Dewesoft C++ Script

DEWESoft®

SOFTWARE USER MANUAL

Dewesoft C++ Script V21-1

1. Table of contents

2. About this document

2.1. Legend

The following symbols and formats will be used throughout the document.

Important
It gives you important information about the subject.
Please read carefully!

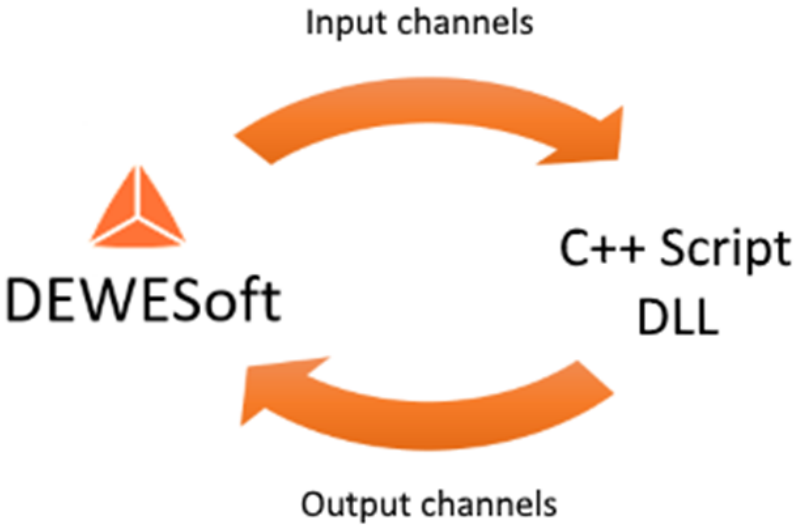
Hint
It gives you a hint or provides additional information about a subject.

Example
Gives you an example of a specific subject.

3. What is Dewesoft's C++ Script?

C++ Script is Dewesoft's tool for creating custom math modules. It exists as a compromise between formula's simplicity and full plugins' power, allowing you to write complex Dewesoft math modules with ease. It can be configured to operate on arbitrarily many input channels and produce arbitrarily many output channels, all while processing the data with the power of modern C++.

3.1. How it works

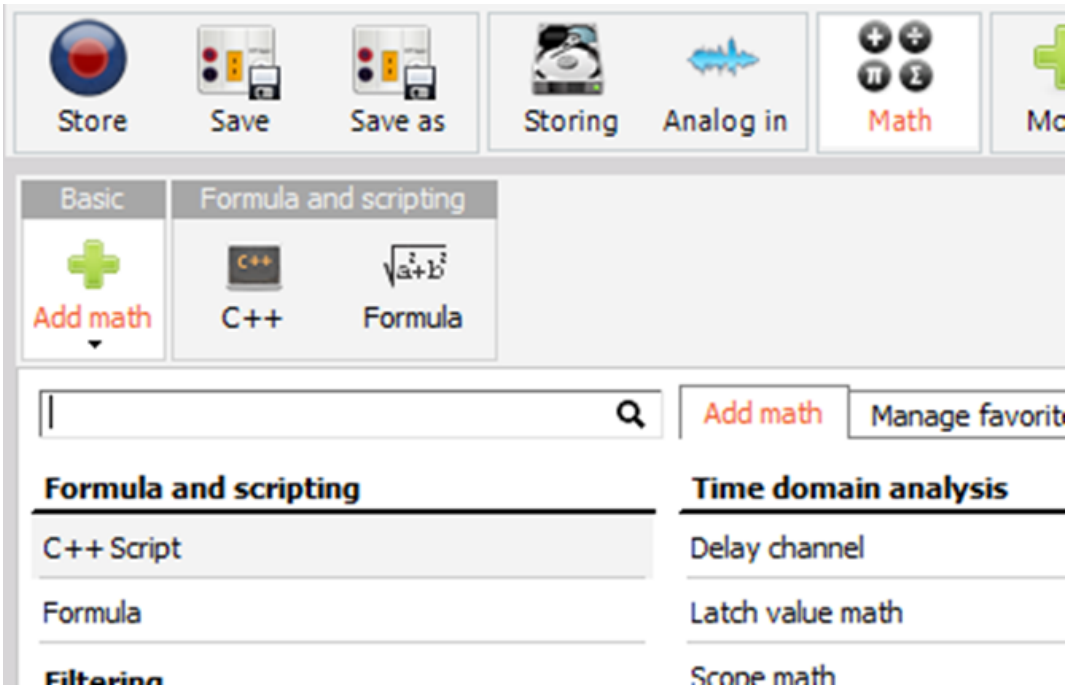


C++ Script's data cycle

The way C++ Script works behind the scenes is by taking your C++ code and using MinGW's C++ compiler to compile it into a DLL. Dewesoft is then able to load this DLL, feed it data from input channels, and retrieve the processed data into output channels. Because the hard work of supporting all the different channel types is handled by Dewesoft and C++ Script, all you are left with is a nice abstraction layer in C++.

3.2. Installation

Since Dewesoft X3 SP6, C++ Script is a first-class citizen of Dewesoft. You can find it under **Math**, next to **Formula** in **Formula and scripting** section.



Location of C++ Script

In case you want to write your own C++ Scripts, you will need DSMinGW installed on your system. DSMinGW is a package of compilers for C++, that allows you to write and compile C++ Scripts in Dewesoft. If you didn't select DSMinGW option while installing Dewesoft, you can manually install it from Dewesoft's webpage https://download.dewesoft.com under Support > Downloads > Developers > C++ Script (you will need to be logged in to access the Developers download section).

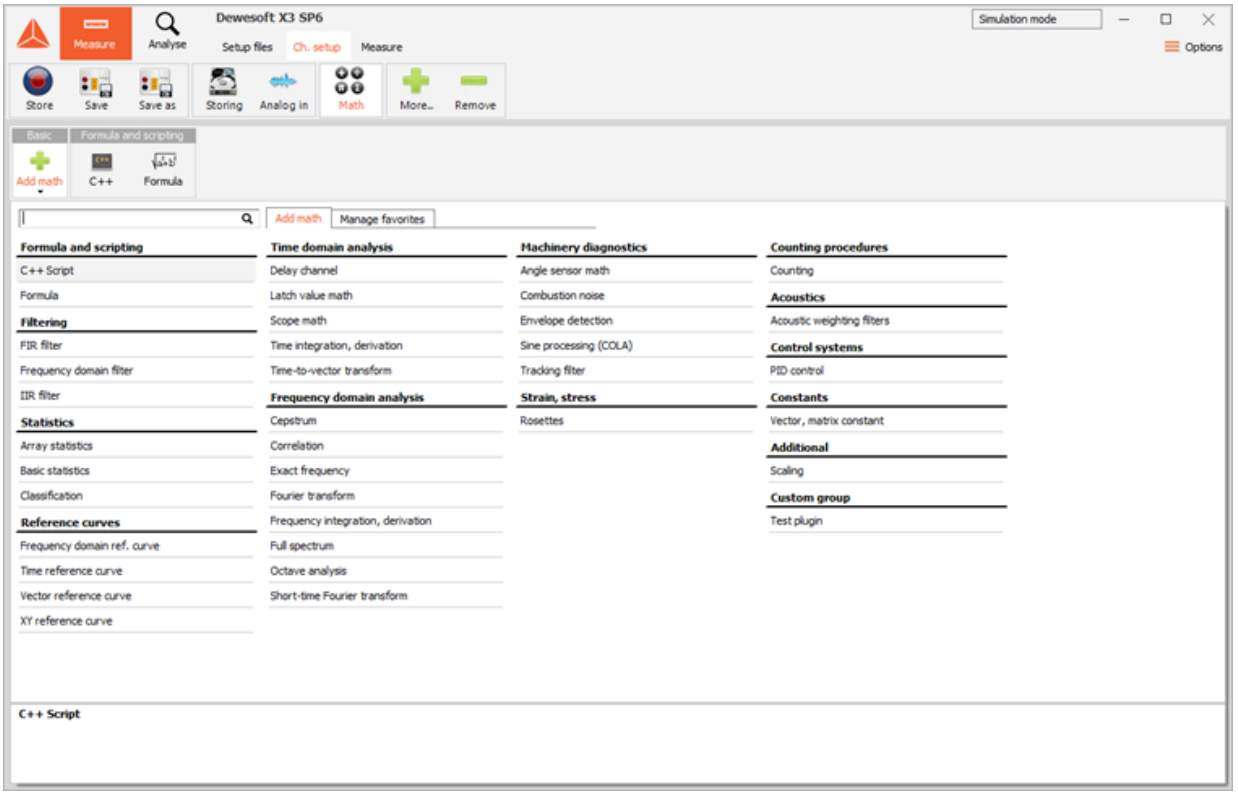
4. Tutorial: Simple Signal Averaging Module

To demonstrate both the simplicity and power of the new C++ Script the first part of this manual shows you how to implement a simple averaging module. At the end of this section you should end up with a working module which allows you to select a channel to average, and produce an output channel with averaged values.

Note that the tutorial is only meant to introduce you to the steps required to use the C++ Script and we will discuss all the different options the module has to offer in detail later on in the manual.

4.1. Step Zero: Creating a new C++ Script

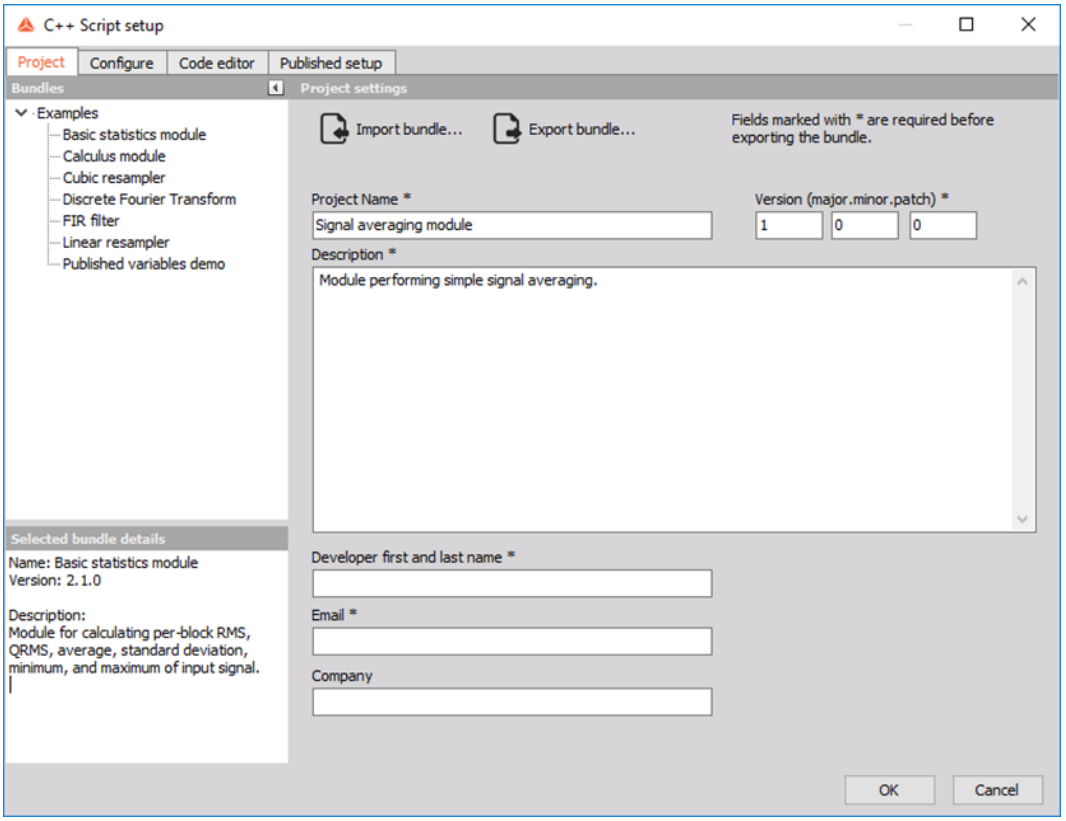
Tutorial is split into 4 steps, which correspond to the tabs at the top of the C++ Script setup form. To start, create a new C++ Script by clicking the **New Setup** button in Dewesoft's ribbon menu. Next, click the **Math** button and **Add Math** button which appears underneath it. In the menu that just opened up, find the section called **Forumula and scripting** and finally click on **C++ Script**, which brings up a new C++ Script window.



Creating a new C++ Script

4.2. Step One: Project tab

In the new setup window you are greeted with the **Project details** tab. Here you are expected to specify your module name, a brief description of what your module does, and the version of your module. Proceed by filling out the *Project Name* field with "*Signal averaging module*", *Description* with "*Module performing simple signal averaging*." and leave the rest of the fields as they are.

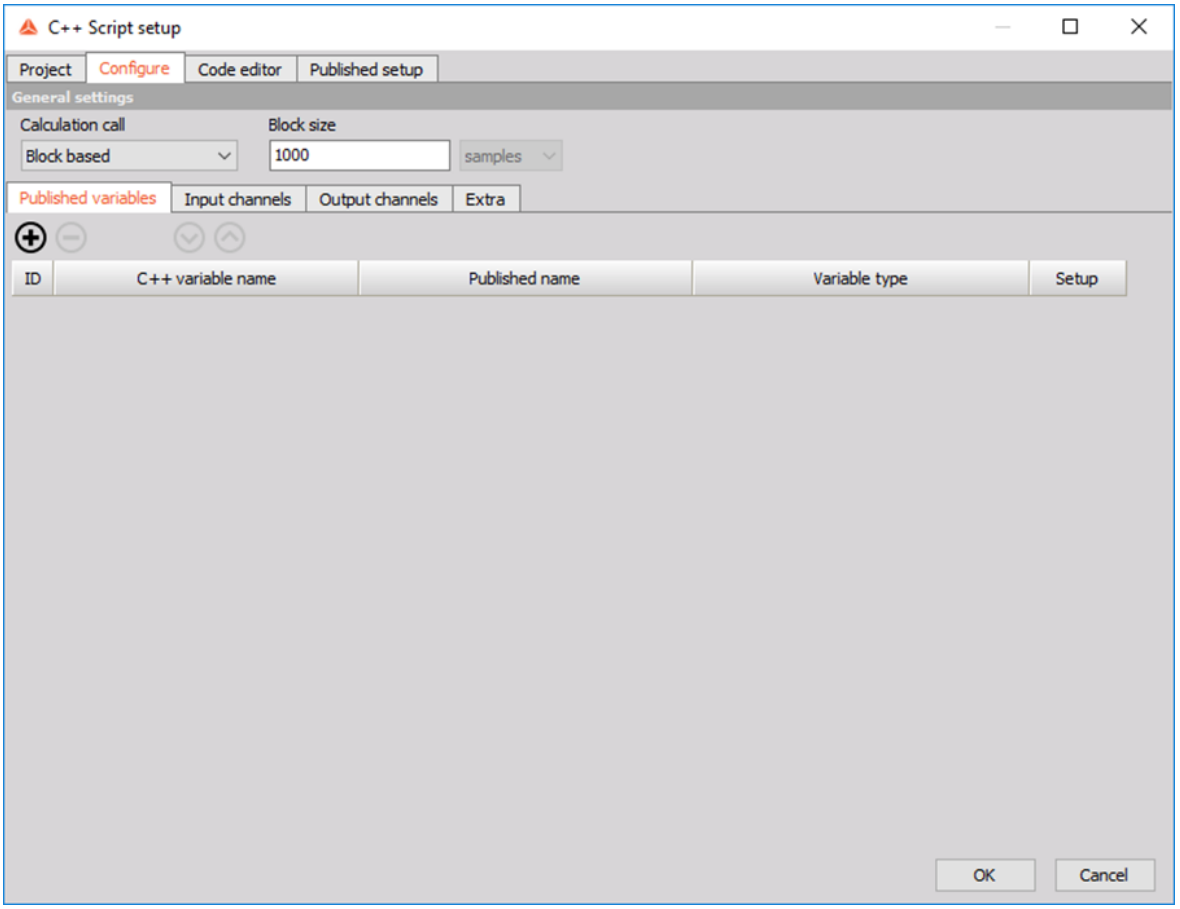


Filling out the Project tab

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

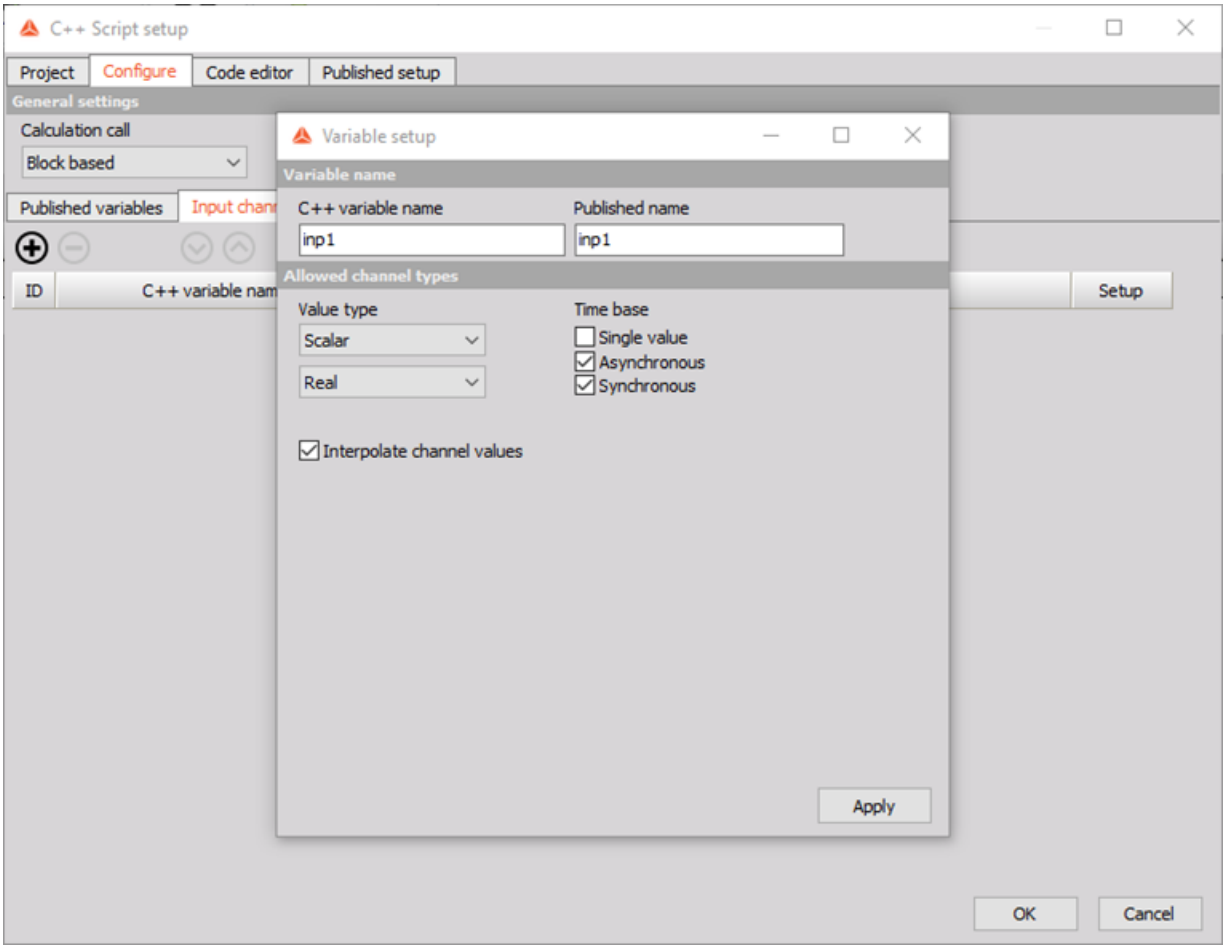
4.3. Step Two: Configure tab

In the **Configure** tab, first change the **Calculation call** from Sample based to Block based. This should enable the **Block size** edit field right next to it and fill it with 1000. For now leave it at that number.



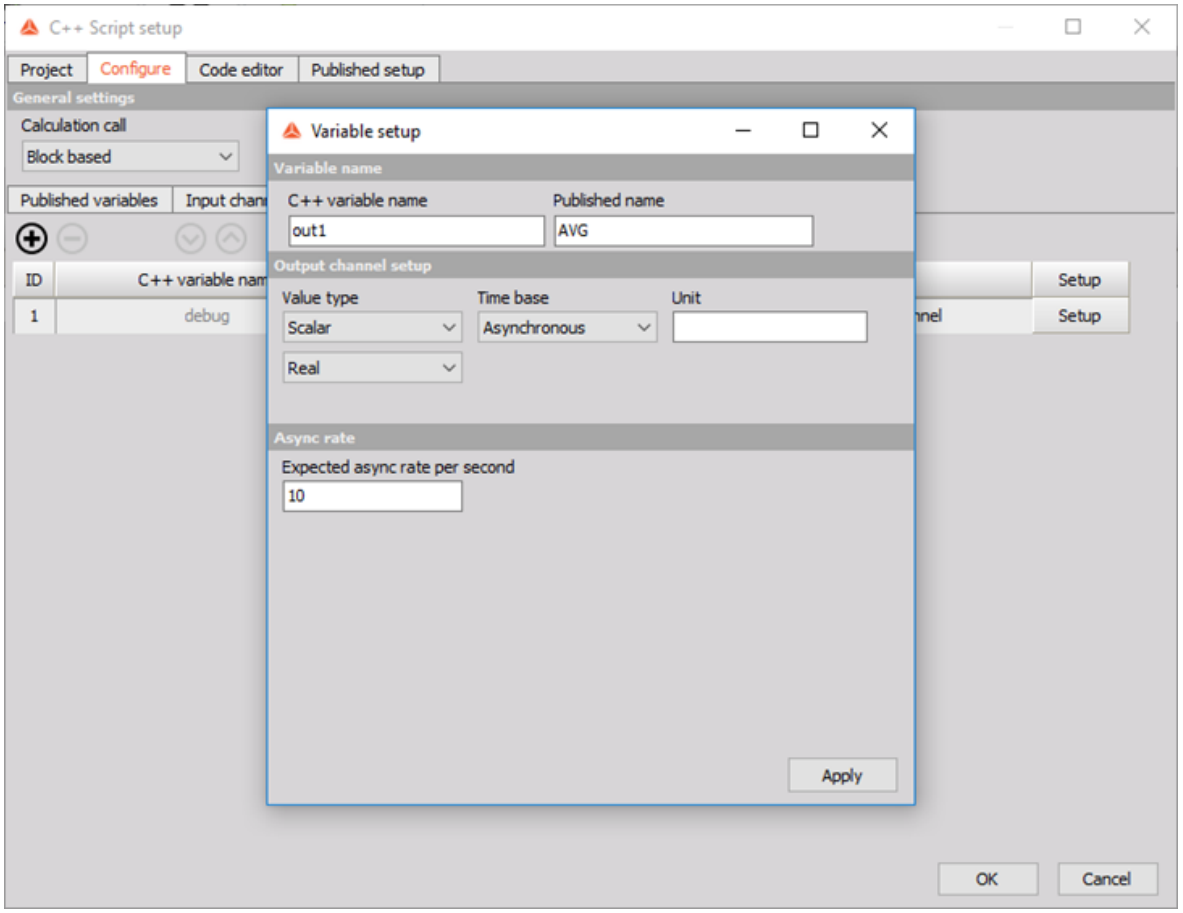
Changing calculation type to 'block based'

Under the *Input channels* tab click the **(+)** button. This brings up a *Variable setup* form for configuring your input channel. Leave both fields in **Variable name** as they are, and **Value type** as *Scalar* and *Real*. Tick the *Synchronous* checkbox under Time base. Clicking *Apply* you should find that the table now has a single row, containing *inp1* in the *C++ variable* name column and *(I) Sync/Async Real Scalar under Channel type*. This means your module is going to have exactly one channel of either synchronous or asynchronous type as input.



Input channel

Under the *Output channels* tab you will be able to find a 'debug' channel. Ignore it and click the **(+)** button again. This brings up a form similar to last, except Time base now needs to be specified exactly. Set channel's **Published name** to *'AVG'*, leave the *Value type* as *Scalar* and *Real*, and set the **Timebase** to *Asynchronous*. Set *Expected async rate per second* field to the current sample rate of your setup (found under **Analog in** tab under **Dynamic acquisition rate**) divided by 1000; in our case we set the sampling rate to 10000 to simplify this calculation so the value in this field should be 10000/1000=10. Clicking apply creates a new row in the table, this time containing *out1*.



Changing output channel's type

Close the variable setup form and get ready to write some code in the Code editor tab!

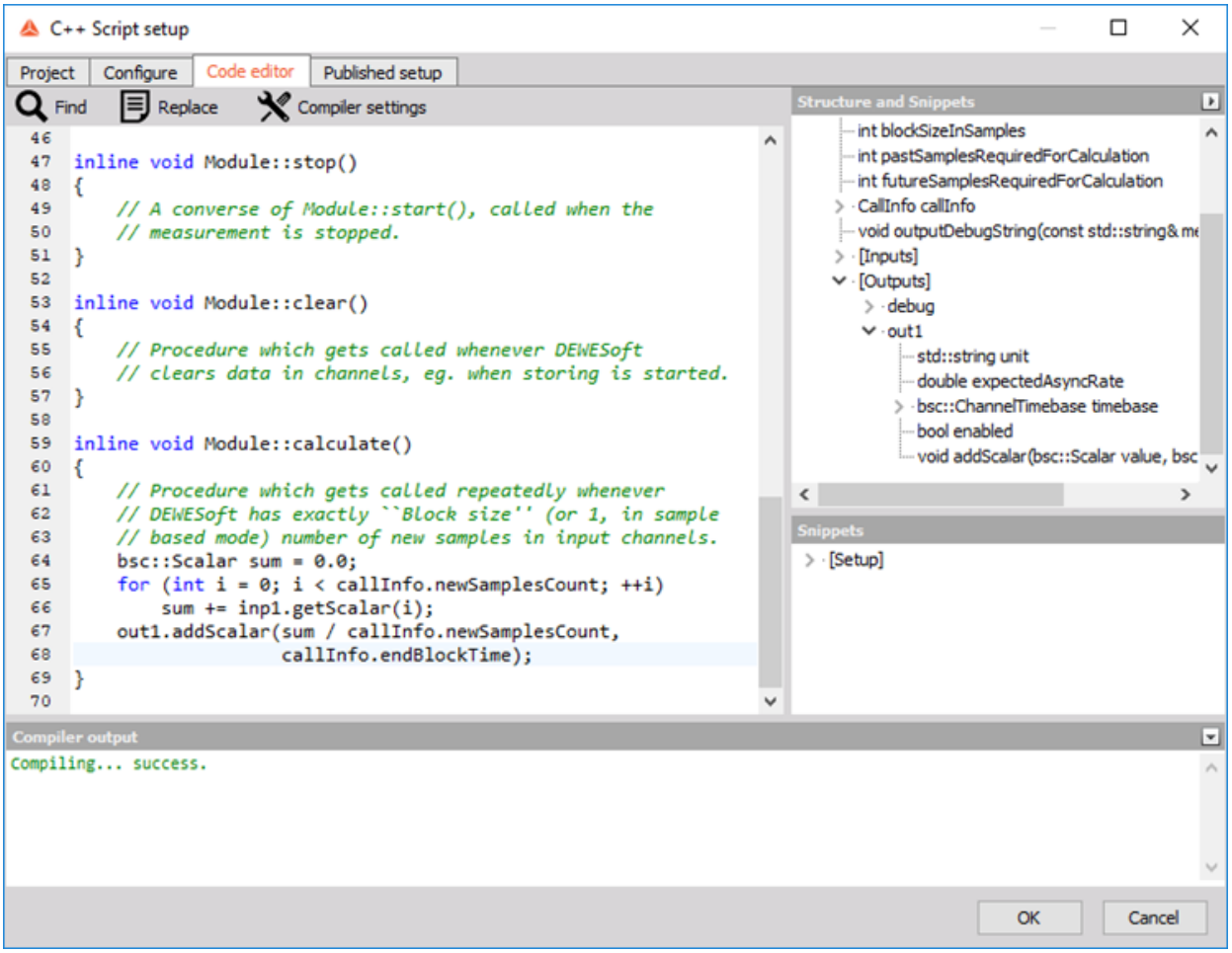
4.4. Step Three: Code editor tab

In the **Code editor** tab you can see the main code of your module. Ignoring most of it scroll to the very bottom to find `inline void Module::calculate()` function. Fill it in with the following code:

```
inline void Module::calculate()
{
    bsc::Scalar sum = 0.0;
    for (int i = 0; i < callInfo.newSamplesCount; ++i)
        sum += inp1.getScalar(i);
    out1.addScalar(sum / callInfo.newSamplesCount,
                   callInfo.endBlockTime);
}
```

The code should be pretty self-explanatory, but to write it out in english, for each Dewesoft call to `calculate` we sum all the scalar values in the block of samples from input channel *inp1*, and write their average to output channel out1. Since we defined our output channel as asynchronous, we also have to specify the time of our output sample. For this we simply use the time of the last sample in block, which

is automatically set for us in the `callInfo.endBlockTime` variable. Note that inp1 and out1 correspond to the default **C++ variable name** values of input and output channels from **Configure** tab respectively.



Code editor tab

Hint

Click on the **Structure view** 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`, `inp1` and `out1` with their respective methods for reading and writing from the channels, as well as some other variables and methods we didn't mention. Note that elements in the tree view can be double-clicked to populate them in the code editor.

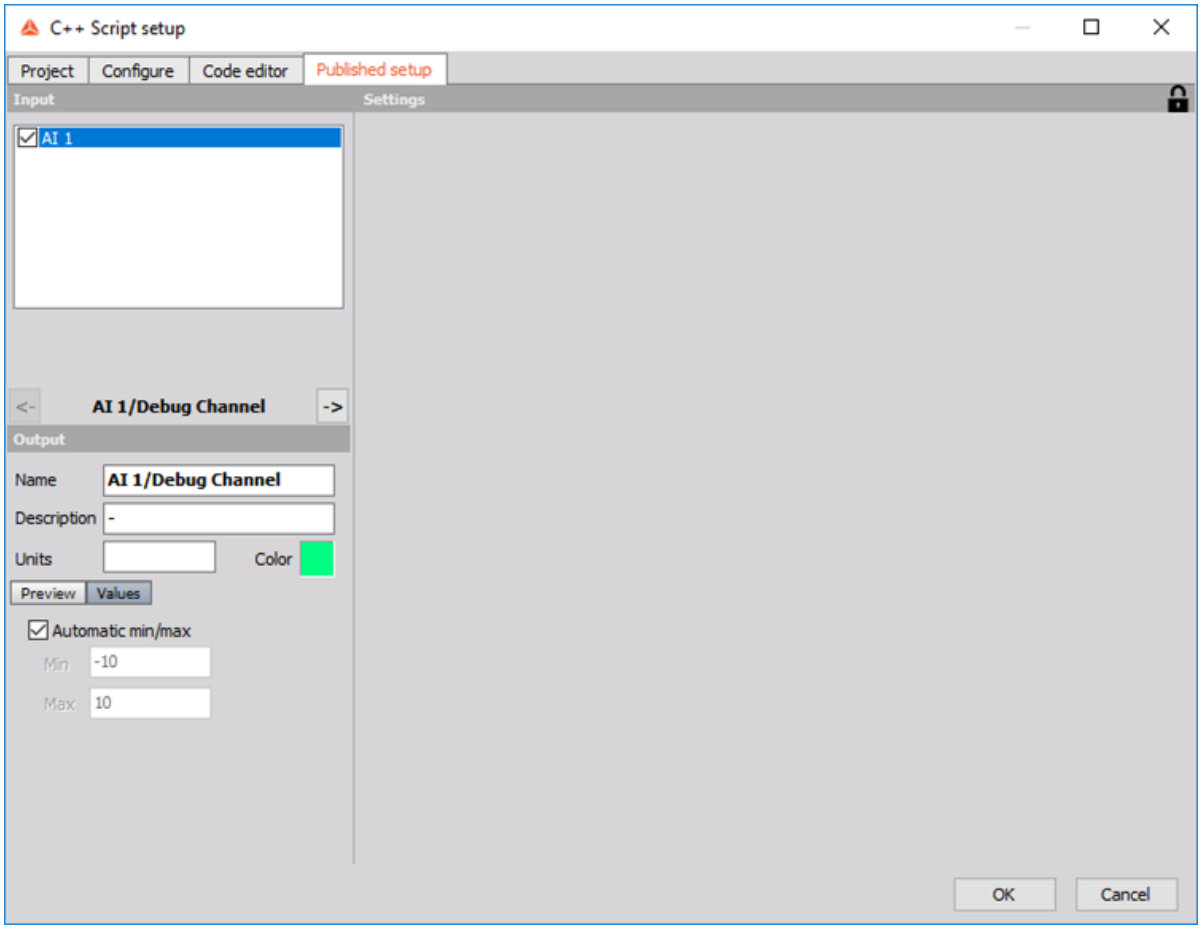
With that we can move on to the last tab.

4.5. Step Four: Published tab

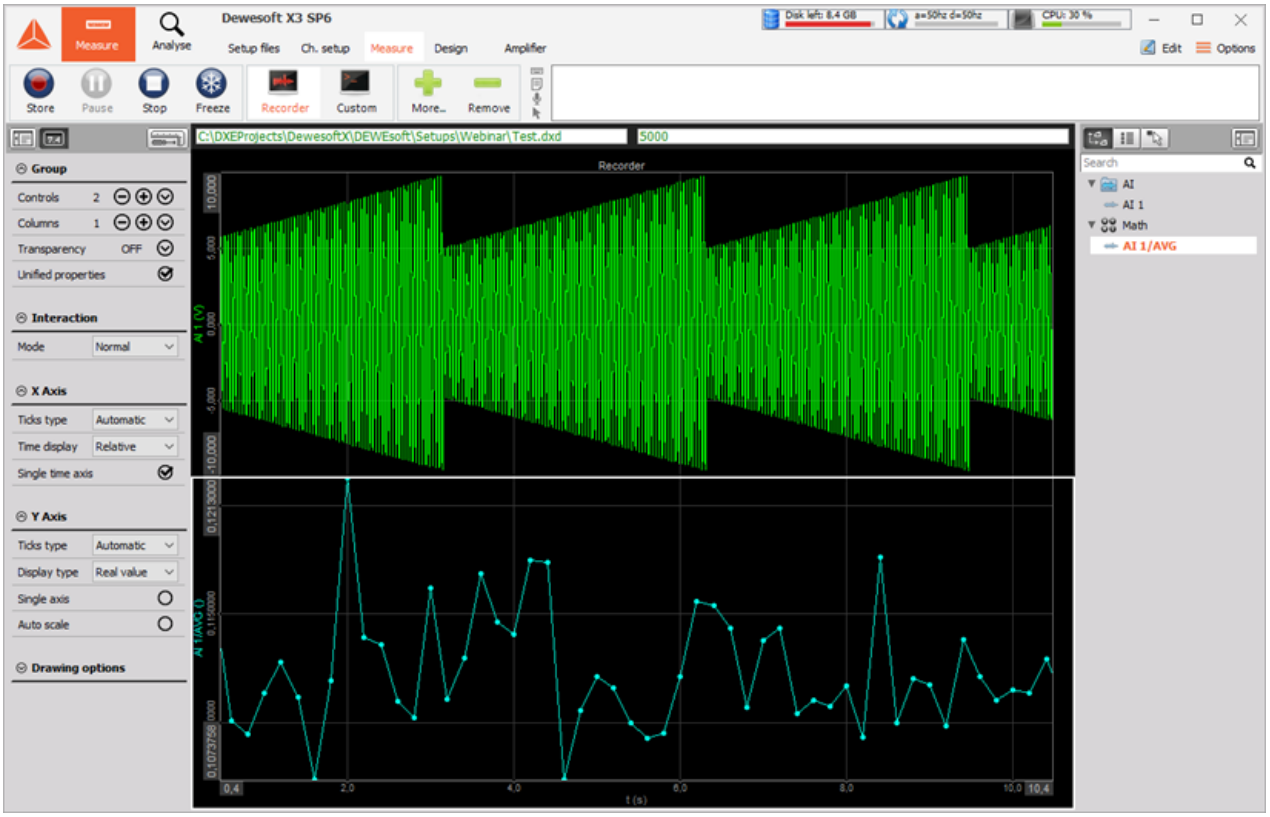
The user interface in the **Published** tab should look relatively familiar, as it tries to mimic other math modules found throughout Dewesoft. It should also make it a bit more clear what you were doing in the **Configure** tab: on the left hand side you have a list of synchronous input channels, which was the type of our input channel. Ticking the checkbox next to any of the channels will create a Math object with a

single asynchronous output channel (in addition to the default 'debug' channel), which corresponds to our output channel *AVG*.

To test your new module, tick one (or more) channels in the **Input** list and click the **OK** button at the very bottom of the setup form. Initiate Dewesoft's measurement mode by clicking on **Measure** tab and observe your channel with averaged input signal.



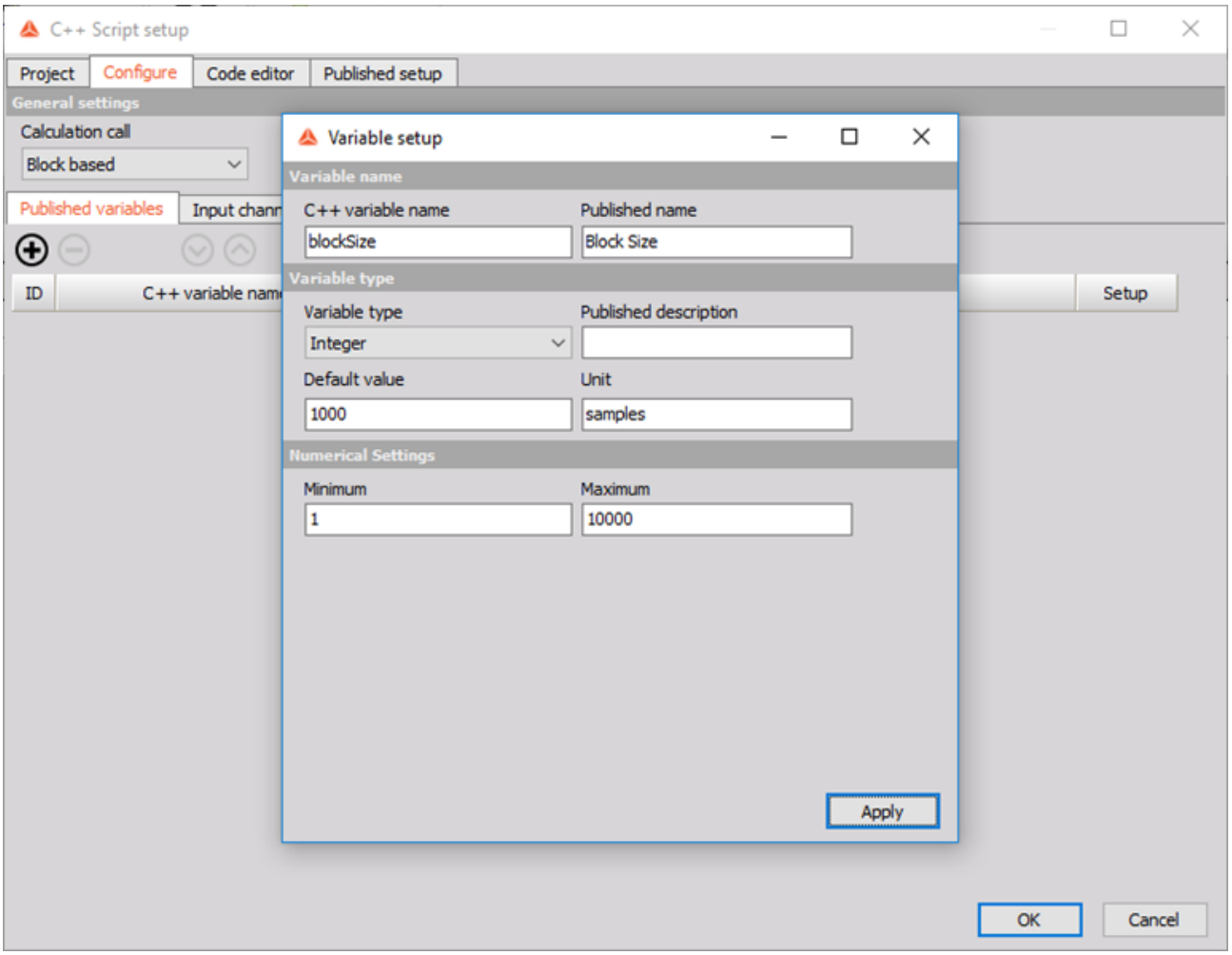
Final settings



Measurement

5. Small addition to our averaging example

We have now created a module performing averaging of an input signal, but it has some real deficiencies: the most glaring one being that the size of block for averaging is fixed at 1000 samples. Let's refactor this into a variable on the **Published** tab meaning the user will be able to simply open our module up and set it to whichever value he desires.



Adding a published variable for controlling the block size

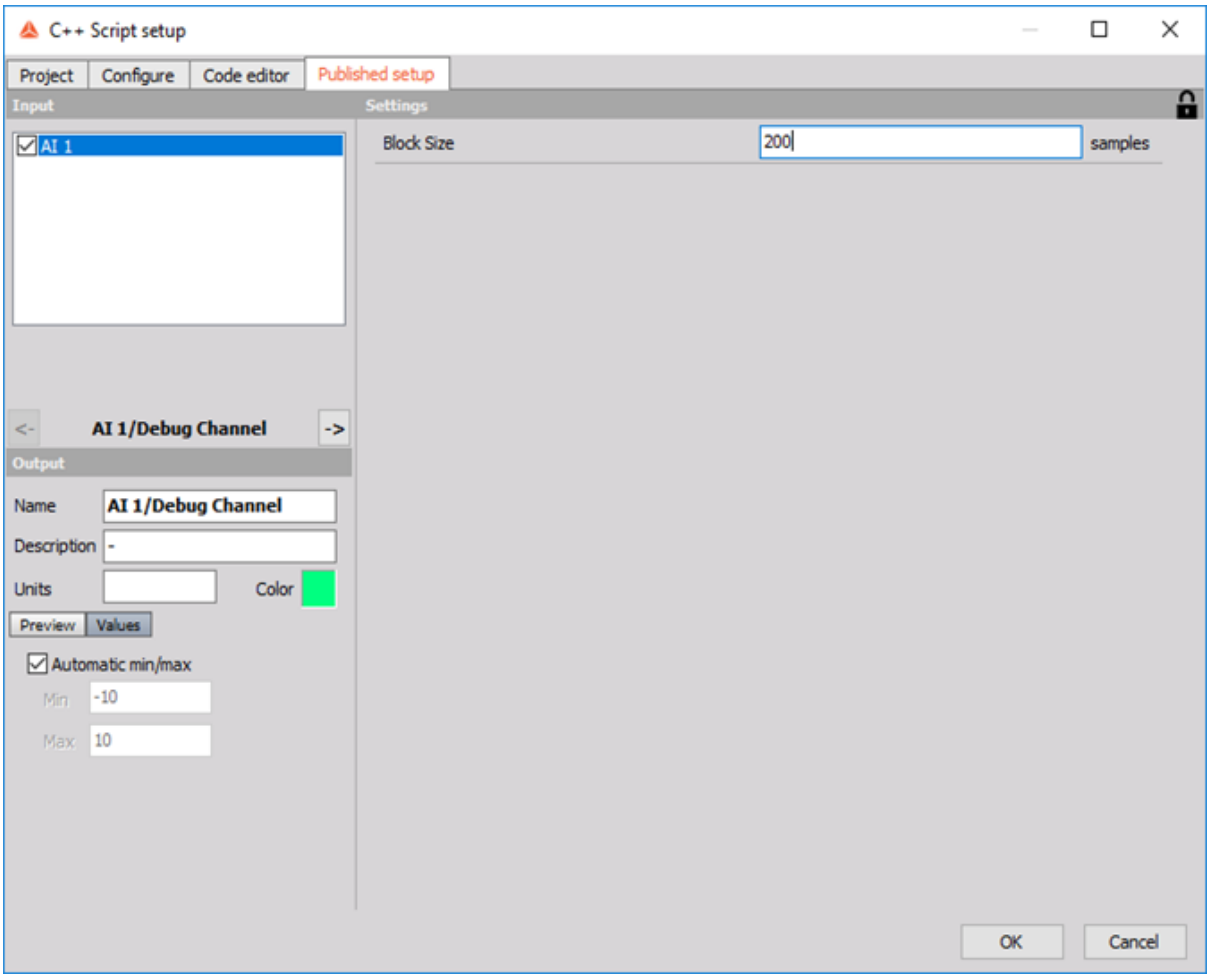
To do that, navigate to Configure tab, go to Published variables tab, and click on **(+)**. In the popup form set **Variable type** to *Integer*, **C++ variable name** to "*blockSize*", **Published name** to "*Block Size*", **Default value** to "*1000*" and **Unit** to "*Samples*". Under **Numerical Settings** set **Minimum** to "*1*" and **Maximum** to "*10000*". Click **Apply** and move to **Code editor** tab, where you need to change the inline void Module::configure() function to look like this:

```
inline void Module::configure()
{
    blockSizeInSamples = published.blockSize;
    out1.expectedAsyncRate = bsc::core.acqSampleRate / blockSizeInSamples;
```

```
}
```

With these changes the block size is going to be dynamically determined by the user, and the expected async rate of our output channel will be automatically set correctly regardless of the Dewesoft's acquisition sample rate. Note that, since we used callInfo.newSamplesCount in the for loop inside Module::calculate(), we do not need to touch the rest of the code for it to work as we want it to.

Move on to the **Published** tab where you will now be able to find a new field under **Settings**, containing our **Block Size** as defined in **Configure** tab and try playing around with different values!



Final settings

6. C++ Script Features

The tutorial hopefully demonstrated the most basic features of the C++ Script. In the rest of the manual we present all the features of the C++ Script in detail.

6.1. Pre-packaged bundles

A good way to learn about C++ Script is to study pre-made examples, so we have prepared a selection of so-called bundles which you can download from the developer downloads section of our website, https://www.dewesoft.com/developers. To see the bundles in the **Project** tab of the C++ Script's setup form (under **Bundles** panel), follow the README instructions inside the zip file.

Clicking on any of these bundles in the **Bundles** panel will populate the **Selected bundle details** area with the details about the selected item, which can give you a better idea about what the bundle contains. Double-clicking any item irreversibly clears your current C++ Script and populates it with bundle's contents for you to explore and experiment with.

There is nothing special about these bundles -- in fact, they are bundles created with C++ Script itself by clicking on the **Export bundle**... button from the **Bundle** option in the main menu. C++ Script's Bundles explorer searches for all the bundle files found within Dewesoft's Scripts\Cpp\ directory, which is also the default export directory. You can also import a bundle located in some different directory by clicking on the **Import bundle.**.. menu option and locating it manually.

6.2. Exporting C++ Scripts as stand-alone math modules

The C++ Scipts can be exported as a standalone bundles. The bundle will contain the settings necessary to create the setup forms for the end user as well as precompiled DLL binaries, meaning end-user of the module will not need to have DSMinGW installed on their machine to use the script.

Since bundles are self-contained, you can also put them into Dewesoft's bin\addons\ folder, and they will appear in Dewesoft as regular, built in math modules under **Custom C++ Scripts**. In this case, the first line of the **Description** field in **Project** tab will appear as the description when the user hovers over the module in **Add math** menu.

By clicking on **Export bundle**... button C++ Script will create a self-contained bundle. The **Export bundle settings** has an option to *Export as upgrade of existing bundle* with the ability to search for an existing bundle and an option to write a new bundle version. When exporting a bundle without *Export as upgrade of existing* enabled the exported bundle is always treated as a completely new plugin and existing setups, that contain the previous version of this plugin, will not properly work with the newly exported plugin.

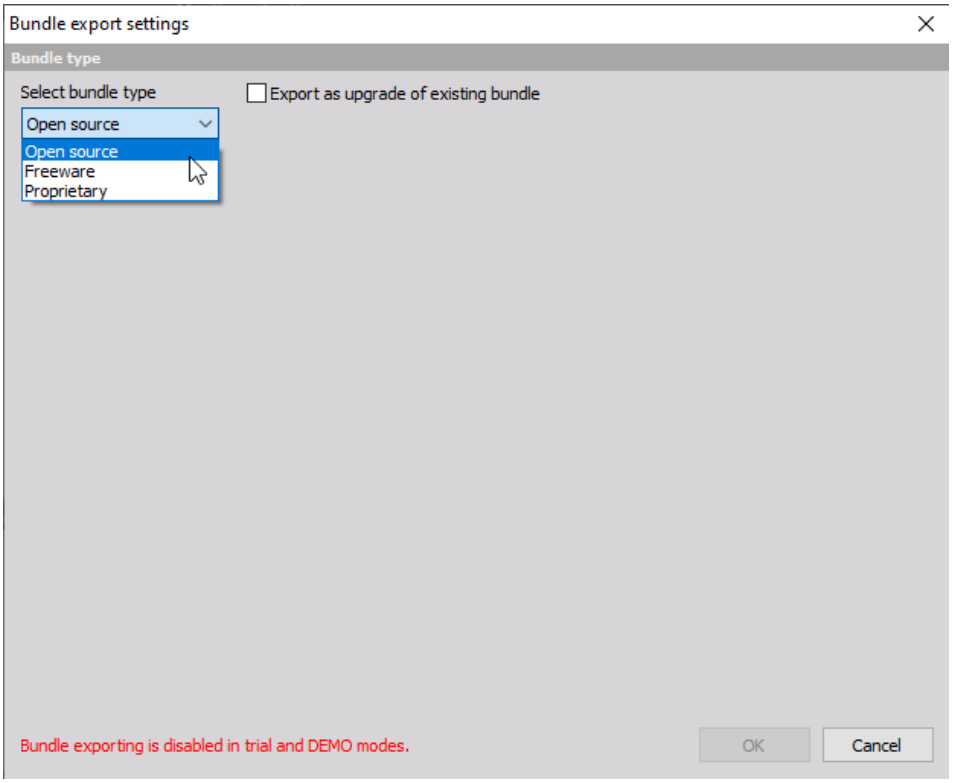
Dialog also lets you specify one of three ways to export your bundle:

- *Open source*: bundle is exported with the source code included. When such a bundle is imported via **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 **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.

- *Proprietary*: enables the locking mechanism and expands the form to enable adding multiple serial numbers. With a new grid there is an option to add specific serial numbers of devices to lock the bundle. On the left side there is a list box which is populated with serial numbers of currently connected devices. There are two locking mechanisms: *Automatic* and *Manual from code*.
    - The *Automatic* locking: works the same like it does for other plugins and maths - this means no additional work for the programmer. During the global licensing check there is an additional step to check for the existence of valid serial numbers.
    - The *Manual from code* locking: enables the bundle to be visible regardless of the HW connected to the Dewesoft, but can be controlled from the code with *bsc::core.isRunningOnLicensedHardware* boolean.

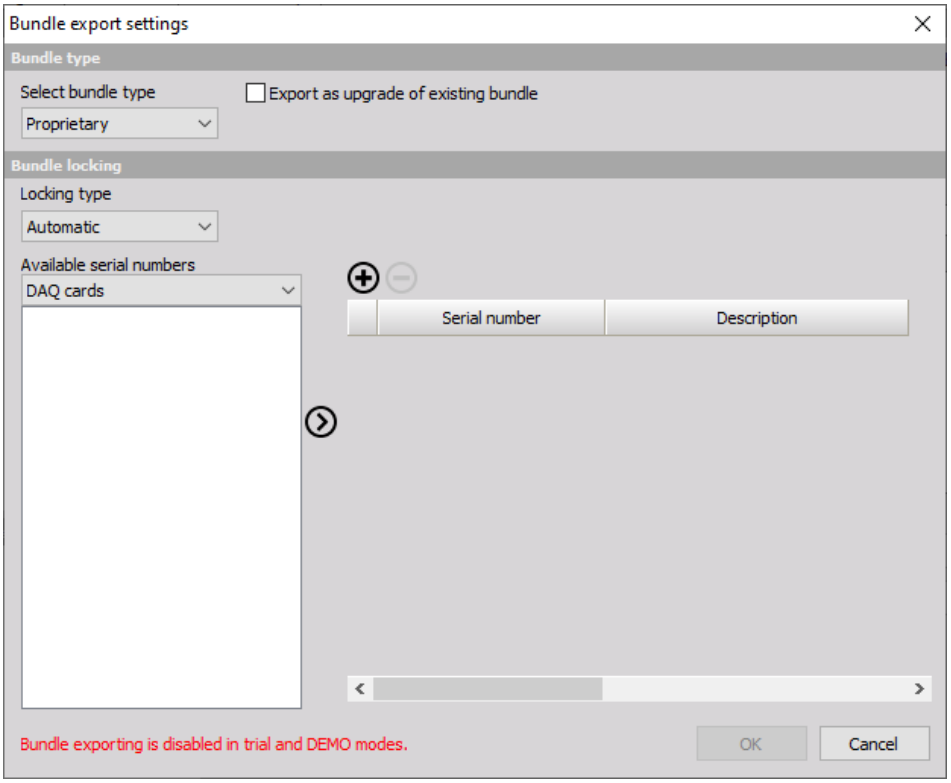
Hint

The Motherboard serial number differs from the Serial number of the device and the Motherboard serial number should not be used for locking a bundle.



Export bundle settings dialog

Exporting a bundle as an upgrade is also useful when serial numbers, that a bundle should be locked to, are not known in advance. When upgrading an existing bundle additional serial numbers can be added and the old *.cbu* bundle file, in the addons folder, can be replaced with the newly exported one and all previously created setups should work.

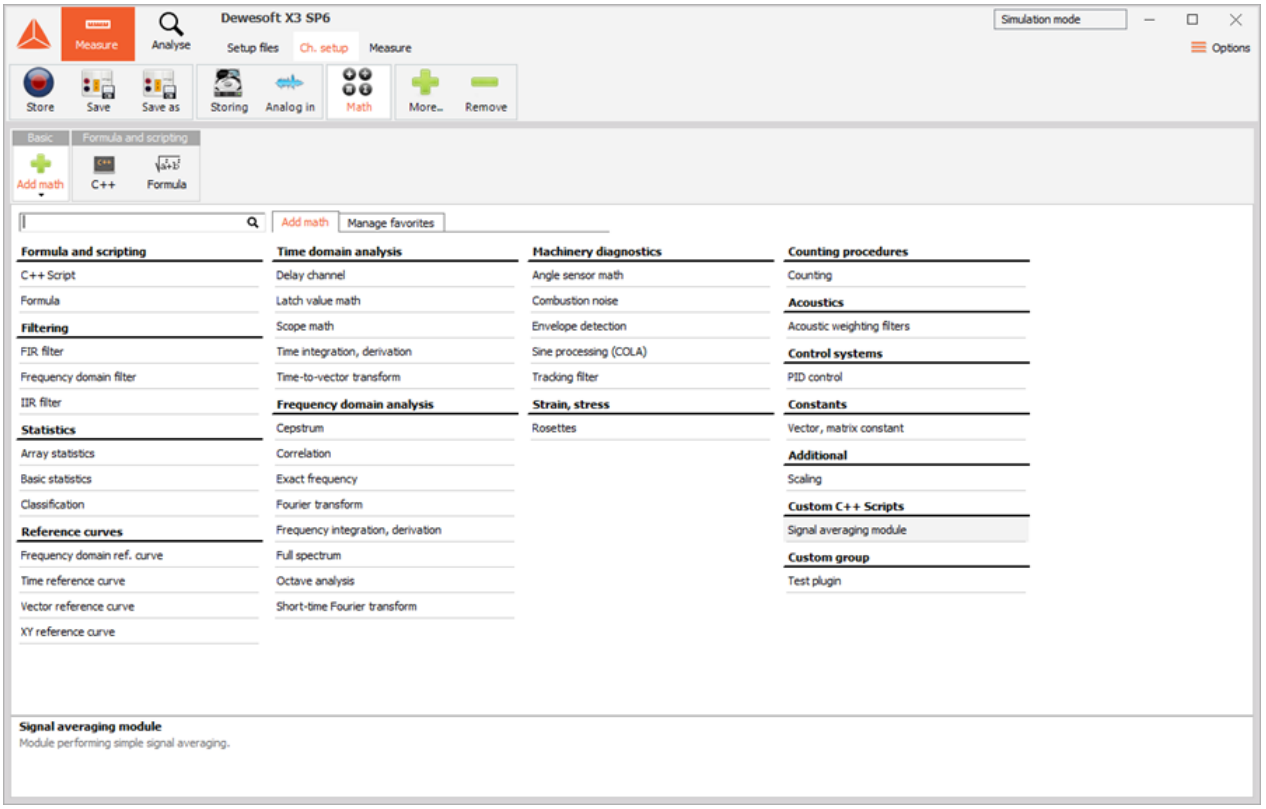


Proprietary bundle type with bundle locking options

To lock a bundle from within the code use the code:

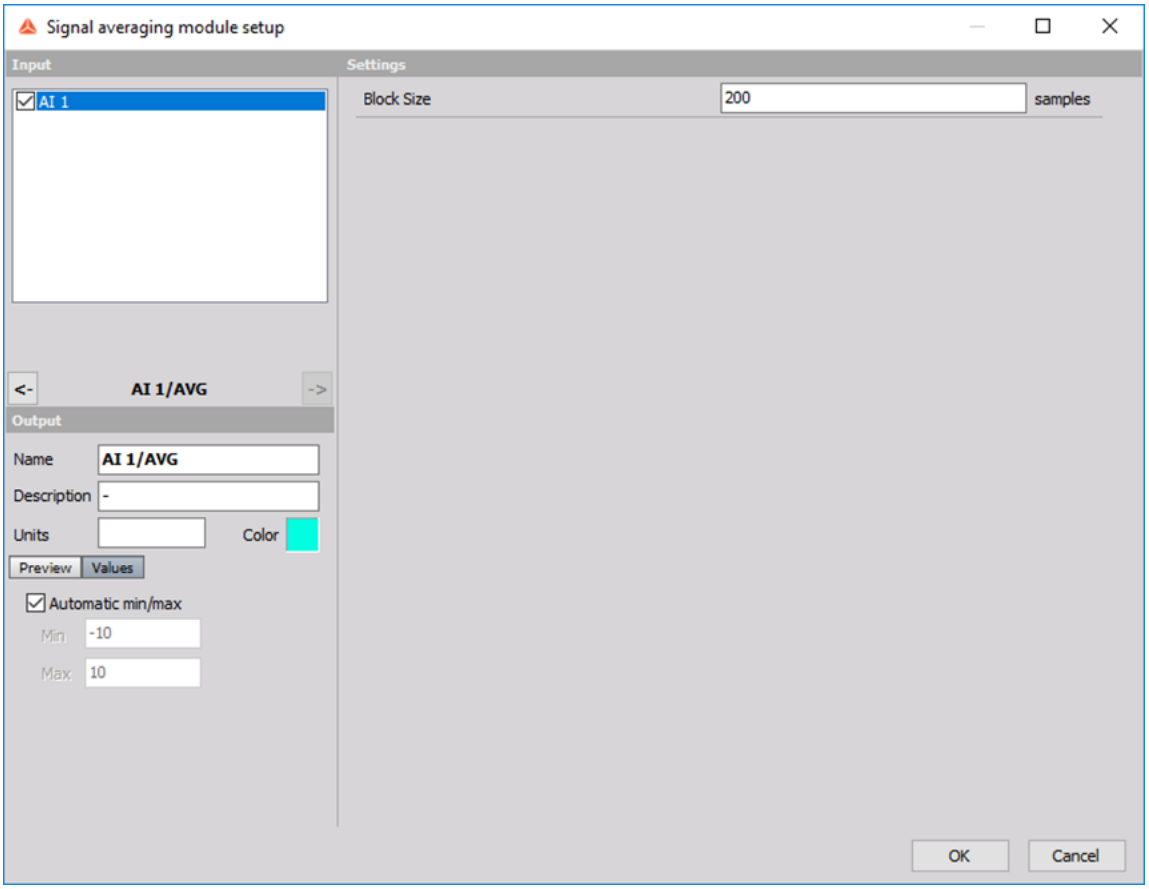
```
inline void Module::calculate()
{
    // Procedure which gets called repeatedly whenever DEWESoft has exactly "Block size"
    // (or 1, in sample based mode) number of new samples in input channels.
    if (!bsc::core.isRunningOnLicensedHardware && callInfo.startBlockTime > 10.0)
        return 0;
    // custom CPP code
}
```

With the above code the custom CPP code will only be executed when connected to the devices with supported serial number, and will not be executed after 10 seconds otherwise.



Exported script from start of manual

Users of such bundles will always only have access to **Published** tab, meaning they will be able to change only the settings creator of the script decided to expose on that tab, regardless of whether the bundle was exported as *Open source or Freeware*.



Signal averaging module as stand-alone math

6.3. Dewesoft::Math::Api::Basic namespace

All C++ Script's types and some of its structures reside in the `Dewesoft::Math::Api::Basic` namespace. For brevity reasons this manual uses bsc as the alias for this namespace, and, as typing out the entire namespace every time you need to access it can be a chore, the very top of the template in a fresh C++ Script module contains the following line:

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

6.4. Published variables

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

Creating a variable by clicking on the **(+)** button will bring up a form asking you to specify the type of the variable and some of its basic features. The field C++ **variable name** specifies the name with which you will be able to access the variable from within C++ code, where the variable will be stored inside a special published structure. The **Published name** field specifies what the variable will be named in the

Published tab. Other fields are mostly dependent on the type of variable you select, but should be relatively self-explanatory.

Inside the C++ code these variables are at all times read-only and are set before any of the Module methods are called by Dewesoft. The following table shows the connection between selected variable types and the types of created variables in C++ code:

Variable type	C++ counterpart
Boolean	`bool`
Integer	`int`
Float	`double`
Enumeration	`enum`
String	`std::string`
File name	`std::string`

When it comes to *File name* variables, there is a special rule: the end-user can on **Published** tab specify the path to file either as:

- absolute, in which case the absolute path is always used; or
- relative, in which case C++ Script will transform the path to absolute by:
  - prepending it with the location of current setup file if the setup is saved; or
  - prepending it with the location of Dewesoft's setup folder, as defined in Options > Settings > Files and folders > Default folder for setup files.

6.5. Core variables

`bsc::core` structure contains 2 read-only variables:

- `double bsc::core.acqSampleRate`: acquisition sample rate Dewesoft is currently running with. This is set to the correct value before the call to `Module::configure()`; and
- `double bsc::core.absoluteTimeAtStart`: POSIX time at the start of measurement/storing. Integral part of the number represents the number of seconds elapsed since 00:00:00 Thursday, 1 January 1970 in UTC time, and the fractional part represents the number of milliseconds. This variable is only set to the correct value after the call to `Module::start()`. Note that Dewesoft uses a different time format, as described on this link: http://docwiki.embarcadero.com/Libraries/Rio/en/System.TDateTime. To convert from POSIX time to Dewesoft time you will need to use the following equality:

  $$Dewesoft = POSIX / (24 * 60 * 60) + 25569,$$

  where 24*60*60 is the number of seconds in a day, and 25569 is the difference in seconds between the starts of POSIX and Dewesoft times.

6.6. Module variables

Module itself contains 4 variables (alongside callInfo structure):

- `int moduleIndex`: the index of current module, where first module has `moduleIndex = 0`, second `moduleIndex = 1`, and so on;
- `int pastSamplesRequiredForCalculation`: number of additional samples from past passed to input channels;
- `int futureSamplesRequiredForCalculation`: number of additional samples from future passed to input channels.
- `int blockSizeInSamples`: number of new samples in input channels for each call to `Module::calculate()`.

For further explanation of `pastSamplesRequiredForCalculation` and `futureSamplesRequiredForCalculation` variables refer to section Channel delays.

6.7. Module's callInfo structure

Module's callInfo structure contains 4 read-only variables, which get updated before each call to `Module::calculate()` with the latest values:

- `int64_t bsc::core.sampleCountSinceStart`: number of samples acquired since start of measurement;
- `int callInfo.newSamplesCount`: number of new samples in input channels;
- `bsc::Time callInfo.startBlockTime`: time in seconds of the first new sample in input channels;
- `bsc::Time callInfo.endBlockTime`: time in seconds of the last new sample in input channels.

6.8. Channel types

C++ Script supports various different channel types found throughout Dewesoft. The channels can be split in 3 different ways: according to channel time base (single value, synchronous, asynchronous), channel value type (scalar, vector, matrix), and channel type itself (input, output).

Note that synchronous channels with sample rate divider other than 1 are currently unsupported in all forms.

6.9. Input channels

For input channels you can retrieve the value of i-th sample in the block by using `getScalar(i)` (or, `getVector(i)` for vector value type channels, or `getMatrix(i)` for matrix value type channels). Similarly you can retrieve the time stamp of the i-th sample by using channel's `getTime(i)` member function. If the **calculation type** is *sample based* parameter i can be omitted, as there is always exactly one sample in the input channel.

In case you set `pastSamplesRequiredForCalculation` (refer to section void Module::configure()), input channels will also contain exactly that many samples from past, which you can access by using negative indices. Similarly for `futureSamplesRequiredForCalculation` the input channels will contain this many additional samples from future, which can be accessed with indices i ≥ `callInfo.newSamplesCount`. Refer to section Channel delays for further explanation.

In the background, `bsc::Scalar type` is simply an alias for type double, while `bsc::Vector` and `bsc::Matrix` types are custom types as described in sections bsc::Vector type and bsc::Matrix type.

For channels with vector and matrix value type you can access their dimensions via channel's axes member; i.e. to access the vector dimensions of a channel you can use `axes[0].size()`, and for matrix `axes[d].size()`, d ∈ {0,1} for d-th dimension. Aside from the dimension size, the axes structure's elements also contains individual axis `name`, `unit`, and `values`.

6.10. Output channels

To add samples to output channels you can use the channel's `addScalar(const Scalar& value, bsc::Time time)` function (or, `addVector(const Vector& value, bsc::Time time)` for vector value type channels, or `addMatrix(const Matrix& value, bsc::Time time)` for matrix value type channels). Note that if the timebase of your output channel is synchronous, you are expected to output exactly `callInfo.newSamplesCount` number of samples per function call. Additionally, for synchronous and single value timebase channels the time argument in the function call can be omitted, as it is implied.

Important: output channels with synchronous or asynchronous timebase need to have samples added in strictly ascending order by time.

When adding an output channel with asynchronous timebase you will be asked to specify *expected async rate per second value*. This should be an approximate (within an order of magnitude) rate at which you will be adding samples to the channel. Specifying too big of a number means Dewesoft will allocate too much memory, but specifying too small of a number means the samples will get lost. Since this setting will likely depend on Dewesoft's sample rate, you can also set channel's expected async rate per second at runtime by modifying channel's `expectedAsyncRate` variable in `Module::configure()` function. Refer to void Module::configure() for more details.

For channels with vector and matrix value types can also change the dimensions by modifying axes structure in the `Module::configure()` function call. Again, refer to void Module::configure() for more details.

Note that any *inf* and *nan* values in the output channels do not trigger an exception, but instead get automatically changed to 0.

6.11. Custom C++ Script types

This section contains description of types defined and used throughout Dewesoft C++ Script's c++ code.

6.11.1. bsc::Time type

`bsc::Time` type is an alias for type double, used to denote time in seconds.

6.11.2. bsc::Scalar type

`bsc::Scalar` type is an alias for type double, while `bsc::ComplexScalar` is an alias for type `std::complex<double>`.

6.11.3. bsc::Vector type

Input and output channels with vector value type operate via `bsc::Vector` (or, `bsc::ComplexVector` for complex values) types. The following are `bsc::Vector's` operations:

- Construct new vector `V` with `V(size)` OR `V = initializer_list`

```
// Example:
bsc::Vector V = {1, 2, 3}; // vector of 3 elements
bsc::Vector V(3); // vector with 3 zeros
```

- Copy vector `V` into vector `V2`

```
// Example:
bsc::Vector V2 = V;
bsc::Vector V2(V);
```

- Access vector `V`'s elements with `V[i]` (zero-based indexing)

```
// Example:
V[1] = 10.0; // set second element in vector V to value 10.0
bsc::Scalar u = V[3]; // set u to value of V's fourth element
```

- Retrieve number of elements of vector `V` with `V.size()`

```
// Example:
bsc::Vector V(3);
int i = V.size(); // i will become 3 as V has 3 elements
```

Same operations are available for `bsc::ComplexVector` (simply replace `bsc::Vector` and `bsc::Scalar` with `bsc::ComplexVector` and `bsc::ComplexScalar` respectively in the list above).

Expert note: behind the scenes, vector V is a wrapper for an 1-d array with `M.size()` elements (where element is either of type double in case of real vectors, or `std::complex<double>` in case of complex vectors). You can access the underlying array directly via `M.data()` method, which returns the pointer to the first element of the array.

6.11.4. bsc::Matrix type

Input and output channels with matrix value type operate via `bsc::Matrix` (or, `bsc::ComplexMatrix` for complex values) types. The following are `bsc::Matrix's` operations:

- Construct new matrix `M` with `M(rows, columns)` OR `M = initializer_list`

```
// Example:
bsc::Matrix M = {{1, 2, 3}, // matrix of 2 rows by 3 columns
                 {4, 5, 6}}; // with specified initial elements
bsc::Matrix M(2, 3); // 2 by 3 matrix of zeros
```

- Copy matrix `M` into matrix `M2`

```
// Example:
bsc::Matrix M2 = M;
bsc::Matrix M2(M);
```

- Access matrix `M`'s elements with `M[row][column]` (zero-based indexing)

```
// Example:
M[0][1] = 10.0; // set element in row 0 column 1 to value 10.0
bsc::Scalar v = M[3][2]; // set v to value in row 3 column 2
```

- Retrieve number of rows of matrix `M` with `M.m()`

```
// Example:
bsc::Matrix M(2, 3);
int i = M.m(); // i will become 2 as M has 2 rows
```

- Retrieve number of columns of matrix `M` with `M.n()`

```
// Example:
bsc::Matrix M(2, 3);
int j = M.n(); // j will become 3 as M has 3 columns
```

Same operations are available for `bsc::ComplexMatrix` (simply replace `bsc::Matrix` and `bsc::Scalar` with `bsc::ComplexMatrix` and `bsc::ComplexScalar` respectively in the list above).

Expert note 1: behind the scenes, matrix M is stored as an 1-d array with `M.m()` · `M.n()` elements, in column-major order (where element is either of type double in case of real matrices, or `std::complex<double>` in case of complex matrices). You can access the underlying array directly via `M.data()` method, which returns the pointer to the first element of the array, and `M.size()` method to retrieve its length (which always equals `M.m()` · `M.n()`).

Expert note 2: for faster access to elements you can use `M(row, column)` (for which `M[row][column]` is just syntactic sugar), which might be faster as it avoids generating intermediate object created by `M[row]`. For consistency sake a similar syntactic sugar (with no performance gain) is available for `bsc::Vector`, with `V(i)` being equivalent to `V[i]`.

6.11.5. Module::calculate()'s sample rate

The timebases of input and output channels of each individual Module will determine the rate at which `Module::calculate()` function gets called. Here are the rules:

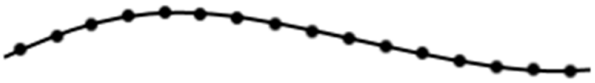
- if any one of output channels is a synchronous channel, you can not specify vector or matrix input channels, and every input channel will be resampled to synchronous rate.
- if none of the output channels are synchronous, the first input channel will be taken as master channel and every other input channel will be resampled to its time base:
    - if the first input channel is synchronous, all the input channels shall be resampled to a synchronous timebase.
    - if the first input channel has asynchronous timebase, all the input channels shall be resampled to its timebase.
    - if the first input channel has a single value timebase, `Module::calculate()` function will be called every time its value could potentially be changed. In this case the calculation ignores **calculation type** as there is only ever going to be exactly 1 new sample in input channels.

Wherever we say 'resampled' in the list above we mean linear resampling if the interpolate channel values check box is ticked in channel's setup form, otherwise Dewesoft will take the closest last available value from the channel.

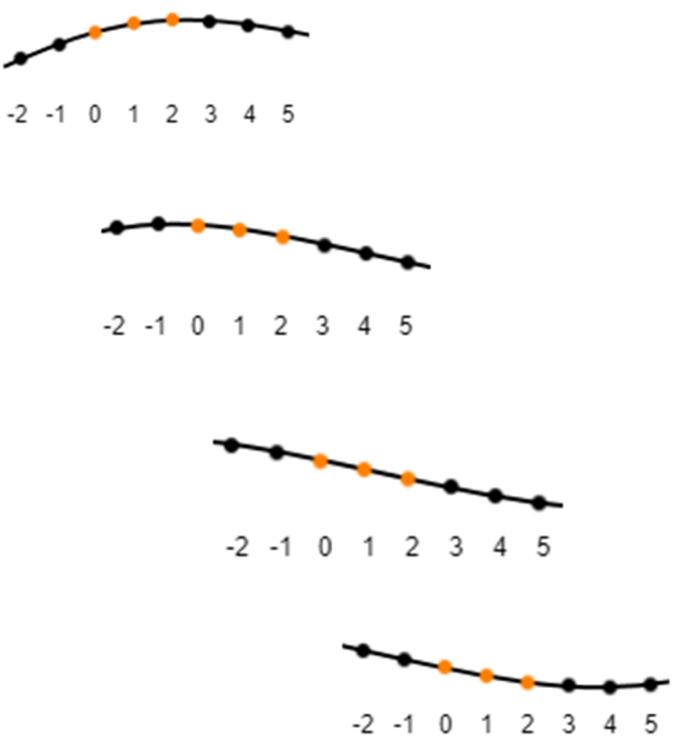
6.12. Channel delays

In case your calculation requires past and future samples from channel (e.g. say you are implementing FIR filter or custom resamplers), you can set `pastSamplesRequiredForCalculation` and `futureSamplesRequiredForCalculation` inside `Module::configure()` to number of samples required on each end. With this setting your input channels then contain this many additional samples per call to `Module::calculate()`.

To better explain this let's consider an example where we are using Block based calculation type with block size equal to 3 and we set `pastSamplesRequiredForCalculation` to 2 and `futureSamplesRequiredForCalculation` to 3 inside `Module::configure()` function. Let's pretend that the following figure shows our input signal (with x axis representing time and black dots marking the individual samples):



Then, the following 4 figures show which samples are set in input channel for 4 successive calls of `Module::calculate()`:



The numbers below the samples show valid i parameters for querying samples from the input channel (`via get[value type](i) and getTime(i)` functions) and orange dots represent samples which would get set regardless of the `pastSamplesRequiredForCalculation` and `futureSamplesRequiredForCalculation` setting. This means that valid i for retrieving samples lies on the closed interval [*-pastSamplesRequiredForCalculation, callInfo.newSamplesCount-1+futureSamplesRequiredForCalculation*].

You can see that despite the fact that we are processing 8 samples per call to `Module::calculate()`, we are still expected to only output 3 samples in case our output channel is synchronous (at timestamps of orange samples), and that channel only advances by block size number of samples at once, so some samples are sent to the function multiple times.

Note that you can think of *Sample based* calculation type to be just a special case of Block based calculation type with block size equal to 1.

6.12.1. Module class' methods

Your C++ Script interacts with Dewesoft by letting it invoke certain methods of your implemented Module's code (all of which come pre-populated in the C++ Script's code editor by default). What follows is a brief description of each of the methods.

6.12.2. Module::Module()

`Module::Module()` function gets called exactly once whenever your module is created. It is guaranteed that variables in the published structure are set to their correct values before the module is created, but other variables (eg. variables in core structure, `blockSizeInSamples`, channel properties, ...) are not. As such this procedure is perfect for starting any slower jobs, reading data from files, etc.

6.12.3. Module::~Module()

For each `Module::Module()` function call there is a guaranteed corresponding call to `Module::~Module()` whenever your module is destroyed. This happens for example when Dewesoft is closed, when another setup is loaded, or when your module gets removed. This gives you an opportunity to clean up anything done in `Module::Module()`, for example releasing locks on files, closing libraries, etc.

6.12.4. void Module::configure()

Inside `Module::configure()` function you can modify properties of your output channels and manually set calculation type/block size and delays. Note that after the `Module::configure()` call you should not further modify these values. Also note that `Module::configure()` may get called multiple times before the start of measurement.

The following is a list of things Dewesoft looks at after `Module::configure()` call is over:

- asynchronous output channels' `expectedAsyncRate`: if your module contains asynchronous output channels, you can set their expected rate per second by modifying their `expectedAsyncRate` member. This can be useful if you want to make asynchronous channels work regardless of Dewesoft's sample rate, for example by setting the value to `bsc::core.acqSampleRate / blockSizeInSamples` if you plan to output one sample per block of samples. Another useful case can be if you know the rate of your output channel is somehow going to be connected to the rate of some other input channel I, in which case you can simply set the e`xpectedAsyncRate` to `I.expectedAsyncRate`.

- output channels' enabled: for each output channel you can decide to disable it and hide it from the end user. This can be useful if you want to disable channels based on some user-selected option in the *Published* tab.
- output channels' unit: for each output channel you can set its unit, since it might depend on input channels' units (which you can access via their own unit field).
- vector/matrix output channels' `axes`: if your module contains vector or matrix output channels, you can also change their dimensions and axis information here. You can do that by modifying the channel's `axis` data, more specifically setting the `axis[d]`'s name (of type `std::string`), unit (of type `std::string`), and values (of type `bsc::Vector`) fields, where d ∈ {0} if channel is a vector channel, and d ∈ {0,1} if channel is a matrix channel. Before the call to `Module::configure()` the values vector gets initialized to {1, 2, ..., n_d} where n_d is the dimension size you specified in the *Variable setup* tab. The size of output channel's d-th dimension at the end of `Module::configure()` is equal to the number of elements in `axes[d].values` vector.
- `blockSizeInSamples`: if you want to manually change calculation type/block size, setting this variable to 1 is equivalent to setting *calculation type* to *Sample based*, and any number greater than 1 to Block based with Block Size value equal to it.
- `pastSamplesRequiredForCalculation`: refer to section Channel delays.
- `futureSamplesRequiredForCalculation`: refer to section Channel delays.

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.

6.12.5. void Module::start()

`Module::start()` function gets called at the very start of measurement, and is guaranteed to be called after `Module::Module()` and `void Module::configure()`. It should be used for lightweight tasks such as initializing your variables.

6.12.6. void Module::stop()

For each `Module::start()` function call there is a guaranteed corresponding call to `Module::stop()` at the end of measurement, giving you opportunity to clean up; for example to free memory allocated in the `Module::start()` call.

6.12.7. void Module::clear()

`Module::clear()` function gets called whenever Dewesoft clears the data from its channels. This occurs when you start storing the data and might be useful eg. for resetting intermediate values from your calculations.

6.12.8. void Module::calculate()

`Module::calculate()` function gets called repeatedly during the measurement mode. Before each call, the Dewesoft populates input channels with *pastSamplesRequiredForCalculation+callInfo.newSamplesCount+futureSamplesRequiredForCalculation* number of samples. This is where you can obtain the samples from input channels (refer to section

Input channels), process them, and depending on the sample rate (refer to section Module::calculate()'s sample rate) and output channel types (refer to section Channel types) add samples to output channels.

6.13. 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. On top of this the debug channel will also 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 (refer to section Output channels) either in channel's setup or by setting debug.expectedAsyncRate to appropriate value in `Module::configure()` in your c++ code.

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

6.14. Output channel names

By default, C++ Script prepends output channel names with the names of input channels the output channels depend on. This might sometimes prove to be undesirable, as the names of such channels can get really long, especially if the script has multiple input channels; e.g.: 'input 1, input 2, input 3/output 1'. For such reason the checkbox *Do not prefix output channel names with corresponding input channel names found* in **Extra** tab under **Configure** can be toggled to turn off prepending of input channel names.

6.15. Compiler settings in Code editor

While C++ Script is primarily intended to be used for light scripting, sometimes we would want to use functionality implemented by external libraries or files. For this reason C++ Script exposes the compiler flags, with which you can control which libraries or other source files are included during the compilation of C++ Script.

You can access compiler settings via **Code editor** tab, by clicking the **Compiler settings** button in the editor toolbar. This opens a form where user can specify the following compiler options (in parenthesis are the compiler flags used for g++ compiler):

- Include directories `(-I)`: used for specifying which directories the compiler will look into when searching for include files.
- Source files: used for specifying additional source files.
- Linker directories `(-L)`: used for specifying which directories the linker will search when looking for libraries specified under Linker files.
- Linker files `(-l)`: used to specify additional libraries for the linker. Libraries should be specified without the extension (e.g. for "mylib.dll" you should only add "mylib" as the flag).

- Compiler flags: used for specifying any additional compiler flags. These are added to the compile command as-is.

You can use **(+)** button to add a new flag/file/directory, or **(...)** button whenever available to browse your filesystem for files/directories.

Wherever the compiler flags takes a file path, the same rules as described in Published variables section apply when resolving. In other words you can use relative paths for compiler commands, which might simplify copying the setups between computers.

For all the flags, a special token can be used to differentiate between the 32 and 64 bit compilers: `$(ARCH)` gets replaced by empty string when 32 bit compiler is used, and string "64" with 64 bit compiler. For example, adding "mylib$(ARCH)" under linker files gets transformed into "mylib" with 32 bit compiler and into "mylib64" with 64 bit compiler.

Note that in order to export the bundle on **Project** tab both the 32 bit and 64 bit compilers must successfully compile the code. Also note that when exporting the bundle, any additional libraries or source files are not included in the bundle, meaning you will need to manually copy-paste them if you plan to continue development of the bundles on other computers.

Clicking **Show compiler command** also shows the exact command line command that gets executed every time C++ Script is recompiled, which might help the user figure out why compiling fails.

6.15.1. Example of using external source files

Say we have a header and source files "C:\Dewesoft\MyLibrary\include\mylib.h" and "C:\Dewesoft\MyLibrary\src\mylib.cpp" we want to use in our C++ Script. To use them, we would need to add *C:\Dewesoft\MyLibrary\include* under **Include directories** and *C:\Dewesoft\MyLibrary\src\mylib.cpp* under **Source file**s. In the Editor we can then add `#include "mylib.h"` command to the top of the code, and use any functions defined in the included files.

6.15.2. Example of using external libraries

Say we have a library "C:\Dewesoft\MyLibrary\mylib.dll" with the accompanying header file at "C:\Dewesoft\MyLibrary\include\mylib.h". In this case we would need to add *C:\Dewesoft\MyLibrary* under **Linker directories**, and mylib under **Linker files**. To actually use the library we also need to add #include "mylib.h" at the top of the source code in Editor, and since we are using an include directive, we also need to tell compiler where to find the specified header file, which we do by adding *C:\Dewesoft\MyLibrary*\include to **Include directories**.

Note again that if we wanted to export the bundle, both 32 and 64 bit compilers would have to succeed in compiling our script, but with current configuration this could possibly fail. To solve this issue we can use 2 libraries: "C:\Dewesoft\MyLibrary\mylib.dll" for 32 bit library and "C:\Dewesoft\MyLibrary\mylib64.dll" for the 64 bit version. Then we would change the flag in **Linker files** to *mylib$(ARCH)*, meaning the 32 bit compiler will still use *mylib.dll*, while the 64 bit compiler will use the *mylib64.dll*, solving our problem.

6.16. Tips from developers

Here we share some tips which might prove useful during development of your C++ Script module:

- If you can, set Dewesoft's acquisition sample rate to a low number, for example 500 or 1000 Hz.
- Save your setups often, especially before testing them in *Measurement* mode.
- Add a *Digital meter* visual control in the Measurement mode and attach the default *Debug Channel* to it as soon as you start developing your module; it can save you a lot of headaches.
- For vector / matrix input channels, `getVector(i) / getMatrix(i)` operations are *slow*, especially if you don't plan to access all the elements in the retrieved vector / matrix (e.g. `I.getVector(i)[j]`). If you can, create a local variable of type `bsc::Vector / bsc::Matrix` and assign the vector / matrix to it, limiting the `getVector(i) / getMatrix(i)` calls to at most one per `Module::calculate()` call.
- When you are outputting samples to your asynchronous output channel O with the same sample rate as some input asynchronous channel I, you don't have to manually calculate its `expectedAsyncRate`; simply set it to that input channel's `expectedAsyncRate` in the `Module::configure()` function like so:

```
O.expectedAsyncRate = I.expectedAsyncRate;
```

- Resizing the axis of a vector output channel VO to eg. 21 can be achieved with the following pattern inside `Module::configure()`:

```
VO.axes[0].values = bsc::Vector(21);
for (int i = 0; i < 21; ++i)
    VO.axes[0].values[i] = i;
```

- Looping over all samples contained in input channels when you have `pastSamplesRequiredForCalculation` and `futureSamplesRequiredForCalculation` specified should look something like this:

```
for (int i = -pastSamplesRequiredForCalculation;
         i < callInfo.newSamplesCount + futureSamplesRequiredForCalculation;
         ++i)
{
    bsc::Scalar v = inp1.getScalar(i);
    bsc::Time t = inp1.getTime(i);
    // your code
}
```

7. Warranty information

Notice
The information contained in this document is subject to change without notice.

Note:
Dewesoft d.o.o. shall not be liable for any errors contained in this document. Dewesoft MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. DEWESOFT SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Dewesoft shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

The copy of the specific warranty terms applicable to your Dewesoft product and replacement parts can be obtained from your local sales and service office. To find a local dealer for your country, please visit https://dewesoft.com/support/distributors.

7.1. Calibration

Every instrument needs to be calibrated at regular intervals. The standard norm across nearly every industry is annual calibration. Before your Dewesoft data acquisition system is delivered, it is calibrated. Detailed calibration reports for your Dewesoft system can be requested. We retain them for at least one year, after system delivery.

7.2. Support

Dewesoft has a team of people ready to assist you if you have any questions or any technical difficulties regarding the system. For any support please contact your local distributor first or Dewesoft directly.

Dewesoft d.o.o.
Gabrsko 11a
1420 Trbovlje Slovenia

Europe Tel.: +386 356 25 300
Web: http://www.dewesoft.com
Email: Support@dewesoft.com
The telephone hotline is available Monday to Friday from 07:00 to 16:00 CET (GMT +1:00)

7.3. Service/repair

The team of Dewesoft also performs any kinds of repairs to your system to assure a safe and proper operation in the future. For information regarding service and repairs please contact your local distributor first or Dewesoft directly on https://dewesoft.com/support/rma-service.

7.4. Restricted Rights

Use Slovenian law for duplication or disclosure. Dewesoft d.o.o. Gabrsko 11a, 1420 Trbovlje, Slovenia / Europe.

7.5. Printing History

Version 2.0.0, Revision 217 Released 2015 Last changed: 23. July 2018 at 16:54.

7.6. Copyright

7.7. Trademarks

We take pride in our products and we take care that all key products and technologies are registered as trademarks all over the world. The Dewesoft name is a registered trademark. Product families (KRYPTON, SIRIUS, DSI, DS-NET) and technologies (DualCoreADC, SuperCounter, GrandView) are registered trademarks as well. When used as the logo or as part of any graphic material, the registered trademark sign is used as a part of the logo. When used in text representing the company, product or technology name, the ® sign is not used. The Dewesoft triangle logo is a registered trademark but the ® sign is not used in the visual representation of the triangle logo.

8. Safety instructions

Your safety is our primary concern! Please be safe!

8.1. Safety symbols in the manual

Warning
Calls attention to a procedure, practice, or condition that could cause the body injury or death

Caution
Calls attention to a procedure, practice, or condition that could possibly cause damage to equipment or permanent loss of data.

8.2. General Safety Instructions

Warning

The following general safety precautions must be observed during all phases of operation, service, and repair of this product. Failure to comply with these precautions or with specific warnings elsewhere in this manual violates safety standards of design, manufacture, and intended use of the product. Dewesoft d.o.o. assumes no liability for the customer's failure to comply with these requirements.

All accessories shown in this document are available as an option and will not be shipped as standard parts.

8.2.1. Environmental Considerations

Information about the environmental impact of the product.

8.2.2. Product End-of-Life Handling

Observe the following guidelines when recycling a Dewesoft system:

8.2.3. System and Components Recycling

Production of these components required the extraction and use of natural resources. The substances contained in the system could be harmful to your health and to the environment if the system is improperly handled at its end of life! Please recycle this product in an appropriate way to avoid unnecessary pollution of the environment and to keep natural resources.

This symbol indicates that this system complies with the European Union's requirements according to Directive 2002/96/EC on waste electrical and electronic equipment (WEEE). Please find further information about recycling on the Dewesoft web site www.dewesoft.com

Restriction of Hazardous Substances

This product has been classified as Monitoring and Control equipment and is outside the scope of the 2002/95/EC RoHS Directive. However, we take care of our environment and the product is lead-free.

8.2.4. General safety and hazard warnings for all Dewesoft systems

Safety of the operator and the unit depend on following these rules.

- Use this system under the terms of the specifications only to avoid any possible danger.
- Read your manual before operating the system.
- Observe local laws when using the instrument.
- DO NOT touch internal wiring!
- DO NOT use higher supply voltage than specified!
- Use only original plugs and cables for harnessing.
- You may not connect higher voltages than rated to any connectors.
- The power cable and connector serve as Power-Breaker. The cable must not exceed 3 meters, the disconnect function must be possible without tools.
- Maintenance must be executed by qualified staff only.
- During the use of the system, it might be possible to access other parts of a more comprehensive system. Please read and follow the safety instructions provided in the manuals of all other components regarding warning and security advice for using the system.
- With this product, only use the power cable delivered or defined for the host country.
- DO NOT connect or disconnect sensors, probes or test leads, as these parts are connected to a voltage supply unit.
- Ground the equipment: For Safety Class 1 equipment (equipment having a protective earth terminal), a non-interruptible safety earth ground must be provided from the mains power source to the product input wiring terminals.
- Please note the characteristics and indicators on the system to avoid fire or electric shocks. Before connecting the system, please read the corresponding specifications in the product manual carefully.
- The inputs must not, unless otherwise noted (CATx identification), be connected to the main circuit of category II, III and IV.
- The power cord separates the system from the power supply. Do not block the power cord, since it has to be accessible for the users.
- DO NOT use the system if equipment covers or shields are removed.
- If you assume the system is damaged, get it examined by authorized personnel only.
- Adverse environmental conditions are Moisture or high humidity Dust, flammable gases, fumes or dissolver Thunderstorm or thunderstorm conditions (except assembly PNA) Electrostatic fields, etc.
- The measurement category can be adjusted depending on module configuration.
- Any other use than described above may damage your system and is attended with dangers like short-circuiting, fire or electric shocks.
- The whole system must not be changed, rebuilt or opened.
- DO NOT operate damaged equipment: Whenever it is possible that the safety protection features built into this product have been impaired, either through physical damage, excessive moisture, or any other reason, REMOVE POWER and do not use the product until the safe operation can be verified by service-trained personnel. If necessary, return the product to Dewesoft sales and service office for service and repair to ensure that safety features are maintained.
- If you assume a more riskless use is not provided anymore, the system has to be rendered inoperative and should be protected against inadvertent operation. It is assumed that a more riskless operation is not possible anymore if the system is damaged obviously or causes strange

noises. The system does not work anymore. The system has been exposed to long storage in adverse environments. The system has been exposed to heavy shipment strain.

- Warranty void if damages caused by disregarding this manual. For consequential damages, NO liability will be assumed!
- Warranty void if damage to property or persons caused by improper use or disregarding the safety instructions.
- Unauthorized changing or rebuilding the system is prohibited due to safety and permission reasons (CE).
- Be careful with voltages >25 VAC or >35 VDC! These voltages are already high enough in order to get a perilous electric shock by touching the wiring.
- The product heats during operation. Make sure there is adequate ventilation. Ventilation slots must not be covered!
- Only fuses of the specified type and nominal current may be used. The use of patched fuses is prohibited.
- Prevent using metal bare wires! Risk of short circuit and fire hazard!
- DO NOT use the system before, during or shortly after a thunderstorm (risk of lightning and high energy over-voltage). An advanced range of application under certain conditions is allowed with therefore designed products only. For details please refer to the specifications.
- Make sure that your hands, shoes, clothes, the floor, the system or measuring leads, integrated circuits and so on, are dry.
- DO NOT use the system in rooms with flammable gases, fumes or dust or in adverse environmental conditions.
- Avoid operation in the immediate vicinity of high magnetic or electromagnetic fields, transmitting antennas or high-frequency generators, for exact values please refer to enclosed specifications.
- Use measurement leads or measurement accessories aligned with the specification of the system only. Fire hazard in case of overload!
- Do not switch on the system after transporting it from a cold into a warm room and vice versa. The thereby created condensation may damage your system. Acclimatise the system unpowered to room temperature.
- Do not disassemble the system! There is a high risk of getting a perilous electric shock. Capacitors still might be charged, even if the system has been removed from the power supply.
- The electrical installations and equipment in industrial facilities must be observed by the security regulations and insurance institutions.
- The use of the measuring system in schools and other training facilities must be observed by skilled personnel.
- The measuring systems are not designed for use in humans and animals.
- Please contact a professional if you have doubts about the method of operation, safety or the connection of the system.
- Please be careful with the product. Shocks, hits and dropping it from already- lower level may damage your system.
- Please also consider the detailed technical reference manual as well as the security advice of the connected systems.
- This product has left the factory in safety-related flawlessness and in proper condition. In order to maintain this condition and guarantee safety use, the user has to consider the security advice and warnings in this manual.

EN 61326-3-1:2008

IEC 61326-1 applies to this part of IEC 61326 but is limited to systems and equipment for industrial applications intended to perform safety functions as defined in IEC 61508 with SIL 1-3.

The electromagnetic environments encompassed by this product family standard are industrial, both indoor and outdoor, as described for industrial locations in IEC 61000-6-2 or defined in 3.7 of IEC 61326-1.

Equipment and systems intended for use in other electromagnetic environments, for example, in the process industry or in environments with potentially explosive atmospheres, are excluded from the scope of this product family standard, IEC 61326-3-1.

Devices and systems according to IEC 61508 or IEC 61511 which are considered as “operationally well-tried”, are excluded from the scope of IEC 61326-3-1.

Fire-alarm and safety-alarm systems, intended for the protection of buildings, are excluded from the scope of IEC 61326-3-1.

9. Documentation version history

Version	Date	Notes
1.0	16. 2. 2018	Initial version
V20-1	4. 9. 2020	New template
V21-1	12. 10. 2021	Updated Chapter 6.2. Added the third proprietary option for bundle export. Updated the image of Bundle export settings. Added description of the bundle locking and a corresponding image.