DEWESoft's C++ Script Manual v1.0

What is DEWESoft's C++ Script?

C++ Script is DEWESoft's exciting new feature. You can use it as a compromise between formula's simplicity and full plugins' power to write your very own DEWESoft math modules. 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++.

How it works

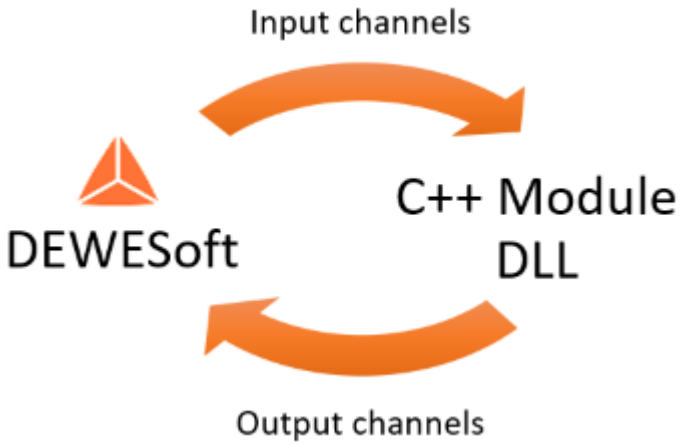


Figure 1: Data cycle

The way the module 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, all you are left with is a nice abstraction layer in C++.

Installation

C++ Script currently resides among *experimental* features in DEWESoft, so you have to manually enable it. Navigate to DEWESoft's *Options* and click on *settings*, then select *Advanced* and finally *Experimental* subsections. There you

will be able to find a feature *C++ Script (Math)* under *Features*; click it so that a checkmark appears next to it, click *Ok*, and restart DEWESoft.

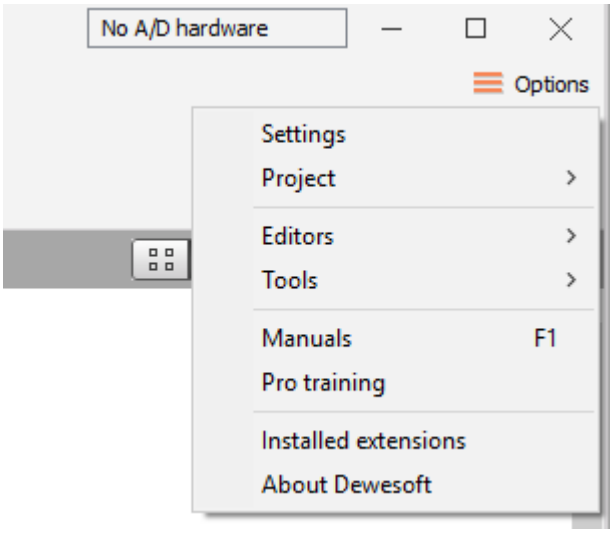


Figure 2: Opening settings

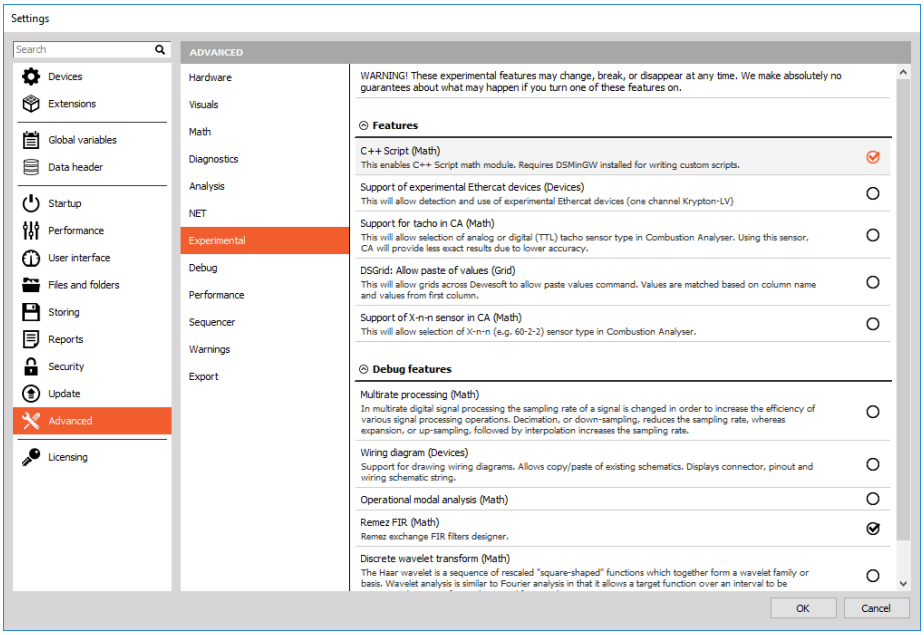


Figure 3: Enabling C++ Script feature

You also need DSMinGW installed on your system.

Tutorial: Simple Signal Averaging Module

To demonstrate both the simplicity and power of the new C++ Script this tutorial shows you how to implement simple averaging module. At the end of the tutorial 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 module has to offer in detail later on in the manual.

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 section called **Forumula and scripting** and finally click on **C++ Script**, which brings up a new C++ Script window.

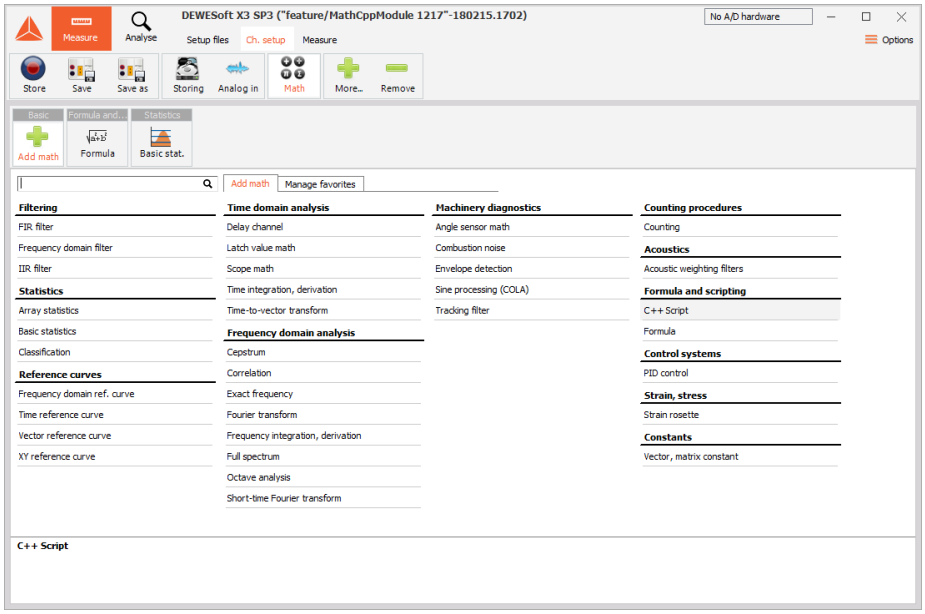


Figure 4: Creating a new C++ Script

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 *Name* field with *"Signal averaging module"*, *Description* with *"Module performing simple signal averaging."* and leave the *Version* fields as they are.

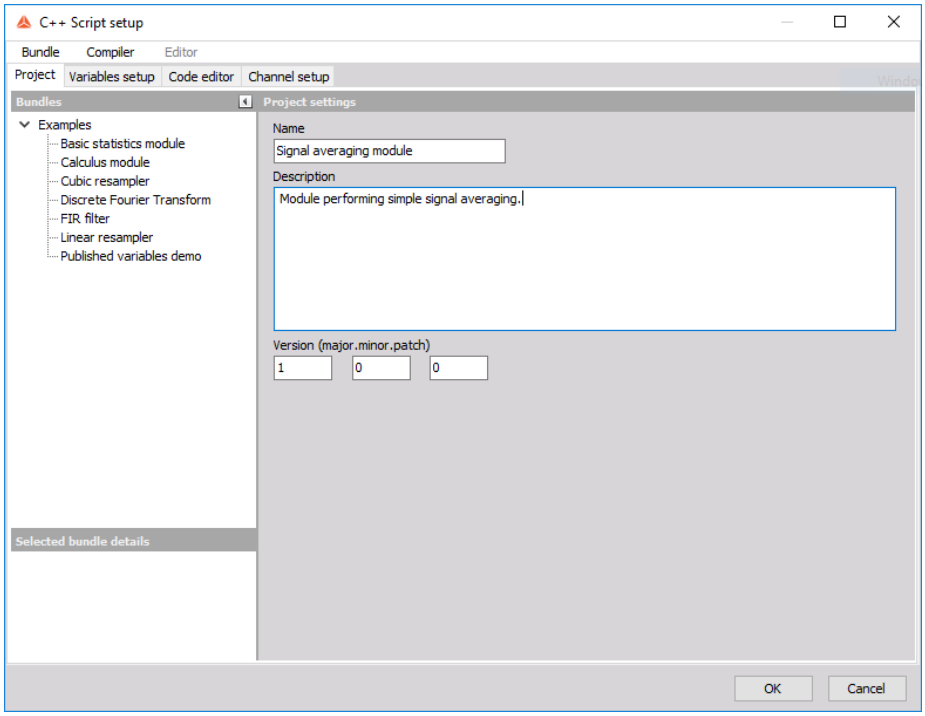


Figure 5: Filling out the **Project** tab

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

Step Two: Variables setup tab

In the **Variables setup** 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.

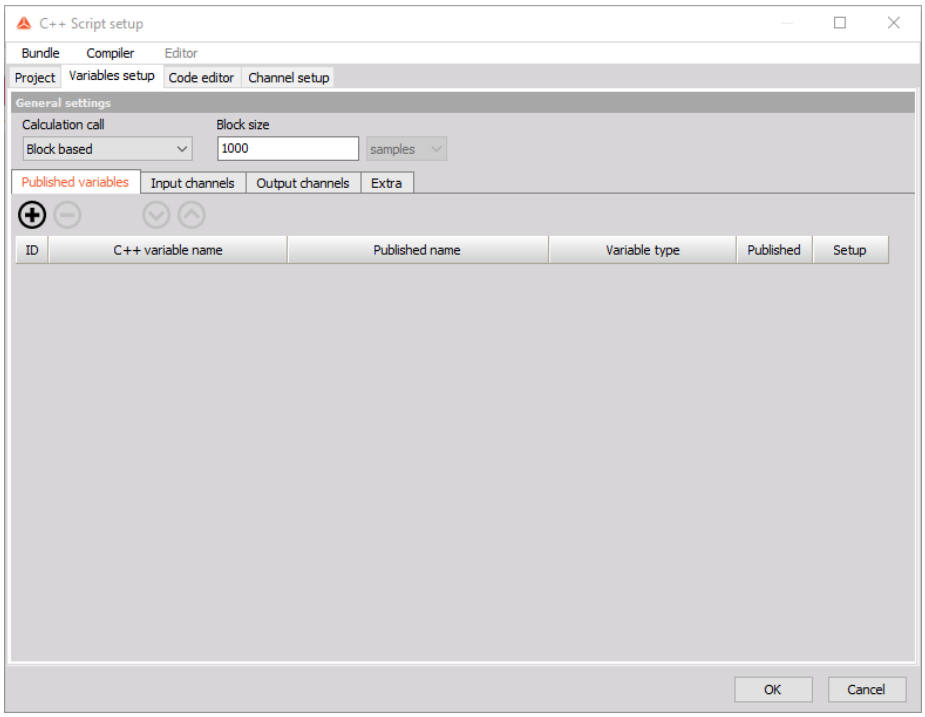


Figure 6: Changing calculation type to 'block based'

Under the *Input channels* tab click the (+) button. This brings up a *Variable setup* form for configuring your input channel. Leave *Value type* as *Scalar* and *Real* and 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.

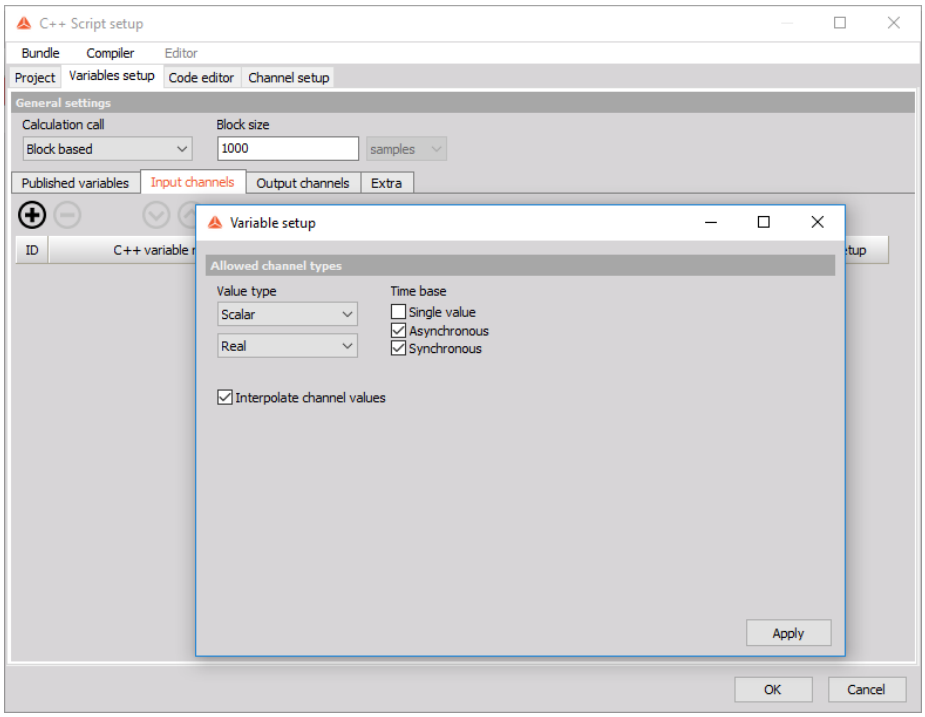


Figure 7: Input channel

Under *Output channels* tab you will already 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 be specified exactly. Leave the *Value type* as *Scalar* and *Real*, *Timebase* as *Asynchronous*, and 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*.

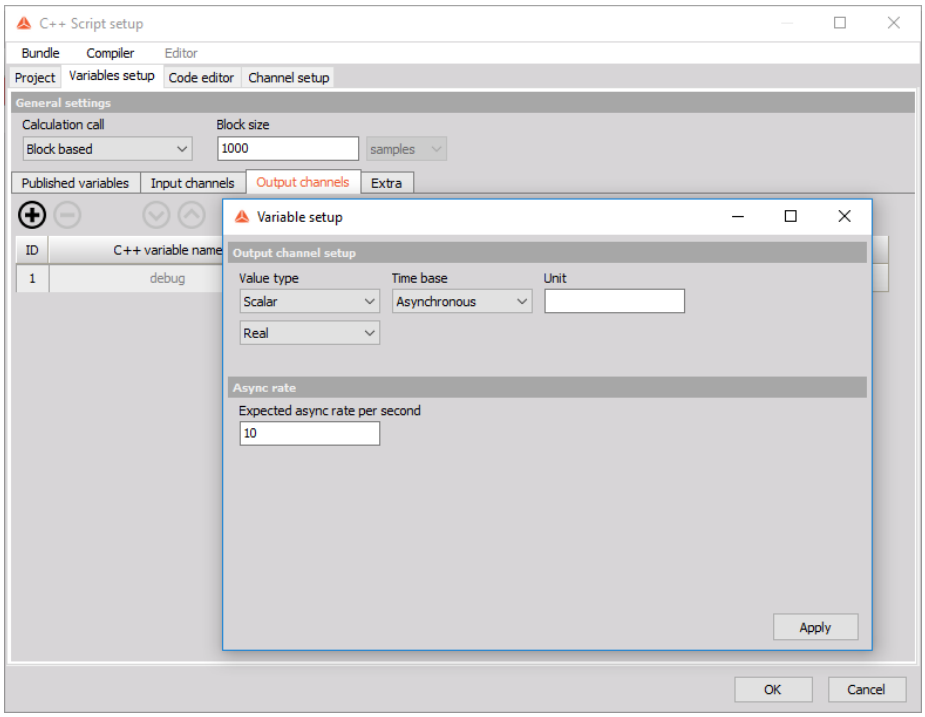


Figure 8: Changing output channel's type

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

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 **Variables setup** tab respectively.

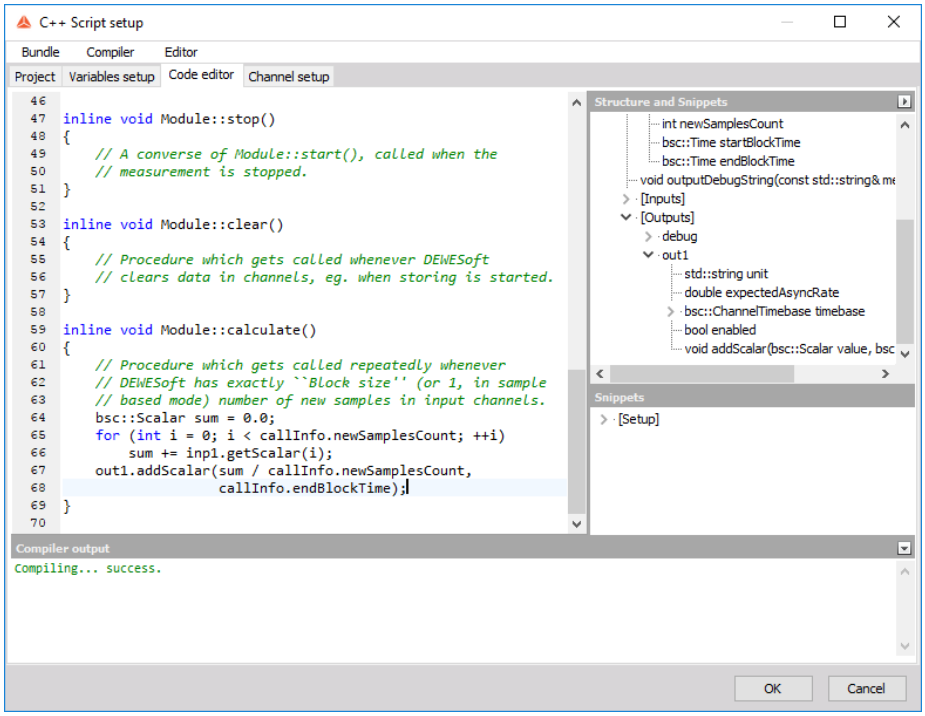


Figure 9: Code editor tab

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.

Step Four: Channel setup tab

The user interface in the **Channel setup** 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 **Variables setup** tab: on the left hand side you have a list of synchronous input channels, which was the type of the *InputChannel1* channel that got pre-populated for you. Ticking the check box 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 the *OutputChanel1*.

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.

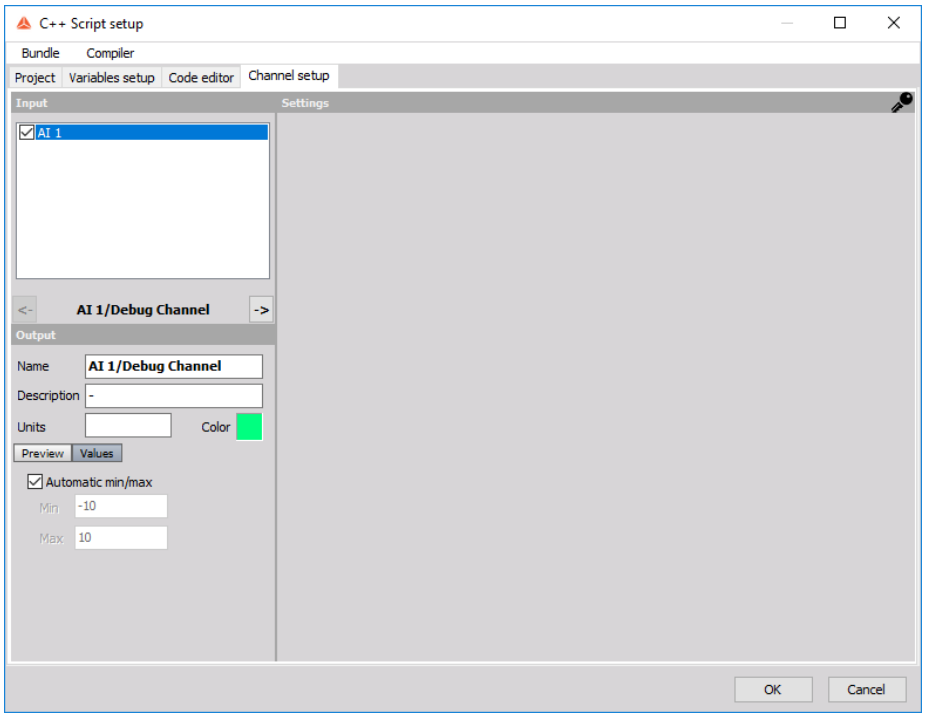


Figure 10: Final settings

Figure 11: Measurement

Small addition to our averaging example

You now have created a module performing averaging of an input signal, but it comes with some 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 *Channel setup* tab meaning user will be able to simply open our module up and set it by himself to whichever value he desires.

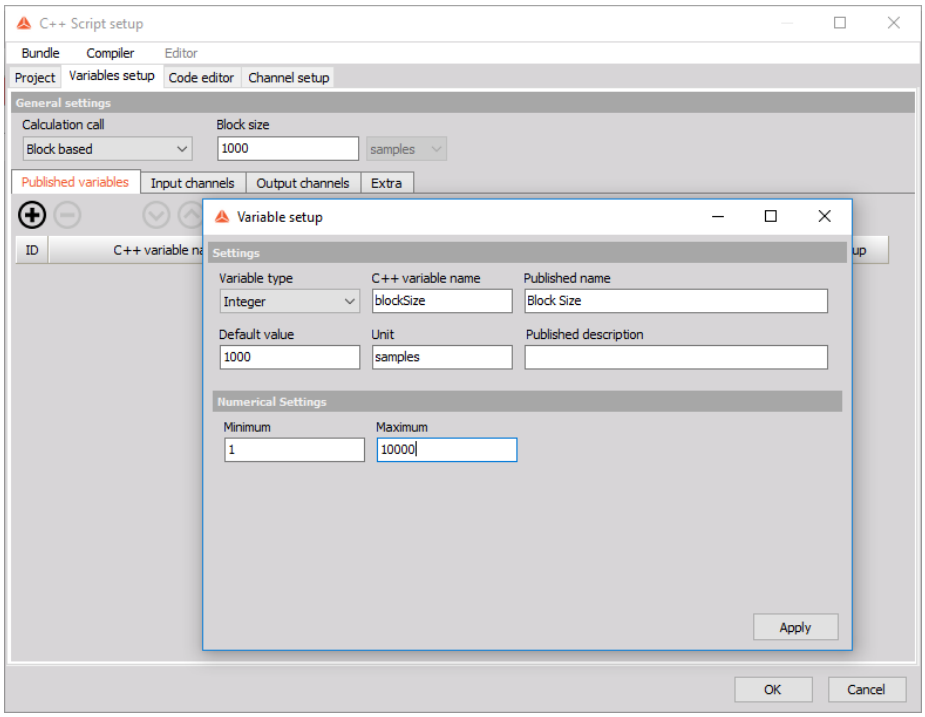


Figure 12: Adding a published variable for controlling the block size

To do that, navigate to **Variables setup** 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 user, and the expected async rate of our output channel will be automatically set correctly regardless of the DEWESoft's acquisition sample rate. Move on to the *Channel setup* tab where you will now be able to find a new field under *Settings*, containing our *Block Size* as defined in *Variables setup* tab and try playing around with different values!

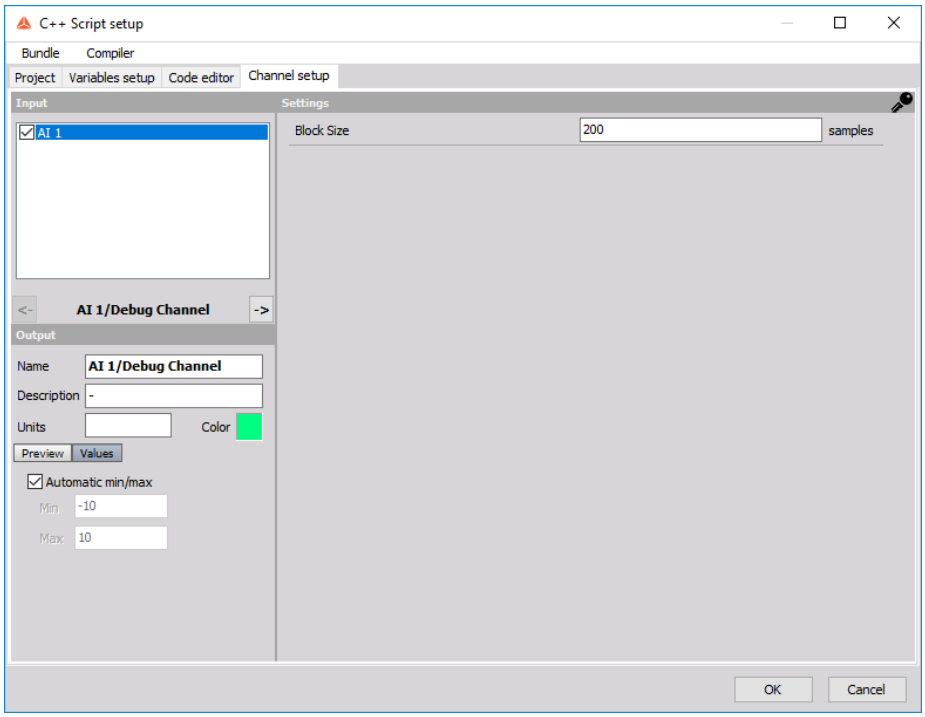


Figure 13: Final settings

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.

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 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 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.

`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;
```

Published variables

Inside **Variables setup** 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 **Channel setup** 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 **Channel setup** tab. Other fields are mostly dependant 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`

Variable type	C++ counterpart
File name	`std::string`

Core variables

`bsc::core` structure contains 1 read-only variable:

- `double bsc::core.acqSampleRate`: acquisition sample rate DEWESoft is currently running with.

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.

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.

Channel types

C++ Script supports various different channel types found throughout DEWE-Soft. 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.

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 `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 $\texttt{i} \geq callInfo.newSamplesCount$. 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()`, $\texttt{d} \in \{0, 1\}$ for `d`-th dimension. Aside from the dimension size, the axes structure's elements also contains individual axis `name`, `unit`, and `values`.

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.

Custom C++ Script types

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

`bsc::Time` type

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

`bsc::Scalar` type

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

`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.

`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]`.

`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:

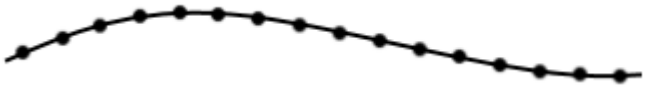
- if any one of output channels is a synchronous channel, you can not specify vector or matrix input channels, and every input channel will be resampled to synchronous rate.
- if none of the output channels are synchronous, the first input channel will be taken as master channel and every other input 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 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.

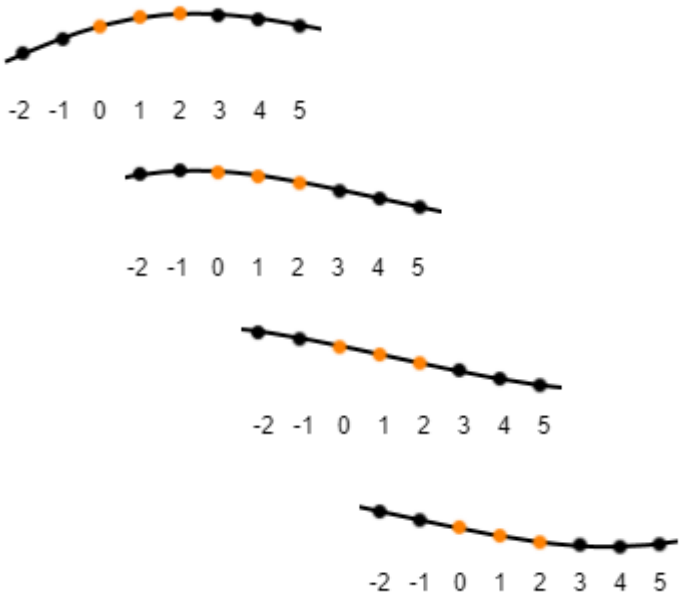
Channel delays

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.

Explanation of C++ Script functions

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.

`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.

`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 opportunity to clean up anything done in `Module::Module()`, for example releasing locks on files, closing libraries, etc.

`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 `expectedAsyncRate` 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 **Channel setup** 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` $\in \{0\}$ if channel is a vector channel, and `d` $\in \{0, 1\}$ if channel is a matrix channel. Before the call to `Module::configure()` the `values` vector gets initialized to $\{1, 2, \ldots, n_d\}$ 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 *Variables setup* tab.

`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.

`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.

`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 calcualtions.

`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.

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 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 **Variable setup**'s **extra** tab.

Tips from developers

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

- Wherever you see `Published X'' of some Y anywhere in the setup form (e.g.` Published name'‘of a variable) that means that that is what Y is going to appear as in the **Channel setup** tab. Conversely, wherever you see "C++ X" that means that that is how it is going to appear in the C++ code.
- 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.
- When you will be 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
}
```