life.augmented

DT0148
Design tip

Using the Functional Block Creator feature in AlgoBuilder

By Zuzana JIRANKOVA

Main components	
AlgoBuilder	Application for the graphical design of algorithms

Purpose and benefits

AlgoBuilder is a graphical design application to build and use algorithms. The application facilitates the process of implementing proof of concept using a graphical interface without writing the whole code. AlgoBuilder reuses previously defined blocks, combines multiple functionalities in a single project and visualizes data using Unicleo-GUI in real time using plot and display.

This design tip focuses on a detailed explanation of configuring a custom block using the Functional Block Creator feature included in AlgoBuilder. This includes the description of the interface, examples of customized codes and integration of custom-made blocks into the user's own designs. Individual steps to create a custom block are explained in detail on a particular example – input multiplier. The design tip also includes more code examples to better understand the possibilities achievable with this tool.

Description

The Functional Block Creator feature allows the user to expand the possibilities of algorithms which can be made in AlgoBuilder. With this feature the user can make his own customized block. This includes defining block information, setting inputs, outputs, properties, and its logic written in C language.

Figure 1. Functional Block Creator in bar menu

The Functional Block Creator can be accessed from the bar menu of the AlgoBuilder interface (as shown in Figure 1) and through the Tools menu (as shown in Figure 2).

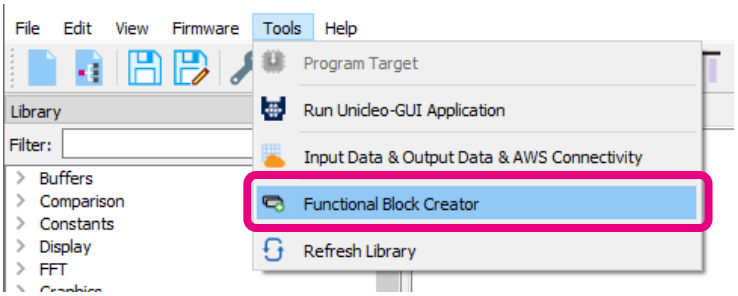

Figure 2. Functional Block Creator in Tools menu

Creating a custom block

In this section the creation of custom blocks is explained in detail. The main components of the block apart from general information and code are: inputs, outputs and properties. The block receives information through inputs and sends the information further using its outputs. Properties can be understood as a configuration which can be adjusted before compilation and does not change during program execution. As an introduction example, a multiplier of an input is created. It has an input and an output both of type INT (integer) and size 1. The multiply factor of the input is the property of the block. Thus, the relation between output and input is the following: *output = input * multiply_factor*.

Block information

The first tab of the Functional Block Creator is *Block Info* (Figure 3). General information about the new custom block is filled in this tab.

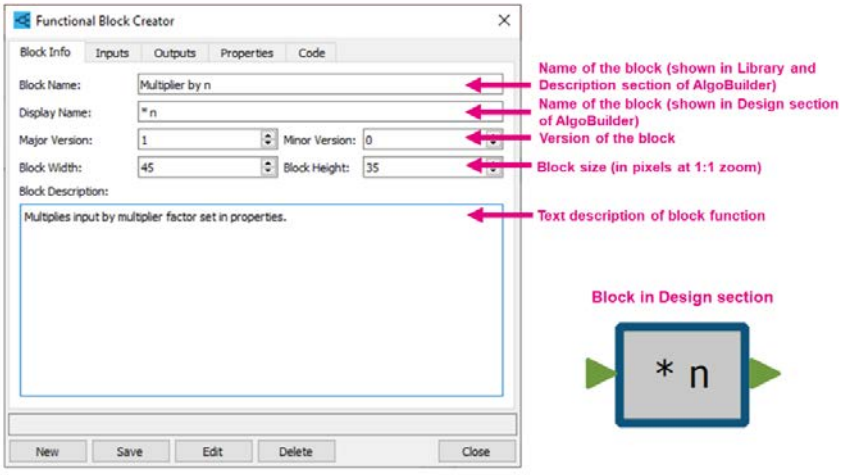

Figure 3. Description of block information tab

Block name is the full name of the block, and it is shown in the Library and Description section of the AlgoBuilder interface. Display name is a shorter version of the name which is shown in the block itself in the design diagram. Major and minor versions help to keep track

of the block version. Version 1.0 is the default version. Another parameter to set is the physical size in the design diagram. This can be done by setting block width and height. It corresponds to the size of the block in pixels for 1:1 zoom. The default size is 25x25 px. The last thing to fill in is the block description, where the functionality of the block can be briefly explained. The description of the block is optional but helps to understand the purpose and behavior.

✎ *The block name can be modified so it shows the multiply factor in the graphics of the Design section chosen by the property "multiply_factor" (which is described in the Properties section setting) to: * %multiply_factor%. Thus, for a multiply factor set to value 1 the block shows "* 1" instead of "* n".*

Input setting

The setting of all inputs of the block is done in the *Inputs* tab (Figure 4). The input is characterized by its name, type, size and description.

The input name is the name of a variable, which is used later in the *Code* tab. The characters, which can be used to name an input, are limited by C language naming rules. This means the name can contain letters (both uppercase and lowercase), digits and underscore. The first character of a name should be a letter or an underscore.

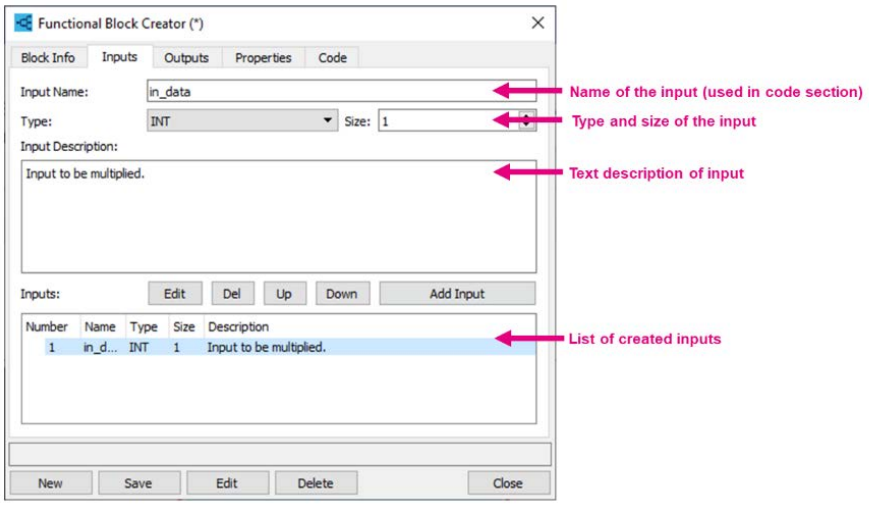

Figure 4. Inputs tab interface

Each input needs to have a type assigned. These are:

- **FLOAT** – 4-byte floating point data type
- **INT** - 4-byte integer data type
- **VARIANT** – takes over the type of the output, which is connected to this input, but the output connected to this input must be the same size

The size of the input defines the number of elements of a certain type. Finally, an optional description field contains a text description of the input.

✎ *Input with name “tmp” of type INT and size of 9 is in C code equivalent to: int tmp[9];*

✎ *Input is created as an array including size 1.*

To add the input to the block, click *Add Input* below the input description. Then you can see the input listed in the *Inputs* list. To delete an input, select the input from the list and press *Del*. To change the order of the inputs in the list, select the input to be shifted and press *Up* or *Down* depending on the desired direction. To edit already created input, select the input from the list and press *Edit*. The information loads into the top part of the interface and the bar with the options changes (Figure 5). This bar is used to save or discard any modifications.

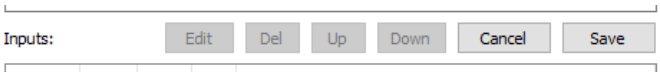

Figure 5. Options bar in Edit mode

Output setting

The setting of all outputs of the block is done in the *Outputs* tab (Figure 6). Similar to input, the output is characterized by name, type, size, and description.

The output name is the name of a variable, which is used later in the *Code* tab. The characters, which can be used to name an output are limited by C language naming rules. This means the name can contain letters (both uppercase and lowercase), digits and underscore. The first character of a name should be a letter or an underscore.

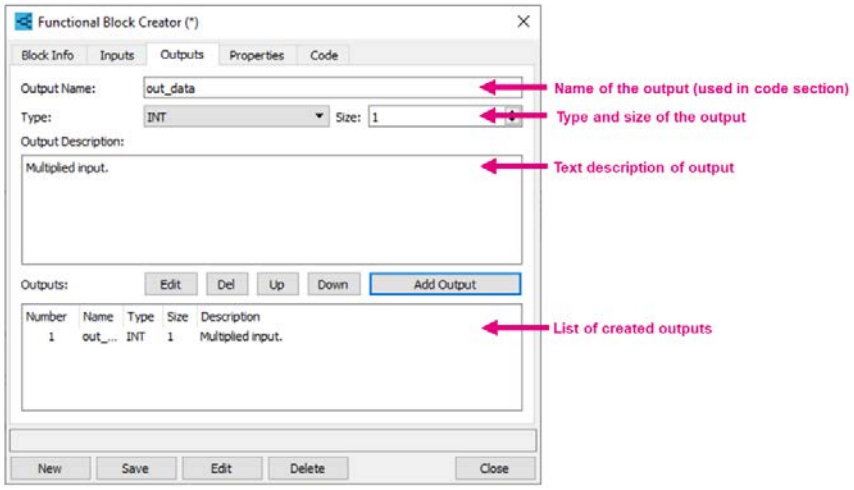

Figure 6. Outputs tab interface

Each output needs to have a type assigned. These are:

- **FLOAT** – 4-byte floating point data type
- **INT** – 4-byte integer data type

The size of the input defines the number of elements of a certain type. Finally, an optional description field should contain a text description of the output.

✎ *Output with name “tmp” of type INT and size of 9 is in C code equivalent to: int tmp[9];*

✎ *Output is created as an array including size 1.*

To add the output to the block, click on *Add Output* below the output description. Then you can see the outputs listed in the *Outputs* list. To delete an output, select the output from the list and press *Del*. To change the order of the outputs in the list, select the output to be shifted and press *Up* or *Down* depending on the desired direction. To edit already created output, select the output from the list and press *Edit*. The information loads into the top part of the interface and the bar with the options changes (Figure 5). This bar is used to save or discard any modifications.

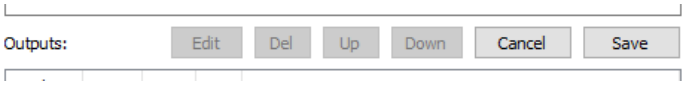

Figure 7. Options bar in Edit mode

Properties setting

The setting of all properties of the block is done in the *Properties* tab (Figure 8). A property is characterized by name, type, value, enum value and description.

The property name is the name of a variable, which is used later in the *Code* tab. The characters, which can be used to name a property are limited by C language naming rules. This means the name can contain letters (both uppercase and lowercase), digits and underscore. The first character of a name should be a letter or an underscore.

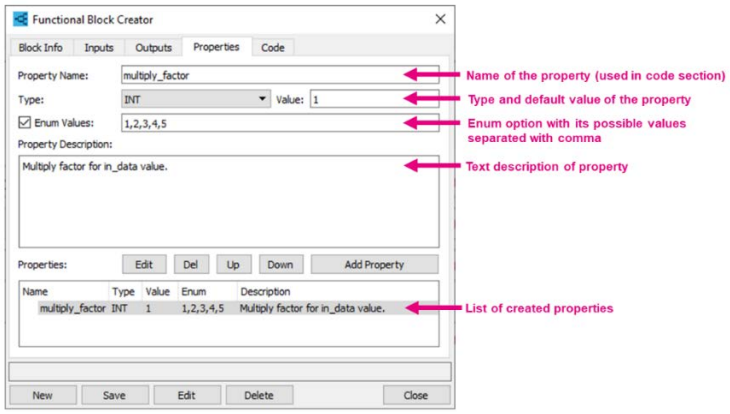

Figure 8. Properties tab interface

Each property needs to have a type assigned. These are:

- **FLOAT** – 4-byte floating point data type
- **INT** - 4-byte integer data type
- **STRING** – array of char data type (1-byte)

Unlike input and output, the property has a value to be set and an enum option. A value is a default value assigned to the property. The user can decide whether the property can be set freely by the user, or if the value should be selected from the predefined options. This can be done by ticking the box *Enum values*. The value options are separated by a comma. Finally, the optional description field should contain a text description of the property.

To add the property to the block, click on *Add property* below the property description. Then you can see the property listed in the *Properties* list. To delete a property, select the property from the list and press *Del*. To change the order of the properties in the list, select the property to be shifted and press *Up* or *Down* depending on the desired direction. To edit already created properties, select the property from the list and change the desired attributes. The information loads into the top part of the interface and the bar with the options changes (Figure 9). This bar is used to save or discard any modifications.

Figure 9. Options bar in Edit mode

Programming the logic of a custom block

In the last tab *Code* (Figure 11) it is possible to program the logic of the block. The programming language used is C. The code itself is divided into 3 parts. These are: Command code, Init code and Function code.

Command Code is a part of the main routine, which is called by AlgoBuilder. The call frequency of the AlgoBuilder project, thus Command Code as well, can be adjusted in the *Sensor hub* block in *Properties* (Figure 10). It is possible to drive the frequency by the ODR of a sensor or a timer. This can be set by the property *Data Rate Control.* The options are: Timer, Accelerometer, Gyroscope, Offline data. The frequency is set using *Data Rate*. If the ODR of a sensor is set as the control, the closest higher frequency is set in the sensors and the routine is driven by a *data ready* event interrupt of the selected sensor. If a timer is selected as the control, the ODR of the sensors is set to closest higher value, but the data are refreshed by a timer event interrupt.

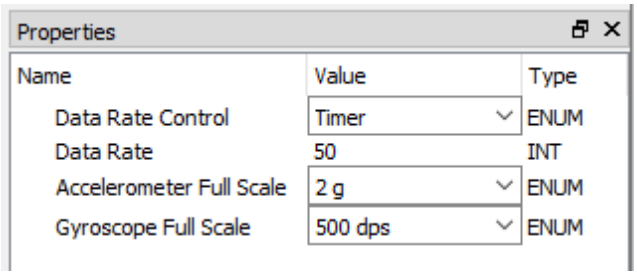

Figure 10. Sensor hub properties

In Command Code the inputs, outputs and properties can be used by referencing its name. For an input with name *input_1* and size 5, its 4th element can be accessed by the array formula: *%input_1%[3]*. Outputs are referenced in the same manner and the formula is

applied to inputs and outputs of size 1 as well. It is important to handle reading and writing correctly when using arrays. This means not going beyond the array range to avoid undefined behavior. The property is referenced only by its name. For a property with the name *block_prop*, it can be referenced by the following formula: *%block_prop%*. It is not possible to define or declare any functions in Command Code, as it is a part of the main routine function called by Algobuilder.

In the introduction example, the input is multiplied by the property value and saved to the output. The following code can be used:

```
%out_data%[0] = %in_data%[0] * %multiply_factor%;
```

The next type of code is Init Code. This type of code is called only once at the beginning of the program. Its use is shown in examples included in the *Other examples* section. Apart from C code, any input, output or property referenced with %% notation can be used here. Inputs and outputs are referenced using array formula including size 1 (e.g. *%input_1%[3]).* It is not possible to define or declare any functions in Init Code, as it is a part of the initialization routine of Algobuilder.

Within Function Code it is possible to define the user's own functions, which can be used in the Command and Init Code. In the Function Code part, it is not possible to reference inputs, outputs or properties of the block and it should be passed to functions as parameters. The usage of functions is included in examples in the section *Other examples*. It is important to keep in mind that if a *static* variable is used in the function, only one instance of the block can be used in the design for correct behavior. All instances of the block call the same function.

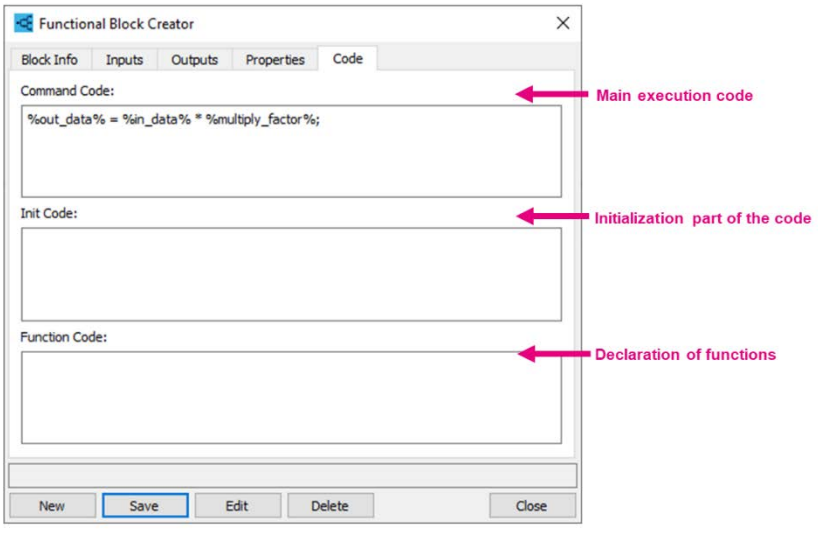

Figure 11. Code tab interface

Managing the library of custom blocks

The custom blocks are stored in Libraries, which are represented by an *.xml* file. The following section describes how to save, edit and delete a custom block.

Saving a custom block

After all fields are filled, the block can be saved by clicking on the *Save* button. The blocks are grouped into libraries. The library name is defined by the name of the xml file, where blocks belonging to this library are saved (Figure 12). All libraries, thus xml files, are stored in the following folder: *C:\Users\<username>\STMicroelectronics\AlgoBuilder\Library*.

Figure 12. Saving block into a library file

After saving the block, the block can be accessed in AlgoBuilder's design Library interface under the name of the xml file (Figure 13).

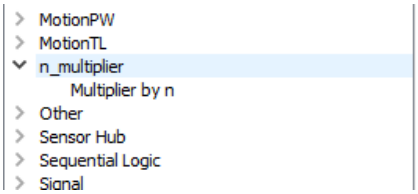

Figure 13. Custom-made library in Library section of AlgoBuilder interface

Editing a custom block

To edit a block, click on *Edit* in the interface of the Functional Block Creator. A list of custom libraries is displayed (Figure 14). Select the desired block to edit from the library and click on Edit. The information about the block is loaded to the interface, where it is possible to edit the information.

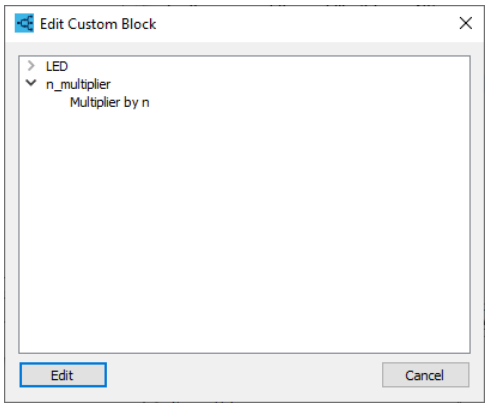

Figure 14. List of custom-made libraries

When editing a block, it is recommended to update the block version (in the *Block Info* tab), too. By updating the block version, it is possible to update the block in the design easily (Figure 15) by right-clicking on the block and selecting *Check for new version of block*.

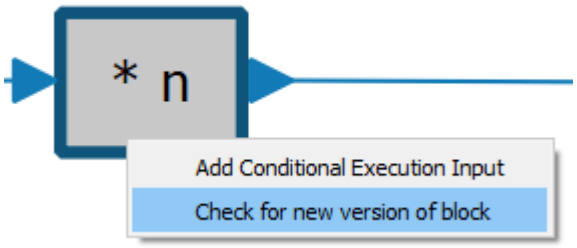

Figure 15. Update of custom block in Design section of AlgoBuilder interface

Deleting a custom block

To delete a block, click on *Delete* in the bottom interface of Functional Block Creator. A list of custom libraries is displayed (same list as in Figure 14). To delete a block, select the desired one and click *Delete*.

If the block was the last one in the xml file, the file won't be deleted. It is possible to save new blocks created later into the library again.

Testing a block

To test a block, a simple design is created. A mandatory block is *[Sensor Hub],* where loop frequency setting is included. The data rate is set to 50 and the control is set to *Timer.* To have the output of this block connected, a simple graph showing accelerometer data is created (top row of Figure 16). The data type of the *Graph* block is set to "Accelerometer".

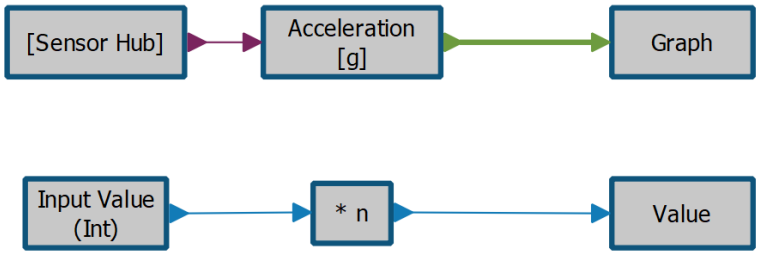

Figure 16.Test design

In the bottom row of Figure 16 a test setup for a custom block * *n* is created. The setup starts with the *Input value* block (from the User Input library). In the block properties we can set the following - Window name to Input and Number of values to 1, Default value to 1 and Value name to Input. To the output we can connect * *n* block. The multiply factor property of this block can be selected by the user (for example value 3). The last block connected in this row is the *Value* block (from the Display library). Its purpose is to show the resulting

multiplied value. The properties are set as follows: Number of values is set to 1, Data type is set to *custom*, value name is set to *Result* and the window name is set to *Output*. All the above mentioned properties can be found in Figure 17.

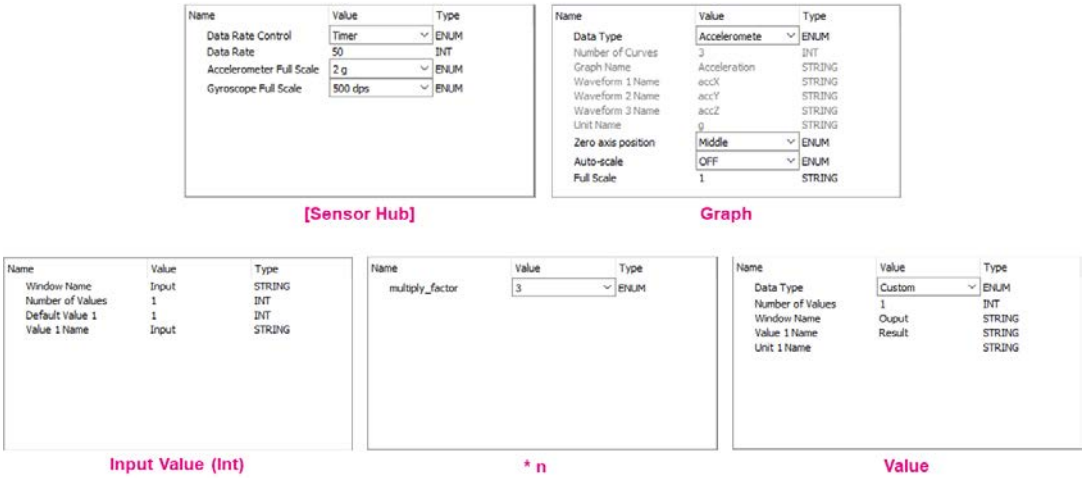

Figure 17. Properties of blocks included in test design

After the design is created and firmware is initialized for the chosen hardware (detailed information about initializing firmware can be found in the AlgoBuilder User Manual), it is possible to *Generate C Code* and *Build Firmware*. After the board is connected to the computer, it is possible to *Program Target* and run the program by clicking on *Run Unicleo-GUI application*. If there is an error in the code, Algobuilder outputs all the errors to the console located at the bottom of the AlgoBuilder interface. Every listed line with an error corresponds to a line in the *algobuilder.c* file, which can be found under the *Show C Code* option (Figure 18).

Figure 18. Show C code button

After the Unicleo-GUI application starts, click *Start* (1) and open windows *Input* (2) and *Output* (3) as shown in Figure 19. The program created in Algobuilder runs and it is possible to type the desired input into the *Input* window and the output (multiplied by 3 in this case) is shown in the *Output* window.

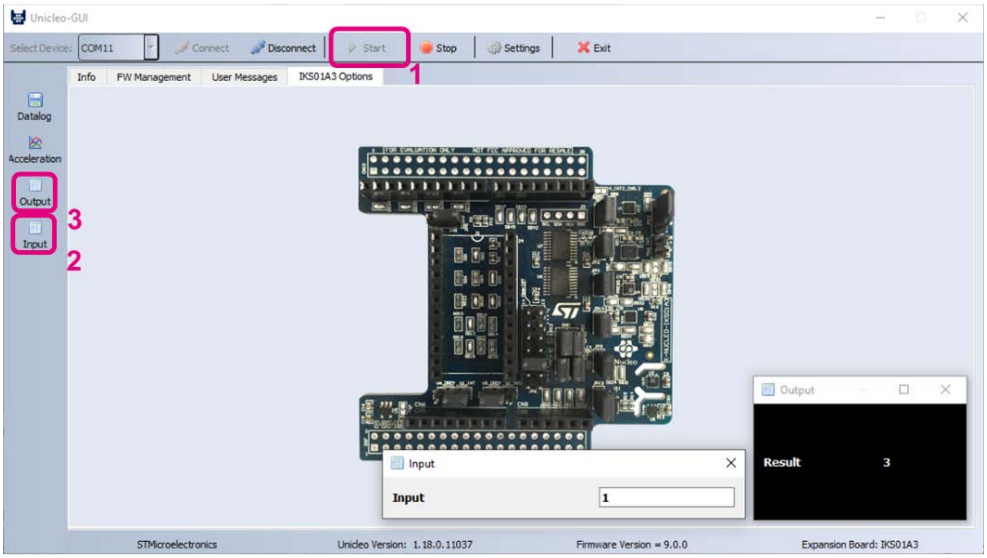

Figure 19. Unicleo-GUI application

Other block examples

More examples of custom-made blocks are listed below.

N-multiplier (using function)

This is functionally the same block as previously described, only solved using a function.

Block info:

- Block name: Multiplier by n
- Display name: * %multiply_factor%
- Block size: 45 x 35

Input 1:

- Input name: in_data
- Input type: INT
- Input size: 1

Output 1:

- Output name: out_data
- Output type: INT
- Output size: 1

Property 1:

- Property name: multiply_factor
- Property type: INT
- Value: 1
- Enum values: 1,2,3,4,5

Code:

- **Command code**:

```
multiply(%in_data%, %out_data%, %multiply_factor%);
```

- **Init code**: -

- **Function code**:

```
void multiply (int* in, int* out, int prop)
{
 out[0] = in[0] * prop;
}
```

Min/max from 3 inputs

Finds maximum out of 3 values if the property *maximum* is set to 1, otherwise finds the minimum. This function uses an input of type VARIANT, meaning the output of type INT or FLOAT can be connected to this input. To handle both output types, _Generic selection is used. This expression is available since the C11 standard. Generic provides a way to control selection during compile time using a type of controlling expression. More information can be found here. This example was tested using STM32CubeIDE 1.5.0.

Block info:

- Block name: Min/Max calculation
- Display name: Min/Max
- Block size: 60 x 35

Input 1:

- Input name: in_data
- Input type: VARIANT
- Input size: 3

Output 1:

- Output name: out_data
- Output type: FLOAT
- Output size: 1

Property 1:

- Property name: maximum
- Property type: INT
- Value: 1
- Enum values: 0,1

Code:

- **Command code**:

```
if (%maximum%)
{
  find_max(%in_data%, %out_data%, 3);
}
```

```
else
{
  find_min(%in_data%, %out_data%, 3);
}
```

- **Init code**: -
- **Function code**:

```
#define find_max(in, out, data) (_Generic((in), float*:
find_max_float, int32_t*: find_max_int))(in, out, data)

#define find_min(in, out, data) (_Generic((in), float*:
find_min_float, int32_t*: find_min_int))(in, out, data)


void find_min_int (int32_t* in, float* out, uint8_t
data_count)
{
  int min_tmp = in[0];
  for (uint8_t i = 1; i < data_count; i++)
  {
    if (min_tmp > in[i])
    {
      min_tmp = in[i];
    }
  }
  out[0] = (float) min_tmp;
}


void find_max_int (int32_t* in, float* out, uint8_t
data_count)
{
  int max_tmp = in[0];
  for (uint8_t i = 1; i< data_count; i++)
  {
    if (max_tmp < in[i])
    {
      max_tmp = in[i];
    }
  }
  out[0] = (float)max_tmp;
}


void find_min_float (float* in, float* out, uint8_t
data_count)
{
  float min_tmp = in[0];
```

```
  for (uint8_t i = 1; i < data_count; i++)
  {
    if (min_tmp > in[i])
    {
      min_tmp = in[i];
    }
  }
  out[0] = min_tmp;
}


void find_max_float (float* in, float* out, uint8_t
data_count)
{
  float max_tmp = in[0];
  for (uint8_t i = 1; i< data_count; i++)
  {
    if (max_tmp < in[i])
    {
      max_tmp = in[i];
    }
  }
  out[0] = max_tmp;
}
```

Average of n samples

Decimation using average of n samples. The number of samples is specified by the property *window_length.* The value can be in the range [1,100]. If a lower or higher number is specified, it is trimmed. It is important to use **only one instance** of the block when using *static* variables as all blocks use the same function and this leads to incorrect behavior.

Block info:

- Block name: Average of n
- Display name: AVG
- Block size: 55x40

Input 1:

- Input name: in_data
- Input type: FLOAT
- Input size: 1

Output 1:

- Output name: avg_data
- Output type: FLOAT
- Output size: 1

Property 1:

- Property name: window_length
- Property type: INT
- Value: 1
- Enum values: -

Code:

- **Command code**:

```
update_avg_value(%in_data%, %avg_data%, window_len);
```

- **Init code**:

```
window_len = %window_length%;
if (window_len < 1) window_len = 1;
else if (window_len > 100) window_len = 100;

%avg_data%[0] = 0;
```

- **Function code**:

```
int32_t window_len = 0;




void update_avg_value (float * in, float* out, int
win_length)
{
  static float avg_buffer = 0;
  static int32_t cnt = 0;

  avg_buffer += in[0];
  cnt ++;

  if (cnt >= win_length){
    out[0] = avg_buffer/win_length;
    avg_buffer = 0;
    cnt = 0;
  }
}
```

Support material

Documentation
User manual, UM2373, Getting started with the AlgoBuilder application for the graphical design of algorithms

Revision history

Date	Version	Changes
21-Jan-2022	1	Initial release

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to *www.st.com/trademarks*. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.