NAVTELECOM
TELEMATICS SYSTEMS

COMPLEX EVENTS
User manual

Version 1.6

Moscow
2022

CONTENTS

1. CHANGE HISTORY

Version 1.0 from 14.10.2021:

- The first version of the document.

Version 1.1 from 10.11.2021:

- Added section "Description of Blocks in the Block Diagram";
- Added block "FROM_FLOAT";
- Added block "CALENDAR";
- Added new functions description in the "Color scheme" section.

Version 1.2 from 23.11.2021:

- Added description of drag'n'drop mode for moving blocks and functions to the diagram;
- Added section describing Undo / Redo operations.

Version 1.3 from 15.12.2021:

- Added section "Flowchart description elements";
- Added new setting description in the "Debug settings" section;
- Added "DELAY" function;
- Added "NTC_CRASH_FILE" function;
- Updated "NTC_ACCEL" function description;
- Updated "TO_FLOAT" and "FROM_FLOAT" functions description.

Version 1.4 from 25.01.2022:

- Fixed description of "TP" function;
- Added description of "EVENT", "SMS", "CALL", "CAM" functions;
- Added "USER_SMS" function;
- Added "RECV_SMS" function;
- Removed unused events in "List of CE Events Codes" section;
- Renamed the blocks from the "Peripheries" group in the "Function Blocks Library";
- Added "PWRSAVE" function.

Version 1.5 from 07.04.2022:

- Changed description in the "General information" section;
- Added section "Description of Functional Blocks Scheme Elements";
- Updated description of "TO_FLOAT", "FROM_FLOAT", "FLEX", "USER_PARAM", "USER_SMS" "OUTPUT" functions;
- Added "APERTURE", "RXD_GET", "RXD_CMP", "RXD_STR2INT", "RXD_STR2FLOAT", "RXD_CHECKSUM", "TXD_INIT", "TXD_SET", "TXD_SET", "TXD_CHECKSUM", "TXD_GET", "RS_TRANS", "RS_SEND", "RS_RECV" functions;
- Added section "Access Functions to Digital Ports".

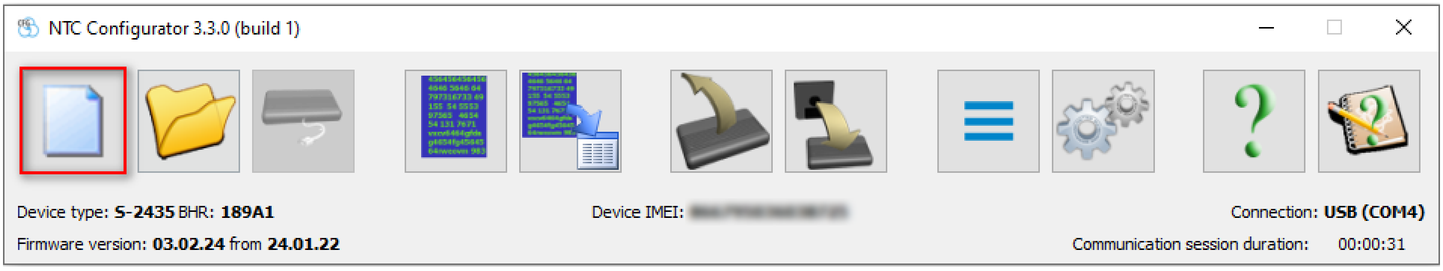
Version 1.6 from 21.06.2022:

- Updated description in the "Creating a Function block diagram" section;
- Added section "Automatic numbering of function blocks";
- Added section "Variables search";
- Updated description of "FLEX", "RXD_GET", "TXD_SET", "TXD_GET" functions;
- Added "INFO", "IMEI", "ICCID", "IMSI", "LOG_MSG", "MODBUS_READ", "MODBUS_WRITE" functions;
- Updated styles of the information frameworks.

2. QUICK START

Let's write a simple program to the value of a variable by 1.

For that:
1. Connect a device with support of Complex Events.
2. Run NTC Configurator.
3. Create a new configuration.



NTC Configurator 3.3.0 (build 1)

Device type: **S-2435** BHR: **189A1**

Device IMEI: [redacted]

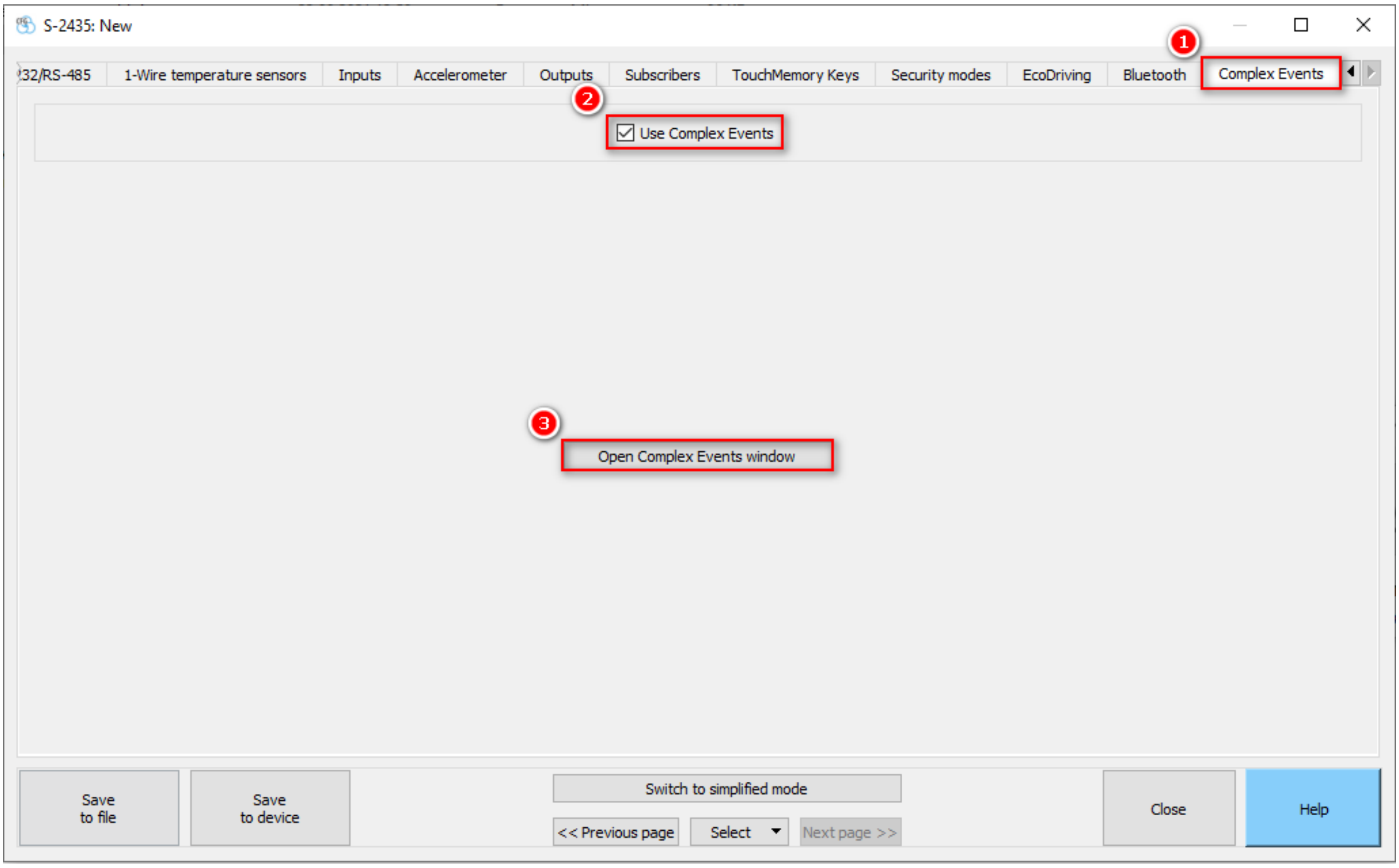
Connection: **USB (COM4)**

Firmware version: **03.02.24** from **24.01.22**

Communication session duration: 00:00:31

Figure 2.1 – Creation of new configuration

4. Click the *Complex Events* tab.
5. Tick – *Use Complex Events* and click the button ***Open Complex Events window***.



S-2435: New

☑ Use Complex Events

Figure 2.2 – Launching of Complex Events editor

6. In the window that opens you should choose menu item ***File – New*** (or click on the button on the toolbar). In the flowchart editor, on the left part of the editor, a simplest flowchart will appear.
7. Click on the ***Action*** block on the left part of the editor. On the right part of the editor there will be the content of the selected ***Action*** block.

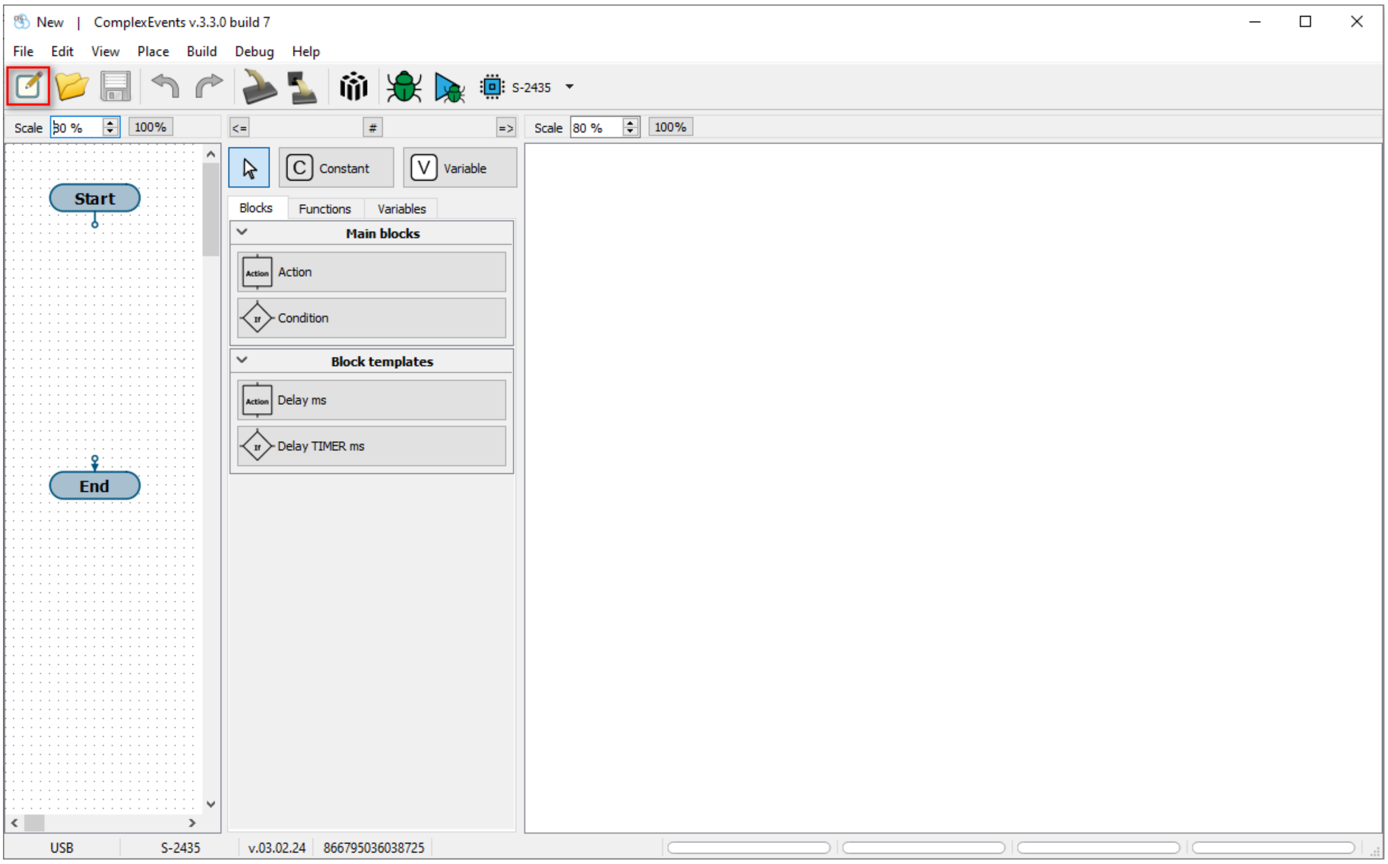
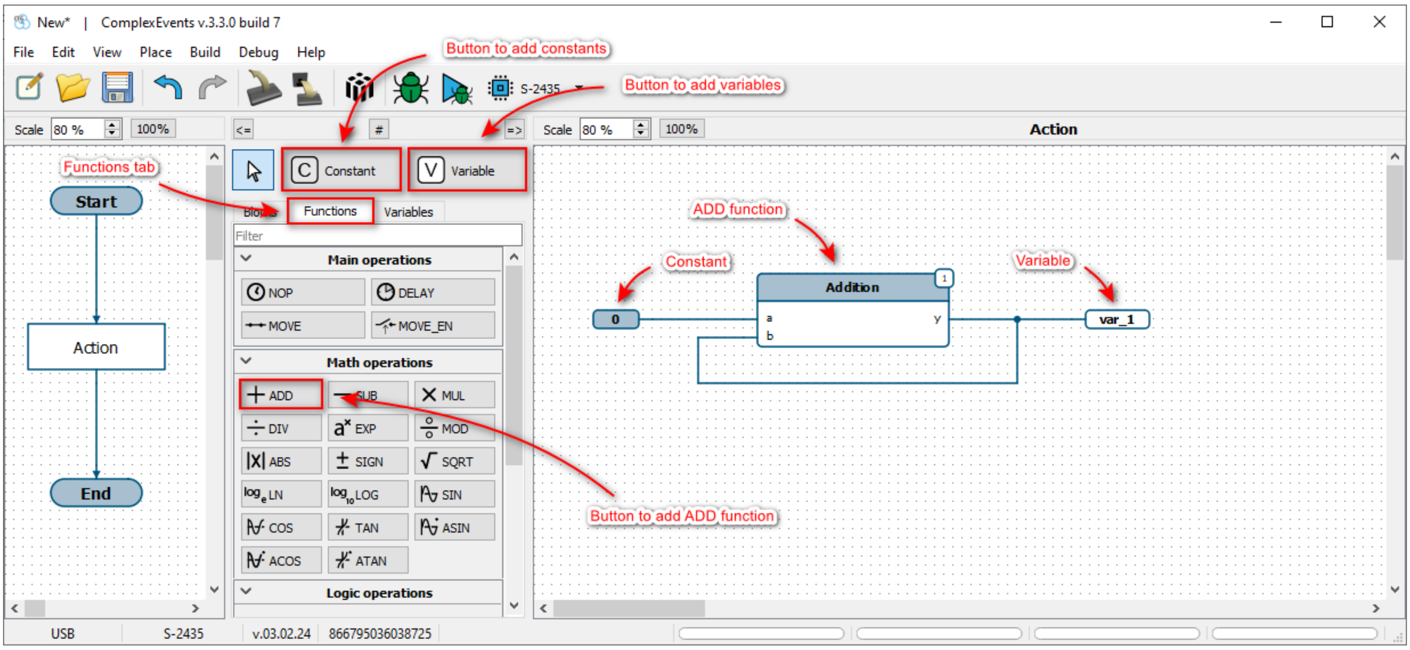


Figure 2.3 – Creation of a new simplest block diagram

8. Select *Functions* tab on the panel on the middle part of the editor.
9. In the displayed window with functions, click on the ***ADD*** button in the *Math operations group.* Then move the cursor to the right part of the editor and click in any place of this field. The *Addition* function will appear in the editor.
10. Click on the ***Variable*** button, move the cursor to the right part of the editor and click in any place of this field.
11. Click on the ***Constant*** button, move the cursors to the right part of the editor and click in any place of this field.
12. Change the value of the constant to **1**. To do this, double-click the constant and in the opened dialog box, in the *Value* field, enter the number 1.
13. Place the added elements (Figure 2.4) and connect them. For connection of two pins, you should click with the left mouse button on the first pin, then click on the second pin.



Button to add constants

Button to add variables

Functions tab

ADD function

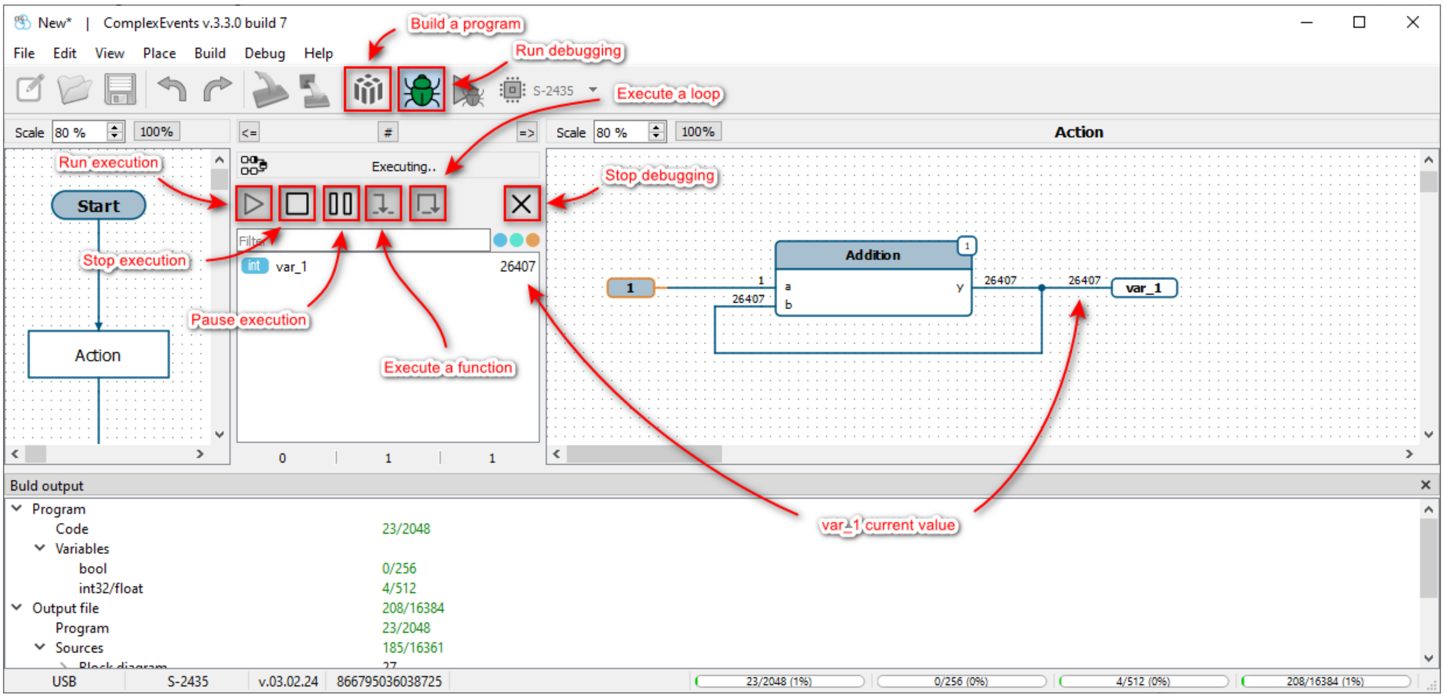
Constant

Variable

Button to add ADD function

Figure 2.4 – Scheme of incrementing the variable var_1 by 1

14. Choose menu item ***Build – To build*** (or click the button on the tollbar). If everything is done correctly, program will build without errors.
15. Choose menu item ***Debug – Start debugging*** (or click the button on the tollbar). If the device is connected, a window prompts upload configuration to the device will appear, click ***Yes** button***. Wait a few second until the device reboots, then the application will automatically load the program and enter debug mode.
16. Click on the ***Step*** ↴ button several times (menu item ***Debug - Step***) and make sure that the value of the variable is increased by 1. The current value of the variable is displayed above its output.
17. Click on the ***Run*** ▷ button (menu item ***Debug - Run***) and make sure that the value of the variable is increased.
18. Click on the ***End debug*** ☒ button (menu item ***Debug – End debug***).



```
Buld output
Program
    Code                23/2048
  Variables
      bool              0/256
      int32/float       4/512
Output file             208/16384
    Program             23/2048
  Sources               185/16361
      Block diagram     27
USB   S-2435   v.03.02.24   866795036038725   23/2048 (1%)   0/256 (0%)   4/512 (0%)   208/16384 (1%)
```

Figure 2.5 – Program debug

3. PROGRAM INTERFACE APPEARANCE

The application is divided into the following areas: *main menu*, *toolbar*, *status bar*, *main area*.

The main menu contains options for interaction with the application.

The toolbar duplicates the most frequently used menu items, can be hidden (menu item ***View – Toolbar***).

The status bar displays information about connected device and necessary recourses for the project performance.

The main area of the application contains:

- Project flowchart editor.
- Function blocks (functions) diagram editor for a particular block of the flowchart.
- Section with elements for creating a project diagram (between editors).
- Additional tabs – issues, build output, break points (displaying is controlled with the ***View** menu)*.

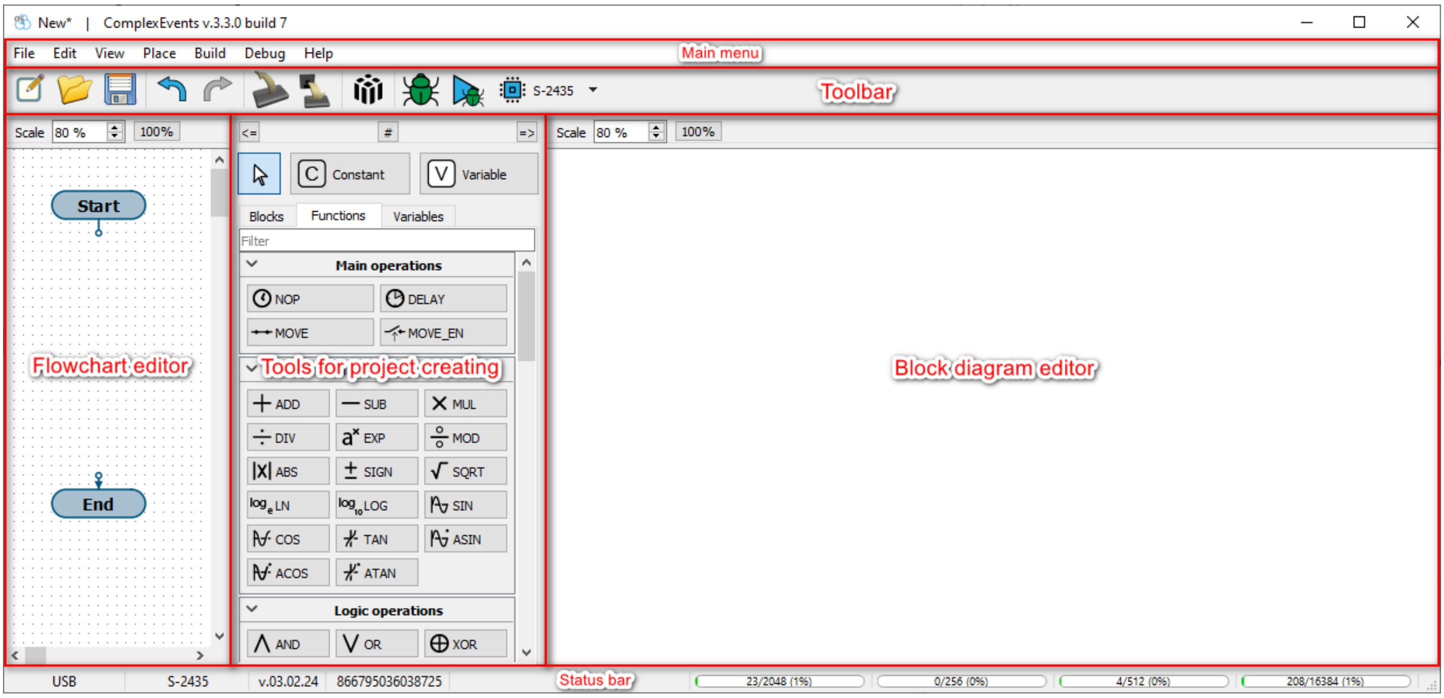


Figure 3.1 – Redactor appearance

4. CREATING AND EDITING THE PROGRAM

4.1 General information

Program flowchart, loaded to device, is drawn up using graphic elements.

First, a flowchart is drawn up in the editor on the left side of the application. Flowchart is a general algorithm of the program, which consists of blocks (steps) interconnected by lines indicating the direction of the execution sequence. The following blocks are supported:

- *Start* – beginning of the program, is always one-off present in the flowchart.
- *End* – ending of the program, is always one-off present in the flowchart.
- *Action* – data processing block.
- *Condition* – data processing block with condition, allows continuing the program in one of two ways. This block allows changing the sequence of program execution for programming conditions and loops.

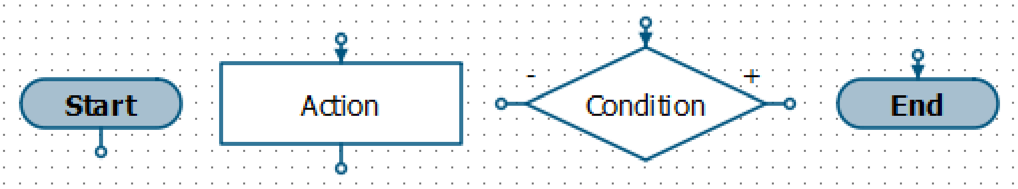


Figure 4.1 – Appearance of Start, Action, Condition and End blocks

Beginning with *Start* block, the blocks are executed one after another in a user-defined sequence (with the help of lines). The program reaches the block *End* - means the end of processing this loop. Loops run endlessly one after the other, from the *Start* block to the *End* block.

On the right side of the application, a data block diagram is drawn up for the particular block (*Action* or *Condition*) from the left part. This diagram consists of interconnected function blocks (functions), constants and variables.

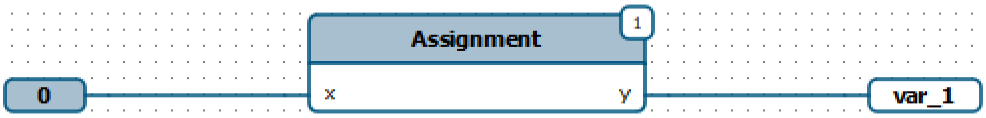


Figure 4.2 – Appearance of Constant, Function and Variable elements

The function block diagram is inherently similar to the CFC (Continuous Function Chart) programming language, which is used for programming PLCs (Programmable Logic Controllers).

4.2 Description of elements in the Flowchart

4.2.1 Start and End blocks

These blocks indicate the beginning and ending of the program. They are one-off present on the diagram and cannot be deleted.

4.2.2 Action block

This block is used to describe one or more functions. The block has one input and one output, which allow placing it in the flowchart and show the direction of the program run.

After execution of the last function of the *Action* block, the program proceeds to the execution of the block which is connected to the output.

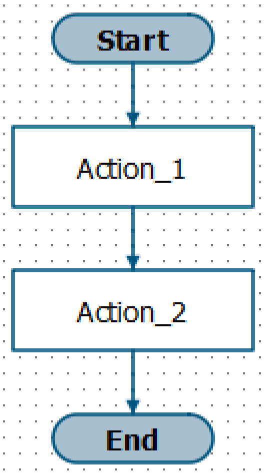


Figure 4.3 – Example of including Action block into flowchart

4.2.3 Condition block

This block is used to describe decision points in the program depending on the conditions specified by the user. As in the Action block, within the Condition block, one or more functions are described. The block has one input and two outputs: «Output +» and «Output -», which allow placing it on the flowchart and show the direction of the program run.

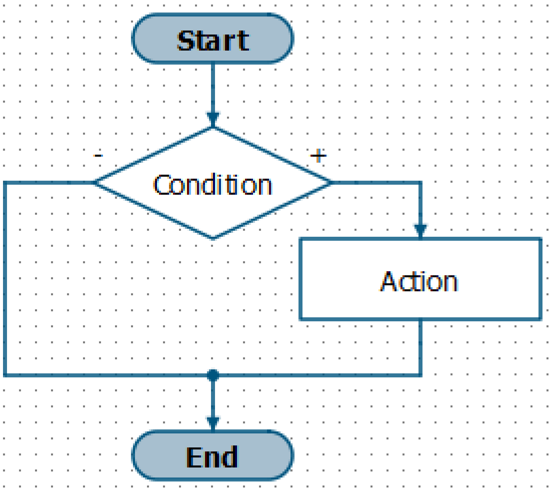


Figure 4.4 – Example of including Condition block into flowchart

A distinctive feature of the block is the presence of the *result* system variable, which is located inside the block (on the right side of the editor). The variable result cannot be deleted or copied.

For the block execution, the *result* variable must contain *True* or *False* condition.

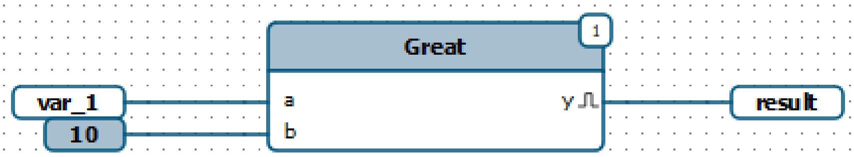


Figure 4.5 – Example of connecting the result variable inside the Condition block

After execution of the last function of the Condition block, the program checks the value of the *result* variable, and if the value is *True*, then the program proceeds to the execution of the block connected to the «Output +», if the value is *False*, the program proceeds to the execution of the block connected to the «Output -».

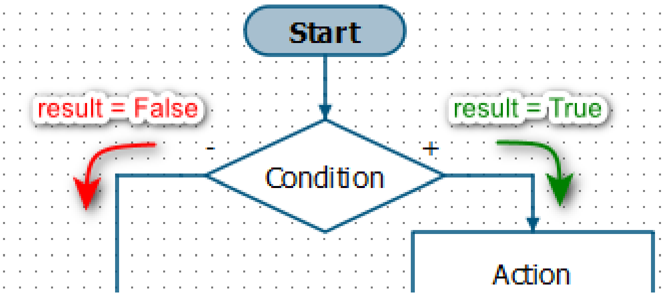


Figure 4.6 – Decision point and program execution depending on the value of the result variable

4.3 Creating a Flowchart

As mentioned above, the *Start* and *End* blocks are one-off present in the program, they cannot be deleted and copied. Only their position can be changed on the flowchart.

The *Action* and *Condition* blocks can be added to the flowchart in required numbers. For that go to the *Blocks* tab on the panel in the middle part of the application and use one of two ways to move blocks to the flowchart:

- Selection by the first click, placement by the second click.
  Left-click on the required block in the *Main blocks* group, then move the cursor to the flowchart editor (in the left part) and left-click again, the selected block will be added to the flowchart.
- Drag'n'drop.
  Point to the required block in the *Main blocks* group, then "grab" it (hold down the left mouse button) and move the cursor to the flowchart editor (in the left part). "Release" the block (release the left mouse button), the selected block will be added to the flowchart.

To cancel adding a block, press the **Esc** key, or click the **Arrow** button. Blocks can be assigned a name and description, for this it is needed to double-click on the block or right-click and select the **Properties** menu item (duplicates the main menu item **Edit - Properties**). Enter the appropriate parameters in the appeared dialog box. Block description is displayed when it is hover the cursor over it.

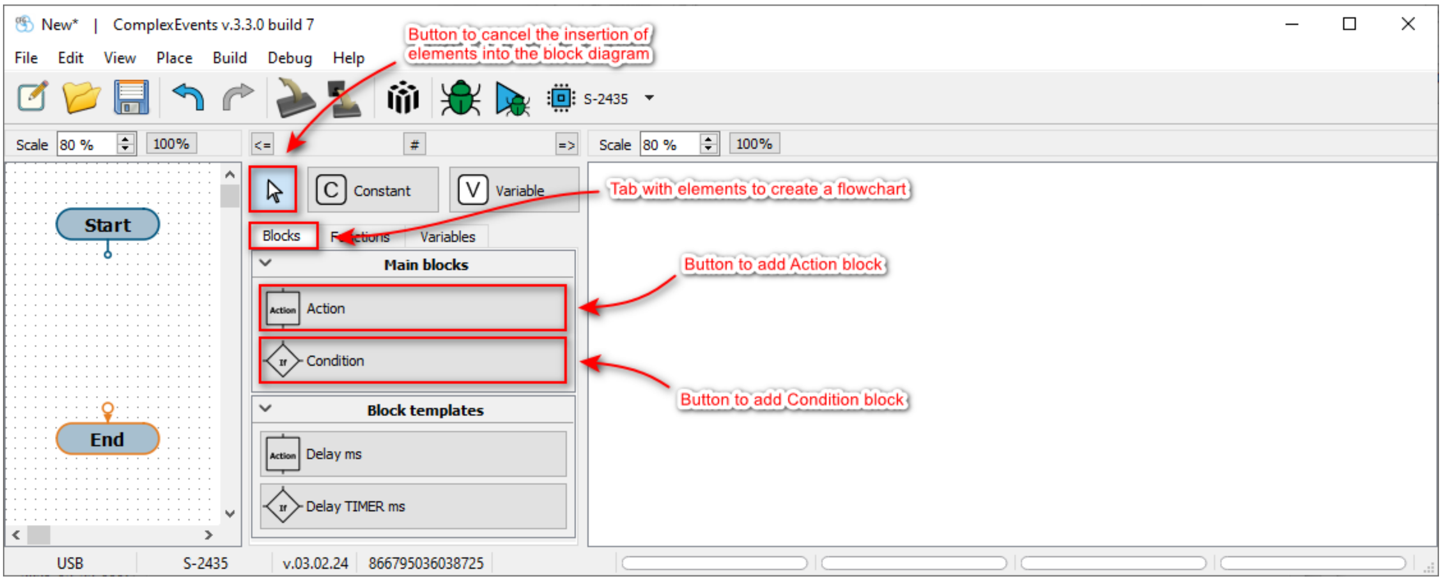

New* | ComplexEvents v.3.3.0 build 7
File Edit View Place Build Debug Help
S-2435
Scale 80 % 100%
<=
#
=>
Scale 80 % 100%
Start
End
Constant
Variable
Blocks
Functions
Variables
Main blocks
Action
Condition
Block templates
Delay ms
Delay TIMER ms
USB
S-2435
v.03.02.24
866795036038725
Button to cancel the insertion of elements into the block diagram
Tab with elements to create a flowchart
Button to add Action block
Button to add Condition block

Figure 4.7 – Elements for creating a flowchart

The order of blocks execution is determined by the user with the help of connecting lines. Connected pins form a chain. To connect two pins, it is needed to click on the first pin, a line will appear, then click on the second pin. The direction of the lines can be changed by clicking in the right places during the creation process (after clicking on the first pin). A pin can be connected to an existing chain by clicking on the pin first, then on the chain. To cancel creating a chain, press the **Esc** key. A chain can contain multiple outputs and one input. The case where the chain contains more than one input will result in a compilation error. Block inputs are indicated by an arrow. All pins of blocks in the flowchart must be connected.

Each block added by the user contains its own function block diagram (on the left side of the application).

4.4 Description of Functional Blocks Scheme Elements

4.4.1 Constants and variables

In order for the user to set the required initial state of the program, as well as to receive and process the results of the program, the elements *Constant* and *Variable* are used.

Constant is a constant value which is determined at the stage of drawing up the program and is not changed during the execution of the program.

Variable is a named memory area which is used to write, read, and store various values. The value of a variable is determined when the program is compiled and can subsequently change constantly during the execution of the program.

Characteristics of constants and variables:

Name	**Applicability**	**Description**
Name	Variable	A text name that allows you to refer to the value of each specific variable (read it or change it). The maximum name length is 16 characters.
Type	Variable Constant	The range of valid values and the size allocated in memory for a constant or variable depend on the type. **"Int32"** Integer number from −2147483648 to 2147483647. Occupies 4 bytes in the memory. **"Float"** Floating-point number. Range of values without loss of precision for numbers with no more than 7 significant digits. For example, -9999999 to 9999999 or -0.999999 to 0.999999. Occupies 4 bytes in the memory. **"Bool"** Boolean (logical) type that has two values *True* or *False.* Occupies less than 1 byte in memory. Type conversion in the Complex Events: "INT32 to FLOAT" and back "FLOAT to INT32": Only the integer part and the sign are transferred: int32 "-123" is converted to float "-123.0"; float "5.99" is converted to int32 "5". "FLOAT/INT32 to BOOL": *True* – values **not** equal to "0" (or "0.0"); *False* – values equal to "0" (or "0.0"). «BOOL to FLOAT/INT32»: *True* is converted to "1" (or "1.0"); *False* is converted to "0" (or "0.0").
Value (displaying)	Variable Constant	Type of displaying the number value with the INT32 type during debugging (during calculations the displaying does not anyhow affect the result). **"DEC"** - Number "26952" in the decimal system '26952' **"HEX"**- Number "26952" in the hexadecimal system '0x6948' **"BIN"** - Number "26952" in the binary system '0b0110100101001000' **"ASCII"** - Number "26952" as ASCII text 'Hi'
Value	Variable Constant	A value that the constant or variable will take when the program starts.

Write access (during debugging)	Variable	If the flag is set, then while the debugger is running, the user can change the value of the variable manually without stopping the program.

4.4.2 Function blocks

Function block (*function*) is a block that has a certain number of inputs and outputs. The inputs of the function receive data (for example, from other blocks), then this data is processed and generated into those, which is sent to the outputs of this function (these outputs can be connected to the inputs of other function blocks). The functions are executed one after the other. The order is determined by the sequence number.

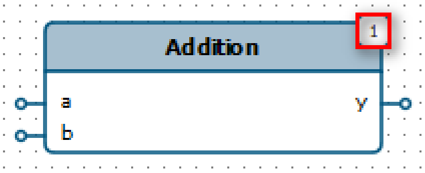


Figure 4.8 – Sequence number for function execution

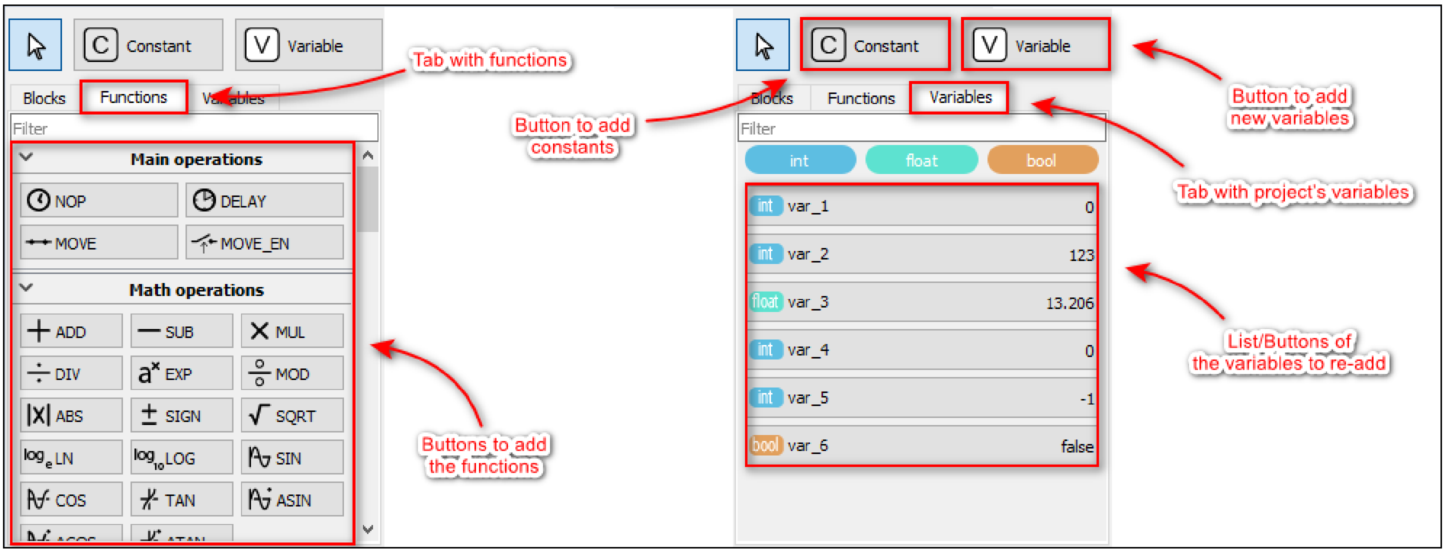
4.5 Creating a Function block diagram

In selecting a block in the flowchart (on the left side), its function diagram appears on the right side of the application. A function diagram can contain function blocks (functions), variables and constants interconnected. Lines in the function block diagram indicate the direction of data flow.

To add a function to the diagram, go to the *Functions* tab on the panel in the middle part of the application and use one of two ways to move blocks to the flowchart:

- Selection by the first click, placement by the second click.
  Left-click on the required function, then move the cursor to the function block diagram editor (in the right part) and left-click again, the selected function will be added to the diagram.
- Drag'n'drop.
  Point to the required function, then "grab" it (hold down the left mouse button) and move the cursor to the function block diagram editor (in the right part). "Release" the block (release the left mouse button), the selected function will be added to the flowchart.

To cancel adding a function, press the **Esc** key, or press the **Arrow** button. Constants and variables are added in the same way as functions, for this there are **Constant** and **Variable** buttons. When using the **Variable** button, a new variable will always be added to the diagram. To add a previously created variable to the diagram, go to the *Variables* tab and select the one you need from the list. This tab displays all user-added variables. The same variable can be added to different function block diagrams.



Tab with functions

Button to add constants

Buttons to add the functions

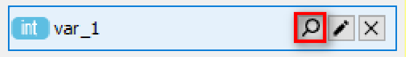
Button to add new variables

Tab with project's variables

List/Buttons of the variables to re-add

Figure 4.9 – Elements for creating a function block diagram

To quickly find variable on the diagram, you need to click the ***Search*** button for the required variable. You can read more about the search interface in the "Variables search" section.



Function inputs are always located on the left part and outputs are always located on the right part. Connection of the elements pins of the functional diagram is carried out as in the flowchart. Connected pins form a chain. There can be only one function output in a chain, only one variable or constant. If there is an output in the chain, then the constant should not be connected to this chain. It is not necessary to connect all function pins to the chain.

The functions are executed sequentially one after the other. In the upper right corner, the sequence number of the function is displayed, which determines the order of execution. The lower the sequence number, the sooner the function is executed. To change the sequence number is possible by double-clicking on the function or by right-clicking and selecting the **Properties** menu item. Then in the appeared dialog box, change the value of the *Execution index.* In this dialog box, it is possible to change other parameters of the functions, if they are provided for it.

The editor provides mechanism for automatic numbering of functions, detailed description is given in the section "Automatic numbering of function blocks".

4.6 Flowchart description elements

To improve the information content of the flowchart, the editor provides the following mechanisms:

- Adding/changing the *Name* and *Description* of the block on the left side of the flowchart;
- Adding *Text* and *Rectangle* elements to the left or right side of the flowchart;

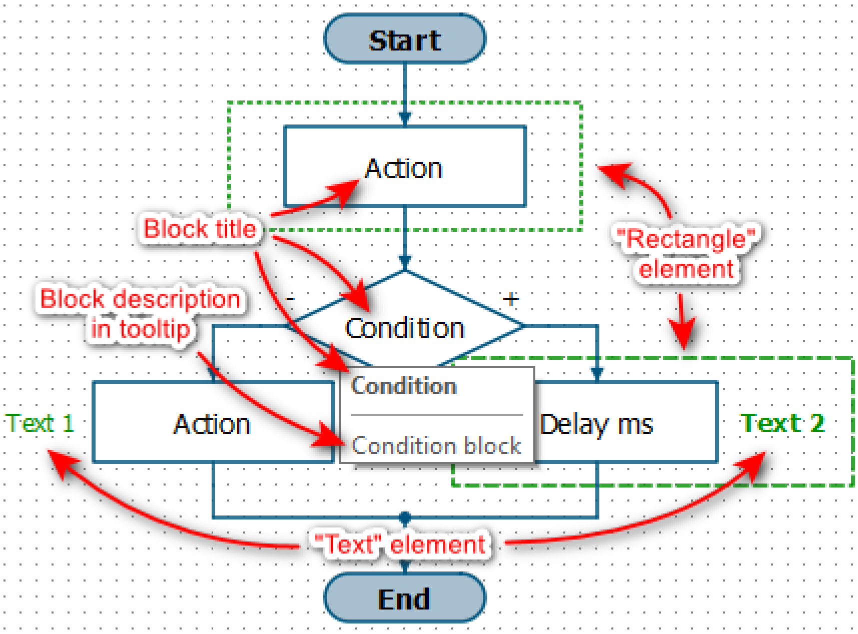


Figure 4.10 - Undo and redo control buttons

4.6.1 Name and Description of the block

Each block on the left side of the flowchart has a *Name* and *Description*. The *Name* is displayed on the flowchart inside the block and in the title of the tooltip that appears when you point to the block. The *Description* of the block is displayed only in the tooltip. To change the *Name* or *Description*, you need to right-click on the block and select ***Properties*** from the context menu.

4.6.2 Text

To place text boxes on the diagram, you can use the *Text* element. The element can be placed both on the left and on the right side. To place it, select the menu item ***Place - Text***. The following parameters can be configured for text boxes:

- Font size
- Style of writing (bold, italic, underlined)
- Vertical alignment (left, center, right)

The text color is determined by the global color scheme setting (see the "Color scheme" section).

4.6.3 Rectangle

To place frames on the diagram, you can use the *Rectangle* element. The element can be placed both on the left and on the right side. To place it, select the menu item **Place - Rectangle**. For frames, only the line type can be configured.

The frames color and their thickness are determined by the global color scheme setting (see the "Color scheme" section).

4.7 Undo/Redo

Undo and redo operations are available in the editor.

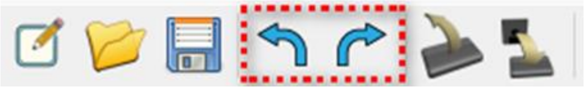


Figure 4.11 - Undo and redo buttons

The memory stores the last 100 user actions. The program controls all basic user manipulations: creating deleting/moving of block, lines, variables and constants, changing the names of variables, changing the values of variables and constants, changing the properties of blocks and functions, etc. This makes working in the editor much easier.

4.8 Automatic numbering of function blocks

The editor provides mechanism for automatic numbering of function blocks (menu item ***Edit*** - ***Number function blocks***). This function allows you to quickly number blocks depending on their location on the diagram.

Two numbering algorithms are available:
Down then right - column numbering from top to bottom, from left to right.
Right then down – line numbering from left to right, from top to bottom.

The numbering mechanism focuses only on the visual arrangement of the circuit elements and does not adjust its work depending on the order of connecting the elements or their functionality.

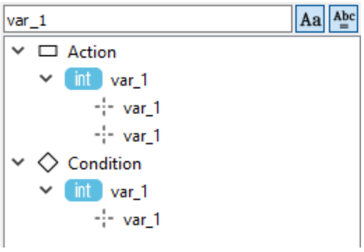
> ⚠ *The automatic numbering mechanism is designed for fast draft numbering. After it, it is recommended to check the result and make manual adjustments.*

4.9 Variables search

For the convenience of working with variables, you can use the search box, it can be open by two ways:

- In the menu ***View – Variables search***
- In the list of used variables on the *Variables* tab, click the ***Search*** 🔍 button

The area with the list of variable usage points will open. When you double-click on any of the mentions, the editor centers the viewport at the required location in the diagram. If you open the search with the ***Search*** button, then when you open it, the name of the selected variable will be indicated in the search box.



var_1

- Action
  - int var_1
    - var_1
    - var_1
- Condition
  - int var_1
    - var_1

4.10 Operation with files

At Complex Events startup, an empty project is created; only the *Start* and *End* blocks are present on the block diagram. The menu item ***File – New*** (the 📝 button on the toolbar)*, creates a new project with a simple block diagram.*

The created project can be saved to a file (the menu items ***File – Save***, ***File – Save as*** or the 💾 button on the toolbar), open from a file (the menu item ***File – Open*** or the 📂 button on the toolbar). For quick access to recent projects, there is a list of recently saved and opened files, using the menu item ***File – Recent files***.

Block diagrams can be saved and opened to/from a file. For that select the required block on the block diagram and use the menu items ***Edit – Import***, ***Edit – Export*** (or right-click and select similar menu items from the list). For quick access, blocks can be saved to templates (the menu item ***Edit – Send to templates***). Saved templates are available in the *Templates* group on the *Blocks* tab in the panel in the middle part of the application.

5. PROGRAM BUILD

Building the program is activated through the menu item ***Build – To build*** (or clicking the button on the toolbar). Building includes:

- compilation of the project (can be compiled on its own, the menu item ***Build- Compile***)
- building the output program file for uploading into the device
- checking device configuration
- displaying errors and warnings
- displaying resources needed to build the program.

During compilation, the program is built and the necessary resources are allocated. If the program contains errors or the device lacks the necessary resources for building, then the corresponding messages are added to the *Issues* tab (automatically opens).

In building the output file, a file uploaded to the device is generated, it contains a program executed by the Complex Events interpreter and the source file of the project. If the size of the file exceeds the allowable size, the corresponding messages are added to the *Issues* tab.

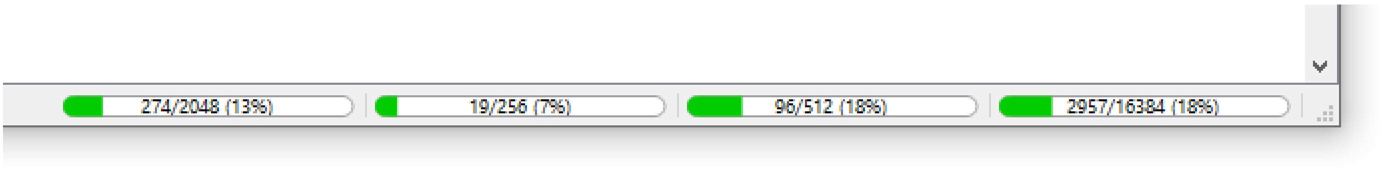
In order for program to work properly, Complex Events support must be enabled in the device configuration, and if the program uses functions that work with the device's peripherals, then this peripheral must be configured accordingly. If the configuration contains incorrect settings, the corresponding messages are added to the *Issues* tab

The *Issues* tab displays error and warning messages. The tab is accessed through the **View - Issues** menu item. At left double-clicking on the message, the application shows the problematic element: shows in the graphical editor, displays the required device configuration tab, etc.

The *Build output* tab displays the resources consumed by the program. The tab is accessed through the **View – Build output** menu item. Program resources:

- Program code – 2048 bytes
- Variables
  bool - 256 pcs
  int / **float** - 512 bytes (4 bytes for each)
- Total size of the file uploaded to the device is 16384 bytes.

Information about the resources used is located on the right side of the status bar:



274/2048 (13%) 19/256 (7%) 96/512 (18%) 2957/16384 (18%)

Figure 5.1– Program resources

6. PROGRAM DEBUG

All debug options are available when device is connected.

6.1 Start debugging

To debug the program on the device, select the ***Debug – Start debugging*** or click the button on the toolbar. After that, the application will perform the following actions:

- Builds the project. If the build results error messages, debugging will be interrupted.
- If the build results messages about incorrect configuration of the device, it will be prompted to interrupt debugging (if other is not selected in the settings).
- Proposes to upload the configuration to the device (if other is not selected in the settings). If accepted, configuration will be uploaded, after that device will be rebooted.
- Uploads the program to the device.
- Enters debug mode, stopping at the first executable function.

6.2 Debugging for a running program

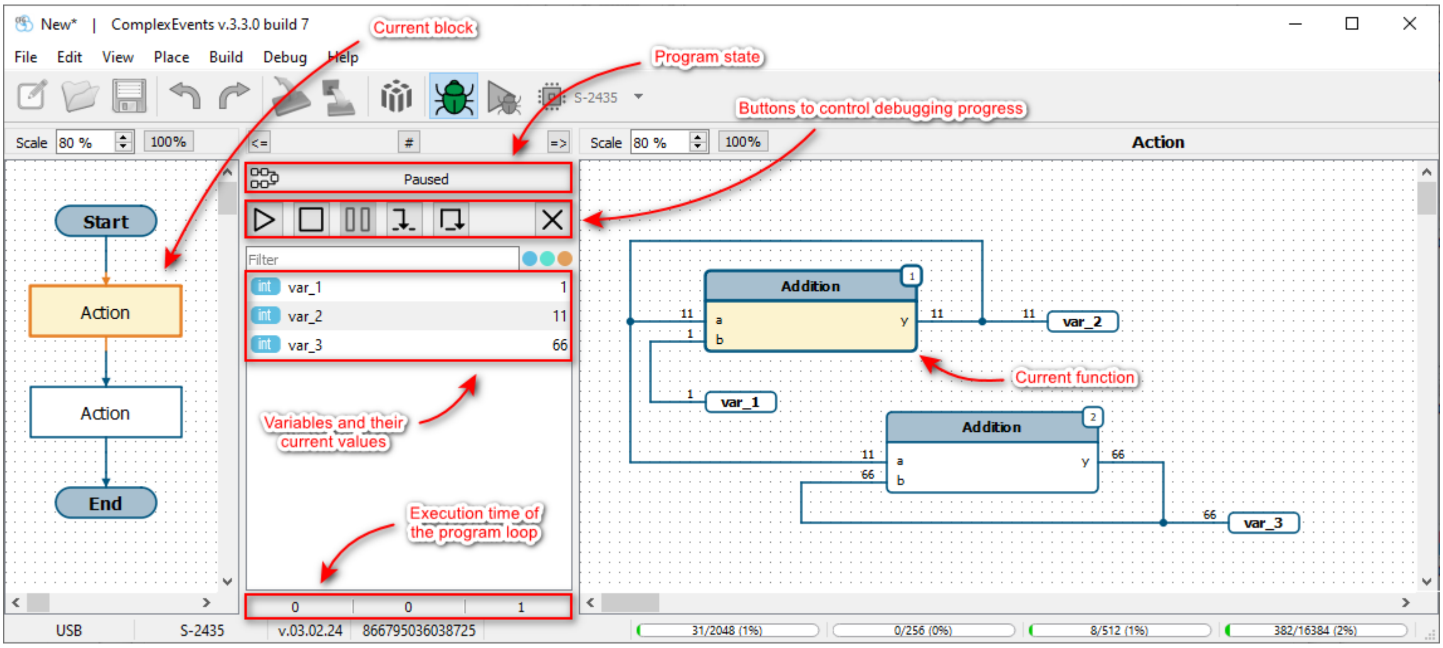
To debug the already working device, select the ***Debug – Connect to the running device*** or click the button on the toolbar. After that, the application will perform the following actions:

- Proposes to download the configuration from the device (if other is not selected in the settings).
- Downloads the program from the device and opens it in the editor.
- Starts building the program, if there are error messages, the connection will be interrupted.
- Enters debug mode, while the program will continue to run.

6.3 Operation in debug mode

In debug mode, the application prohibits changing the current diagram. The panel with diagram elements in the middle part is replaced by a debug panel.

The debug panel displays: the program status bar, buttons to manage the program, a list of variables, information about the execution time of the program loop.



int	var_1	1
int	var_2	11
int	var_3	66

Current block

Program state

Buttons to control debugging progress

Current function

Variables and their current values

Execution time of the program loop

0	0	1

Figure 6.1 – Appearance of the editor in debug mode

6.3.1 Program status bar

The program can be in the following states:

- *No program* – no program has been loaded into the device, or has been loaded with an error.
- *Error* – an error occurred during the execution of the program.
- *Stopped* – the execution of the program is stopped. At the next start of the program, the variables will be initialized and launched from the first function.
- *Download* – recording program to the device.
- *Paused* – the application has been paused. At resuming operation, the program will continue execution with the current function. In this mode, the current function and its block are highlighted in the editor windows.
- *Execution* – the device is executing the program.

After exiting the debug mode, the device will start or continue the program (depending on the current state), but only if the program was not in the modes: *No program* or *Error*.

6.3.2 Controlling program execution

To control the program execution, there are special buttons under the program status bar (these buttons are duplicated in the **Debug** menu):

- ***Continue*** ▷ – if the program is in the *Stopped* state - starts the program for execution, if the program is in the *Paused* state - continues working with the current function.
- ***Stop*** □ – stops the program execution (puts it in the *Stopped* state).
- ***Pause*** ⏸ – pauses the program execution (puts it in the *Paused* state).
- ***Step*** ⤵ – performs one function and pauses at the next one.
- ***Cycle*** ↻ – executes all functions until it goes to the beginning of the program, pauses at the first function.
- ***Send user command*** ▭ – opens a dialog for sending data to a user command. This command is displayed in the middle panel only if the program uses the **CMD** function.
- ***End debug*** ☒ – ends debugging the program, takes the device out of debug mode, and switches the editor to normal mode.

To stop the program before executing a specific function, the application provides breakpoints. To set and remove a breakpoint, it is needed to right-click on the required function and select the **Toggle breakpoint** menu item (duplicated in the **Debug** menu). Setting breakpoints is also available in the project diagram editing mode. The device physically supports up to 8 breakpoints. The list of current breakpoints can be viewed on the *Breakpoints* tab (opened with the **View - Breakpoints** menu item). Through this tab, breakpoints can be deleted by selecting the required ones and pressing the **Del** key. At double-clicking on a breakpoint in the list, the application will show the function on which it is set.

6.3.3 Viewing diagram data values

In debug mode, the application in the diagram displays the current values at the inputs and outputs of the functions (directly above each pin). Data is read from the device with a period specified in the application settings.

Under the buttons in the middle part of the program there is a list of used variables with their current values. This list can be filtered by variable name or data type.

6.3.4 Execution time of the program loop

The program runs cyclically. The time of one loop may vary, depending on the state of the program data or on the degree of device workload. To estimate the execution time of the program, the device measures the loop period. The panel in the middle part of the program, under the list of variables, displays the minimum, maximum and average loop value in milliseconds.

6.4 Writing and reading a program without debugging

The editor allows writing the program to the device without entering debug mode. To do this, there is a menu item ***Debug – Write program to device*** (or the button on the toolbar). By analogy, it is allowed reading the program from the device and opening it in the editor with ***Debug – Read program from device*** menu item (or the button on the toolbar).

It is important to remember that with this method of loading the program, the editor does not check the compatibility of the device configuration with the loaded program.

7. SETTINGS

Settings are opened through the menu item ***File – Settings***. The settings are divided into the following groups: ***Main***, ***Debug***, ***Color scheme***.

7.1 Main

In this window, it is possible to configure the interval for automatic saving of project changes – the *Auto save* field. The following values are available:

- *10 seconds*
- *30 seconds*
- *1 minute*
- *5 minutes*
- *10 minutes*
- *No* (Autosave disabled)

7.2 Debug settings

In this window, following parameters are configured:

- *Data update period* is the time period in milliseconds, with which debug information is read if the device is connected via USB.
  The minimum value is 100 ms.
- *Data update period for low-speed connection* is the time period in milliseconds with which debug information is read if the device is connected via Bluetooth or RCS server.
  The minimum value is 1000 ms.
- *Run debugger with incorrect device configuration* – possible values: *Ask* (by default), *No*, *Yes*
- *Upload configuration before starting debugger* – possible values: *Ask* (by default), *No, Yes*
- *Download configuration before connection to device* – possible values: *Ask* (by default, *No*, *Yes*

7.3 Color scheme

In this window, it is possible to customize the color scheme of the graphical elements of the editor.

The field *Default color scheme* allows selection one of the standard color schemes. To apply the selected color scheme it is necessary to select it in the dropdown list and click on the ***Apply*** button.

For manual editing of the editor color schemes it is possible to use group of settings described below.

Options in the *Block diagram* group refer to the interface located on the left side of the editor. Options in the *Function diagram* group refer to the interface located on the right side of the editor. Options in the *General* group refer to general graphic items.

Field *For state* defines the state in which the graphic elements are located. There are four possible states, the first two are general, other two are related to debug mode:

- *Normal* – normal state when no item is selected.
- *Selected* – when user selected a given item, one or more.
- *Current* – in debug mode, the program is paused on this element.
- *Current selected* – in addition to the previous state, the item is selected.

8. APPENDIX

8.1 Shortcut keys

Working with project:	
CTRL + N	Create a new project
CTRL + O	Open project from file
CTRL + S	Save project to file
Build:	
CTRL + B	Build a project
CTRL + SHIFT + B	Compile the project
Debug:	
F5	Start debugging, continue execution
F2	Finish debugging
F10	Perform one loop
F11	Perform one function
F9	Toggle breakpoint

8.2 Event codes Complex Events

With function running, device can generate events with following codes (event_code):

Code (HEX)	Code (DEC)	Text for SMS	Description
0xA056	41046	CMPLXEVNT_A	Complex Events. Custom event #1
0xA057	41047	CMPLXEVNT_S	Complex Events. Custom event #2
0xA058	41048	CMPLXEVNT_F	Complex Events. Custom event #3
0xA22F	41519	C_CVNT_U	Complex Events. Program update.

8.3 Function block library

List of function blocks:

Name	#	Description	Number of operands				Size, bytes	Operands type
			IN	OUT	INT	CONST		
Main operations								
NOP	3	No operation (delay)	-	-	-	-	1	-
DELAY	78	Delay	1	-	-	-	3	int32
MOVE	4	Move assignment	1	1	-	-	5	Any
MOVE_EN	5	Move conditional assignment	2	1	-	-	7	Any
~~TO_FLOAT~~	*6*	*Convert to float*	*1*	*1*	*-*	*-*	*5*	*int32*
~~FROM_FLOAT~~	*75*	*Convert from float*	*1*	*1*	*-*	*-*	*5*	*float*
Math operations								
ADD	7	Addition	2	1	-	-	7	float|int32
SUB	8	Subtraction	2	1	-	-	7	float|int32
MUL	9	Multiplication	2	1	-	-	7	float|int32
DIV	10	Division	2	1	-	-	7	float|int32
EXP	11	Exponentiation	2	1	-	-	7	float|int32
MOD	12	Modulo division	1	1	-	-	5	float|int32
ABS	13	Absolute value	1	1	-	-	5	float|int32
SIGN	14	Definition of sign	1	1	-	-	5	float|int32
SQRT	15	Square root	1	1	-	-	5	float
LN	16	Natural logarithm	1	1	-	-	5	float
LOG	17	Common logarithm	1	1	-	-	5	float
SIN	18	Sine	1	1	-	-	5	float
COS	19	Cosine	1	1	-	-	5	float
TAN	20	Tangent	1	1	-	-	5	float
ASIN	21	Arcsine	1	1	-	-	5	float
ACOS	22	Arccosine	1	1	-	-	5	float
ATAN	23	Arctangent	1	1	-	-	5	float
Logical operations								
AND	24	Logical AND	2	1	-	-	7	bool
OR	25	Logical OR	2	1	-	-	7	bool
XOR	26	Logical exclusive OR	2	1	-	-	7	bool
NOT	27	Logical NOT	1	1	-	-	5	bool
Bitwise operations								
BAND	28	Bitwise AND	2	1	-	-	7	int32
BOR	29	Bitwise OR	2	1	-	-	7	int32
BXOR	30	Bitwise exclusive OR	2	1	-	-	7	int32
BNOT	31	Bitwise NOT	1	1	-	-	5	int32
BSHL	32	Bitwise left shift	2	1	-	-	7	int32
BSHR	33	Bitwise right shift	2	1	-	-	7	int32
CODER	34	Coder	N	1	-	-	4+2*(N+1)	int32
DECODER	35	Decoder	1	N	-	-	4+2*(N+1)	int32
Relational operations								
EQ	36	Equal	2	1	-	-	7	float|int32
NE	37	Not equal	2	1	-	-	7	float|int32
GT	38	Greater	2	1	-	-	7	float|int32
GE	39	Greater or equal	2	1	-	-	7	float|int32
Selection and limit operations								
SEL	40	Selection value	3	1	-	-	9	float|int32
MAX	41	Maximum value	2	1	-	-	7	float|int32
MIN	42	Minimum value	2	1	-	-	7	float|int32
LIMIT	43	Limitation	3	1	-	-	7	float|int32
MUX	44	Multiplexer	1+N	1	-	-	4+2*(N+2)	float|int32
DMUX	45	Demultiplexer	2	N			4+2*(N+2)	float|int32
APPERTURE	94	Value change control	2	1	1	-	9	float|int32
Triggers, generators and counters								
SR	46	Set-Reset trigger	2	1	-	-	7	bool

RS	47	Reset-Set trigger	2	1	-	-	7	bool
TT	48	Toggle (T-trigger)	1	1	1	-	7	bool
TP	49	One pulse generator	2	2	2	-	11	bool
BLINK	50	Pulse generator	3	3	1	-	15	bool
TON	51	On delay timer	2	2	2	-	13	bool
TOFF	52	Off delay timer	2	2	2	-	13	bool
RISING	53	Rising edge detector	1	1	1	-	7	bool
FALLING	54	Falling edge detector	1	1	1	-	7	bool
CNT	55	Counter	5	3	2	-	21	bool
RAND	56	Random number generator	-	1	-	-	3	int32
PWM	57	PWM generator	2	2	1	-	11	int32
Special functions								
EVENT	58	Event generator	2	-	1	2	9	int32
CMD	59	Command from device	-	6	-	-	13	int32
FLEX	60	Reading a value from FLEX table	-	1	-	3	6	int32
USER_PARAM	61	Writing a value to user parameter	2	-	-	1	6	int32
SMS	62	Send SMS	1	1	1	3+N	10+N	bool
USER_SMS	79	Send user SMS	1+N	1	1	M+L	7+2·N +M+N	bool
RECV_SMS	80	SMS received	0	1	0	1+N+M	4+N+M	bool
CALL	63	Make a call	1	1	1	2	9	bool
CAM	64	Take a picture	1	1	1	-	7	bool
GEOZONE	65	Geofence	5	1	1	1	16	float|int32
CALENDAR	76	Calendar	2	7	-	-	19	int32
INFO	95	About device	-	2	-	-	5	int32
IMEI	96	Modem IMEI	-	2	-	-	5	int32
ICCID	97	SIM card ICCID	1	2	-	-	7	int32
IMSI	98	SIM card IMSI	1	2	-	-	7	int32
LOG_MSG	106	Send a message to the log	1+N	0	1	M	5+2·N+M	bool
Peripheries								
INPUT	66	Input	1	2	-	1	8	int32
OUTPUT	67	Output	1	-	1	1	6	int32
HYGRO	68	Hygrometer	-	2	-	1	6	float
ACCEL	69	Accelerometer	-	9	-	-	19	int32
ECODRIVE	70	EcoDrive	-	9	-	-	19	int32
ONEWIRE_KEY	71	1-Wire key	-	3	-	-	7	int32
RFID	72	RFID	-	5	-	-	11	int32
TACHOGRAPH	73	Tachograph driver	-	7	-	1	16	int32
GUARD	74	Security mode	1	2	1	1	9	float|int32
CRASH_FILE	77	Accident file generating	2	3	2	-	15	bool
PWRSAVE	81	Energy saving control	6	-	-	-	13	bool
Access Functions to Digital Ports								
~~*RXD_GET*~~	*82*	*Read value from RXD buffer*	*2*	*1*	*-*	*1*	*8*	*float|int32*
RXD_CMP	83	Data search in RXD buffer	1	1	-	1+N	6+N	int32
RXD_STR2INT	84	Convert string from RXD buffer to integer number	1	1	-	-	5	int32
RXD_STR2FLOAT	85	Convert string from RXD buffer to float	1	1	-	-	5	float
RXD_CHECKSUM	86	Verify checksum in RXD buffer	3	1	-	2	11	bool
TXD_INIT	87	TXD buffer initialization	1	-	-	1+N	4+N	bool
~~*TXD_SET*~~	*88*	*Write value to TXD buffer*	*4*	*-*	*-*	*1*	*10*	*float|int32*
TXD_CHECKSUM	89	Write checksum to TXD buffer	4	-	-	2	11	bool
TXD_GET	90	Read value from TXD buffer	2	1	-	1	8	float|int32
RS_TRANS	91	Request/response via serial port	3	3	-	2	15	bool

RS_SEND	92	Transmit data to serial port	2	1	-	1	8	bool
RS_RECV	93	Receive data from serial port	2	3	-	2	14	bool
RXD_GET	107	Read value from RXD buffer	2	N	-	1	7+2·N	float|int32
TXD_SET	108	Write value to TXD buffer	3+N	-	-	1	9+2·N	float|int32
MODBUS_READ	109	Reading data via Modbus RTU protocol	1	2+N	1	7	9+2·N + 10	float|int32 |bool
MODBUS_WRITE	110	Writing data via Modbus RTU protocol	1+N	2	1	7	9+2 N + 10	float|int32 |bool

8.3.1 Main operations

8.3.1.1 NOP – no operation

The instruction does nothing and has no inputs or outputs.

8.3.1.2 DELAY – delay

	Signature	Type	Description
Inputs	period	int32	The duration of the delay in ms.

*The block delays the operation of Complex Events for the time specified by the period input. Therefore, when the program is running and debugging, it is not possible to see the current remaining delay time. But if pause the execution at the time of the delay execution, the debugger will highlight the required **DELAY** block on the diagram.*

8.3.1.3 MOVE - assigment

	Signature	Type	Description
Inputs	x	float, int32, bool	Input operand. The value at the input *x* is copied to the value at the output *y*
Outputs	y	float, int32, bool	Output operand

*The block type is determined by the type of the value at the input **x***

8.3.1.4 MOVE_EN – condition assigment

	Signature	Type	Description
Inputs	x	float, int32, bool	Input operand
	enable	bool	Copy condition. The value at input *x* is copied to the value at output *y* if enable = *true*, otherwise *y* is not changed.
Outputs	y	float, int32, bool	Output operand

*The block type is determined by the type of the value at the input **x***

8.3.1.5 (deprecated) ~~TO_FLOAT~~ – convert int32(IEEE754) to float

> ⚠ *Function is hidden, starting for the editor version v.3.3.0.*

	Signature	Type	Description
Inputs	x	int32	Input operand
Outputs	y	float	Output operand

The block interprets the integer value received at the input *x* as a number with floating point written in accordance with the IEEE754 standard and translates it into a more readable and computable representation.

INT32		FLOAT
1095977927	**=**	**13.206**

8.3.1.6 (deprecated) ~~FROM_FLOAT~~ – convert float to int32(IEEE754)

> ⚠ *Function is hidden, starting for the editor version v.3.3.0.*

	Signature	Type	Description
Inputs	x	float	Input operand
Outputs	y	int32	Output operand

The block converts a number with floating point received at the input *x* to the integer record format according to the IEEE754 standard.

FLOAT		INT32
13.206	**=**	**1095977927**

8.3.2 Math operations

8.3.2.1 ADD – addition

	Signature	Type	Description
Inputs	*a*	float, int32	Summand 1
	b	float, int32	Summand 2
Outputs	*y*	float, int32	Sum

$$y = a + b$$

⚠ *The block type is determined by the type of the value at the input **a***

8.3.2.2 SUB – subtraction

	Signature	Type	Description
Inputs	*a*	float, int32	Minuend
	b	float, int32	Subtrahend
Outputs	*y*	float, int32	Difference

$$y = a - b$$

⚠ *The block type is determined by the type of the value at the input **b***

8.3.2.3 MUL – multiplication

	Signature	Type	Description
Inputs	*a*	float, int32	Multiplier 1
	b	float, int32	Multiplicand 2
Outputs	*y*	float, int32	Product

$$y = a \cdot b$$

⚠ *The block type is determined by the type of the value at the input **a***

8.3.2.4 DIV – division

	Signature	Type	Description
Inputs	*a*	float, int32	Dividend
	b	float, int32	Divisor
Outputs	*y*	float, int32	Fraction

$$y = \frac{a}{b}$$

⚠ *The block type is determined by the type of the value at the input **a***

8.3.2.5 EXP – exponentiation

	Signature	Type	Description
Inputs	*a*	float, int32	Base
	b	float, int32	Exponent
Outputs	*y*	float, int32	Power

$$y = a^b$$

⚠ *The block type is determined by the type of the value at the input **a***

8.3.2.6 MOD – modulo division

	Signature	Type	Description
Inputs	*a*	float, int32	Dividend
	b	float, int32	Divisor
Outputs	*y*	float, int32	Remainder

$$y = a \,\%\, b$$

⚠ *The block type is determined by the type of the value at the input **a***

8.3.2.7 ABS – absolute value

	Signature	Type	Description
Inputs	*x*	float, int32	Input operand
Outputs	*y*	float, int32	Result

$$y = |x|$$

*The block type is determined by the type of the value at the input **x***

⚠ *The block type is determined by the type of the value at the input **x***

8.3.2.8 SIGN – definition of sign

	Signature	Type	Description
Inputs	*x*	float, int32	Input operand
Outputs	*y*	float, int32	Result

$$y = \begin{cases} x > 0, & 1 \\ x = 0, & 0 \\ x < 0, & -1 \end{cases}$$

⚠ *The block type is determined by the type of the value at the input* ***x***

8.3.2.9 SQRT – square root

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \sqrt{x}$$

8.3.2.10 LN – natural logarithm

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \ln x$$

8.3.2.11 LOG – common logarithm

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \log x$$

8.3.2.12 SIN – sine

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \sin(x)$$

8.3.2.13 COS – cosine

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \cos(x)$$

8.3.2.14 TAN – tangent

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \mathrm{tg}(x)$$

8.3.2.15 ASIN – arcsine

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \ asin(x)$$

8.3.2.16 ACOS – arccosine

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \ \mathrm{acos}(x)$$

8.3.2.17 ATAN – arctanangent

	Signature	Type	Description
Inputs	*x*	float	Input operand
Outputs	*y*	float	Result

$$y = \ \mathrm{atg}(x)$$

8.3.3 Logical operations

8.3.3.1 AND – locigal AND

	Signature	Type	Description
Inputs	*a*	bool	Operand 1
	b	bool	Operand 2
Outputs	*y*	bool	Conjunction

$$y = a \bigwedge b$$

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

8.3.3.2 OR – logical OR

	Signature	Type	Description
Inputs	*a*	bool	Operand 1
	b	bool	Operand 2
Outputs	*y*	bool	Disjunction

$$y = a \bigvee b$$

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

8.3.3.3 XOR – logical exclusive OR

	Signature	Type	Description
Inputs	*a*	bool	Operand 1
	b	bool	Operand 2
Outputs	*y*	bool	Exclusive disjunction

$$y = a \bigoplus b$$

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

8.3.3.4 NOT – logical NOT

	Signature	Type	Description
Inputs	*X*	bool	Input operand
Outputs	*Y*	bool	Negation

$$y = \overline{x}$$

x	y
0	1
1	0

8.3.4 Bitwise operations

8.3.4.1 BAND – bitwise AND

	Signature	Type	Description
Inputs	*a*	int32	Operand 1
	b	int32	Operand 2
Outputs	*y*	int32	Bitwise conjunction

$$y = a \bigwedge b$$

Operand	Value (DEC)	Value (HEX)	Value (BIN)							
			Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
a	150	0x96	1	0	0	1	0	1	1	0
b	85	0x55	0	1	0	1	0	1	0	1
y	20	0x14	0	0	0	1	0	1	0	0

8.3.4.2 BOR – bitwise OR

	Signature	Type	Description
Inputs	*a*	int32	Operand 1
	b	int32	Operand 2
Outputs	*y*	int32	Bitwise disjunction

$$y = a \bigvee b$$

Operand	Value (DEC)	Value (HEX)	Value (BIN)							
			Bit7	Bit6	Bit5	Bit4	Bit7	Bit2	Bit1	Bit0
a	150	0x96	1	0	0	1	0	1	1	0
b	85	0x55	0	1	0	1	0	1	0	1
y	215	0xD7	1	1	0	1	0	1	1	1

8.3.4.3 BXOR – bitwise exclusive OR

	Signature	Type	Description
Inputs	*a*	int32	Operand 1
	b	int32	Operand 2
Outputs	*y*	int32	Bitwise exclusive disjunction

$$y = a \bigoplus b$$

Operand	Value (DEC)	Value (HEX)	Value (BIN)							
			Bit7	Bit6	Bit5	Bit4	Bit7	Bit2	Bit1	Bit0
a	150	0x96	1	0	0	1	0	1	1	0
b	85	0x55	0	1	0	1	0	1	0	1
y	193	0xC3	1	1	0	0	0	0	1	1

8.3.4.4 BNOT – bitwise NOT

	Signature	Type	Description
Inputs	*x*	int32	Input operand
Outputs	*y*	int32	Bitwise negation

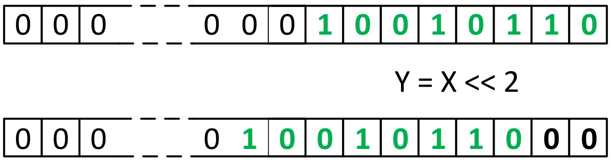
$$y = \overline{x}$$

Operand	Value (DEC)	Value (HEX)	Value (BIN)							
			Bit7	Bit6	Bit5	Bit4	Bit7	Bit2	Bit1	Bit0
x	150	0x96	1	0	0	1	0	1	1	0
y	105	0x69	0	1	1	0	1	0	0	1

8.3.4.5 BSHL – bitwise left shift

	Signature	Type	Description
Inputs	*x*	int32	Shifted
	n	int32	Shift amount (shifted-in bits)
Outputs	*y*	int32	Shift result

$$y = x \ll n$$



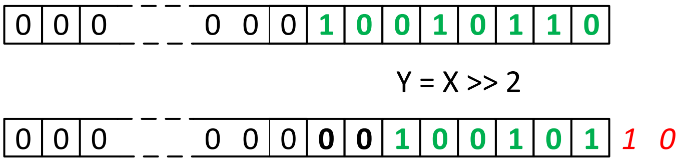
X = 150 (dec) = 0x96 (hex)

Y = 600 (dec) = 0x258 (hex)

8.3.4.6 BSHR – bitwise right shift

	Signature	Type	Description
Inputs	*x*	int32	Shifted
	n	int32	Shift amount (shifted-in bits)
Outputs	*y*	int32	Shift result

$$y = x \gg n$$



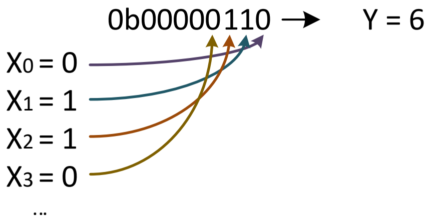
X = 150 (dec) = 0x96 (hex)

Y = 37 (dec) = 0x25 (hex)

8.3.4.7 CODER – coder

	Signature	Type	Description
Inputs	*x_0*	bool	Bit 0
	x_1	bool	Bit 1
	…		
	x_{N-1}	bool	Bit *N-1*
Outputs	*y*	int32	Bitwise sum

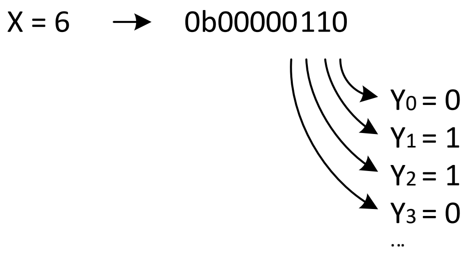
$$y = \sum_{i=0}^{N-1} x_i \ll i$$



8.3.4.8 DECODER – decoder

	Signature	Type	Description
Inputs	*x*	int32	Input value
Outputs	*y_0*	bool	Bit 0
	y_1	bool	Bit 1
	…		
	y_{N-1}	bool	Bit *N-1*

$$y_i = \left(x \bigwedge (1 \ll i)\right) \gg i, \qquad i \in 0..N-1$$



8.3.5 Relational operations

8.3.5.1 EQ – equal

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
Outputs	*y*	bool	Result *true*, if *a* = *b*

$$y = \begin{cases} a = b, & true \\ & false \end{cases}$$

⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.5.2 NE – not equal

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
Outputs	*y*	bool	Result *true*, if *a* ≠ *b*

$$y = \begin{cases} a \neq b, & true \\ & false \end{cases}$$

⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.5.3 GT – greater

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
Outputs	*y*	bool	Result *true*, if *a* > *b*

$$y = \begin{cases} a > b, & true \\ & false \end{cases}$$

⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.5.4 GE – greater or equal

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
Outputs	*y*	bool	Result *true*, if *a* ≥ *b*

$$y = \begin{cases} a \geq b, & true \\ & false \end{cases}$$

⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.6 Selection and limit operations

8.3.6.1 SEL – selection value

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
	n	bool	Select operand. If *n* equal to «1», then the value of input *b* will be transmitted to the output. Otherwise, the value of input *a* will be transmitted to the output.
Outputs	*y*	float, int32	Result

$$y = \begin{cases} n = false, & a \\ & b \end{cases}$$

> ⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.6.2 MAX – maximum value

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
Outputs	*y*	float, int32	Maximum value

$$y = \begin{cases} a > b, & a \\ & b \end{cases}$$

> ⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.6.3 MIN – minimum value

	Signature	Type	Description
Inputs	*a*	float, int32	Operand 1
	b	float, int32	Operand 2
Outputs	*y*	float, int32	Minimum value

$$y = \begin{cases} a < b, & a \\ & b \end{cases}$$

> ⚠ *The block type is determined by the type of the value at the input* ***a***

8.3.6.4 LIMIT – limitation

	Signature	Type	Description
Inputs	*x*	float, int32	Input operand
	max	float, int32	Maximum
	min	float, int32	Minimum
Outputs	*y*	float, int32	If *x* is less than *min* then the output will be set to *min*. If *x* is less than *max* then the output will be set to *max*. Otherwise, the output will be set to *x.*

$$y = \begin{cases} x > max, & max \\ x < min, & min \\ & x \end{cases}$$

⚠ *The block type is determined by the type of the value at the input* ***x***

8.3.6.5 MUX – multiplexer

	Signature	Type	Description
Inputs	*x_0*	float, int32	Input 0
	x_1	float, int32	Input 1
	…		
	x_{N-1}	float, int32	Input *N-1*
	k	int32	The number of the input, the value of which will be transmitted to the output.
Outputs	*y*	float, int32	The output takes on the value of one of the inputs.

$$y = x_k, \qquad k \in 1 \dots N-1$$

⚠ *The block type is determined by the type of the value at the input* ***x_0***

8.3.6.6 DMUX – demultiplexer

	Signature	Type	Description
Inputs	*x*	float, int32	Input. The value that will be transmitted to one of the outputs.
	k	int32	The number of the output to which the input value will be transmitted.
Outputs	*y_0*	float, int32	Output 0
	y_1	float, int32	Output 1
	…		
	y_{N-1}	float, int32	Output *N-1*

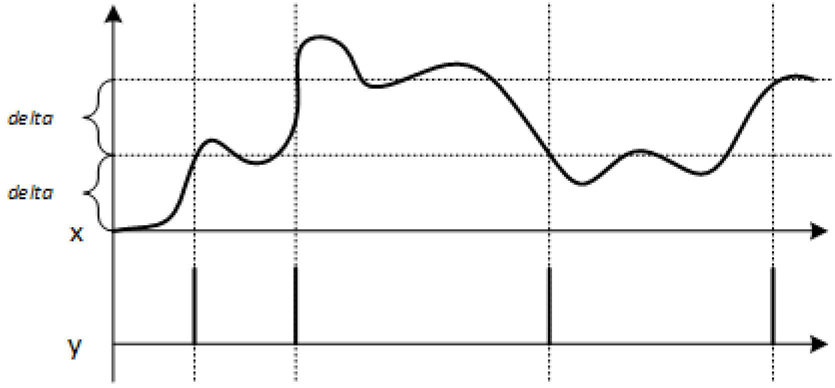
$$y_i = \begin{cases} x, & i = k \\ & 0 \end{cases}, \qquad k \in 0 \dots N-1$$

⚠ *The block type is determined by the type of the value at the input* ***x***

8.3.6.7 APPERTURE – Value change control

	Signature	Type	Description
Inputs	*x*	float, int32	Input operand
	delta	float, int32	Value that changes the output *y* to *true*
Outputs	*y*	bool	Result
Internal	*x_old*	float, int32	The value of *x*, at the previous commit

$$y = \begin{cases} |x - x\_old| \geq delta, & true \\ & false \end{cases}$$



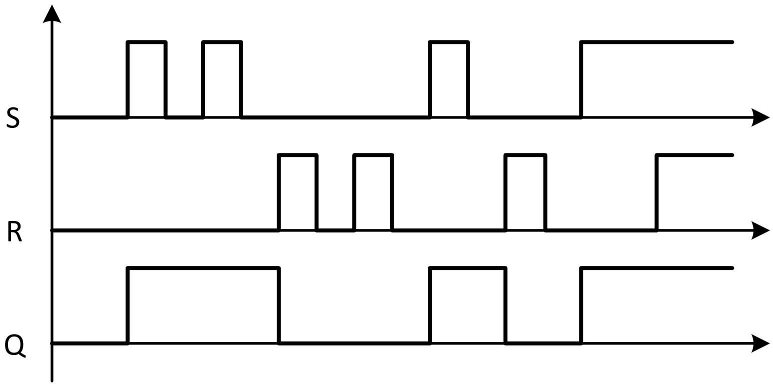
⚠ *The block type is determined by the type of the value at the input* ***x***

8.3.7 Triggers, generators, counters

8.3.7.1 SR – Set-Reset trigger

	Signature	Type	Description
Inputs	*S*	bool	Set output. When 1 comes to input *S*, output *Q* is set to 1. Input *S* is «dominant», i.e. if inputs *S* and *R* are set to 1, then output *Q* will be set to 1.
	R	bool	Reset output. When 1 comes to input *R*, output *Q* is set to 0.
Outputs	*Q*	bool	Output

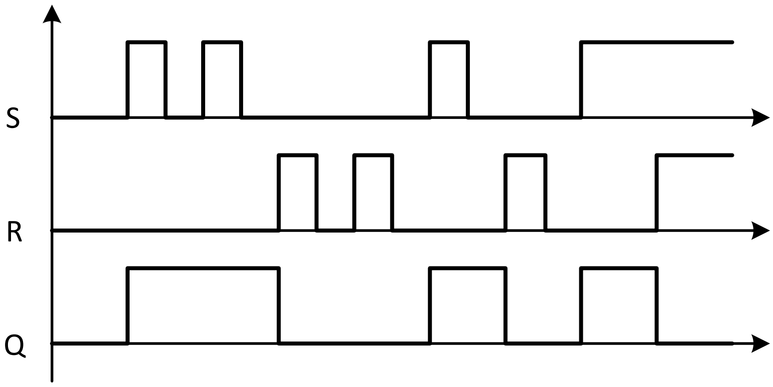
$$Q = (\overline{R} \bigwedge Q) \bigvee S$$



8.3.7.2 RS – Reset-Set trigger

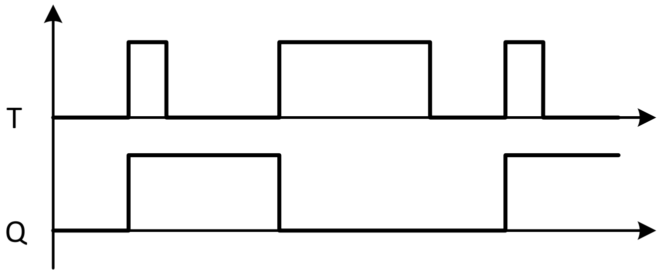
	Signature	Type	Description
Inputs	*S*	bool	Set the output. When 1 comes to input S, output *Q* is set to 1.
	R	bool	Reset the output. When 1 comes to input *R*, output *Q* is set to 0. Input *R* is «dominant», i.e. if inputs *S* and *R* are set to 1, then output *Q* will be set to 0.
Outputs	*Q*	bool	Output

$$Q = \overline{R} \bigwedge (Q \bigvee S)$$



8.3.7.3 TT – Toggle (T-trigger)

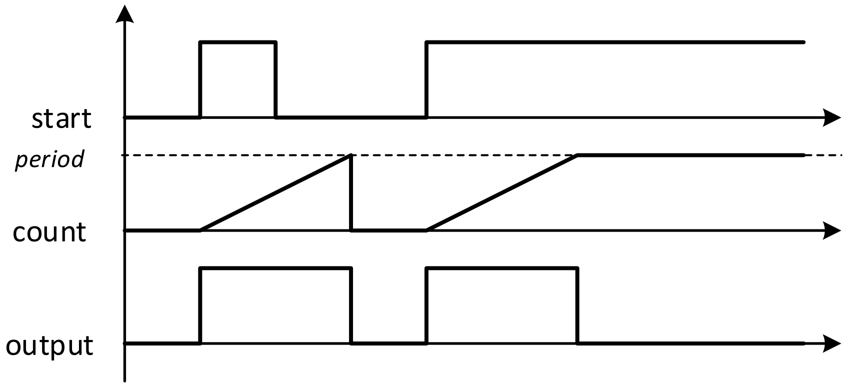
	Signature	Type	Description
Inputs	*T*	bool	Input. When transition from 0 to 1 comes to input T, output Q is negated.
Outputs	*Q*	bool	Output
Internal	*old*	bool	*T* value in the previous step.



8.3.7.4 TP – one pulse generator

	Signature	Type	Description
Inputs	*start*	bool	Trigger (rising edge)
	period	int32	Pulse duration in ms
Outputs	*output*	bool	Output. Set to 1 by the *start* input. It is reset to 0 when the *count* counter reaches the value of the *period* input.
	count	int32	Counter value in ms. Triggered by the *start* input.
Internal	*tick*	int32	Origin of the counter *count*
	old	bool	*start* value in the previous step.

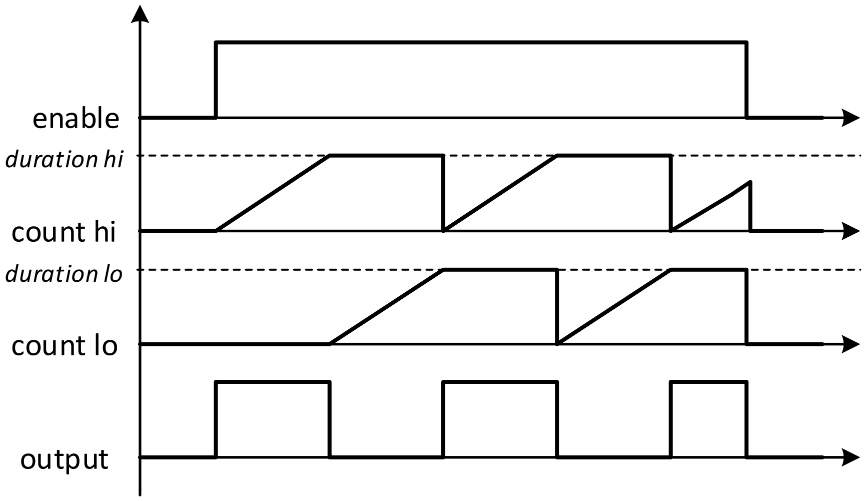
When 1 comes to *start* input, *output* is set to 1 and internal counter *count* is started. When counter reaches the *period* value, count stops and *output* is reset to zero.



8.3.7.5 BLINK – pulse generator

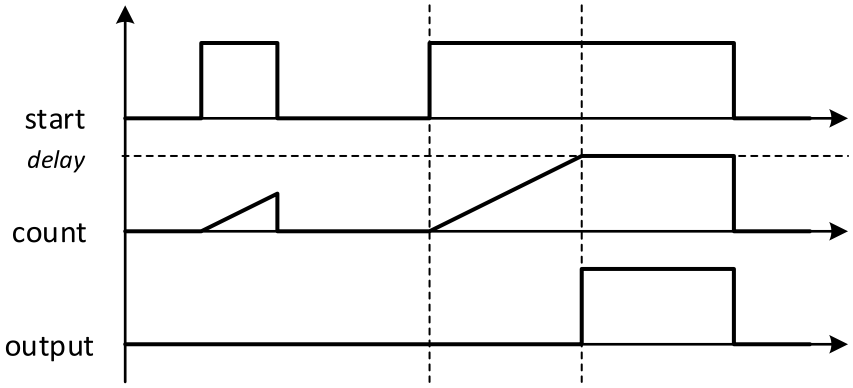
	Signature	Type	Description
Inputs	*enable*	bool	Enable signal
	duration hi	int32	Duration of states of logical 1 in ms
	duration lo	int32	Duration of states of logical 0 in ms
Outputs	*output*	bool	Output. It is in the state of logical 1, at counting the counter *count hi*, and in the state of logical 0 at counting the counter *count lo*.
	count hi	int32	Counter state of logical 1 in ms. Resets at the same time as *count lo* resets.
	count lo	int32	Counter state of logical 0 in ms. Resets when *duration lo* is reached.
Internal	*TCK*	int32	Origin of the counters *count hi* and *count lo*

When *enable* input is set to logical 0, generator is off all outputs are equal to zero. When *enable* input is set to logical 1, generator is on, counters *count hi* and *count lo* count in turn from zero to *duration hi*, *duration lo* values respectively. When *count hi* counts, *output* is 1, when *count lo* counts, *output* is 0.



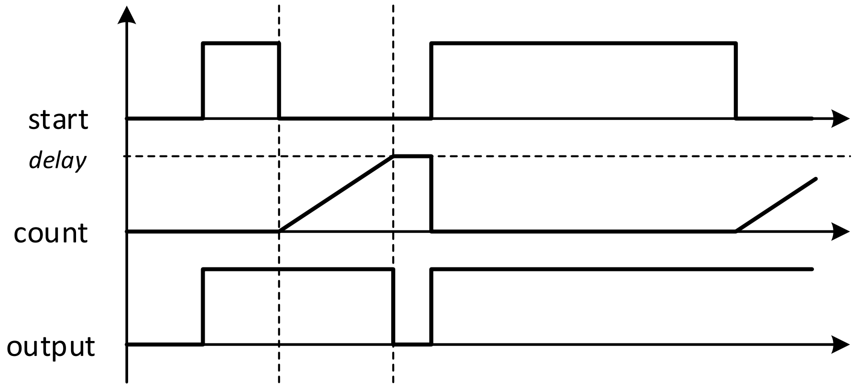
8.3.7.6 TON – On delay timer

	Signature	Type	Description
Inputs	*start*	bool	Trigger
	delay	int32	Duration of switching-on in ms
Outputs	*output*	bool	Set in 1, when the counter *count* reaches input *delay*.
	count	int32	Counter value in ms. Triggered by the *start* input.
Internal	*old*	bool	*start* value in the previous step
	TCK	int32	Origin of *count* counter



8.3.7.7 TOFF – Off delay timer

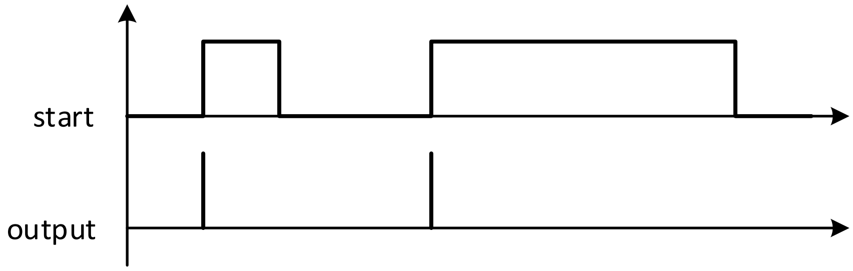
	Signature	Type	Description
Inputs	*start*	bool	Trigger
	delay	int32	Duration of switching-off in ms
Outputs	*output*	bool	Set in 1, by *start* input. Resets in 0, when counter *CNT* reaches *delay* input.
	count	int32	Counter value in ms. Triggered at changing *start* input from 1 to 0.
Internal	*old*	bool	*start* value in the previous step
	TCK	int32	Origin of *CNT* counter



8.3.7.8 RISING – rising edge detector

	Signature	Type	Description
Inputs	*input*	bool	When *input* goes from 0 to 1 for one cycle, *output* is set to logical 1.
Outputs	*output*	bool	Output. Set in 1, when *input* changes from 0 to 1.
Internal	*old*	bool	*input* value in the previous step

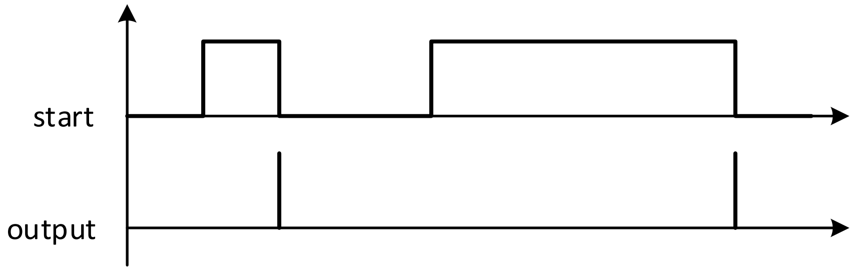
$$\mathrm{output} = input \bigwedge \overline{old}$$



8.3.7.9 FALLING – falling edge detector

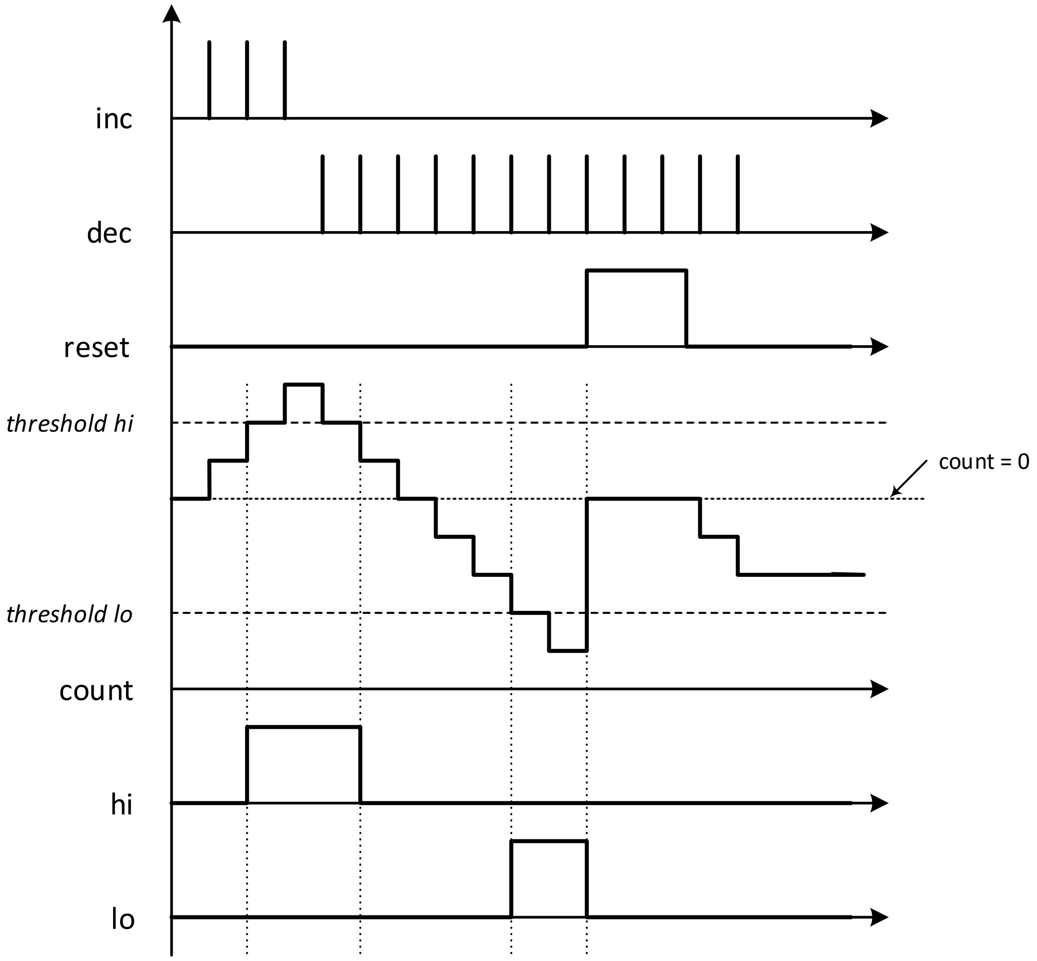
	Signature	Type	Description
Inputs	*input*	bool	When *input* goes from 0 to 1 for one cycle, *output* is set to logical 1.
Outputs	*output*	bool	Output. Set in 1, when *input* changes from 1 to 0.
Internal	*old*	bool	*input* value in the previous step

$$\mathrm{output} = old \bigwedge \overline{input}$$



8.3.7.10 CNT – counter

	Signature	Type	Description
Inputs	*inc*	bool	Incremental input. When *inc* input goes from 0 to 1, counter *count* increases its value by 1.
	dec	bool	Decremental input. When *dec* input goes from 0 to 1, counter *count* decreases its value by 1.
	reset	bool	Reset *count.* When input *reset* is 1, counter *count* resets to zero.
	threshold hi	int32	High threshold. When counter *count* reaches *threshold hi* value, output *hi* is set to logical 1.
	threshold lo	Int32	Low threshold. When *count* value is less *threshold lo* value, output *lo* is set to logical 1.
Outputs	*count*	int32	Counter value
	hi	bool	Counter value *count* ≥ *threshold hi*
	lo	bool	Counter value *count* ≤ *threshold lo*
Internal	*inc old*	bool	*inc* value in the previous step
	dec old	bool	*dec* value in the previous step

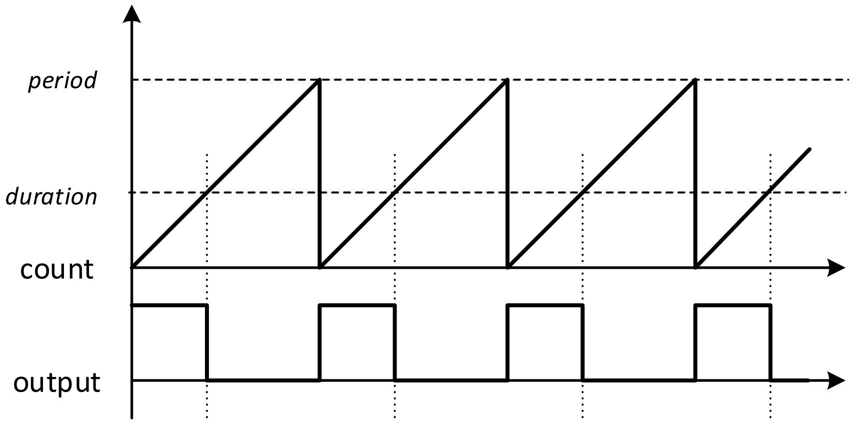


8.3.7.11 RAND – random number generator

	Signature	Type	Description
Outputs	*output*	int32	pseudorandom number

8.3.7.12 PWM – PWM generator

	Signature	Type	Description
Inputs	*duration*	int32	PWM pulse duration in ms
	period	int32	PWM period in ms
Outputs	*output*	bool	Output. Equal to 1 when counter *count* is greater or equal to *duration*.
	count	int32	PWM counter in ms. Counts from *0* to *period-1*
Internal	*tick*	int32	Origin of *count* counter



8.3.8 Special functions

8.3.8.1 EVENT – event generator

	Signature	Type	Description
Inputs	*generate*	bool	Event generation signal. When *generate* input goes from 0 to 1, block generates an event.
	force	bool	Priority event. If parameter *force* is equal to *true*, then event will be sent to the server with priority, otherwise the event will be sent with general priority.

Settings	*index*	uint8	Event number. There are 3 codes available, which will be substituted in field #2 (event_code) of the FLEX protocol:
			CE_EVT_1 - Event #41046;
			CE_EVT_2 - Event #41047;
			CE_EVT_3 - Event #41048.

	format	uint8	Packet format *(Currently feature is in development)*
Internal	*old*	bool	Signal value of event generation, on the previous cycle

> ⚠ *- If a constant is connected to the **generate** input and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to generate an event.*
> *- If a variable or output of another function is connected to the **generate** input, then the triggering occurs when switching from False to True.*

8.3.8.2 CMD – command from device

	Signature	Type	Description
Outputs	*active*	bool	Command receiving signal
	param1	int32	Parameter 1
	param2	int32	Parameter 2
	param3	int32	Parameter 3
	param4	int32	Parameter 4
	param5	int32	Parameter 5

To receive parameters from a user or a monitoring system, there is a command provided, which device can receive by USB, Bluetooth, SMS, Internet.

When command is received, device sets the active output to 1 for one cycle of block operation (later it will be reset to 0).

Outputs *paramX* are set to the values of the last command received (outputs are reset to 0 when it is the first run of Complex Events, or when the value 0 is received in a command).

Command format:

Request	*!CEVT<s><param1>[,<param2>,<param3>,<param4>,<param5>] Пример: *!CEVT 120,300 // It is allowed not to add the last values *!CEVT 10,,,,200 // To skip intermediate it is needed to use commas	
Respond	*@CEVT	
Exchange channel	Internet, USB, Bluetooth, SMS	
Signature	**Description**	**Data format**
<s>	Delimiter – space (0x20)	char
<param1>	Value set at param1 output. Text value is converted to I32 number. Empty value is treated as 0.	char[]
<param2>	Similar to the <param1> parameter, but for param2	char[]
<param3>	Similar to the <param1> parameter, but for param3	char[]
<param4>	Similar to the <param1> parameter, but for param4	char[]
<param5>	Similar to the <param1> parameter, but for param5	char[]

8.3.8.3 FLEX – reading a value from FLEX table

⚠ *Function was updated. The current implementation has been used from the editor version v3.4.1*

	Signature	Type	Description
Outputs	*value*	int32	Value returned by block
Settings	*index*	uint8	Number of the FLEX field from which it is needed to get the value.
	offset	uint8	Offset in bytes from the beginning of the field (some fields contain several tens of bytes)

	type	uint8	Parameter type for reading:
			uint8 – one-byte unsigned number;
			int8 – one-byte signed number;
			uint16 – two-byte unsigned number;
			int16 – two-byte signed number;
			int32/float – four-byte signed/real number.

❗ *The logic of the function depends on the data type:*
*- If a **variable** with the **FLOAT** type is connected to the output **value** and the parameter **type = int32/float**, then the function reads data from memory according to the IEEE754 standard. This method must be used for FLEX parameters that are stored in the Float format (For example, the "speed" parameter)*
- Otherwise, the function reads the data as an INT32 number. This method must be used for FLEX parameters that are stored in any format other than Float.
The conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.

8.3.8.4 USER_PARAM – writing a value to user parameter

	Signature	Type	Description
Input	*value*	int32/float	Value to be written to the corresponding user parameter
	enable	bool	Recording condition. Input *value* is recorded, if *enable = true*, otherwise value is not recorded.
Settings	*index*	uint8	Index of user parameter, to which the record will be made.

ℹ *For block operation, transfer of the corresponding user parameter must be configured in the device configuration. First, it is needed to place a block on the diagram, then (before compiling) make changes to the configuration.*
Configuration> Protocol Settings:
..> select «FLEX3.0»
..> User Parameters> Assign Parameters «User Parameter CEx».

❗ *The logic of the function depends on the data type:*
*- If a **variable** with the **FLOAT** type is connected to the input **value**, then the function writes data to memory according to the IEEE754 standard. This method must be used for parameters that will be read by the server in the Float format (For example, the number 12.016 should be written this way). To send such a value to the server, you must use a user parameter of 4 bytes.*
- Otherwise, the function writes the data as an integer number. This method must be used for parameters that will be read by the server in Int or Uint format (For example, the number 43605 should be written this way). You can use a custom parameter of any size to send it to the server.
The conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.

8.3.8.5 SMS – send SMS

	Signature	Type	Description
Input	*start*	bool	Signal about sending SMS. When input *start* value changes state from 0 to 1, device starts sending SMS.
Output	*active*	bool	Execution. Output returns *true* until device makes a previous attempt to send SMS.
Settings	*user*	uint8	Subscriber number in the device memory.
	type	uint8	Message type
	message	string	Custom string to be added to the message, up to 32 characters only. NOT used if type = «Standard SMS».
Internal	*old*	bool	*start* value on the previous cycle

> ⚠ *If a constant is connected to the **start** input and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to send SMS.*
> *If a variable or output of another function is connected to the **start** input, then the triggering occurs when switching from False to True.*

8.3.8.6 USER_SMS – send custom SMS

	Signature	Type	Description
Inputs	*start*	bool	Signal about sending SMS.
	$value_0$	int32/float	Argument 0
	$value_1$	int32/float	Argument 1
	…		
	$value_{N-1}$	int32/float	Argument *N-1*
Outputs	*active*	bool	Execution. The output value is *true* until the device makes an ongoing attempt to send an SMS.
Settings	*user*	string	A string with an arbitrary phone number or with a subscriber number from the configuration.
	message	string	Message text. Arguments can be added to the message body. Example: Voltage is **{0}** V, Temperature is **{1}** *C
Internal	*old*	bool	*start* value on the previous cycle

> ⚠ *If a constant is connected to the **start** input and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to send SMS.*
> *If a variable or output of another function is connected to the **start** input, then the triggering occurs when switching from False to True.*

8.3.8.7 RECV_SMS - SMS receipt indicator

	Signature	Type	Description
Output	*active*	bool	Signal about receiving SMS, that matched the *message* template and checked against the *flags* conditions. The output for one cycle of program execution is *true.*
Settings	*phone*	string	String with arbitrary phone number
	message	string	Template text (up to 16 characters)
	flags	uint8	Checking options

8.3.8.8 CALL – make a call

	Signature	Type	Description
Input	*start*	bool	Signal to make a call. When input *start* value changes state from 0 to 1, device starts making calls.
Output	*active*	bool	Execution. Output returns *true* until device makes a previous attempt to make a call.
Settings	*user*	uint8	Subscriber number in the device memory
	type	uint8	Call type
Internal	*old*	bool	*start* value on the previous cycle

> ⚠ *If a constant is connected to the* ***start*** *input and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to make a call.*
> *If a variable or output of another function is connected to the* ***start*** *input, then the triggering occurs when switching from False to True.*

8.3.8.9 CAM – make a picture

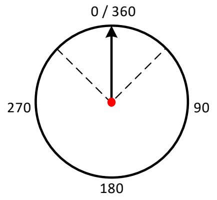
	Signature	Type	Description
Input	*start*	bool	When input *start* value changes state from 0 to 1, device takes a picture.
Output	*active*	bool	Execution. Output returns *true* until device is generating and saving the picture.
Internal	*old*	bool	*start* value on the previous cycle

> ℹ *For block operation, device must be configured to work with the camera.*
> *Configuration> RS-232 / RS-485> Use As> «Camera».*

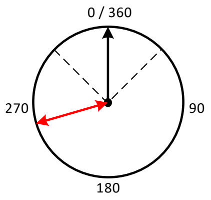
> ⚠ *If a constant is connected to the* ***start*** *input and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to take a picture.*
> *If a variable or output of another function is connected to the* ***start*** *input, then the triggering occurs when switching from False to True.*

8.3.8.10 GEOZONE – Geofence

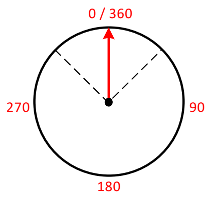
	Signature	Type	Description
Input	*latitude*	float	Geofence center latitude (Example: 55.755669)
	longitude	float	Geofence center longitude (Example: 37.616802)
	radius	float	Geofence circle radius in meters
	course	int32	Direction of movement (course) to fix the entrance to the geofence
	course delta	int32	Entry angle range. If *course delta* is set to 360, then control of course for entering the geofence is not performed.
Output	*active*	bool	*true* value, if the object is inside a geofence.
Settings	*speed min*	int16	Speed, below which the *current course* is not updated
Internal	*current course*	int32	Current course



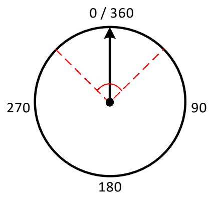
latitude, longitude



radius



course



course delta

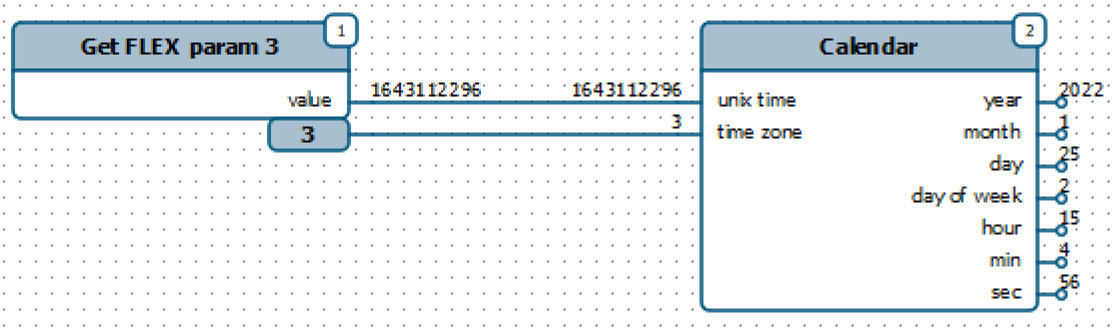
8.3.8.11 CALENDAR – calendar

	Signature	Type	Description
Input	*UNIX time*	int32	Time in UNIX-time format.
	timezone	int32	Time zone. Integer number from -12 to 12.
Output	*year*	int32	Year
	month	int32	Month number. Integer number from 1 to 12. For example: 1 –January etc.
	day	int32	Day of the month. Integer number from 1 to 31.
	day of week	int32	Day of the week. Integer number from 1 to 7. For example: 1 – Monday etc.
	hour	int32	Hour. Integer number from 0 to 24.
	min	int32	Minute. Integer number from 0 to 59.
	sec	int32	Second. Integer number from 0 to 59.

Time in UNIX-time format is integer number, which is the number of seconds passed from 00:00:00 01.01.1970

This block converts time in UNIX-time format, taking into account the time zone, into more convenient for use separate parameters: year, month, day and others.

To convert the current time of the device, it is needed to create FLEX block to obtain field No.3 [time] and connect it to the UINX time input.



8.3.8.12 INFO – Information about device

	Signature	Type	Desription
Output	*model*	int32	Numerical designation of the device model.
	version	int32	Firmware version of the device, represented as an integer number, where the lower 2 digits are in the first byte, the middle 2 digits are in the second byte and the higher 2 digits are in the third byte.

For example, device S-2435 with firmware v03.02.31:
model=2435
version=197151 (0x0003021F)

8.3.8.13 IMEI – Modem IMEI

	Signature	Type	Desription
Output	*digits 8..0*	int32	Number representing the lower 9 digits of the IMEI.
	digits 14..9	int32	Number representing the higher 6 digits of the IMEI.

For example, IMEI 866795030518573:
digits 8..0 = 30518573
digits 14..9 = 866795

⚠ *In the example for **digits 8..0**, not 030518573 is written, but 30518573. Extreme zeros on the left are not displayed when displaying numeric values.*

8.3.8.14 ICCID – SIM card ICCID

	Signature	Type	Desription
Input	*SIM index*	bool	SIM card slot number: **"0"** - external; **"1"** - internal.
Output	*digits 8..0*	int32	Number representing the lower 9 digits of the ICCID.
	digits 16..9	int32	Number representing the higher 6 digits of the ICCID.

For example, ICCID 8970199201010570553:
digits 8..0 = 10570553
digits 16..9 = 70199201

⚠ *In the example for **digits 8..0**, not 010570553 is written, but 10570553. Extreme zeros on the left are not displayed when displaying numeric values.*

ℹ *The length of the ICCID number is usually 19 to 20 digits. The function allows you to get only the lower 17 digits. The higher 2 digits for any SIM cards of ISO/IEC 7812-1 standard must be '89'.*

8.3.8.15 IMSI – SIM card IMSI

	Signature	Type	Desription
Input	*SIM index*	bool	SIM card slot number: **"0"** - external; **"1"** - internal.
Output	*digits 8..0*	int32	Number representing the lower 9 digits of the IMSI.
	digits 14..9	int32	Number representing the higher 6 digits of the IMSI.

For example, IMSI 250991039698855:
digits 8..0 = 39698855
digits 14..9 = 250991

> ⚠ *In the example for **digits 8..0,** not 039698855 is written, but 39698855. Extreme zeros on the left are not displayed when displaying numeric values.*

> ℹ *The first three digits of the IMSI are the MCC (country code, for example, 250 - Russia). It is followed by two or three digits MNC (mobile network code, for example, 99 - Beeline). All subsequent digits are the MSIN user ID.*

8.3.8.16 LOG_MSG – Send a message to the log

	Signature	Type	Desription
Inputs	*send*	bool	Message send signal.
	$value_0$	int32/float	Argument 0
	$value_1$	int32/float	Argument 1
	…		
	$value_{N-1}$	int32/float	Argument N-1
	message	string	Message text. Arguments can be added to the message body. Example: Voltage = **{0}** V, Temperature = **{1}** *C
Internal	*old*	bool	*start* value on the previous cycle

This function outputs arbitrary text with arguments to the user log window of the NTC Configurator program.

> ℹ *To view the logs, in the main window of the NTC Configurator program, you should go to "Advanced" > "Show log window" > set the "Complex Events" flag.*

> ⚠ *If a constant is connected to the **send** input and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to send a message.*
> *If a variable or output of another function is connected to the **send** input, then the triggering occurs when switching from False to True.*

8.3.9 Peripheries

8.3.9.1 INPUT

	Signature	Type	Description
Inputs	*reset*	bool	Reset counter *(if the input is configured as a «pulse counter»)*
Outputs	*voltage*	int32	Voltage *(goes through a quick filtering by device algorithms)*

	value	int32	Type of value depends on the input setting:
			«Discrete» - trigger state 1 or 0;
			«Analog» - voltage in mV (without filtering);
			«Frequency» - frequency in Hz;
			«Counting» - number of counted impulses.

Settings	*index*	uint8	Device input number

ℹ *For block operation, in device configuration corresponding input must not be disconnected Configuration> Inputs> Use as> Any value other than "Not Used".*

8.3.9.2 OUTPUT

	Signature	Type	Description
Inputs	*value*	int32	Output state to be set. The logic depends on the output setup. **"Of general purpose"** **«1»** - enable (short to ground) **«0»** - disable. **"Buzzer"** *(only OUT_1):* The frequency (Hz) to be generated at the output.
Settings	*index*	uint8	Device output number

ℹ *For block operation, in device configuration corresponding output must be configured in a certain way.*
Configuration > Outputs> Use as> "Of general purpose".
For OUT_1 "Buzzer" setting is allowed.

⚠ *Block operates in the «pressure signal» mode. At each execution, the block tries to set the state of the output, which is specified by the input value.*

8.3.9.3 HYGRO – hygrometer

	Signature	Type	Description
Outputs	*temperature*	float	Temperature, °C
	humidity	float	Humidity, %
Settings	*Index*	uint8	Temperature/Humidity sensor number for displaying

8.3.9.4 ACCEL – accelerometer

	Signature	Type	Description
Outputs	*x*	int32	Current acceleration along the X axis of the accelerometer
	y	int32	Current acceleration along the Y axis of the accelerometer
	z	int32	Current acceleration along the Z axis of the accelerometer
	acc sqrt	int32	Square root of the sum of the squares of the accelerations along each axis
	int sqrt	int32	--
	angle	int32	Tilt angle relative to the local (temporary) vertical
	pitch	int32	Pitch angle: forward tilt <0 backward tilt > 0
	roll	int32	Roll angle: roll to the left <0 roll to the right > 0
	calibrated	bool	Accelerometer calibration status (true - calibrated)

8.3.9.5 ECODRIVE – Eco Driving

	Signature	Type	Description
Outputs	*speed*	int32	Current speed value
	boost	int32	Current acceleration value (after calibration)
	retard	int32	Current braking value (after calibration)
	drift_right	int32	Current value of acceleration to the right (after calibration)
	drift_left	int32	Current value of acceleration to the left (after calibration)
	jump	int32	Current value of vertical acceleration (after calibration)
	belt	int32	--
	light	int32	--
	prm	int32	--

> *For block operation, device must be configured to work with Eco Driving.*
> *Configuration> EcoDriving> Enable driving quality control*

8.3.9.6 ONEWIRE_KEY – Information about short-range current tag on the 1-Wire or RS-232/485 interfaces

	Signature	Type	Description
Outputs	*lo*	int32	Low 4 bytes of code
	hi	int32	High 4 bytes of code
	valid	bool	Code is in the list of device proxy codes

8.3.9.7 RFID – Information about long-range RFID current tag on the RS-232/485 interfaces

	Signature	Type	Description
Outputs	*lo*	int32	Low 4 bytes of code
	hi	int32	High 4 bytes of code
	pwr	int32	Signal power
	type	int32	--
	valid	bool	Code is in the list of device proxy codes

> *For block operation, device must be configured to work with RFID readers.*
> *Configuration > RS-232/RS-485 > Device X > "RFID tag reader".*

8.3.9.8 TACHOGRAPH – Tachograph driver

	Signature	Type	Description
Outputs	*code0_3*	int32	0 .. 3 bytes of card code
	code4_7	int32	4 .. 7 bytes of card code
	code8_11	int32	8 .. 11 bytes of card code
	code12_15	int32	12 .. 15 bytes of card code
	state	int32	Driver state
	type	int32	--
	active	bool	--
Settings	*index*	uint8	Driver number (1st or 2nd)

For block operation, device must be configured to work with tachograph.
Configuration > RS-232/RS-485 > Device X > "Tachograph".

8.3.9.9 GUARD – Security mode

	Signature	Type	Description
Inputs	*enable*	bool	Enable/disable security mode: **«0»** – surveillance **«1»** – security
Outputs	*mode*	int32	Current operating mode: **«0»** – surveillance **«1»** – security
	error	int32	Error code when switching security mode: **«1»** – security mode disabled in device configuration; **«2»** – timeout not expired for switching mode ban; **«3»** – mode enabled: do not enable security mode with running ignition; **«4»** – device is already in this mode; **«5»** – mode enabled: do not enable security mode, if one of security sensors triggered.
Settings	*type*	uint8	Type of switching of operating mode: **«By level»** - at each execution, this block sets the operating mode according to the value of the input; **«By rising edge»** - at each execution, this block switches the operating mode to the opposite if the state of the input has changed from 0 to 1.
Internal	*old*	bool	*enable* value on the previous cycle

For block operation, device must be configured to work with security functions.
Configuration > Security mode > "Use security modes".

If the type of switching of operating mode is set "By level", then this block works in "Pressure signal" mode. At each execution, the block tries to set the state of the output, which is specified by the input value.

8.3.9.10 CRASH_FILE – Accident file generating

	Signature	Type	Description
Inputs	*generate*	bool	On the rising edge, generate an accident file
	unlock	bool	On a rising edge, release the overwrite lock
Outputs	*active*	bool	The accident file is generated. The value *true* is set at the beginning of file generating, the value *false* is set when the file generating is completed.
	time	int32	Time of file creating in UNIX format *(0 – no accident file)*
	locked	bool	The value *true* is set if the file is protected from overwriting.
Internal	*generate_old*	bool	*generate* value on the previous cycle
	unlock_old	bool	*Unlock* value on the previous cycle

ℹ *For block operation, device must be configured to work with accident detection function.*
Configuration > Accelerometer > Road accident detection > "Enable road accident detection …"

⚠ *If a <u>constant is connected</u> to input **generate** and its value is True, then the function works in the "pressure signal" mode. On each execution, the function tries to generate or unlock an accident file.*
*If a <u>variable or output of another function is connected</u> to input **generate**, then the triggering occurs when switching from False to True.*

8.3.9.11 PWRSAVE – Energy saving control

	Signature	Type	Description
Inputs	*gsm off*	bool	Disable GSM module power If *true*, device will close all established Internet connections and disable power of the GSM module. If *false*, then GSM module is enabled.
	gnss off	bool	Disable GNSS module power If *true*, device will power off the navigation module. If *false*, then GNSS module is enabled.
	battery off	bool	Disable battery charge If *true*, device will power off the back-up battery charging (but will continue to be powered by it). If *false*, back-up battery charging is enabled.
	periph off	bool	Disable periphery. If *true*, device will power off the digital interfaces that can be disabled. If *false*, digital interfaces are enabled.
	events off	bool	Disable generating events If *true*, device will prohibit generating events. If *false*, generating events is performed in the normal mode in accordance with the configuration.
Hidden	*sleep*	bool	Enter low power mode (input is provided for the future functionality)

ℹ *For block operation, device must be configured to work with Energy saving mode:*
Configuration > System settings:
.. > Enable "Use energy saving mode"
.. > Select "… controlled by Complex Events"

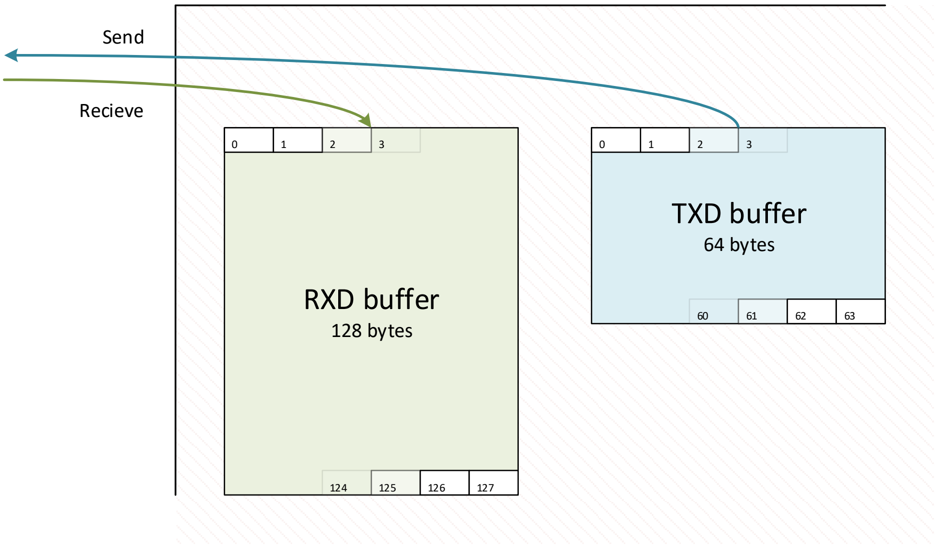
8.3.10 Access Functions to Digital Ports

At operation with all digital ports, two buffers are used to receive and transmit data: RXD (receive buffer) and TXD (transmit buffer).

Buffer sizes are fixed:

- RXD buffer - 128 bytes;
- TXD buffer - 64 bytes.

When debugging, buffers are shown as an array of bytes indexed from 0 to *(buffer_size - 1)* in the editor.



Data transmission process can be divided into several main stages:

- Write data to TXD buffer;
- Transmit data from the TXD buffer through the interface.

8.3.10.1 RS_SEND - Transmit data to serial port

	Signature	**Type**	**Description**
Inputs	*start*	bool	If *true*, the function attempts to transmit data.
	send size	int32	The number of bytes to transmit data through the interface.

Outputs	*state*	int32	Transmitter status:
			"0" - no activity;
			"1" - transmitting data;
			"-1" – the interface is unavailable (not configured).

Settings	*port*	uint8	Selecting the digital interface.

The function transmits data through the serial interface *port*. For that data is taken from the TXD buffer from position 0 to *(send_size - 1)*.

> *For function operation, the appropriate interface must be configured in the device configuration. Configuration > RS-232/RS-485 > Device 1 > "Complex Events (asynchronous mode)".*

Data reception process can be divided into several main stages:

- Receive data from the interface to RXD buffer;
- Read data from RXD buffer.

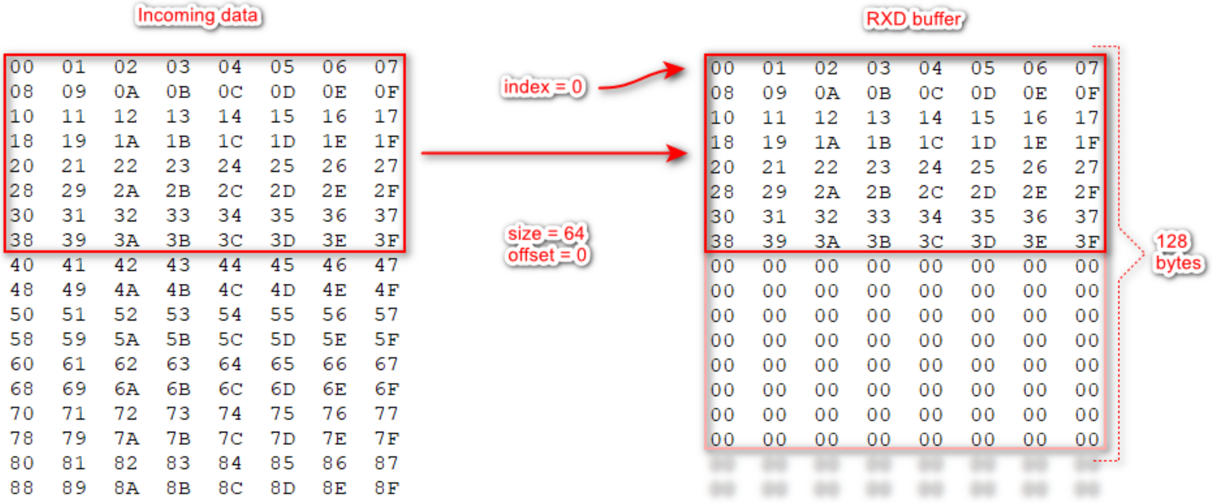
Unlike the data transmission, data receiving has not quite simple process. It is important to take into account an important feature of data processing – the device can receive unlimited amount of data, but the RXD buffer can store no more than 128 bytes. In this case, for one cycle of receive function execution, the device places no more than 64 bytes of data from the interface into the RXD buffer.

If the device receives data larger than 128 bytes, the RXD buffer will overflow. On overflow, the RXD buffer only stores the last 128 bytes of received data.

Therefore, if it is necessary to process data that exceeds 128 bytes, the program should be compiled in such a way that after each cycle of the function for receiving data from the interface, the current contents of the RXD buffer are processed. This approach will allow processing the entire required amount of data in several iterations.

Below is a visual representation of the process of receiving data, the volume of which slightly exceeds the size of the RXD buffer:

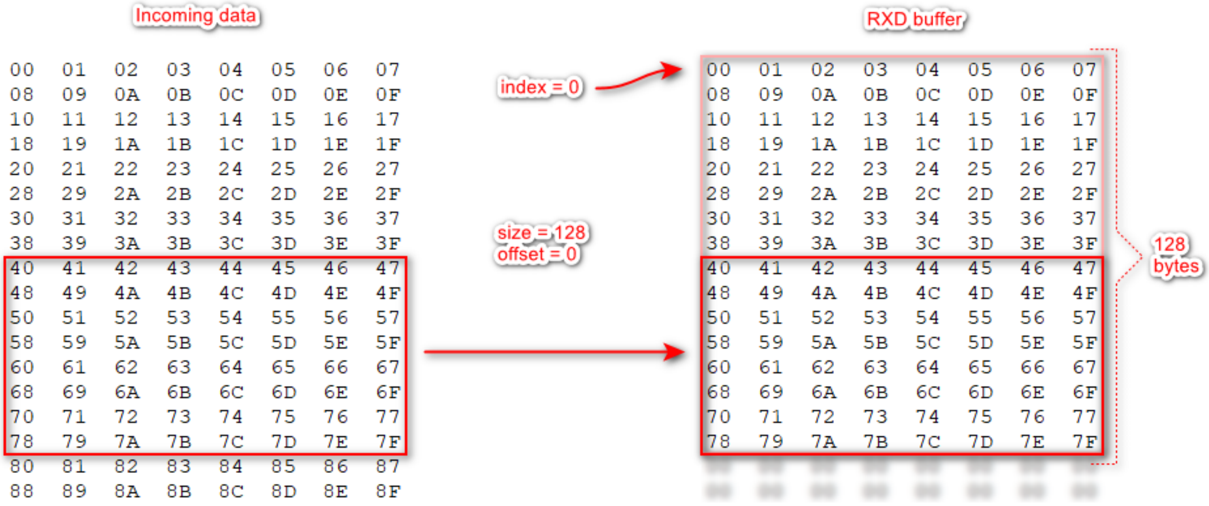
1. The receive data function detects a new incoming data stream. On the first cycle, the function receives 64 bytes, increments the received data counter *size* by 64, and places the data in the RXD buffer, starting at index 0.



```
00 01 02 03 04 05 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27
28 29 2A 2B 2C 2D 2E 2F
30 31 32 33 34 35 36 37
38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47
48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57
58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67
68 69 6A 6B 6C 6D 6E 6F
70 71 72 73 74 75 76 77
78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87
88 89 8A 8B 8C 8D 8E 8F
```

```
00 01 02 03 04 05 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27
28 29 2A 2B 2C 2D 2E 2F
30 31 32 33 34 35 36 37
38 39 3A 3B 3C 3D 3E 3F
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

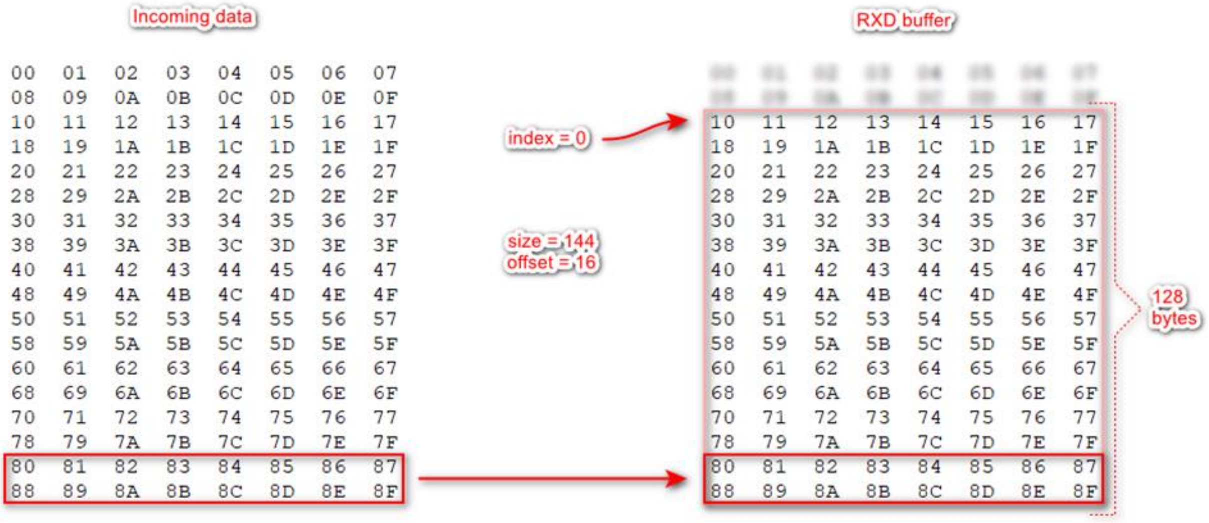
2. On the next cycle, the function receives another 64 bytes, increments the received data counter *size* by 64, and places the data in the RXD buffer, starting at index 64.



```
00 01 02 03 04 05 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27
28 29 2A 2B 2C 2D 2E 2F
30 31 32 33 34 35 36 37
38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47
48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57
58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67
68 69 6A 6B 6C 6D 6E 6F
70 71 72 73 74 75 76 77
78 79 7A 7B 7C 7D 7E 7F
80 81 82 83 84 85 86 87
88 89 8A 8B 8C 8D 8E 8F
```

```
00 01 02 03 04 05 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27
28 29 2A 2B 2C 2D 2E 2F
30 31 32 33 34 35 36 37
38 39 3A 3B 3C 3D 3E 3F
40 41 42 43 44 45 46 47
48 49 4A 4B 4C 4D 4E 4F
50 51 52 53 54 55 56 57
58 59 5A 5B 5C 5D 5E 5F
60 61 62 63 64 65 66 67
68 69 6A 6B 6C 6D 6E 6F
70 71 72 73 74 75 76 77
78 79 7A 7B 7C 7D 7E 7F
```

3. On the next cycle, the function receives the remaining data (*X bytes*), increments the received data counter *size* by X. Erases the first *X* bytes in the RXD buffer. Shifts the contents of the RXD buffer *X* bytes "to the left" (i.e. the byte at index *X-1* will now be at index 0). Increases the overflow counter *offset* by *X*. Puts data into the RXD buffer, starting at index *(128-X)*.



```
00  01  02  03  04  05  06  07
08  09  0A  0B  0C  0D  0E  0F
10  11  12  13  14  15  16  17
18  19  1A  1B  1C  1D  1E  1F
20  21  22  23  24  25  26  27
28  29  2A  2B  2C  2D  2E  2F
30  31  32  33  34  35  36  37
38  39  3A  3B  3C  3D  3E  3F
40  41  42  43  44  45  46  47
48  49  4A  4B  4C  4D  4E  4F
50  51  52  53  54  55  56  57
58  59  5A  5B  5C  5D  5E  5F
60  61  62  63  64  65  66  67
68  69  6A  6B  6C  6D  6E  6F
70  71  72  73  74  75  76  77
78  79  7A  7B  7C  7D  7E  7F
80  81  82  83  84  85  86  87
88  89  8A  8B  8C  8D  8E  8F
```

```
00  01  02  03  04  05  06  07
08  09  0A  0B  0C  0D  0E  0F
10  11  12  13  14  15  16  17
18  19  1A  1B  1C  1D  1E  1F
20  21  22  23  24  25  26  27
28  29  2A  2B  2C  2D  2E  2F
30  31  32  33  34  35  36  37
38  39  3A  3B  3C  3D  3E  3F
40  41  42  43  44  45  46  47
48  49  4A  4B  4C  4D  4E  4F
50  51  52  53  54  55  56  57
58  59  5A  5B  5C  5D  5E  5F
60  61  62  63  64  65  66  67
68  69  6A  6B  6C  6D  6E  6F
70  71  72  73  74  75  76  77
78  79  7A  7B  7C  7D  7E  7F
80  81  82  83  84  85  86  87
88  89  8A  8B  8C  8D  8E  8F
```

8.3.10.2 RS_RECV - Receive data from serial port

	Signature	Type	Description
Inputs	*enabled*	bool	If *true*, the function expects input data from the interface.
	reset	Bool	If *true*, the function will clear the RXD buffer (all bytes will be set to 0x00) and the next data will be written starting at index 0.

Outputs	*state*	int32	Receiver status:
			"0" - reception is disabled;
			"1" - waiting for data;
			"2" - data reception;
			"3" - data accepted;
			"-1" - the interface is unavailable (not configured);

	size	int32	The size of the received data array. The received data is immediately placed in the RXD buffer.
	offset	int32	The amount of data lost due to RXD buffer overflow (if more than 128 bytes are received). The buffer always contains the last 128 bytes of received data.
Settings	*port*	uint8	The digital interface controlled by the function. If the selected interface is not configured, the function will generate an error *state = -1*.
	timeout	uint16	The time after receiving the last byte, after which it is considered that the data reception is completed *state = 3*. The next data will be considered new and will be written to the RXD buffer from index 0.

The function receives data through the serial interface *port*. When the function fixes the start of data transmission (*state = 1*), then the first received bytes are copied to the RXD buffer, starting from index 0. In one operation cycle, the function is able to receive 64 bytes from the interface. If the amount of incoming data is more than 64 bytes, then the receiving process will be completed in several cycles (*state = 2*), while the remaining data will be added to the RXD buffer starting from index 64. The function will fix the end of data reception (*state = 3*), if after receiving the last byte *timeout* has expired. The next data will be considered new and will be written to the RXD buffer at index 0.

For function operation, the appropriate interface must be configured in the device configuration. Configuration > RS-232/RS-485 > Device 1 > "Complex Events (asynchronous mode)".

As a special case of data exchange, there is a function for performing a request/response transaction. This process can be divided into several main steps:

- Write data to TXD buffer;
- Transmit data from the TXD buffer through the interface;
- Receive data from the interface to the RXD buffer;
- Read data from RXD buffer.

8.3.10.3 RS_TRANS - Request/response via serial port

	Signature	**Type**	**Description**
Inputs	*start*	bool	If *true*, the function attempts to start a transaction.
	send size	int32	The size of the data array from the TXD buffer to transmit.
	require size	int32	The size of the expected response.
Outputs	*ended*	bool	Transaction completion signal. The signal is not set if the interface is not configured *(state = -1).*
	state	int32	Transaction status: **"0"** - no activity; **"1"** - waiting for access to the interface; **"2"** - access to the interface is received; **"3"** - transaction in progress; **"4"** - transaction completed successfully; **"-1"** - the interface is unavailable (not configured); **"-2"** - the timeout for waiting for a response has expired; **"-3"** - unknown error.
	recv size	int32	The size of the received data array. The received data is immediately placed in the RXD buffer.
Settings	*port*	uint8	The digital interface controlled by the function. If the selected interface is not configured, the function will generate an error *state = -1.*
	timeout	uint16	Time the function waits for a response after data transmission. If the number of bytes ≤ *require size* is received within the allotted time, then the transaction ends with the error *state = -2.*

The function sends data through the serial interface *port*. For sending, data is taken from the TXD buffer in the range from 0 to *(send size - 1)*. Next, the function waits for a response during the *timeout* time or until data of length ≥ *require size* arrives in the RXD buffer.

For function operation, the appropriate interface must be configured in the device configuration. Configuration > RS-232/RS-485 > Device X > "Complex Events (transaction)".

To work with RXD and TXD buffers, there is a set of functions used to allow performance of basic read/write and data conversion operations.

8.3.10.4 RXD_GET - Read value from RXD buffer

⚠ *Function was updated. The current implementation has been used from the editor version v3.4.1*

	Сигнатура	Тип	Описание
Inputs	*index*	int32	The position in the RXD buffer from which to read. The very first element of the buffer has index 0.
	size	int32	The number of bytes to read from the RXD buffer in the each *valueX* output. Valid values are from 1 to 4.
Outputs	*$value_0$*	int32/float	The read value 0.
	$value_1$	int32/float	The read value 1.
	…		
	$value_{N-1}$	int32/float	The read value *N-1.*
Settings	*N*	uint8	Number of outputs *value*
	endian	uint8	The byte order to be used when copying buffer elements to output *value.* For example RXD = [01,02,03,04,05,...], *index* = 0, *size* = 4: **"Little-endian"** *value* = 0x04030201. **"Big-endian"** *value* = 0x01020304. **"Big-endian (2 bytes)"** *value* = 0x03040102.
	sign	bool	If the flag is set, then the function will treat the read data as a negative number if the most significant bit is "1".

The function performs sequential reading of the RXD buffer for each *valueX* output. Reading starts at the *index*. The function reads *size* bytes and passes them to the *valueX* output. Then reading *index* is shifted by *size*, after which reading is made for the next *valueX* output. As a result, a range of bytes from *index* to *(index+(size*N)-1)* will be read from the buffer.

⚠ *The logic of the function depends on the data type:*
*- If a variable with the **FLOAT** type is connected to the **valueX** output and **size = 4**, then the function reads data from the buffer according to the IEEE754 standard. This method must be used for values that are stored in the Float format (for example, the value 12.6).*
- Otherwise, the function reads the data as INT32.
The conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.

8.3.10.5 RXD_CMP - Data search in RXD buffer

	Signature	Type	Description
Input	*index*	int32	The position in the RXD buffer from which to search. The very first element of the buffer has index 0.
Output	*result*	int32	Search results: **"≥0"** - Data found, index of the buffer element immediately following the found data sequence. **"-1"** - Data not found.
Settings	*data*	bin	Sequence to search in the RXD buffer. Specified in HEX "3120322033" or ASCII "1 2 3".
Internal	*size*	uint8	The size of the *data* field.

Example:
If in RXD = [01,02,03,04,05,06...], *index* = 0, *data* = [0203], then *value* = 3
If in RXD = [01,02,03,04,05,06...], *index* = 2, *data* = [0203], then *value* = -1
If in RXD = [01,02,03,04,05,06...], *index* = 0, *data* = [3322], then *value* = -1

8.3.10.6 RXD_STR2INT - Convert string from RXD buffer to integer number

	Signature	Type	Description
Input	*index*	int32	The position in the RXD buffer where the INT value is located.
Output	*value*	int32	The read value. If the value is not read, then *value* = 0

Starting at position *index*, the function attempts to read an INT value stored as an ASCII string.

Example:
Buffer RXD = [7a,67,2d,32,2e,36,66...]. In ASCII it is the string "zg-2.6f".
If *index* = 2 then *value* = -2
If *index* = 3 then *value* = 2
If *index* = 4 then *value* = 0

8.3.10.7 RXD_STR2FLOAT - Convert string from RXD buffer to float

	Signature	Type	Description
Input	*index*	int32	The position in the RXD buffer where the FLOAT value is located.
Output	*value*	int32	The read value. If the value is not read, then *value* = 0

Starting at position *index*, the function attempts to read an FLOAT value stored as an ASCII string.

Example:
Buffer RXD = [7a,67,2d,32,2e,36,66...]. In ASCII it is the string "zg-2.6f".
If *index* = 2 then *value* = -2.6
If *index* = 3 then *value* = 2.6
If *index* = 4 then *value* = 0

8.3.10.8 RXD_CHECKSUM - Verify checksum in RXD buffer

	Signature	Type	Description
Input	*index*	int32	Position in the RXD buffer, starting from which the calculation is performed.
	size	int32	The length of the data array to calculate the CRC.
	valued index	int32	The position in the RXD buffer that contains the value against which the computed CRC will be compared.
Output	*valid*	bool	CRC check result.
Settings	*type*	uint8	CRC calculation algorithm: **"CRC-16 (Modbus)"** Standart algorithm CRC-16 Modbus. **"CRC-8 (Maxim/Dallas)"** Standart algorithm CRC-8 Maxim/Dallas. **"XOR (8 bits)"** Sequential operation XOR. **"Sum (8 bits)"** Sequential addition of elements.
	options	uint8	**"Byte order"** - Byte order when comparing CRC (if calculated CRC = 0x0201). **"Little-endian"** The value 0x0102 will be used. **"Big-endian"** The value 0x0201 will be used. **«Invert»** If the flag is set, then before comparing the CRC will be bit-wise inverted. For example, if it was 0x0201, then it will be 0xfdfe. **«Add 1»** If the flag is set, then before comparing the CRC will be increased by 1. For example, if it was 0x0201, then it will be 0x0202.

The function performs CRC calculation on RXD buffer starting from *index* to *(index+size-1)*. The calculated CRC is compared with the value stored in the RXD buffer starting at *value index*.

The operations "Byte Order", "Invert", "Add 1" are performed after the CRC calculation in turn in the order of enumeration and affect the final value used in the comparison.

8.3.10.9 TXD_INIT - TXD buffer initialization

	Signature	Type	Description
Input	*enable*	bool	If *true*, then the TXD buffer is initialized with user data.
Settings	*data*	bin	The sequence for initializing the TXD buffer. Specified in HEX "3120322033" or ASCII "1 2 3".
Internal	*size*	uint8	Number of bytes to be written to the TXD buffer.

The function fills the TXD buffer with data entered by the user, starting at index 0. If the length of the user sequence is less than the length of the buffer, then the remaining cells are filled with 0x00.

8.3.10.10 TXD_SET - Write value to TXD buffer

⚠ *Function was updated. The current implementation has been used from the editor version v3.4.1*

	Signature	Type	Description
Inputs	*enable*	bool	If *true*, then the value is written to the buffer.
	index	int32	The position in the TXD buffer from which to write.
	size	int32	The number of bytes to be written to the buffer (from 1 to 4 bytes).
	$value_0$	int32/float	The value 0 to write to the buffer.
	$value_1$	int32/float	The value 1.
	…		
	$value_{N-1}$	int32/float	The value *N-1*
Settings	*N*	uint8	Number pf outputs *value*
	endian	uint8	The byte order to be used when writing to the buffer. For example TXD = [01,02,03,04,05,...], *index* = 1, *size* = 4, *value* = 0x44332211: **"Little-endian"** After writing TXD = [01,11,22,33,44,...] **"Big-endian"** After writing TXD = [01,44,33,22,11,...] **"Big-endian (2 bytes)"** After writing TXD = [01,22,11,44,33,...]

The function writes the *valueX* from 1 to 4 bytes into the TXD buffer starting from *index* position. Unlike the initialization function, this function is only applied to bytes in the range from *index* to *(index+(size*N)-1).*

⚠ *The logic of the function depends on the data type:*
*- If a **variable** with the **FLOAT** type is connected to the **value** output and **size = 4**, then the function writes data according to the IEEE754 standard. This method must be used for values that are stored in the Float format (for example, the value 12.016).*
- Otherwise, the function writes the data as an integer. This method must be used for values in the formats Int or Uint (For example, this is how the number 43605 should be written).
The conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.

8.3.10.11 TXD_CHECKSUM - Write checksum to TXD buffer

	Signature	Type	Description
Inputs	*enable*	bool	If *true*, the function performs CRC calculation.
	index	int32	Position in the TXD buffer, starting from which the calculation is performed.
	size	int32	The length of the data array to calculate the CRC.
	value index	int32	The position in the TXD buffer from which the calculated CRC will be written.
Settings	*type*	uint8	CRC calculation algorithm: **"CRC-16 (Modbus)"** Standart algorithm CRC-16 Modbus. **"CRC-8 (Maxim/Dallas)"** Standart algorithm CRC-8 Maxim/Dallas. **"XOR (8 bits)"** Sequential operation XOR. **"Sum (8 bits)"** Sequential addition of elements.
	options	uint8	**"Byte order"** - Byte order when comparing CRC (if calculated CRC = 0x0201). **"Little-endian"** The value 0x0102 will be written. **"Big-endian"** The value 0x0201 will be written. **"Invert"** If the flag is set, then before writing the CRC will be bit-wise inverted. For example, if it was 0x0201, then it will be 0xfdfe. **"Add 1"** If the flag is set, then before writing the CRC will be increased by 1. For example, if it was 0x0201, then it will be 0x0202.

The function performs CRC calculation on RXD buffer starting from *index* to *(index+size-1)*. The calculated CRC is compared with the value stored in the RXD buffer starting at *value index*.

The operations "Byte Order", "Invert", "Add 1" are performed after the CRC calculation in turn in the order of enumeration and affect the final value used in the comparison.

8.3.10.12 TXD_GET - Read value from TXD buffer

⚠ *Function was updated. The current implementation has been used from the editor version v3.4.1*

	Signature	Type	Description
Inputs	*index*	int32	The position in the TXD buffer from which to read. The very first element of the buffer has index 0.
	size	int32	The number of bytes to read from the TXD buffer. Valid values from 1 to 4.
Outputs	*value*	int32 | float	The read value.
Settings	*endian*	uint8	The byte order to be used when copying buffer elements to output *value.* For example RXD = [01,02,03,04,05,...], *index* = 0, *size* = 4: **«Little-endian»** *value* = 0x04030201. **«Big-endian»** *value* = 0x01020304. **«Big endian (2 bytes)»** *value* = 0x03040102.
	sign	bool	If the flag is set, then the function will treat the read data as a negative number if the most significant bit is “1”.

❗ *The logic of the function depends on the data type:*
*If a **variable** with the **FLOAT** type is connected to the **value** output and **size = 4**, then the function reads data from the buffer according to the IEEE754 standard. This method must be used for values that are stored in the Float format (for example, the value 12.6).*
- Otherwise, the function reads the data as INT32.
Conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.

For the convenience of receiving and sending data via the ModBus protocol, special functions MODBUS_READ and MODBUS_WRITE are provided, which are actually modified versions of RS_TRANS. The data exchange process is greatly simplified in relation to the universal functions of data exchange, because the function itself composes the request/command, controls the receipt of the response itself and parses the data itself.

8.3.10.13 MODBUS_READ – Reading data by Modbus RTU protocol

	Signature	**Type**	**Description**
Inputs	*enable*	bool	Sending requests is allowed
Outputs	*valid*	bool	*True* if the last request received a valid response and the *valueX* outputs are relevant
	state	int32	State: **"0"** – not active **"1"** – waiting for access to the interface **"2"** – access to the interface is received **"3"** – transaction in progress **"4"** – transaction completed successfully **"-1"** – interface is unavailable (not configured) **"-2"** – response timeout expired **"-3"** – unknown error
	value$_0$	int32/float/bool	The last read value 0.
	value$_1$	int32/float/bool	The last read value 1
	…		
	value$_{N-1}$	int32/float/bool	The last read value *N-1*
Settings	*N*	uint8	Number of outputs *value*
	port	uint8	The digital interface controlled by the function. If the selected interface is not configured, the function will generate an error *state = -1.*
	period	uint16	Resend request period if the *true* value is kept at the *enable* input. Re-send is performed both in case of error and in case of successful completion of the transaction.
	timeout	uint16	Time the function waits for a response after sending data. If the correct response is not received, then the transaction ends with an error *state = -2.*
	function	uint8	ModBus function to read data
	number	uint8	Network number of the polled sensor
	address	uint16	Requested data address

	type	uint8	Parameter type for reading:
			uint8 – one-byte unsigned number;
			int8 – one-byte signed number;
			uint16 – two-byte unsigned number;
			int16 – two-byte signed number;
			int32/float – four-byte signed/real number.

	endian	uint8	The byte order to be used when copying buffer elements to output *value.* For example, data = [01,02,03,04], *type* = int32: **"Little-endian"** *value* = 0x04030201. **"Big-endian"** *value* = 0x01020304. **"Big-endian (2 bytes)"** *value* = 0x03040102.
Internal	*count*	int32	Internal timeout counter

For example, let us set up the function as follows:

Parameter	Value
N	3
port	RS-485
period	1000 ms
timeout	100 ms
function	(03) Reading input registers
number	17
address	107 (0x6B)
type	int16
endian	Big-endian

Examples of a generated request and an expected response:

Request		Response	
Value (HEX)	**ModBus field name**	**Value (HEX)**	**ModBus field name**
11	Sensor network number	11	Sensor network number
03	Modbus function	03	Modbus function
00	First register address (Hi bytes)	06	Number of data bytes
6B	First register address (Lo bytes)	AE	Register value 0x006B (Hi bytes)
00	Number of registers (Hi bytes)	41	Register value 0x006B (Lo bytes)
03	Number of registers (Lo bytes)	56	Register value 0x006C (Hi bytes)
76	CRC (Hi bytes)	52	Register value 0x006C (Lo bytes)
87	CRC (Lo bytes)	43	Register value 0x006D (Hi bytes)
		40	Register value 0x006D (Lo bytes)
		49	CRC (Hi bytes)
		AD	CRC (Lo bytes)

Device will generate a request and try to send it via the RS-485 interface. After sending, the device will wait for a response within 100 ms.

After receiving the data, the device will check the packet format for compliance with the ModBus protocol, check the expected function and checksum. If all checks are passed, then the outputs will have the following values:

valid =true
value0 = 0xAE41
value1 = 0x5652
value2 = 0x4340

If the answer is not received within the allotted time, then the previous values will remain at the *valueX* outputs, the *valid* output will take the value false.

If the *enable* input remains true, then 1000 ms after the start of the previous transaction, the function will repeat sending the request and parsing the response.

ℹ *The function uses universal buffers RXD и TXD*

ℹ *For function operation, the appropriate interface must be configured in the device configuration. Configuration > RS-232/RS-485 > Device X > "Complex Events (transaction)".*

⚠ *The logic of the function depends on the data type:*
*If a **variable** with the **FLOAT** type is connected to the **value** output and **type = int32/float**, then the function reads data from the buffer according to the IEEE754 standard. This method must be used for values that are stored in the Float format (for example, the value 12.6).*
- Otherwise, the function reads the data as INT32.
Conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.

8.3.10.14 MODBUS_WRITE – Writing data via Modbus RTU protocol

	Signature	Type	Description
Inputs	*enable*	bool	Sending commands is allowed
	$value_0$	int32/float/bool	The written value 0
	$value_1$	int32/float/bool	The written value 1
	…		
	$value_{N-1}$	int32/float/bool	The written value *N-1*
Outputs	*actual*	bool	*True* if the last command received a valid response and the *valueX* inputs were successfully written
	state	int32	State: **"0"** – not active **"1"** – waiting for access to the interface **"2"** – access to the interface is received **"3"** – transaction in progress **"4"** – transaction completed successfully **"-1"** – interface is unavailable (not configured) **"-2"** – response timeout expired **"-3"** – unknown error
Settings	*N*	uint8	Number of outputs *value*
	port	uint8	The digital interface controlled by the function. If the selected interface is not configured, the function will generate an error *state = -1.*
	period	uint16	Resend request period if the *true* value is kept at the *enable* input. Re-send is performed both in case of error and in case of successful completion of the transaction.
	timeout	uint16	Time the function waits for a response after sending data. If the correct response is not received, then the transaction ends with an error *state = -2.*
	function	uint8	ModBus function to write data
	number	uint8	Network number of the polled sensor
	address	uint16	Requested data address

	type	uint8	Parameter type for writing:
			uint8 – one-byte unsigned number;
			int8 – one-byte signed number;
			uint16 – two-byte unsigned number;
			int16 – two-byte signed number;
			int32/float – four-byte signed/real number.

	endian	uint8	The byte order to be used when copying the values from the *valueX* inputs to the command body. For example, value = 0x01020304, *type* = int32/float: **"Little-endian"** *TXD* = [04,03,02,01] **"Big-endian"** *TXD* = [01,02,03,04] **"Big-endian (2 bytes)"** *TXD* = [03,04,01,02]
Internal	*count*	int32	Internal timeout counter

For example, let us set up the function as follows:

Parameter	Value	Input	Value
N	1	*value0*	3
port	RS-485		
period	1000 ms		
timeout	100 ms		
function	(06) Write one storage register		
number	17		
address	1 (0x01)		
type	uint8		
endian	Big-endian		

Examples of a generated request and an expected response:

Command		Response	
Value (HEX)	**ModBus field name**	**Value (HEX)**	**ModBus field name**
11	Sensor network number	11	Sensor network number
06	Modbus function	06	Modbus function
00	First register address (Hi bytes)	00	First register address (Hi bytes)
01	First register address (Lo bytes)	01	First register address (Lo bytes)
00	Value to ser (Hi bytes)	00	Set value (Hi bytes)
03	Value to set (Lo bytes)	03	Set value (Lo bytes)
76	CRC (Hi bytes)	76	CRC (Hi bytes)
87	CRC (Lo bytes)	87	CRC (Lo bytes)

When sending a command to set one storage register, an echo is expected in response

The device will generate a command and try to send it via the RS-485 interface. After sending, the device will wait for a response within 100 ms.

After receiving the data, the device will check the packet format for compliance with the ModBus protocol, check the expected function and checksum. If all checks are passed, then the *actual* output will be set to true.

If no response is received within the allotted time, then the *actual* output will be set to false.

If the *enable* input remains true, then 1000 ms after the start of the previous transaction, the function will repeat sending the command and parsing the response.

The function uses universal buffers RXD и TXD

For function operation, the appropriate interface must be configured in the device configuration. Configuration > RS-232/RS-485 > Device X > "Complex Events (transaction)".

The logic of the function depends on the data type:
*If a **variable** with the **FLOAT** type is connected to the **value** input and **type = int32/float**, then the function writes data according to the IEEE754 standard. This method must be used for values that are stored in the Float format (for example, the value 12.016).*
- Otherwise, the function writes the data as an integer number. This method must be used for values in the formats Int or Uint (For example, this is how the number 43605 should be written).
Conversion is performed automatically using the FROM_FLOAT and TO_FLOAT functions.