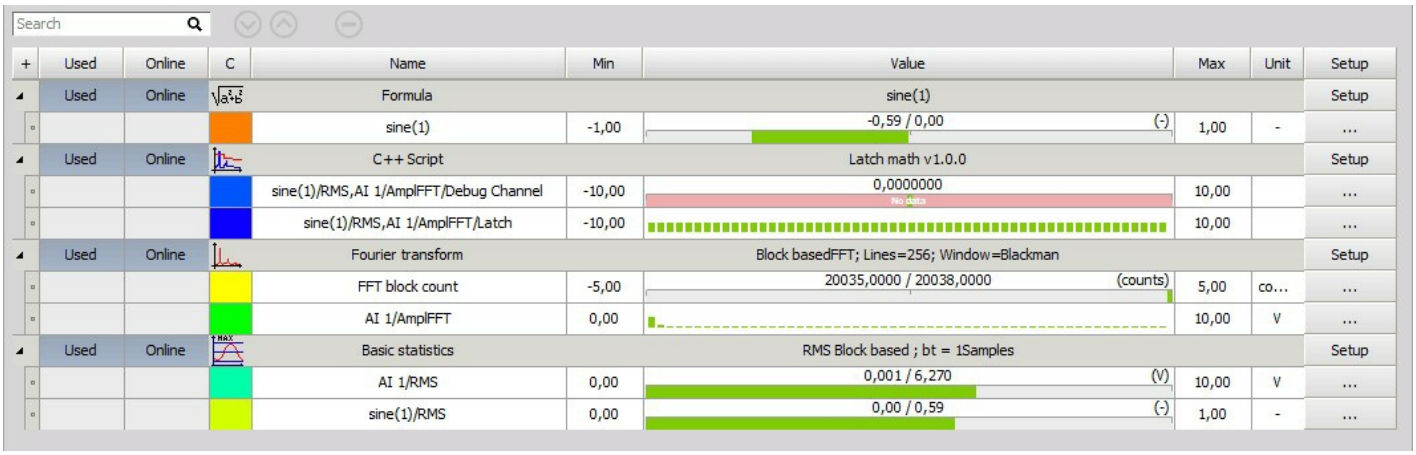


Image 30: Basic statistics setup

Our setup now looks like this:



+	Used	Online	C	Name	Min	Value	Max	Unit	Setup
	Used	Online		Formula		sine(1)			Setup
				sine(1)	-1,00	-0,59 / 0,00 (-)	1,00	-	...
	Used	Online		C++ Script		Latch math v1.0.0			Setup
				sine(1)/RMS,AI 1/AmplFFT/Debug Channel	-10,00	0,0000000	10,00		...
				sine(1)/RMS,AI 1/AmplFFT/Latch	-10,00		10,00		...
	Used	Online		Fourier transform		Block basedFFT; Lines=256; Window=Blackman			Setup
				FFT block count	-5,00	20035,0000 / 20038,0000 (counts)	5,00	co...	...
				AI 1/AmplFFT	0,00		10,00	V	...
	Used	Online		Basic statistics		RMS Block based ; bt = 1Samples			Setup
				AI 1/RMS	0,00	0,001 / 6,270 (V)	10,00	V	...
				sine(1)/RMS	0,00	0,00 / 0,59 (-)	1,00	-	...

Image 31: All the equations

We can now assign the Input channels in the C++ Script setup in the ***Published setup*** tab. We assign the "sine(1)/RMS signal" to the Criteria channel and we assign the "AI 1/AmplFFT" signal to the Input channel then click ***Ok***.

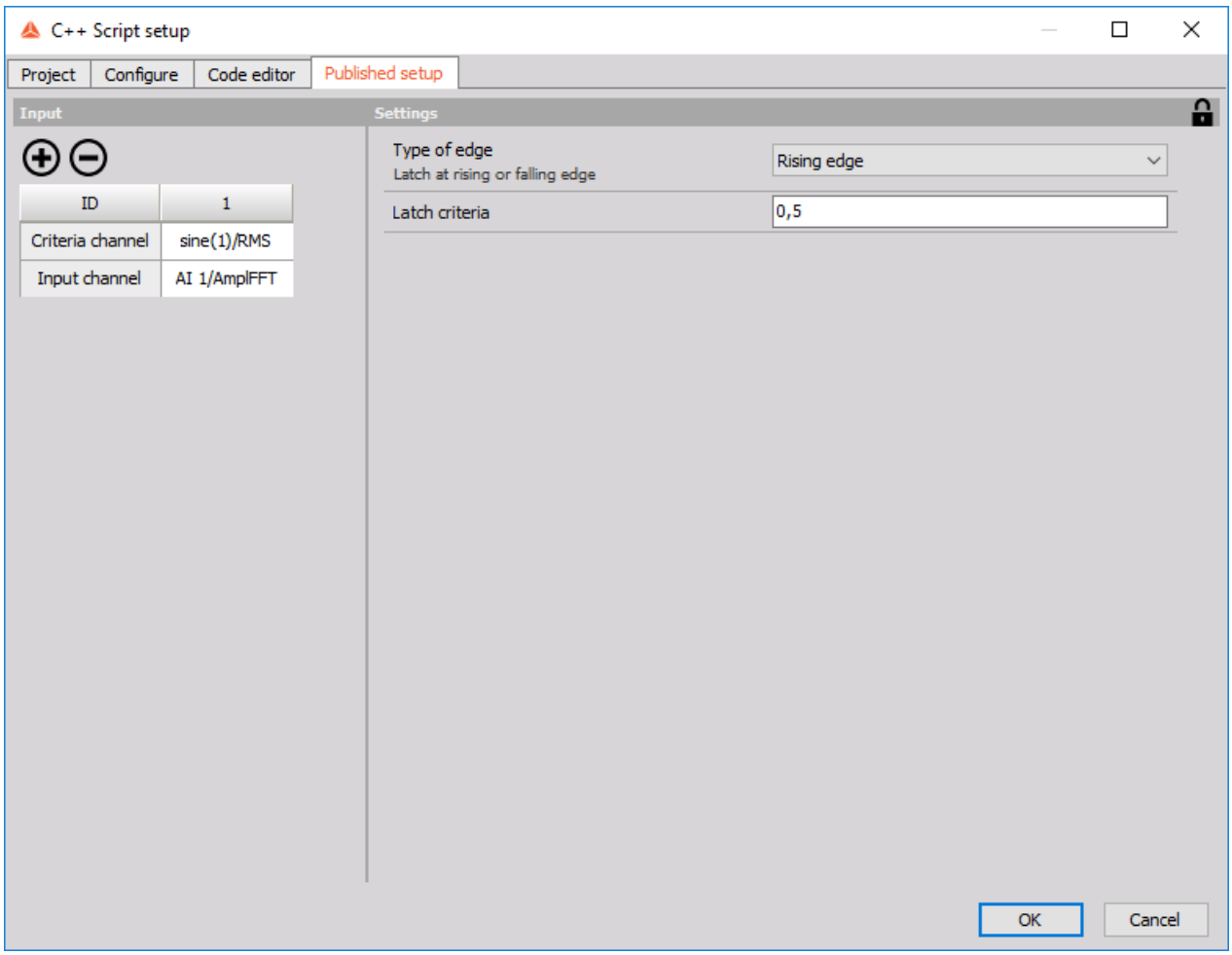
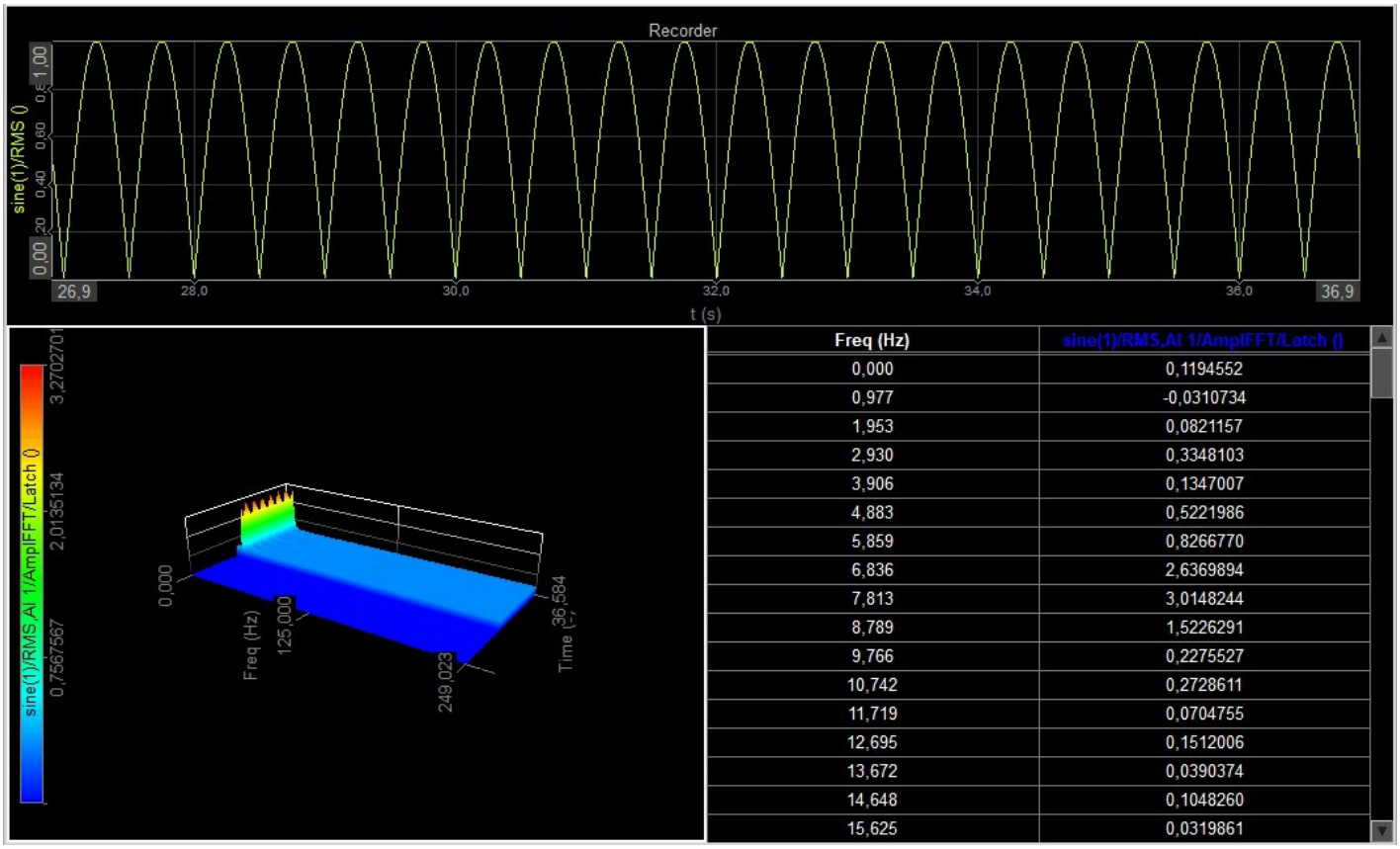


Image 32: Assign the channels

Now we can see results by going into Measurement mode in Dewesoft.

C++ Example II: Latch math - Output

In the image 33 we see the outputted vectors on a 3D graph. A new vector is outputted every time the sine(1)/RMS signal passes the value 0,5. We can also output the vector in a 2D/3D table and see the values in the vector.



Freq (Hz)	sine(1)/RMS.AI 1/AmplFFT/Latch ()
0,000	0,1194552
0,977	-0,0310734
1,953	0,0821157
2,930	0,3348103
3,906	0,1347007
4,883	0,5221986
5,859	0,8266770
6,836	2,6369894
7,813	3,0148244
8,789	1,5226291
9,766	0,2275527
10,742	0,2728611
11,719	0,0704755
12,695	0,1512006
13,672	0,0390374
14,648	0,1048260
15,625	0,0319861

Image 33: Outputted vectors on 3D graph

But if we now decide that the latch condition should occur on the falling edge instead, we can again just change the variables in the *Published setup* of the C++ Script setup.

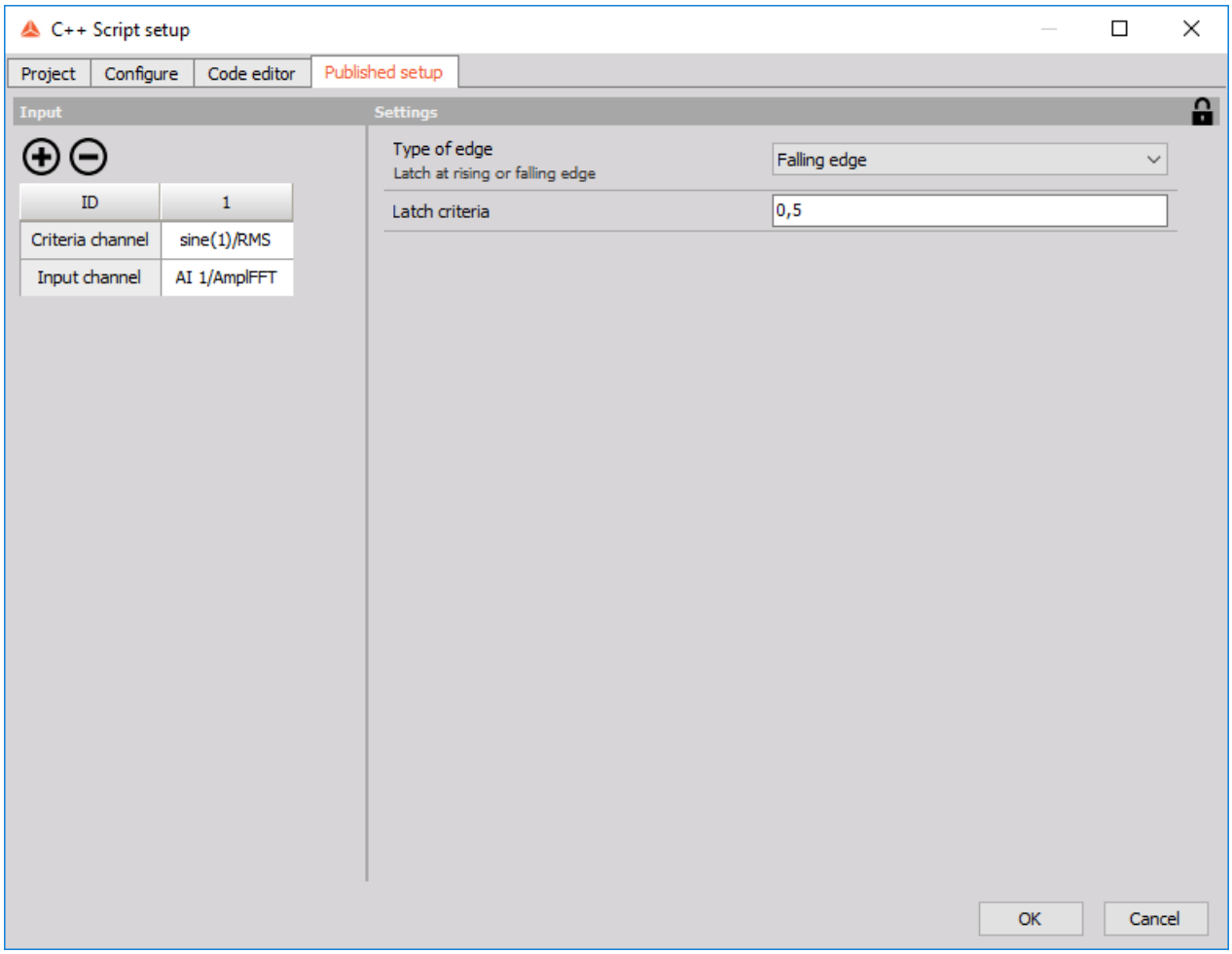
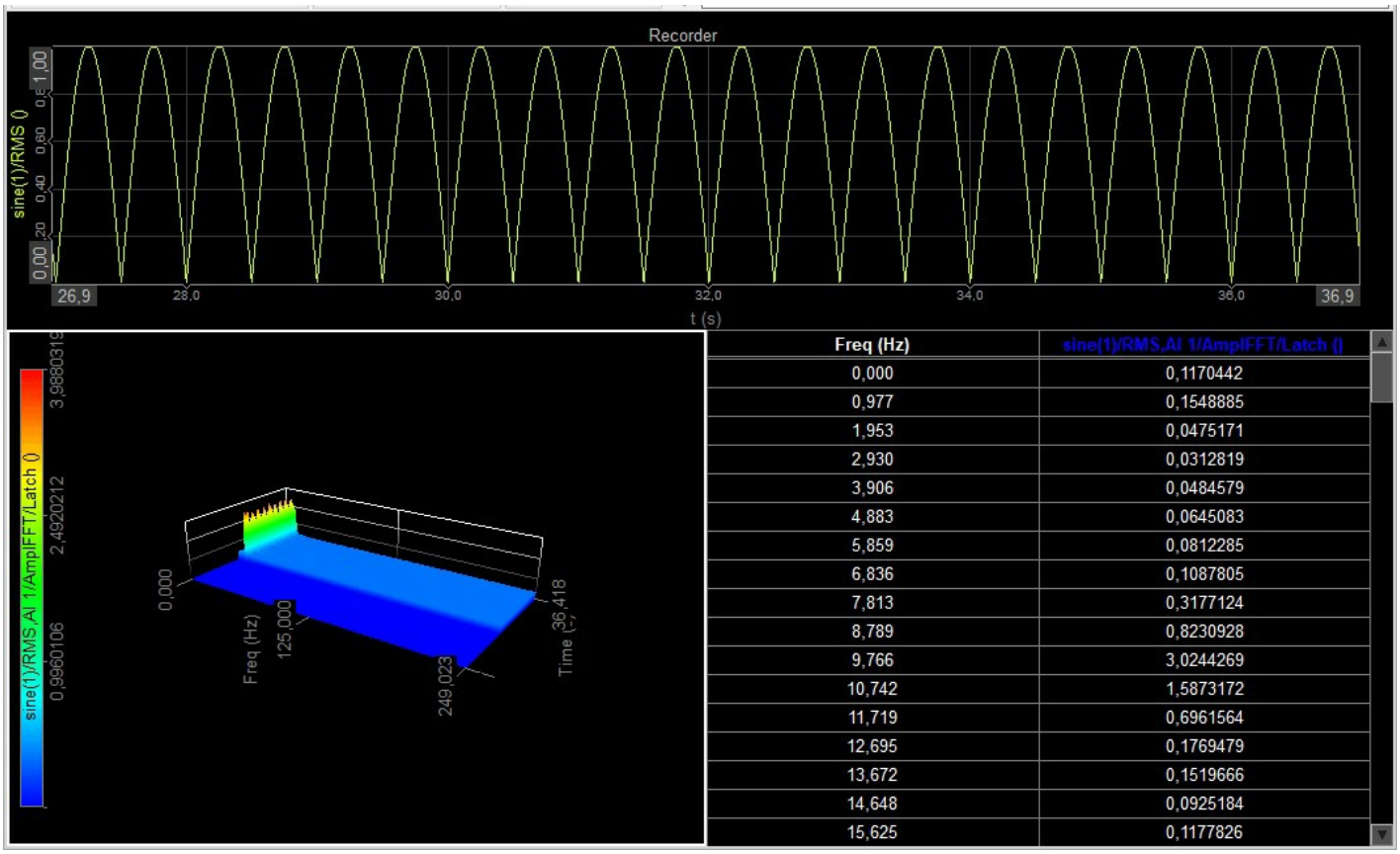


Image 34: Change the variables in the Published setup

The output will now look like this:



Freq (Hz)	sine(1)/RMS,AI 1/AmplFFT/Latch ()
0,000	0,1170442
0,977	0,1548885
1,953	0,0475171
2,930	0,0312819
3,906	0,0484579
4,883	0,0645083
5,859	0,0812285
6,836	0,1087805
7,813	0,3177124
8,789	0,8230928
9,766	3,0244269
10,742	1,5873172
11,719	0,6961564
12,695	0,1769479
13,672	0,1519666
14,648	0,0925184
15,625	0,1177826

Image 35: New output

C++ Example II: Latch math - Debug channel

To aid with the development of your C++ Script a special debug output channel (of type asynchronous string) is enabled by default whenever you create a new script. With this debug channel you can use a special `void outputDebugString(const std::string& message, bsc::Time timestamp)` function to output arbitrary string messages to it from `Module::calculate()`. On top of this, the debug channel will also automatically contain any exceptions thrown from inside your `Module::calculate()` function.

To see these messages in measurement mode, add a Digital meter visual control to your display and bind the debug channel to it. Note that since the debug channel is an asynchronous channel, you will need to properly specify the expected async rate per second value either in channel's setup or by setting `debug.expectedAsyncRate` to an appropriate value in `Module::configure()` in your C++ code.

Debug channel can easily catch all the C++ exceptions triggered from your `Module::calculate()`, but it cannot do much in case of segmentation faults and other undefined behavior. There is no simple way of finding these from C++ Script as they will usually crash Dewesoft.

An example of a message outputted by the debug channel can be seen below. It shows a warning we get if we forget to set the `pastSamplesRequiredForCalculation` in `Module::configure()` method of our example and still want to access the previous sample of the Criteria channel using `criteriaChannel.getScalar(-1)` call.

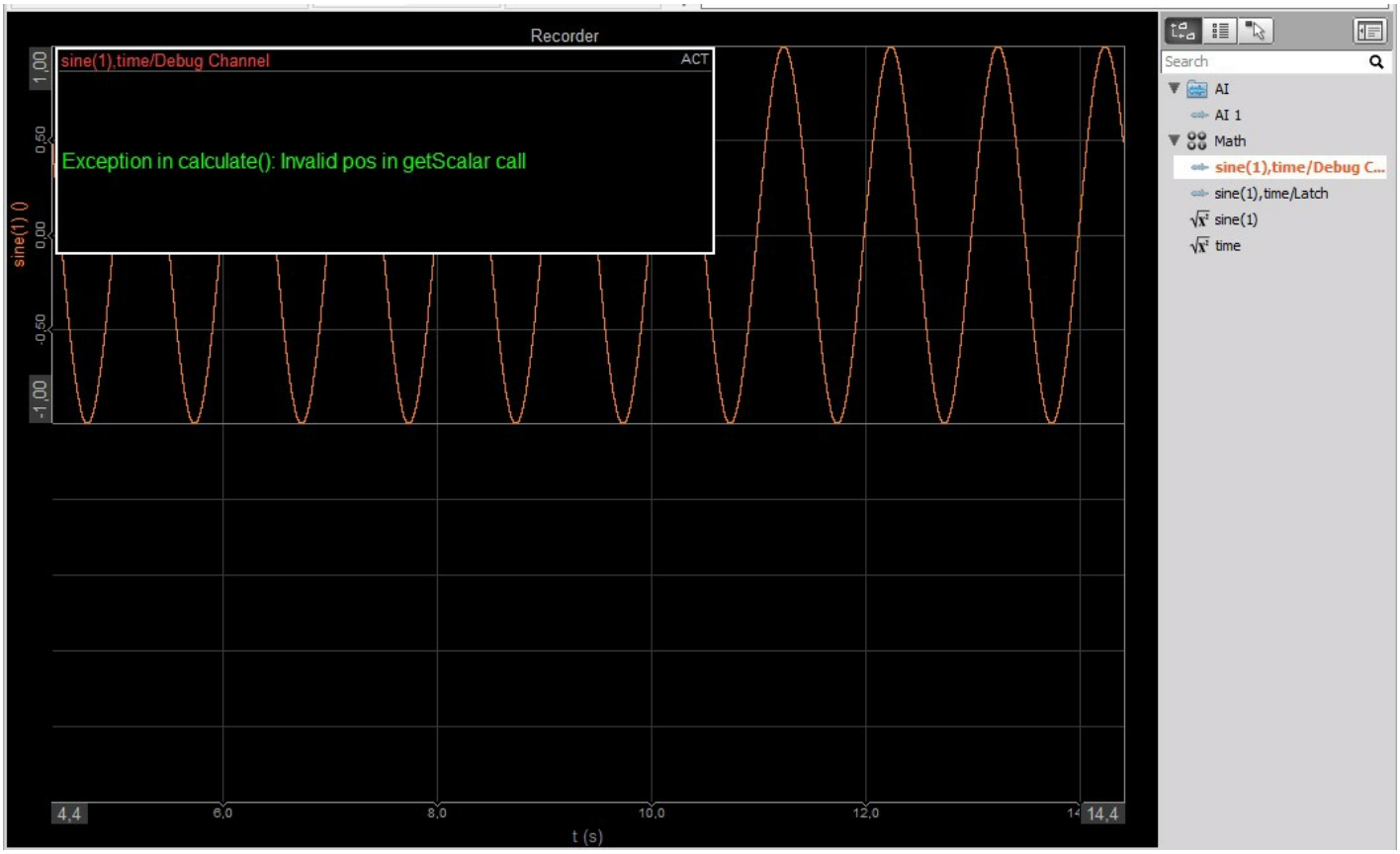


Image 36: Debug channel output

When you are done developing your module, you can easily remove the debug channel by unticking the checkbox in *Configure*'s *Extra* tab.

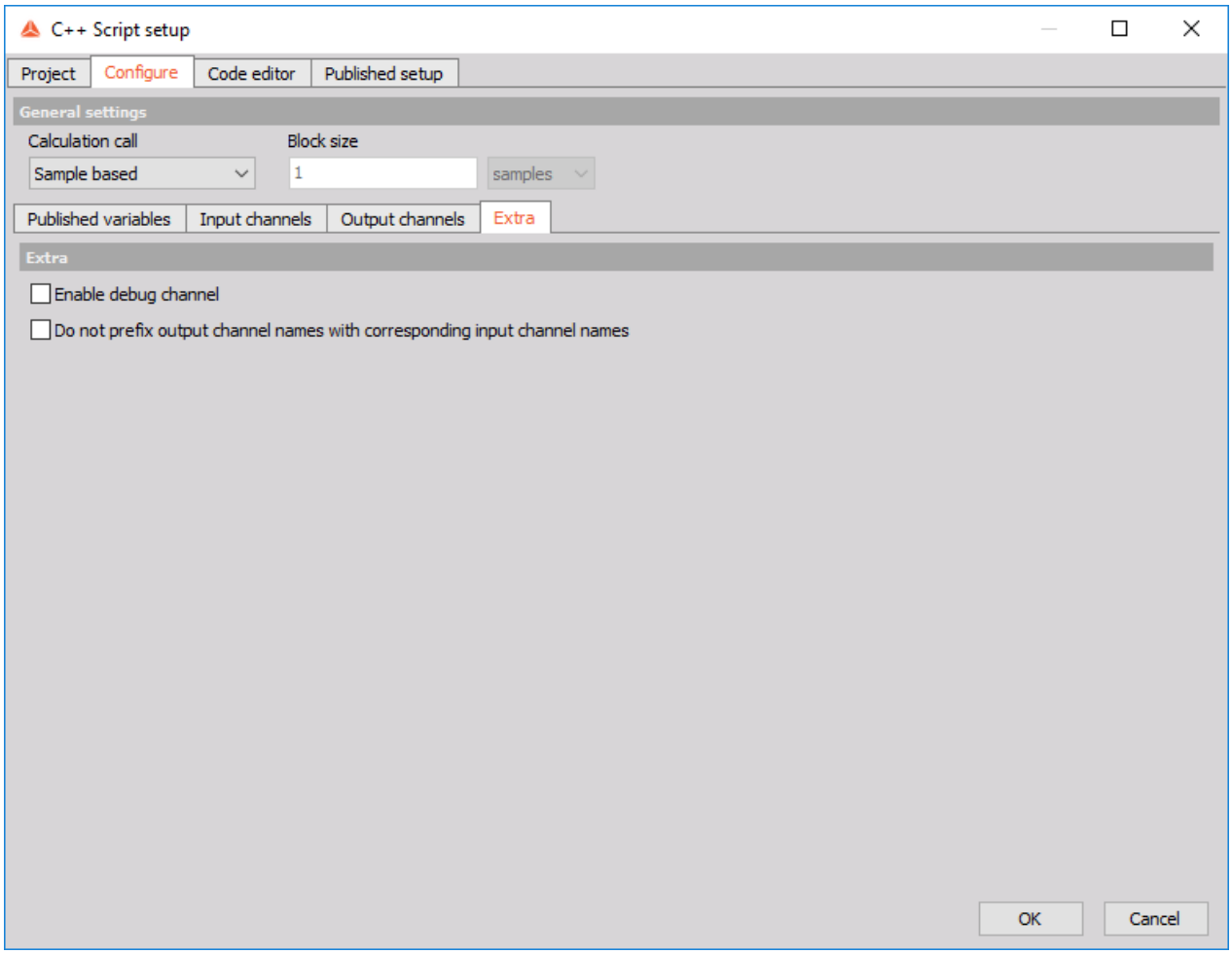


Image 37: Remove the debug channel by unticking the checkbox in Configure's Extra tab

How to Import/Export C++ bundle?

In this pro training we have created a pretty useful, general *Math module*. One thing we might want to do next is use our latch module in other setups, possibly on other computers. The way to do this is by using C++ Script's bundle functionality, which can bundle together all the settings along with the precompiled module, meaning we don't need to compile it again if we just want to use it on a different machine.

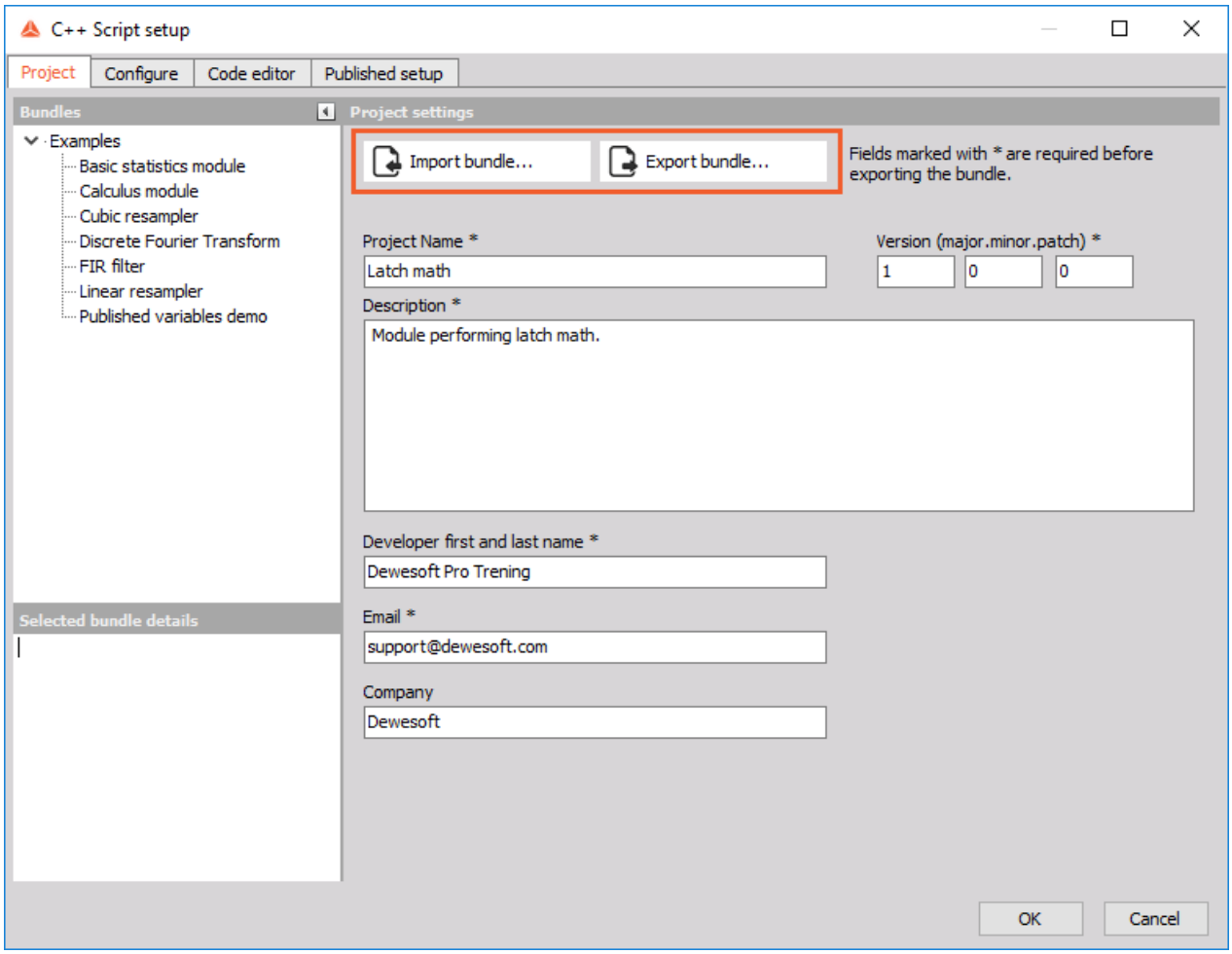


Image 38: Import and export bundle in Project tab

Export the script by going to the Project tab of our setup and click on the Export bundle... button. A window will appear asking you for the way in which you want to bundle the module:

- **Open source**: bundle is exported with the source code included. When such a bundle is imported via the *Import bundle...* button, the user will have access to all 4 tabs in C++ Script.
- **Freeware**: bundle is exported without the source code. When such a bundle is imported via the *Import bundle...* button, the user will only have access to the *Published* tab. Be careful, as there is no way to recover the source code from a Freeware bundle.

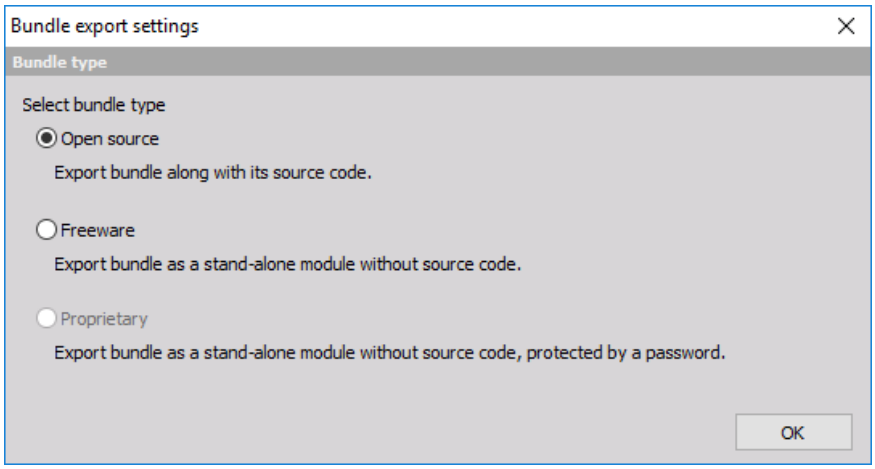


Image 39: Bundle export settings

Select whichever type you want and click ***Ok***. A new window will open prompting you to choose a destination on the computer where you want the bundle to be saved. After choosing the destination click ***Save*** and your bundle will be recompiled to work with all the supported architectures and saved to your desired location with the ".cbu" extension.

Import the bundle on another computer or in a different setup by creating a new C++ Script and clicking the ***Import bundle...*** button in the ***Project*** tab and locating the "*.cbu" file you want to load. Click ***Open*** and the bundle will be imported to your current setup.

Stand-alone math

Exported ".cbu" bundles can be added to Dewesoft's `bin\addons\` folder, just like regular plugins. Dewesoft is able to recognize such bundles, and display them in the *"Add math"* menu, just like built-in Dewesoft math modules. Since bundles contain all the precompiled binary files, the end-user doesn't need DSMinGW to use such modules.

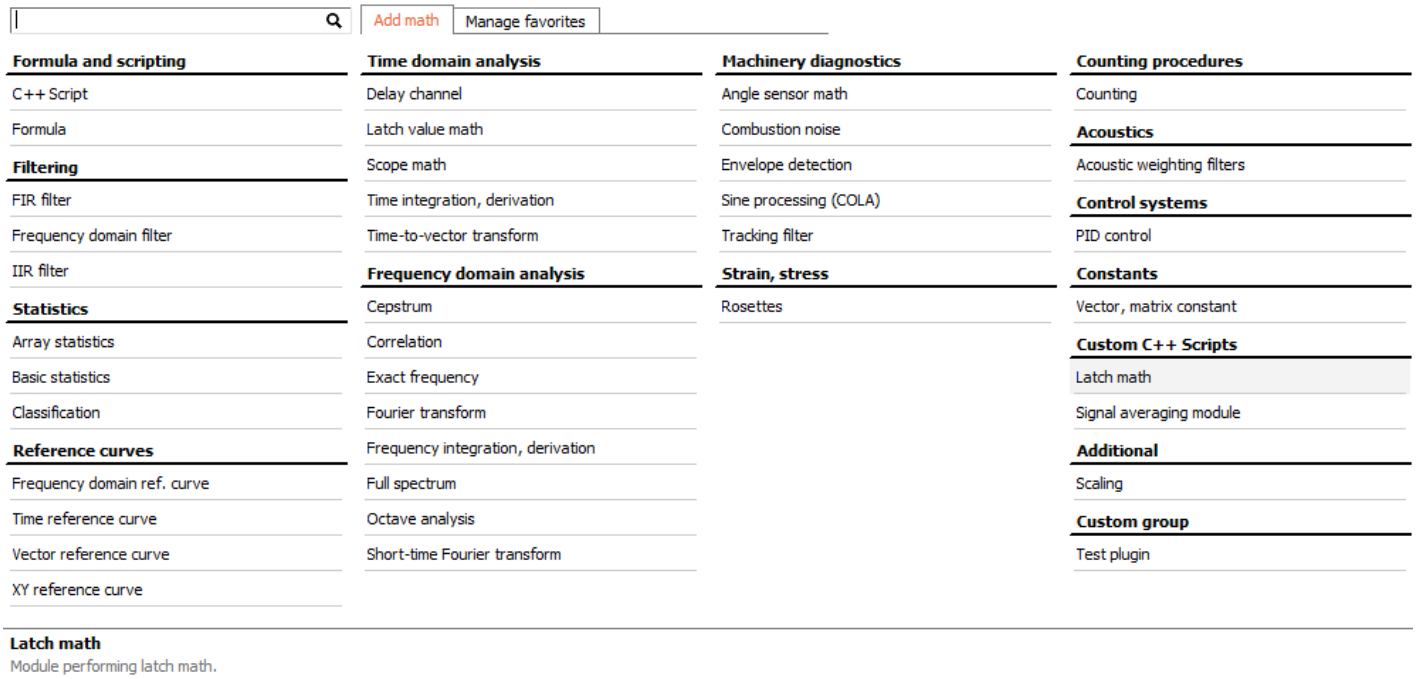


Image 40: Add math section

In the image 40, we have exported our Latch math bundle into `bin\addons\` folder. You can see that after restarting Dewesoft, our module appears under the *Custom C++ Scripts* group, with the name we specified on the *Project* tab under *Project name* and the description as provided in the first line of the *Description* field. We can now use our Latch math module in any setup we want, and we only have access to the *Published* tab with other tabs hidden from the end-user.

In which ways is it possible to extend 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	- The most intuitive of all the approaches, very simple to use. - Integrated fully into Dewesoft meaning no set up required to get running.	- 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	- 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 ***Published setup*** tab. - It can work with an arbitrary amount of input and output channels.	- Requires familiarity with at least basics of programming in C++. - Difficult to test and debug.
Plugins	- 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.	- Requires Visual Studio. - Much harder to learn to use than C++ Script.
Sequencer / DCOM	- 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.	/

Where to go from here?

A great way to learn anything new is to study examples. For this exact reason we created some C++ Script bundles for you, which you can find on Dewesoft's webpage under Support > Downloads > Developers > C++ Script. Note that you have to be logged in to access the C++ Script section. These bundles hopefully demonstrate how versatile the C++ Script is, and may give you a better feel of how and when to use it for your own problems.

For a more in-depth description of the C++ Script you can also check out the C++ Script manual, available on the same webpage as the example bundles. It contains yet another step-by-step example of creating a useful math module, as well as descriptions of all the available structures and features of the C++ Script.

Support > Developers section of Dewesoft website is also a great place for seeking help and additional clarification both from Dewesoft staff, as well as community in general.