



**LOGIC LAB**

# User Manual

LogicLab User Manual  
Revision 5.14 - 2020-07-03  
Published by Axel S.r.l.  
Via del Cannino, 3  
21020 Crosio della Valle (VA)  
© Axel S.r.l. 2020.  
All Rights Reserved.



## Contents

<b>1.</b>	<b>Introduction</b>	<b>1</b>
1.1	Conventions used in this document	1
<b>2.</b>	<b>Overview</b>	<b>3</b>
2.1	The workspace	3
2.1.1	The output window	4
2.1.2	The status bar	4
2.1.3	The document bar	4
2.1.4	The watch window	5
2.1.5	The Operators and blocks window	5
2.1.6	The workspace window	6
2.1.7	The source code editors	7
2.1.8	The toolbars	7
<b>3.</b>	<b>Using the environment</b>	<b>9</b>
3.1	Layout customization	9
3.2	Toolbars	9
3.2.1	Showing/hiding toolbars	9
3.2.2	Moving toolbars	9
3.3	Docking windows	9
3.3.1	Showing/hiding tool windows	9
3.3.2	Floating tool windows	10
3.3.3	Docking tool windows	10
3.3.4	Auto-Hide tool windows	10
3.4	Working with windows	10
3.4.1	The document bar	11
3.4.2	The window menu	11
3.5	Full screen mode	12
3.6	Environment options	12
3.6.1	General	12
3.6.2	Graphic Editor	13
3.6.3	Text Editors	14
3.6.4	Language	14
3.6.5	custom Tools	14
3.6.6	Merge	16
<b>4.</b>	<b>Managing projects</b>	<b>17</b>
4.1	Creating a new project	17
4.2	Uploading the project from the target device	17
4.3	Saving the project	18



4.3.1	Persisting changes to the project	18
4.3.2	Saving to an alternative location	18
4.3.3	Autosave	18
4.3.4	Backup Copies	19
<b>4.4</b>	<b>Managing existing projects</b>	<b>19</b>
4.4.1	Opening an existing LogicLab project	19
4.4.2	Editing the project	19
4.4.3	Closing the project	20
<b>4.5</b>	<b>Distributing projects</b>	<b>20</b>
<b>4.6</b>	<b>Project options</b>	<b>21</b>
4.6.1	general	21
4.6.2	Code generation	21
4.6.3	Build output	22
4.6.4	Download	23
4.6.5	Debug	24
4.6.6	Build events	24
4.6.7	Cross reference	25
<b>4.7</b>	<b>Selecting the target device</b>	<b>26</b>
<b>4.8</b>	<b>Working with libraries</b>	<b>27</b>
4.8.1	The library manager	27
4.8.2	Exporting to a library	28
4.8.3	Importing from a library or another source	29
4.8.4	Updating existing libraries	30
<b>5.</b>	<b>Managing project elements</b>	<b>31</b>
<b>5.1</b>	<b>Program Organization Units</b>	<b>31</b>
5.1.1	Creating a new Program Organization Unit	31
5.1.2	Editing POUs	32
5.1.3	Source code encryption/DECRYPTION	33
<b>5.2</b>	<b>Variables</b>	<b>34</b>
5.2.1	Global variables	34
5.2.2	Local variables	36
5.2.3	Creating multiple VARIABLES	37
5.2.4	Textual editor for variables	38
<b>5.3</b>	<b>Tasks</b>	<b>38</b>
5.3.1	Assigning a program to a task	38
5.3.2	Task configuration	39
<b>5.4</b>	<b>Derived data types</b>	<b>39</b>
5.4.1	Typedefs	39
5.4.2	Structures	41
5.4.3	Enumerations	42





5.4.4	Subranges	43
5.4.5	MACROS	44
5.4.6	INTERfaces	46
<b>5.5</b>	<b>Browse the project</b>	<b>47</b>
5.5.1	Object Browser	47
5.5.2	Search with the Find in project command	55
<b>5.6</b>	<b>Working with LogicLab extensions</b>	<b>56</b>
<b>5.7</b>	<b>Project Custom Workspace</b>	<b>56</b>
5.7.1	Enable The Custom Workspace Into An Existing Project	57
5.7.2	Workspaces Migration	57
5.7.3	Custom Workspace Basic Units	57
5.7.4	Custom Workspace Operations	58
5.7.5	Workspace Elements With Limitations	59
<b>6.</b>	<b>Editing the source code</b>	<b>61</b>
<b>6.1</b>	<b>Instruction List (IL) editor</b>	<b>61</b>
6.1.1	Editing functions	61
6.1.2	Reference to PLC objects	61
6.1.3	Automatic error location	62
6.1.4	Bookmarks	62
<b>6.2</b>	<b>Structured Text (ST) Editor</b>	<b>62</b>
6.2.1	Creating and editing ST objects	62
6.2.2	Editing functions	62
6.2.3	Reference to PLC objects	63
6.2.4	Automatic error location	63
6.2.5	Bookmarks	63
<b>6.3</b>	<b>Ladder Diagram (LD) editor</b>	<b>64</b>
6.3.1	Creating a new LD document	64
6.3.2	Adding/Removing networks	64
6.3.3	Labeling networks	65
6.3.4	Inserting contacts	65
6.3.5	Inserting coils	66
6.3.6	Inserting blocks	66
6.3.7	Editing coils and contacts properties	66
6.3.8	Editing networks	66
6.3.9	Modifying properties of blocks	67
6.3.10	Getting information on a block	67
6.3.11	Automatic error retrieval	67
6.3.12	Inserting variables	68
6.3.13	Inserting constants	68
6.3.14	Inserting expression	68
6.3.15	Comments	68



6.3.16	Branches	69
<b>6.4</b>	<b>Function Block Diagram (FBD) editor</b>	<b>70</b>
6.4.1	Creating a new FBD document	70
6.4.2	Adding/Removing networks	70
6.4.3	Labeling networks	70
6.4.4	Inserting and connecting blocks	71
6.4.5	Editing networks	71
6.4.6	Modifying properties of blocks	72
6.4.7	Getting information on a block	72
6.4.8	Automatic error retrieval	72
<b>6.5</b>	<b>Sequential Function Chart (SFC) Editor</b>	<b>72</b>
6.5.1	Creating a new SFC document	72
6.5.2	Inserting a new SFC element	72
6.5.3	Connecting SFC elements	73
6.5.4	Assigning an action to a step	73
6.5.5	Transitions conditions	74
6.5.6	Specifying the destination of a jump	76
6.5.7	Editing SFC networks	76
<b>6.6</b>	<b>Variables editor</b>	<b>76</b>
6.6.1	Opening a variables editor	77
6.6.2	Creating a new variable	78
6.6.3	Editing variables	78
6.6.4	Deleting variables	80
6.6.5	Sorting variables	81
6.6.6	Copying variables	81
<b>6.7</b>	<b>Object Oriented</b>	<b>81</b>
6.7.1	Enable Object Oriented programming	81
6.7.2	Methods	83
6.7.3	Interfaces	85
6.7.4	Object Oriented in graphic languages	89
<b>7.</b>	<b>Compiling</b>	<b>91</b>
7.1	Compiling the project	91
7.1.1	Image file loading	91
7.2	Compiler output	92
7.2.1	Compiler errors	92
7.3	Command-line compiler	93
<b>8.</b>	<b>Launching the application</b>	<b>95</b>
8.1	Setting up the communication	95
8.1.1	Saving the last used communication port	96
8.2	On-line status	97



8.2.1	Application status	97
8.2.2	Connection status	97
<b>8.3</b>	<b>Downloading the application</b>	<b>98</b>
8.3.1	Controlling source code download	98
<b>8.4</b>	<b>Simulation</b>	<b>99</b>
<b>8.5</b>	<b>Control the PLC execution</b>	<b>99</b>
8.5.1	Halt	100
8.5.2	Cold restart	100
8.5.3	Warm restart	100
8.5.4	Hot restart	100
8.5.5	Reboot target	100
<b>9.</b>	<b>Debugging</b>	<b>101</b>
<b>9.1</b>	<b>Watch window</b>	<b>101</b>
9.1.1	Opening and closing the watch window	102
9.1.2	Adding items to the watch window	102
9.1.3	Removing a variable	105
9.1.4	Refreshment of values	105
9.1.5	Changing the format of data	106
9.1.6	Working with watch lists	107
9.1.7	Autosave watch list	108
<b>9.2</b>	<b>Oscilloscope</b>	<b>108</b>
9.2.1	Opening and closing the oscilloscope	109
9.2.2	Adding items to the oscilloscope	109
9.2.3	Removing a variable	111
9.2.4	Variables sampling	111
9.2.5	Controlling data acquisition and display	111
9.2.6	Saving, restoring and printing the graph	116
<b>9.3</b>	<b>Edit and debug mode</b>	<b>117</b>
<b>9.4</b>	<b>Live debug</b>	<b>117</b>
9.4.1	SFC animation	117
9.4.2	LD animation	118
9.4.3	FBD animation	118
9.4.4	IL and ST animation	119
<b>9.5</b>	<b>Triggers</b>	<b>119</b>
9.5.1	Trigger window	119
9.5.2	Debugging with trigger windows	124
<b>9.6</b>	<b>Graphic triggers</b>	<b>133</b>
9.6.1	Graphic trigger window	133
9.6.2	Debugging with the graphic trigger window	139
<b>9.7</b>	<b>Breakpoints</b>	<b>148</b>



9.7.1	The breakpoint tool	148
9.7.2	Set and remove a breakpoint	149
9.7.3	Working with breakpoints	150
9.7.4	Removing breakpoints	152
<b>10.</b>	<b>LogicLab reference</b>	<b>153</b>
10.1	Menus reference	153
10.1.1	File menu	153
10.1.2	Edit menu	153
10.1.3	View menu	155
10.1.4	Project menu	155
10.1.5	Online Menu	157
10.1.6	Debug menu	157
10.1.7	Scheme MENU FOR FBD	158
10.1.8	Scheme menu for LD	160
10.1.9	Scheme SFC menu	161
10.1.10	Variables menu	162
10.1.11	Window menu	162
10.1.12	Tools menu	163
10.1.13	Help menu	163
10.2	Toolbars reference	163
10.2.1	Main toolbar	163
10.2.2	FBD toolbar	163
10.2.3	LD toolbar	163
10.2.4	SFC toolbar	164
10.2.5	Project toolbar	164
10.2.6	Network toolbar	164
10.2.7	Debug toolbar	164
<b>11.</b>	<b>Language reference</b>	<b>165</b>
11.1	Common elements	165
11.1.1	Basic elements	165
11.1.2	Elementary data types	166
11.1.3	Derived data types	168
11.1.4	Literals	170
11.1.5	Variables	171
11.1.6	Program Organization Units	174
11.1.7	Object Oriented reference	177
11.1.8	IEC 61131-3 standard functions	177
11.2	Instruction List (IL)	198
11.2.1	Syntax and semantics	198
11.2.2	Standard operators	199
11.2.3	Calling Functions and Function blocks	200





11.3	Function Block Diagram (FBD)	200
11.3.1	Representation of lines and blocks	201
11.3.2	Direction of flow in networks	201
11.3.3	Evaluation of networks	201
11.3.4	Execution control elements	202
11.4	Ladder Diagram (LD)	204
11.4.1	Power rails	204
11.4.2	Link elements and states	205
11.4.3	Contacts	205
11.4.4	Coils	206
11.4.5	Operators, functions and function blocks	207
11.5	Structured Text (ST)	207
11.5.1	Expressions	207
11.5.2	Statements in ST	208
11.6	Sequential Function Chart (SFC)	213
11.6.1	Steps	213
11.6.2	Transitions	215
11.6.3	Rules of evolution	216
11.6.4	SFC control flags	218
11.6.5	Check a SFC POU from other programs	219
11.7	LogicLab Language Extensions	220
11.7.1	Macros	220
11.7.2	Pointers	221
11.7.3	Waiting statement	222
<b>12.</b>	<b>Errors Reference</b>	<b>223</b>
12.1	Compile time error messages	223



# 1. INTRODUCTION

## 1.1 CONVENTIONS USED IN THIS DOCUMENT

Text Type	Description
<i>Command, Key</i>	Name of the command or keyboard shortcuts key.
Code	Source code text.
 [Context menu]	Toolbar icon and context menu voice.
[Context menu]	Context menu voice without any icon.
<i>Menu&gt;Item</i>	For menu items hierarchy, the ">" symbol is used. A record File>Open Project is equivalent to "the Open Project item under the File menu".
 <i>Menu&gt;Item</i>	Same as above including the icon shown in the toolbar.
(See Paragraph) (See Chapter)	Link to related subject within this guide.
<i>Terminology</i>	Important term or concept.





## 2. OVERVIEW

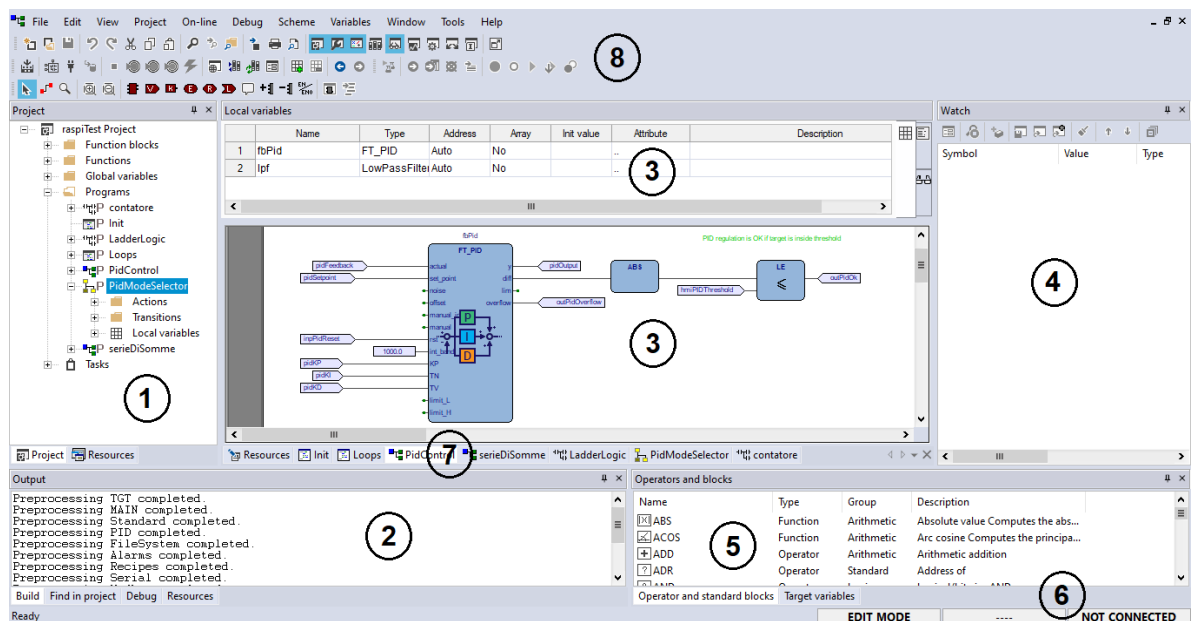
LogicLab is an IEC61131-3 Integrated Development Environment supporting the whole range of languages defined in the standard.

In order to support the user in all the activities involved in the development of an application, LogicLab includes:

- textual source code editors for the Instruction List (briefly, IL) and Structured Text (briefly, ST) programming languages (see Chapter 6.);
- graphical source code editors for the Ladder Diagram (briefly, LD), Function Block Diagram (briefly, FBD), and Sequential Function Chart (briefly, SFC) programming languages (see Chapter 6.);
- a compiler, which translates applications written according to the IEC standard directly into machine code, avoiding the need for a run-time interpreter, thus making the program execution as fast as possible (see Chapter 7);
- a communication system which allows the download of the application to the target environment (see Chapter 8);
- a rich set of debugging tools, ranging from an easy-to-use watch window to more powerful tools, which allows the sampling of fast changing data directly on the target environment, ensuring the information is accurate and reliable (see Chapter 9).

### 2.1 THE WORKSPACE

The figure below shows a view of LogicLab's workspace, including many of its more commonly used components.



**1.** Workspace window **2.** Output window **3.** Source code editors **4.** Watch window **5.** Operators and blocks window **6.** Status bar **7.** Document bar **8.** Toolbars

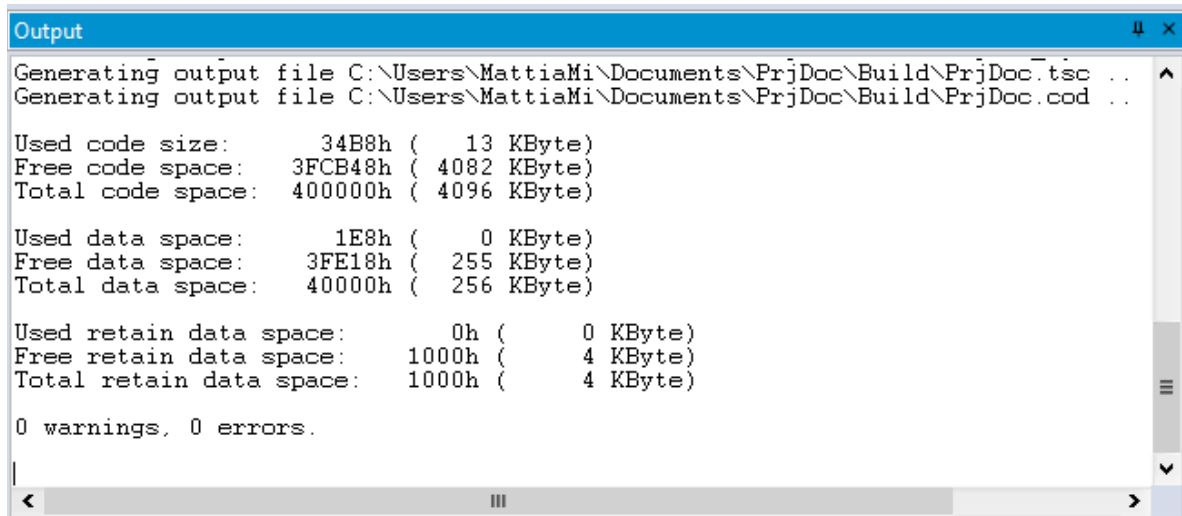
The following paragraphs give an overview of these elements.





### 2.1.1 THE OUTPUT WINDOW

The *Output* window is the place where LogicLab prints its output messages. This window contains four tabs: *Build*, *Find in project*, *Debug*, and *Resources*.



#### Build

The *Build* panel displays the output of the following activities:

- opening a project;
- compiling a project;
- downloading code to a target.

#### Find in project

This panel shows the result of the *Find in project* activity.

#### Debug

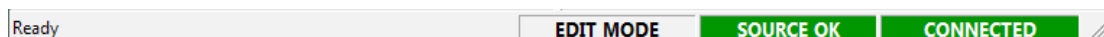
The *Debug* panel displays information about advanced debugging activities (for example, breakpoints). Depending on the target device you are interfacing with, LogicLab can print on this output window every PLC run-time error (for example, division by zero), locating the exact position where the error occurred.

#### Resources

The *Resources* panel displays messages related to the specific target device LogicLab is interfacing with.

### 2.1.2 THE STATUS BAR

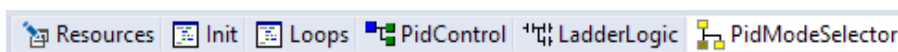
The *Status* bar displays the state of the application at its left border, and an animated control reporting the state of communication at its right border.



For further details see paragraph 8.2.1 and 8.2.2.

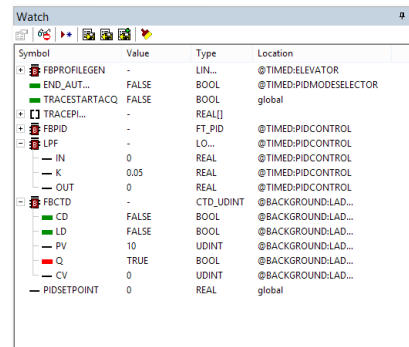
### 2.1.3 THE DOCUMENT BAR

The *Document* bar lists all the documents currently open for editing in LogicLab.



## 2.1.4 THE WATCH WINDOW

The *Watch* window is one of the many debugging tools supplied by LogicLab. Among the other debugging tools, it is worth mentioning the Oscilloscope, the triggers, and the live debug mode (see Paragraph 9.2).



## 2.1.5 THE OPERATORS AND BLOCKS WINDOW

The *Operators and blocks* window contains a set of different panels, which fall into the categories explained in the following paragraphs.

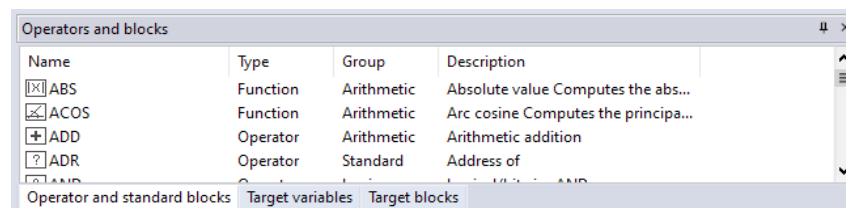
You can choose the display mode by clicking the right button of your mouse. In the [\[View list\]](#) mode, each element is represented by its name and icon. Instead, a table appears in the [\[View detail\]](#) mode, each row of which is associated with one of the embedded elements. The latter mode also displays the *Type* (Operator/Function) and the description of each element.

If you right-click one of the elements of this panel, and you click [\[Object properties\]](#) from the dialog box, then a window appears with further details on the element you selected (input and output supported types, name of input and output pins, etc.).

In the [\[View folder\]](#) mode each element is grouped into the folder to which it belongs. These folders are useful to logically group the elements.

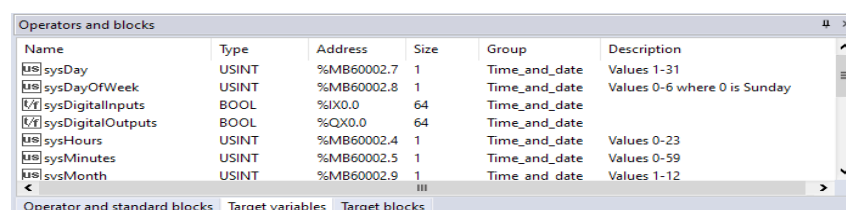
### 2.1.5.1 OPERATORS AND STANDARD BLOCKS

This panel lists basic language elements, such as operators and functions defined by the IEC 61131-3 standard.



### 2.1.5.2 TARGET VARIABLES

This panel lists all the system variables, also called target variables, which are the interface between firmware and PLC application code.



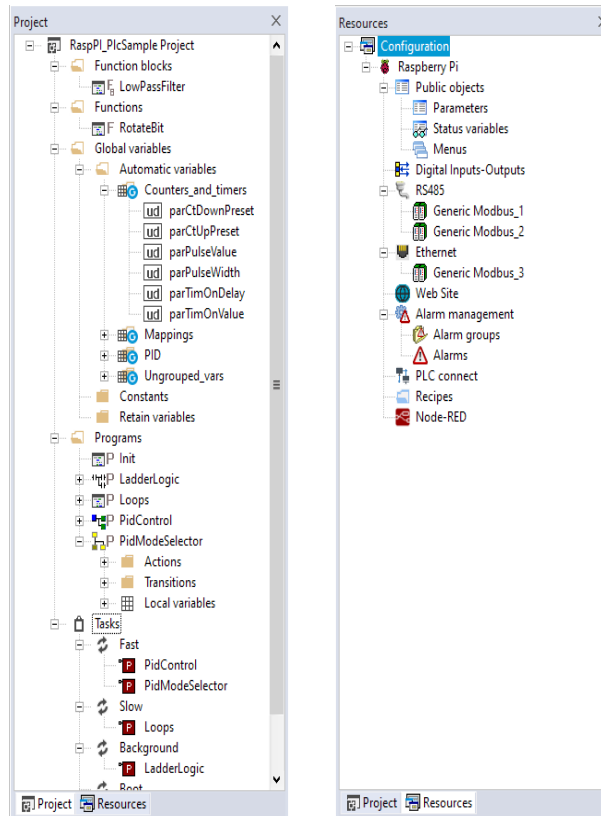
### 2.1.5.3 TARGET BLOCKS

This panel lists all the system functions and function blocks available on the specific target device.

Name	Type	Group	Description
GetTime_us	Function		
PLC_Status	Structure		PLC engine status
SignalTask	Function		
sysLogWriteMsg	Function		
sysTraceLog	Function		

### 2.1.6 THE WORKSPACE WINDOW

The *Workspace* window consists of two distinct panels, as shown in the following picture.



#### 2.1.6.1 PROJECT

The *Project* panel contains all the elements of the project, visualized in a tree-like organization; custom folders can be added to the project tree in order to reorganize the elements accordingly to the user preferences.

Among the other elements, there's also the *task list*, which represent the available tasks and the relative assigned programs (see Paragraph 5.3).

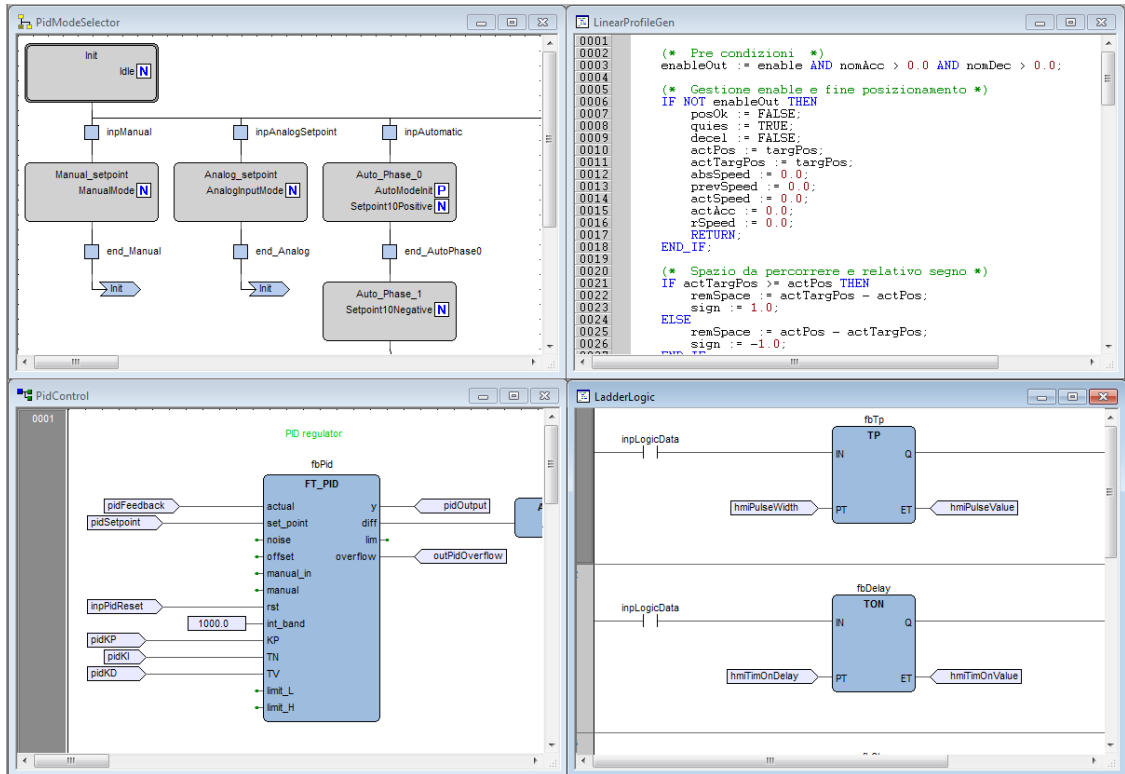
#### 2.1.6.2 RESOURCES

The contents of the *Resources* panel depends on how the target device is interfacing with LogicLab: it may include configuration elements, schemes, wizards, and so on.



## 2.1.7 THE SOURCE CODE EDITORS

The LogicLab programming environment includes a set of editors to manage, edit, and print source files written in any of the 5 programming languages defined by the IEC 61131-3 standard (see Chapter 6).



The definition of both global and local variables is supported by specific spreadsheet-like editors

Local variables								
	Name	Type	Address	Array	Init value	Attribute	Description	
1	fbDelay	TON	Auto	No		..		
2	fbCtu	CTU_UDINT	Auto	No		..		
3	fbCtd	CTD_UDINT	Auto	No		..		
4	fbTp	TP	Auto	No		..		
5	ettp	UDINT	Auto	No		..		

## 2.1.8 THE TOOLBARS

LogicLab allows the user to hide or show specific toolbars to fully customize the workspace. Each operation in the environment, to realize a program (for example add a variable), can be performed through the menus; the toolbars contain icons which work as shortcut for the menus commands (see paragraph 3.2).





## 3. USING THE ENVIRONMENT

This chapter shows you how to deal with the many UI elements LogicLab is composed of, in order to let you set up the IDE in the way which best suits to your specific development process.

### 3.1 LAYOUT CUSTOMIZATION

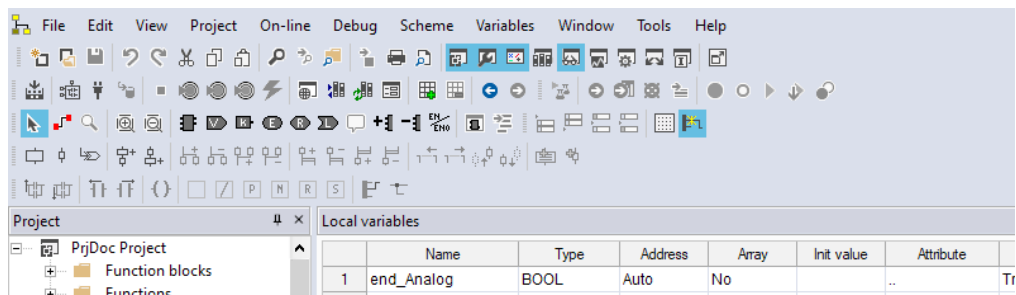
The layout of LogicLab's workspace can be freely customized in order to suit your needs. LogicLab takes care to save the layout configuration on application exit, in order to persist your preferences between different working sessions.

### 3.2 TOOLBARS

#### 3.2.1 SHOWING/HIDING TOOLBARS

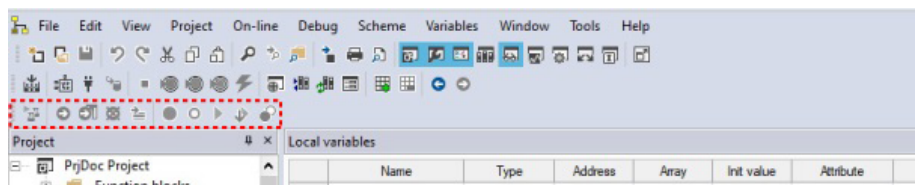
In details, in order to show (or hide) a toolbar, open the [View>Toolbars](#) menu and select the desired toolbar (for example, the *FBD* bar).

The toolbar is then shown (hidden).

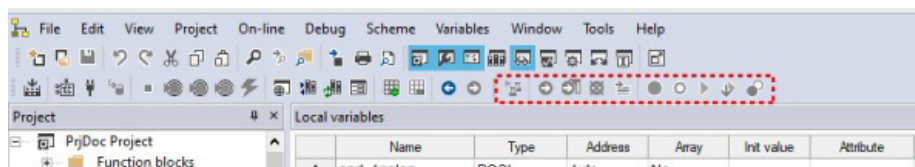


#### 3.2.2 MOVING TOOLBARS

You can move a toolbar by clicking on its left border and then dragging and dropping it to the destination.



The toolbar shows up in the new position.



### 3.3 DOCKING WINDOWS

#### 3.3.1 SHOWING/HIDING TOOL WINDOWS

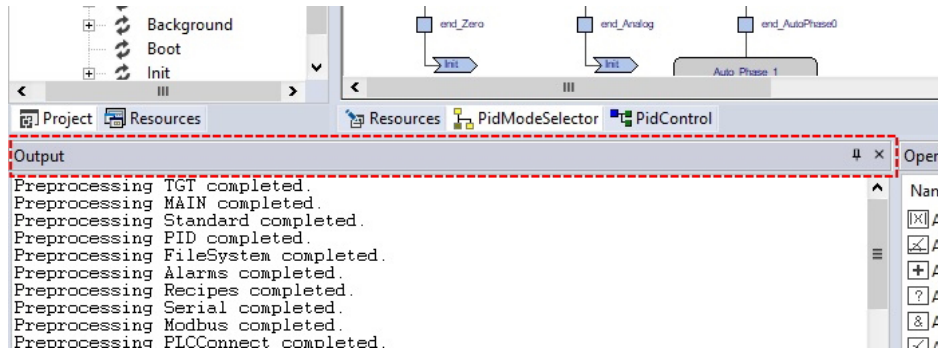
The [View>Tool windows](#) menu allows you to show (or hide) a tool window (for example,



the *Watch* window, the *Properties* window, the *Library* window...). The window is then shown (hidden).

### 3.3.2 FLOATING TOOL WINDOWS

You can undock any window from its default location in LogicLab and move it anywhere, just click on its title bar and drag it to the location you want.

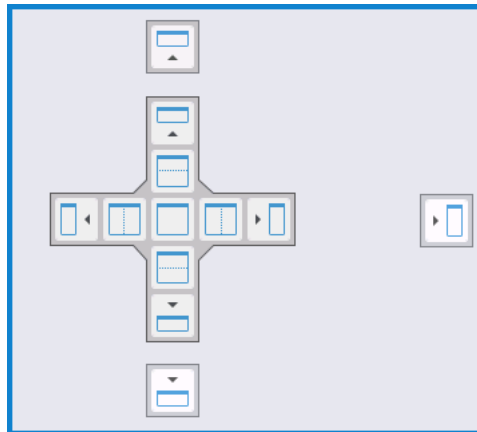


Take back a window to its most recent docked location simply double-click the title bar of the window.

### 3.3.3 DOCKING TOOL WINDOWS

LogicLab shows you a guide diamond when you drag a window to another location to help you easily re-dock the window.

While dragging a window move the mouse cursor on the position of the guide diamond you want to use as new window position.



Tool windows can be fastened to one side of a frame in LogicLab or within a frame.

### 3.3.4 AUTO-HIDE TOOL WINDOWS

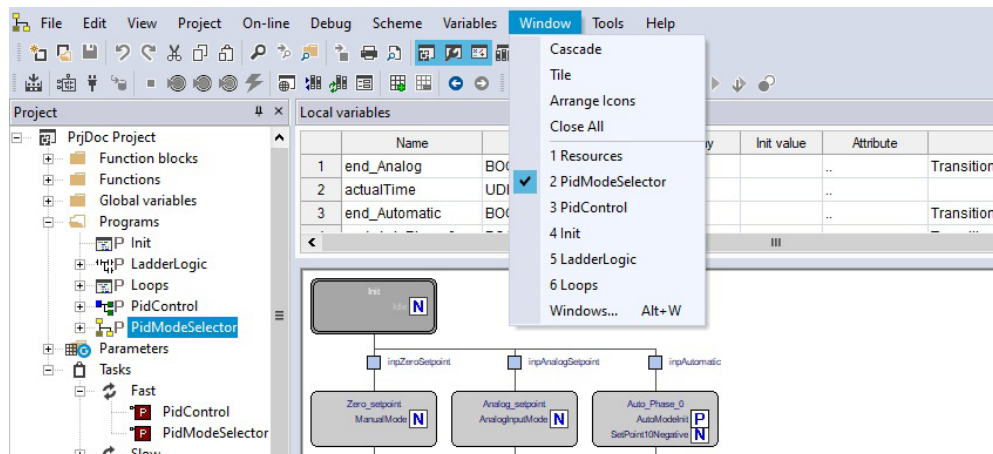
By the pin button on the top right corner of the window you can switch the window to auto-hide mode or to regular docking mode.

## 3.4 WORKING WITH WINDOWS

LogicLab allows to open many source code editors so that the workspace could get rather messy.

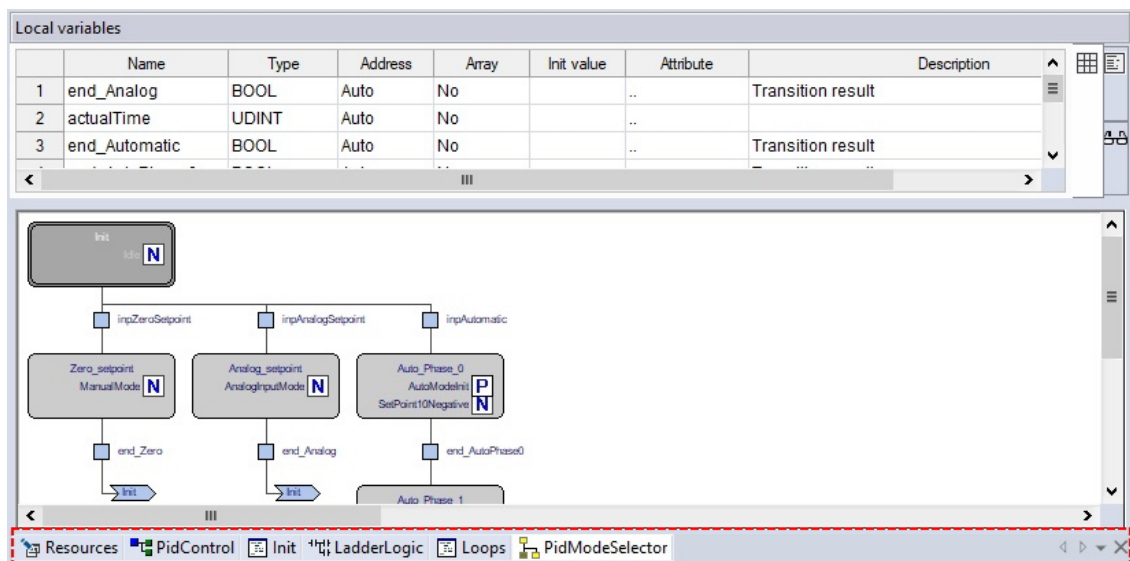


You can easily navigate between these windows through the *Document* bar and the *Window* menu.



### 3.4.1 THE DOCUMENT BAR

The *Document* bar allows to switch between all the currently open editors, simply by clicking on the corresponding name.



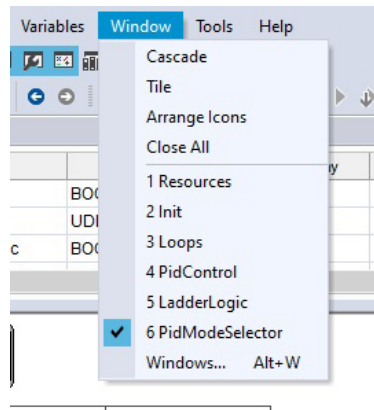
At the right side of the *Document* bar, there are four buttons: the first two allow you to browse the different open editors in case their number exceeds the document bar size; the third shows you the currently open editors in a cascade menu; the last one allows you to close the currently selected editor window.

### 3.4.2 THE WINDOW MENU

The *Window* menu is an alternative to the *Document* bar: it lists all the currently open editors and allows to switch between them.







Moreover, this menu supplies a few commands to automate some basic tasks, such as closing all windows.

### 3.5 FULL SCREEN MODE

In order to ease the coding of your application, you may want to switch on the full screen mode. In full screen mode, the source code editor extends to the whole working area, making easier the job of editing the code, notably when graphical programming languages (that is, LD, FBD, and SFC) are involved.

You can switch on and off the full screen mode with the `View>Full screen`.

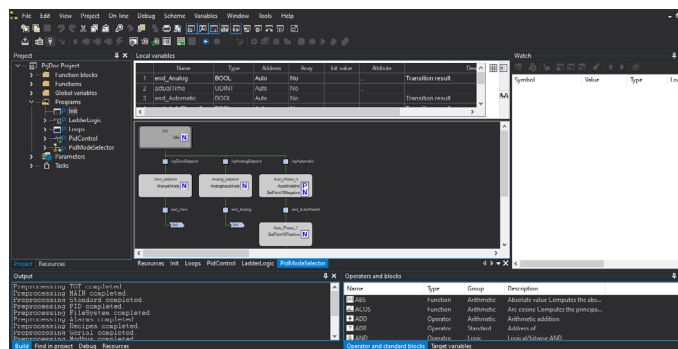
### 3.6 ENVIRONMENT OPTIONS

If you click `File>Options...`, a multi-tab dialog box appears and lets you customize some options of LogicLab.

#### 3.6.1 GENERAL

##### 3.6.1.1 VISUAL THEME

*Colour Theme:* allows you to turn LogicLab to dark theme mode (or return to standard mode), in order to have an environment that best suits your preferences.



##### 3.6.1.2 SAVE OPTIONS

*Max previous version to keep:* if set greater than 0 indicates the maximum number of copies of the project that must be zipped and stored in the `PreviousVersions` folder.



### 3.6.1.3 COMMUNICATION

*Use last port:* if enabled, the last used port will be set as the default one.

### 3.6.1.4 TOOLTIP

*Enable tooltip on editors:* if enabled, small information boxes will appear when user places the cursor over a symbol in the editors.

### 3.6.1.5 OUTPUT WINDOW

You can specify the family and the size of the font used for the output window.

### 3.6.1.6 TOOL WINDOWS

You can specify the family and the size of the font used for the other tool windows.

*Reset bars positions:* the layout of the dock bars in the IDE will be resetted to default positions and dimensions. In order to take effect LogicLab must be restarted.

### 3.6.1.7 SOURCE EDITOR OPTIONS

*ST-LD: auto declaration of variables:* allows the creation of new variables in the moment they're first used in the code. In LD programs, when you name a variable block, that name is suggested as a new variable. In ST program, upon entering a new code line, that line is parsed to look for new variables to suggest.

*Track active object in project tree:* if this option is checked, when using the *Go to symbol (shift+F12)* function, also the project tree will track the selected object.

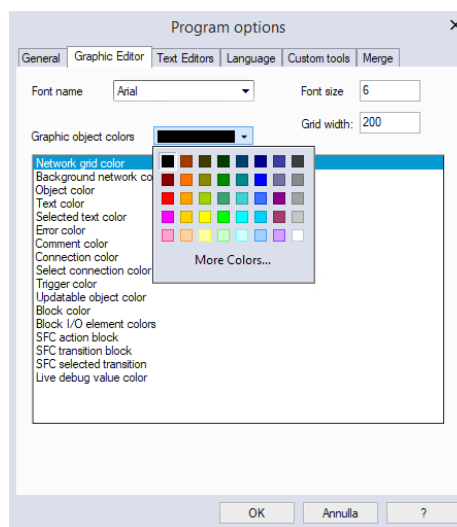
*Automatically restore last open editors:* if this option is checked, LogicLab will save open editors when closing the project. When reopening the project, also the same editors will be restored.

## 3.6.2 GRAPHIC EDITOR

This panel lets you edit the properties of the LD, FBD, and SFC source code editors.

You can specify the family and the size of the font used for graphical editors.

You can modify also the colours of the graphical object.



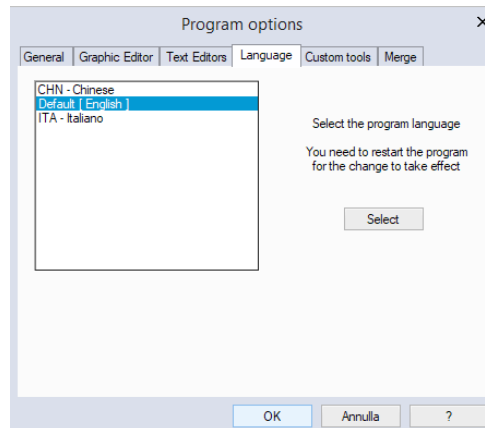
### 3.6.3 TEXT EDITORS

You can specify the family, the size and the color of the font both for code and variable editors.

### 3.6.4 LANGUAGE

You can change the language of the environment by selecting a new one from the list shown in this panel.

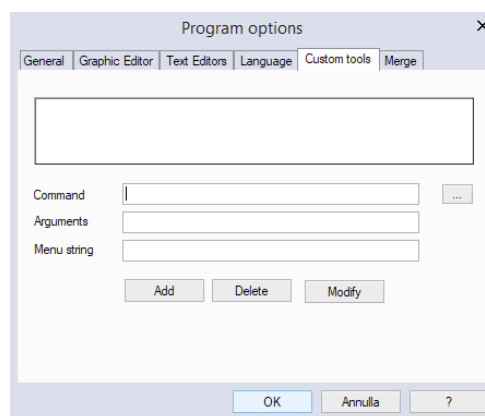
After selecting the new language, press the *Select* button and confirm by clicking *OK*. This change will be effective only the next time you start LogicLab.



### 3.6.5 CUSTOM TOOLS

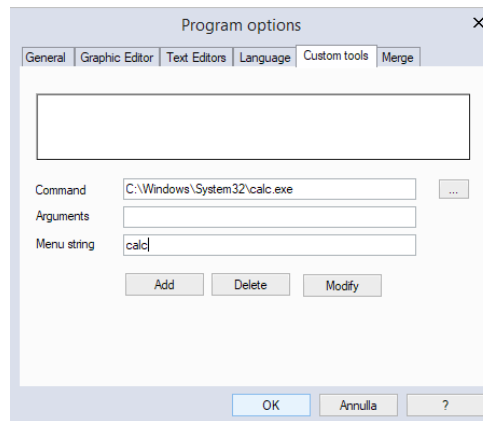
You can add up to 16 commands to the *Custom Tools* menu. These commands can be associated with any program that will run on your operating system. You can also specify arguments for any command that you add to the *Custom Tools* menu. The following procedure shows you how to add a tool to the *Custom Tools* menu.

- 1) Type the full path of the executable file of the tool in the *Command* text box. Otherwise, you can specify the filename by selecting it from Windows Explorer, which you open by clicking the *Browse* button.

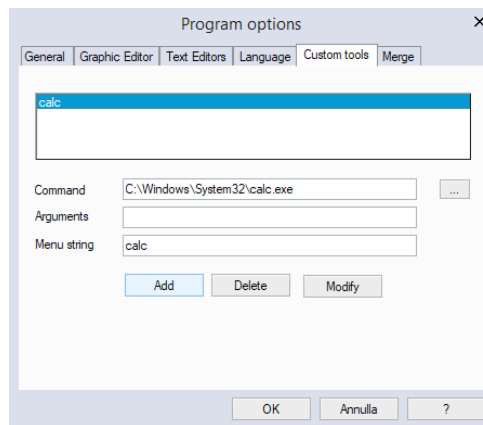


- 2) In the *Arguments* text box, type the arguments - if any - to be passed to the executable command mentioned at step 1. They must be separated by a space.
- 3) Enter in *Menu string* the name you want to give to the tool you are adding. This is the string that will be displayed in the *Tools* menu.
- 4) Press *Add* to effectively insert the new command into the suitable menu.
- 5) Press *OK* to confirm, or *Cancel* to quit.

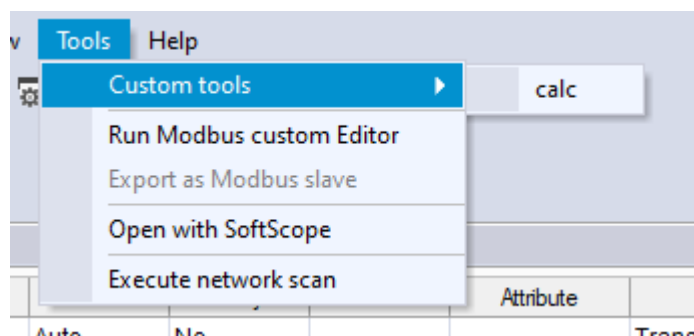
For example, let us assume that you want to add *Windows calculator* to the *Tools* menu:  
 - Fill the fields of the dialog box as displayed.



- Press *Add*. The name you gave to the new tool is now displayed in the list box at the top of the panel.

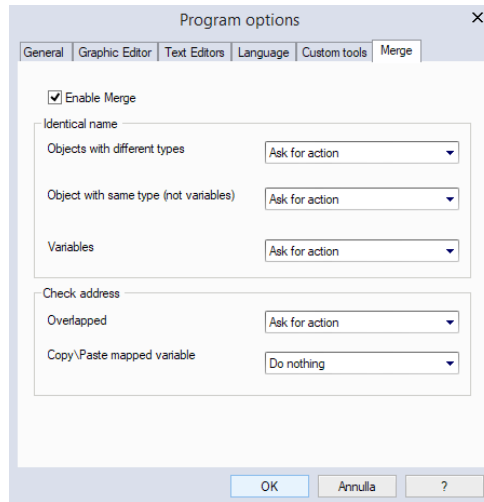


And in the *Custom Tools>Calc* menu as well.



### 3.6.6 MERGE

Here you can set the merge function behaviour (see Paragraph 4.8.3.2 for more details).



## 4. MANAGING PROJECTS

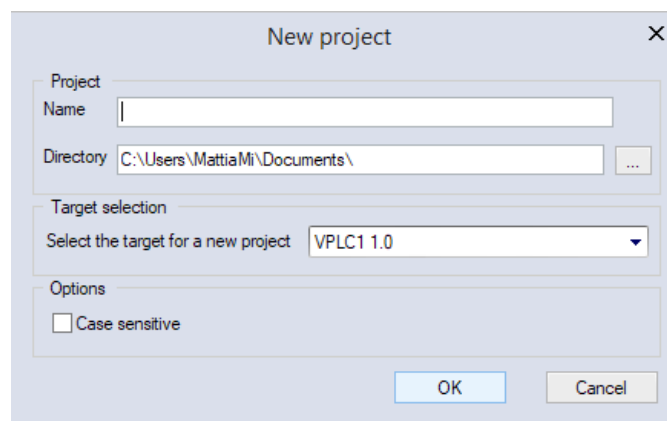
This chapter focuses on LogicLab projects.

A project corresponds to a PLC application and includes all the required elements to run that application on the target device, including its source code, links to libraries, information about the target device and so on.

The following paragraphs explain how to properly work with projects and their elements.

### 4.1 CREATING A NEW PROJECT

To start a new project, click **File>New project** of the LogicLab main window.



You are required to enter the name of the new project in the *Name* text area. The string you enter will also be the name of the folder which will contain all the files making up the LogicLab project. The path name in the *Directory* area indicates the default location of this folder.

*Target selection* allows you to specify the target device which will run the project.

Finally, you can make the project case-sensitive by activating the related option. Note that, by default, this option is not active, in compliance with IEC 61131-3 standard: when you choose to create a case-sensitive project, it will not be standard-compliant.

When you confirm your decision to create a new project and the whole required information has been provided, LogicLab completes the operation, creating the project directory and all project files; then, the project is opened.

The list of devices, from which you can select the target for the project you are creating, depends on the contents of LogicLab catalog, which contains the list of available target devices

If the desired target is missing, either you have run the wrong setup executable or you have to run a separate setup which is responsible to update the catalog to include the target device. In both cases, you should contact your hardware supplier for support.

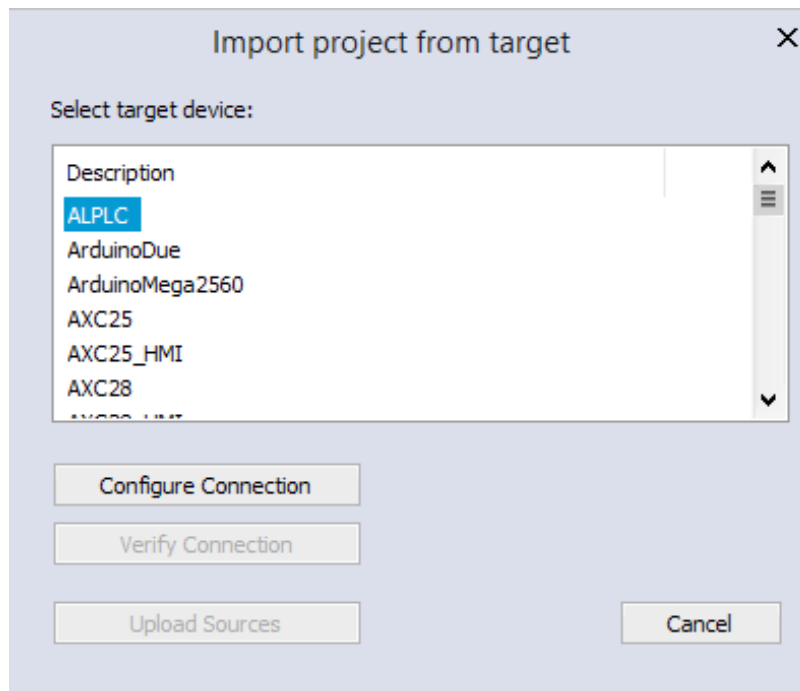
### 4.2 UPLOADING THE PROJECT FROM THE TARGET DEVICE

Depending on the target device you are interfacing with, you may be able to upload a working LogicLab project from the target itself.

In order to upload the project from the target device, follow the procedure below:

- 1) Click the **File>Import project from target** menu voice of the LogicLab main window, which opens the Target list dialog box.





- 2) From the shown list select the target device from which you want to upload the project.
- 3) *Configure Connection* with correct parameters (see Paragraph 8.1 for more details).
- 4) You can test the connection with the target device by *Verify Connection* button. LogicLab tries to establish the connection and reports the test result.
- 5) If the connection is available confirm the operation by clicking on the *Upload Sources* button. When the application upload completes successfully, the project is open and ready for editing.

## 4.3 SAVING THE PROJECT

### 4.3.1 PERSISTING CHANGES TO THE PROJECT

When you make any change to the project (for example, you add a new Program Organization Unit) you are required to save the project in order to persist that change.

To save the project, you can select the corresponding item [File>Save project](#).

### 4.3.2 SAVING TO AN ALTERNATIVE LOCATION

You can also use the [File>Save project As ...](#) command to rename the project, change its format or modify the location of where you want save the file.

LogicLab asks you to select the new destination (which must be an empty directory), then saves a copy of the project to that location and opens this new project file for editing.

### 4.3.3 AUTOSAVE

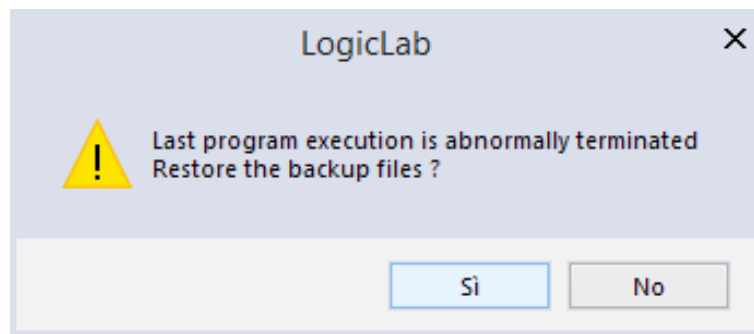
LogicLab includes an *AutoSave* feature that periodically saves your project as you work on it.

*AutoSave* saves data in a separate folder, called *Backup*, stored at the same location of the project folder.

*AutoSave* protects your project in the event that LogicLab unexpectedly quits. When LogicLab is started again, if the *Backup* folder is present, you are asked to restore the last



valid backup file of the project.



When you close LogicLab correctly the *Backup* folder and its contents are deleted. You can specify the interval time (in minutes) between saving.

#### 4.3.4 BACKUP COPIES

LogicLab includes a backup feature of the previous version of the project on which you are working.

When you explicitly save the project, LogicLab saves the current version (before save) of the project in the *PreviousVersions* folder stored at the same location of the project folder; You can set the upper limit of the backup files to be kept on your PC. By default this is 10, set to 0 if you want to disable this feature (see Paragraph 3.6 for more details).

### 4.4 MANAGING EXISTING PROJECTS

#### 4.4.1 OPENING AN EXISTING LOGICLAB PROJECT

To open an existing project, click [File>Open project](#) of LogicLab's main window, or in the *Welcome page* (when no project is open). This causes a dialog box to appear, which lets you load the directory containing the project and select the relative project file.

#### 4.4.2 EDITING THE PROJECT

In order to modify an element of a project, you need first to open that element by double-clicking its name, which you can find by browsing the tree structure of the project tab of the *Workspace* bar.

By double-clicking the name of the object you want to modify, you open an editor consistent with the object type: for example, when you double-click the name of a project POU, the appropriate source code editor is shown; if you double-click the name of a global variable, the variable editor is shown.

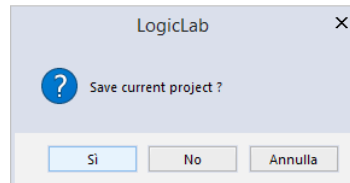
Note that LogicLab prevents you from applying changes to elements of a project, when at least one of the following conditions holds:

- you are in debug mode.
- It is an object of an included library (whereas you can modify an object that you imported from a library).
- The project is opened in read-only mode (view project).



### 4.4.3 CLOSING THE PROJECT

You can terminate the working session either by explicitly closing the project or by exiting LogicLab. In both cases, when there are changes not yet persisted to file, LogicLab asks you to choose between saving and discarding them.



To close the project, select the item `File>Close project`; LogicLab shows the *Welcome page*, so that you can rapidly start a new working session.

## 4.5 DISTRIBUTING PROJECTS

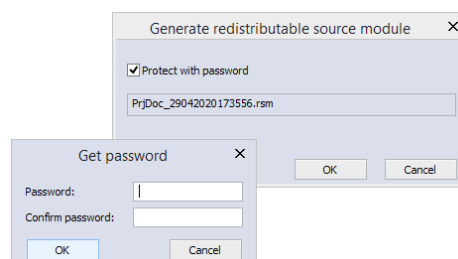
When you need to share a project with another developer you can send him/her either a copy of the project file(s) or a redistributable source module (RSM) generated by LogicLab.

In the former case, the number of files you have to share depends on the format of the project file:

- PLC single project file (`.ppjs` file extension): the project file itself contains the whole information needed to run the application (assuming the receiving developer has an appropriate available target device) including all source code modules, so that you need to share only the `.ppjs` file.
- PLC multiple project file (`.ppjx` or `.ppj` file extension): the project file contains only the links to the source code modules composing the project, which are stored as single files in the project directory. You need to share the whole directory.
- Full XML PLC project file (`.plcprj`): the project file is generated entirely in XML language. The information contained in the project file and its behaviour are the same as `.ppjs` file extension.

Alternatively, you can generate a redistributable source module (RSM) with the corresponding item `Project>Generate redistributable source module`.

LogicLab notifies you of the name of the RSM file and lets you choose whether to protect the file with a password or not. If you choose to protect the file, LogicLab asks you to insert the password.



The advantages of the RSM file format are:

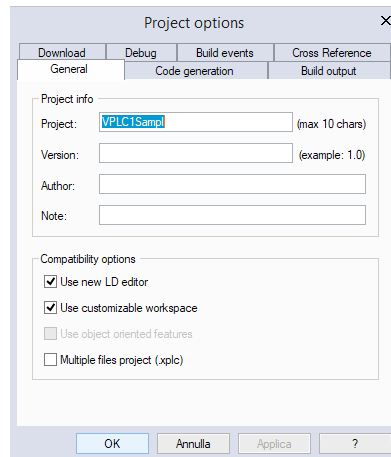
- the source code is encoded in binary format, thus it cannot be read by third parties which do not use LogicLab, making a transfer over the Internet more secure;
- it can be protected with a password, which will be required by LogicLab on file opening;
- being a binary file, its size is reduced.

## 4.6 PROJECT OPTIONS

You can edit some significant project properties choosing **Project>Options...**.

### 4.6.1 GENERAL

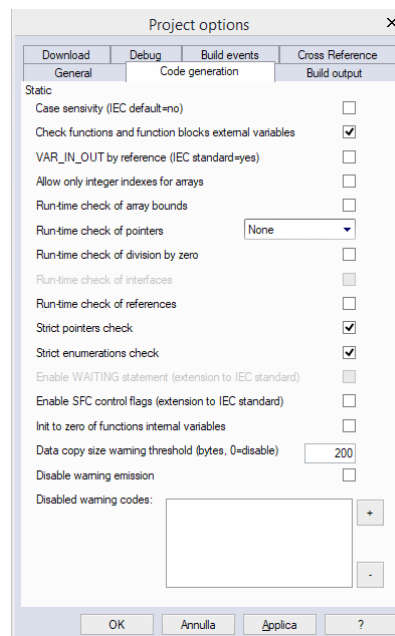
Here you can set some basic properties related to the project, such as its application name and version.



- *Use new LD editor*: the new Ladder Diagram editor is easier to use, by helping you in common operations working on the diagram will be faster and more efficient. Note that, by default, this option is active.
- *Use customizable workspace*: allows you to manage your project tree in order to reach a more efficient workspace. Note that, by default, this option is active.
- *Use object oriented feature*: allows you to use object oriented programming; this feature may not be always available, it depends on the target device implementation.
- *Multiple files project*: allows you to save project in .xplc format.

### 4.6.2 CODE GENERATION

Here you can edit some properties about code generation.

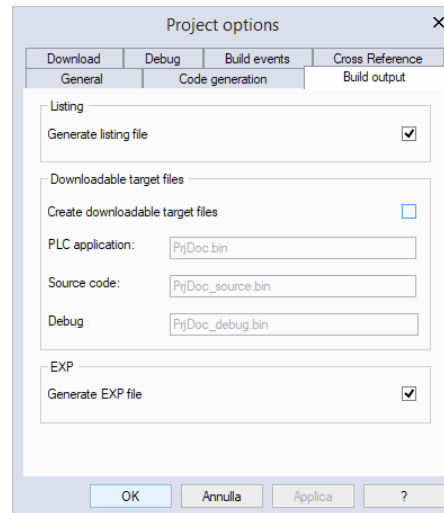


- *Case sensitivity*: you can set the project as case-sensitive checking this option. Note that, by default, this option is not active.
- *Check function and function block external variables*: if this option is disabled, all functions and function blocks can access to global variables without declaring them as external variables. Note that, by default, this option is enabled respecting the IEC 61131-3 standard.
- *VAR\_IN\_OUT by reference*: if checked, the variables declared as VAR\_IN\_OUT of a function block will be treated as reference variables, accordingly to IEC standards.
- *Allow only integer indexes for arrays*: if this option is checked you cannot use *BYTE*, *WORD* or *DWORD* as array indexes.
- *Run-time check of array bounds*: if this option is checked some check code is added to verify that array indexes are not out of bounds during run-time. This option can be set depending on target device.
- *Run-time check of pointers*: this combo allows you to choose if and when the pointer will be tested for their validity before their use. Selecting *NONE*, the check will never be done; selecting *FULL*, the check will always be done, under every circumstances; selecting *ONLY IF NOT NULL*, the check will be done only if the pointer value is not null. The check is done by calling a user-defined function `checkptr` on target. Therefore this option can be set depending on target device.
- *Run-time check of division by zero*: if this option is checked some check code is added to verify that divisions by zero are not performed on arrays during run-time. This option can be set depending on target device.
- *Run-time check of references*: if this option is checked, allows a references validity check; if a reference is dereferenced to *null*, a runtime error is generated. This option can be set depending on target device
- *Strict pointers check*: if enabled, pointer and pointed type validity check is added. The type of a pointer variable must be the same of its pointed variable; if not, an error is generated.
- *Strict enumerations check*: if enabled, enumeration assignment type validity check is added. You can assign enum only to the same type; if not, an error is generated.
- *Enable WAITING statement (extension to standard)*: if this option is checked the *WAITING* construct for the ST language is added as IEC 61131-3 extension (see Paragraph 11.7.3 for more details).
- *Enable SFC control flags (extension to standard)*: if this option is checked, HOLD and RESET flags for SFC POU are enabled.
- *Init to zero of function internal variables*: if this option is checked, the initial value of the internal variables of the functions, will be set to zero as default.
- *Data copy size warning threshold (bytes, 0=disable)*: when arrays or structures are copied, if their dimension exceed the specified threshold, a warning is emitted in order to inform the possible loss of performance of the PLC. If the threshold is set to 0, no warnings are emitted.
- *Disable warning emission*: if this option is checked warning emissions are not printed on the output window.
- *Disable warning codes*: you can specify a list of warning code that will not be printed on the output window.

### 4.6.3 BUILD OUTPUT

Here you can edit some significant properties of the output files generated by compiling operation.





### Listing section

- *Generate listing file*: if this option is checked the compiler will generate a listing file named as *projectname.lst*.

### Downloadable target files section

- *Create downloadable target files*: if this option is checked the compiler will generate the binary files that can be downloaded to the target. You can specify custom filenames or use default ones.

**Please note that only valid Windows filename are accepted!**

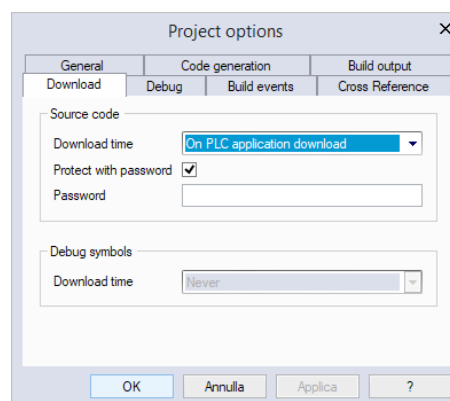
- *PLC application* (active only if *Create downloadable target files* is checked): this field specifies the name of the PLC application binary file. By default *projectname.bin*.
- *Source code* (active only if *Create downloadable target files* is checked): this field specifies the name of the Source code binary file. By default *projectname.\_source.bin*.
- *Debug* (active only if *Create downloadable target files* is checked): this field specifies the name of the Debug symbol binary file. By default *projectname.\_debug.bin*

### Generate EXP file section

- *Generate EXP file*: if this option is checked the compiler will generate an EXP file named as *projectname.exp*

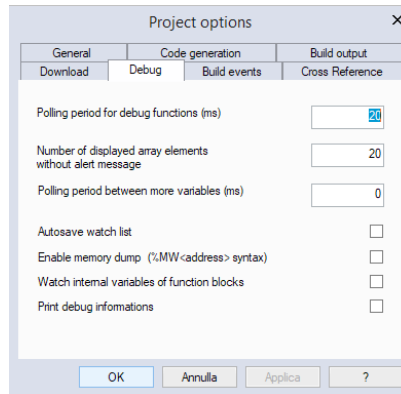
## 4.6.4 DOWNLOAD

Here you can edit some significant properties of the download behaviour (see Paragraph 8.3.1 for more information).



## 4.6.5 DEBUG

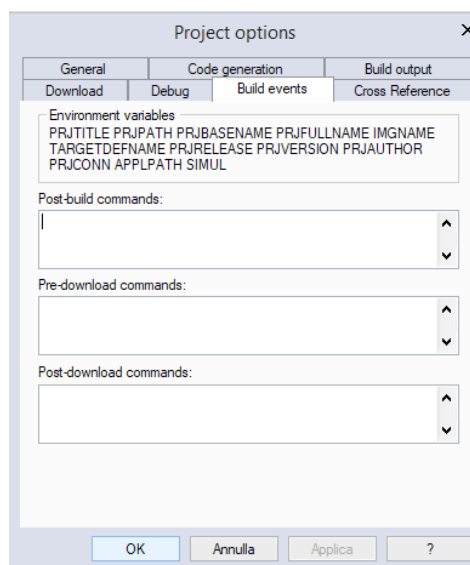
Here you can edit some significant properties of the debug behaviour.



- *Polling period for debug function* (ms): set the active sampling period of the functions' status.
- *Number of displayed array elements without alert message*: specifies the maximum number of array elements to be added in watch window without being alerted.
- *Polling period between more variables* (ms): set the sleep period between sampling two variables.
- *Autosave watch list*: if checked (not by default) the watch list status will be saved into a file, when the project is closed (see Paragraph 9.1.7 for more details).
- *Enable memory dump*: advanced debug feature, allows the user to put in watch directly physical memory addresses (with format %MW without subindexes)
- *Watch internal variables of function block*: when putting a function block instance in watch you'll see also the internal variable values; with this option disabled, only the input and output variables of that function block will be displayed in the watch.
- *Print debug informations*: when compiling, additional information are shown in the output window.

## 4.6.6 BUILD EVENTS

Here you can specify commands that run before the build starts or after the build finishes. You can also use a set of defined environment variables listed on the top of the window.

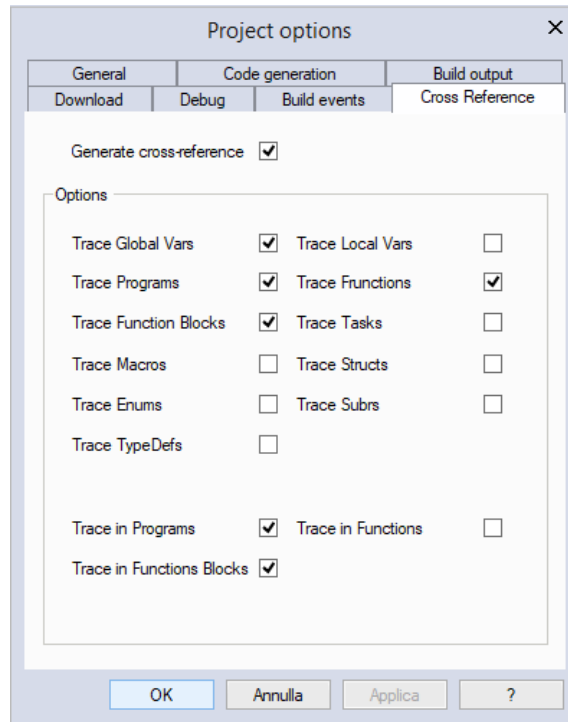


## 4.6.7 CROSS REFERENCE

This window allows you to enable the cross reference.

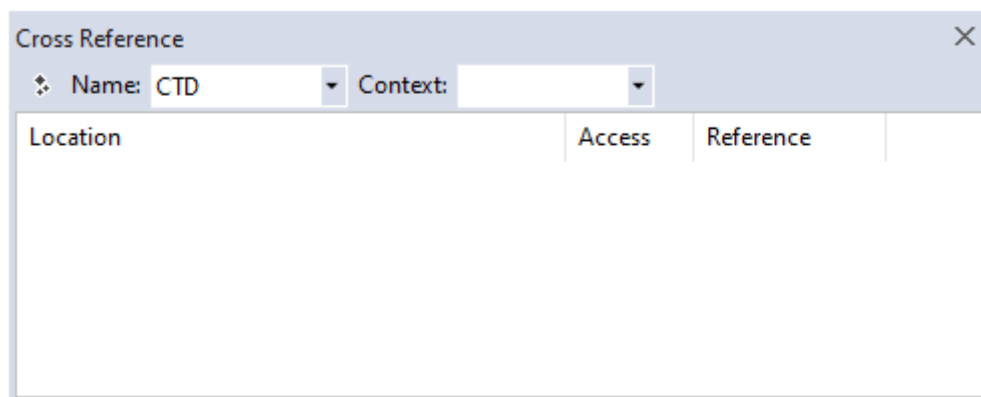
Since using the cross reference will generate additional information, making the project a little heavier, it has to be explicitly enabled.

This window allows you also to set the list of elements that will be searched when executing a cross reference research.



### 4.6.7.1 USING THE CROSS REFERENCE

Once the cross reference has been enabled, open the relative window selecting **View > Tool window > Cross reference**; the following window should be opened:

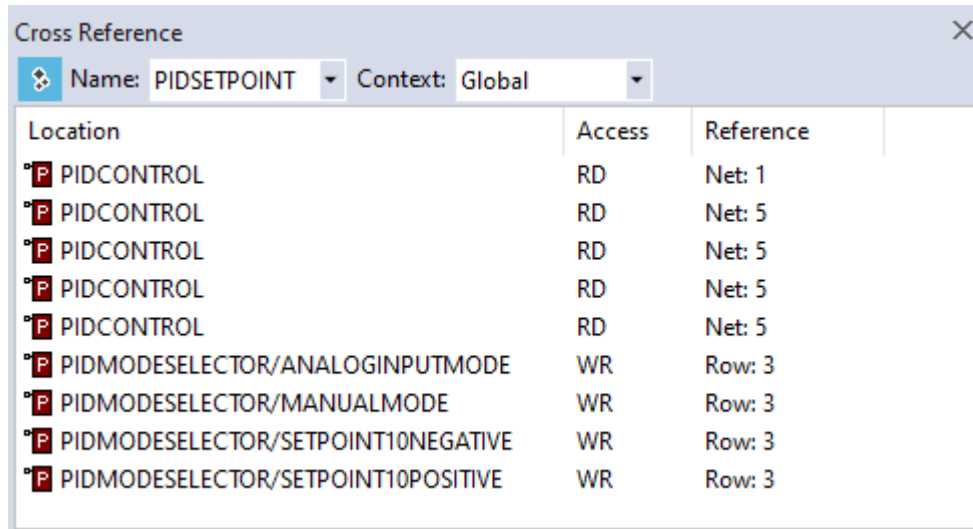


In the menu labeled *Name*: you will see a collection of all the symbols found by the system looking into the groups selected in the Cross reference panel. You can select a symbol from that menu, or you can press the automatic cross reference button:



and simply select a symbol from the project tree.

Once you have selected a symbol, the cross reference window will show you all the occurrences of that symbol and information such as: location of the POU where it's used, how it is used and where it is used inside the POU.



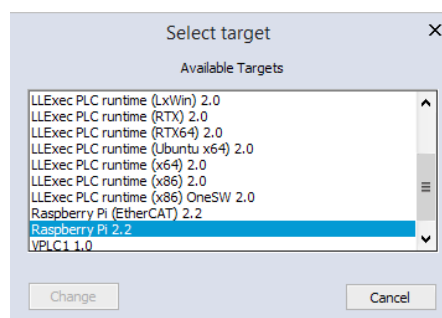
From the menu labeled *Context*: you can filter the result.

By double clicking one of the occurrences in the result list, the relative POU window will be opened and the element (code line or block) where the symbol is used will be highlighted.

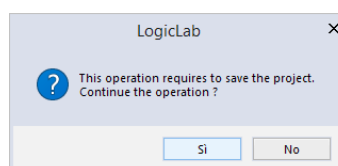
## 4.7 SELECTING THE TARGET DEVICE

You may need to port a PLC application on a target device which differs from the one you originally wrote the code for. Follow the instructions below to adapt your LogicLab project to a new target device.

- 1) Click **Project>Select target** menu of the LogicLab main window. This causes the following dialog box to appear.



- 2) Select one of the target devices listed in the combo box.
- 3) Click *Change* to confirm your choice, *Cancel* to abort.
- 4) If you confirm, LogicLab displays the following dialog box.



Press *Yes* to complete the conversion, *No* to quit.

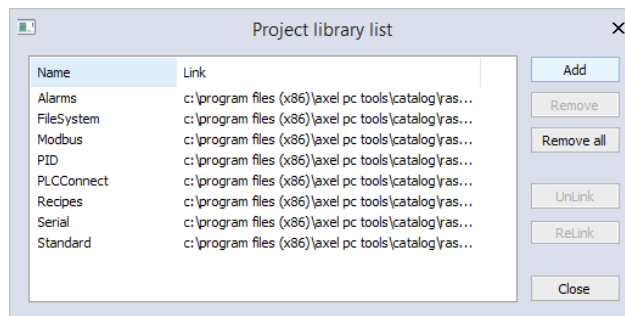
If you press *Yes*, LogicLab updates the project to work with the new target.

It also makes a backup copy of the project file(s) in a sub-directory inside the project directory, so that you can roll-back the operation by manually (i.e., using Windows Explorer) replacing the project file(s) with the backup copy.

## 4.8 WORKING WITH LIBRARIES

Libraries are a powerful tool for sharing objects between LogicLab projects. Libraries are usually stored in dedicated source file, whose extension is `.p11`.

### 4.8.1 THE LIBRARY MANAGER



The library manager lists all the libraries currently included in a LogicLab project. It also allows you to include or remove libraries.

To access the library manager, click `Project > Library manager`.

#### 4.8.1.1 INCLUDING A LIBRARY

The following procedure shows you how to include a library in a LogicLab project, which results in all the library's objects becoming available to the current project.

Including a library means that a reference to the library's `.p11` file is added to the current project, and that a local copy of the library is made. Note that you cannot edit the elements of an included library, unlike imported objects.

If you want to copy or move a project which includes one or more libraries, make sure that references to those libraries are still valid in the new location.

- 1) Click `Project > Library manager`, which opens the *Library manager* dialog box.
- 2) Press the *Add* button, which causes an explorer dialog box to appear, to let you select the `.p11` file of the library you want to open.
- 3) When you have found the `.p11` file, open it either by double-clicking it or by pressing the *Open* button. The name of the library and its absolute pathname are now displayed in a new row at the bottom of the list in the white box.
- 4) Repeat step 1, 2, and 3 for all the libraries you wish to include.
- 5) When you have finished including libraries, press either *OK* to confirm, or *Cancel* to quit.

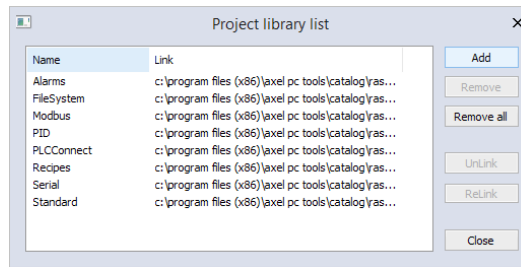
#### 4.8.1.2 REMOVING A LIBRARY

The following procedure shows you how to remove an included library from the current project. Remember that removing a library does not mean erasing the library itself, but the project's reference to it.

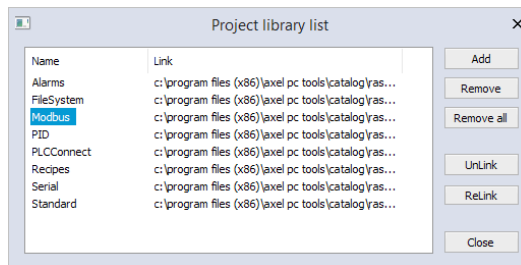
- 1) Click `Project > Library manager` menu of the LogicLab main window, which opens the *Library manager* dialog box.







Select the library you wish to remove by clicking its name once. The *Remove* button is now enabled.

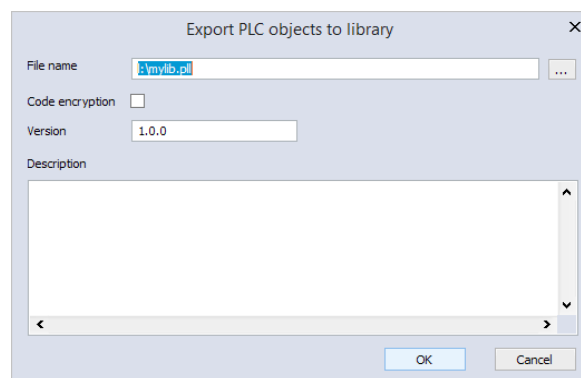


- 2) Click the *Remove* button, which causes the reference to the selected library to disappear from the *Project library* list.
- 3) Repeat for all the libraries you wish to remove. Alternatively, if you want to remove all the libraries, you can press the *Remove all* button.
- 4) When you have finished removing libraries, press either *OK* to confirm, or *Cancel* not to apply changes.

## 4.8.2 EXPORTING TO A LIBRARY

You may export an object from the currently open project to a library, in order to make that object available to other projects. The following procedure shows you how to export objects to a library.

- 1) Look for the object you want to export by browsing the tree structure of the project tab of the *Workspace* bar, then click once the name of the object.
- 2) Click **Project>Export object to library**. This causes the following dialog box to appear.



- 3) Enter the destination library by specifying the location of its *.pll* file. You can do this by:
  - typing the full pathname in the white text box;
  - clicking the *Browse* button, in order to open an explorer dialog box which allows you to browse your disk and the network.
- 4) You may optionally choose to encrypt the source code of the POU you are exporting,

in order to protect your intellectual property.

- 5) If you chose and already existing library file, you can enter a version number in the *version* field in order to keep track of library changes.
- 6) Click *OK* to confirm the operation, otherwise press *Cancel* to quit.

If at Step 3 of this procedure you enter the name of a non-existing *.pll* file, LogicLab creates the file, thus establishing a new library.

#### 4.8.2.1 UNDOING EXPORT TO A LIBRARY

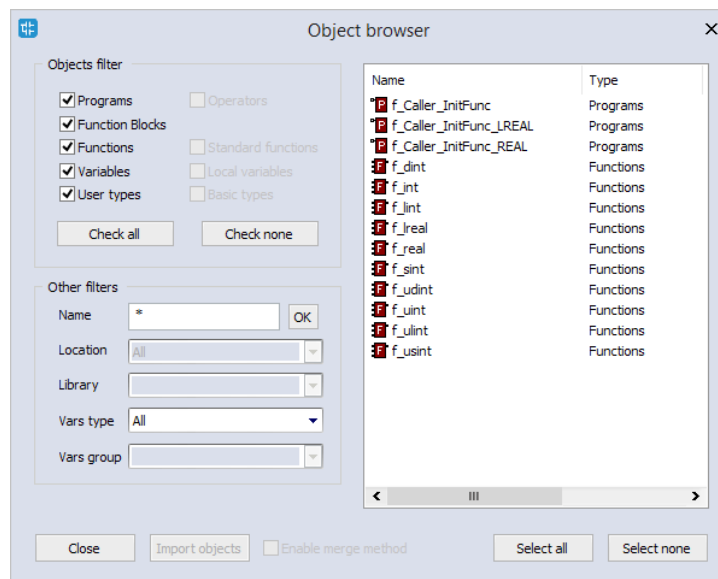
So far, it is not possible to undo export to a library. The only possibility to remove an object is to create another library containing all the objects of the current one, except the one you wish to delete.

### 4.8.3 IMPORTING FROM A LIBRARY OR ANOTHER SOURCE

You can import an object from a library in order to use it in the current project. When you import an object from a library, the local copy of the object loses its reference to the original library and it belongs exclusively to the current project. Therefore, you can edit imported objects, unlike objects of included libraries.

There are two ways of getting a POU from a library. The following procedure shows you how to import objects from a library.

- 1) Click **Project>Import objects**. This causes an explorer dialog box to appear, which lets you select the *.pll* file of the library you want to open.
- 2) When you have found the *.pll* file, open it either by double-clicking it or by pressing the *Open* button. The dialog box of the library explorer appears in foreground. Each tab in the dialog box contains a list of objects of a type consistent with the tab's title.



- 3) Select the tab of the type of the object(s) you want to import. You can also make simple queries on the objects in each tab by using *Filters*. However, note that only the *Name* filter actually applies to libraries. To use it, select a tab, then enter the name of the desired object(s), even using the *\** wildcard, if necessary.
- 4) Select the object(s) you want to import, then press the *Import object* button.
- 5) When you have finished importing objects, press indifferently *OK* or *Cancel* to close the *Library* browser.



### 4.8.3.1 UNDOING IMPORT FROM A LIBRARY

When you import an object in a LogicLab project, you actually make a local copy of that object. Therefore, you just need to delete the local object in order to undo import.

### 4.8.3.2 MERGE FUNCTION

When you import objects in a LogicLab project or insert a copied mapped variable, you may encounter an overlapping address or duplicate naming warning.

By setting the corresponding environment options (see Paragraph 3.6 for more details) you can choose the behaviour that LogicLab should keep when encountering those problems.

The possible actions are:


		Ask	Automatic	Take from library	Do nothing
<b>Naming behaviour</b>	If different types	X	X		X
	If same type but not variables	X	X	X	
	If both variables	X	X	X	
<b>Address behaviour</b>	If address overlaps	X	X	X	
	Copy/paste mapped variable		X		X

- *Ask* (default): user has to decide every time an action is required.
- *Automatic*: a valid name or address is automatically generated by LogicLab and assigned to the imported object.
- *Take from library*: the name or the address is taken from the imported object.
- *Do nothing*: the name or the address of the objects in the project are not modified.

After importing objects, LogicLab generates a log file in the project folder with detailed info.

### 4.8.4 UPDATING EXISTING LIBRARIES

If you edit a linked library file you can refresh its content on the project without closing LogicLab.

- 1) Click  *Project>Refresh all libraries*.
- 2) If the file is correct, LogicLab updates the linked library content and prints a successful message in the output window, otherwise no changes are made on the existing linked library.

## 5. MANAGING PROJECT ELEMENTS

This chapter shows you how to deal with the elements which compose a project, namely: Program Organization Units (briefly, POU), tasks, derived data types, and variables.

### 5.1 PROGRAM ORGANIZATION UNITS

A POU is a Program Organization Unit of type Program, Function or Function block.

This paragraph shows you how to add new POU to the project, how to edit and eventually remove them.

In paragraph 5.1 we will be using a program as example of a generic POU, but the same can be achieved also for functions and function blocks

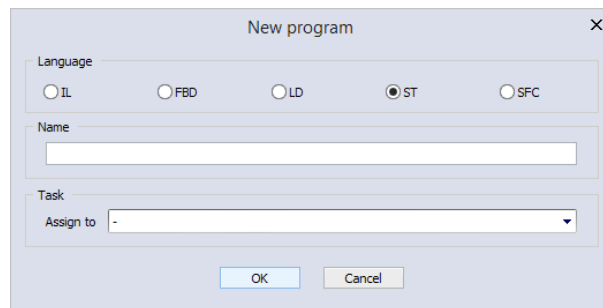
#### 5.1.1 CREATING A NEW PROGRAM ORGANIZATION UNIT

In order to Add a POU select the appropriate voice of the menu

*Project>New Object>New program*

Please note that the item of the sub-menu may change according to the type of the POU you want to create.

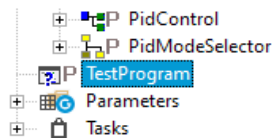
LogicLab will show you a dialog box in where you must select the specific language for the new POU and enter its name.



Confirm the operation by clicking on the OK button.

Alternatively, you can create a new POU from the context menu by selecting a folder or the root element of the project (see Paragraph 5.7.4).

After creating a new program, an alert icon (interrogation mark) appears below the new program icon.

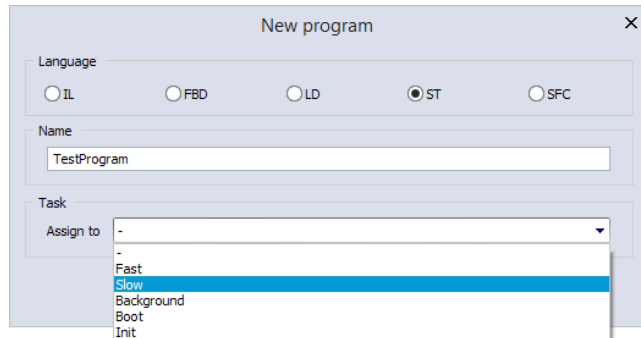


This alert icon indicates that the program is not yet associated to a task. Refer to paragraph 5.3.1 to assign the program to the desired task.



### 5.1.1.1 ASSIGNING A PROGRAM TO A TASK AT CREATION TIME

When creating a new program, LogicLab gives you the chance to assign that program to a task at the same time: select the task you want the program to be assigned to from the list shown in the *Task* section of the *New program* window.



### 5.1.2 EDITING POU S

To edit a POU, open it by double-clicking it from the project tree. The relative editor opens and lets you modify the source code of the POU.

#### Changing the name of the POU:

Select a POU from the project tree then open its context menu by right-clicking on its icon, choose *Rename Program*.

#### Duplicating a POU:

Select a POU from the project tree then choose the appropriate voice of the menu *Project>Duplicate object*.

Enter the name of the new duplicated POU and confirm the operation.

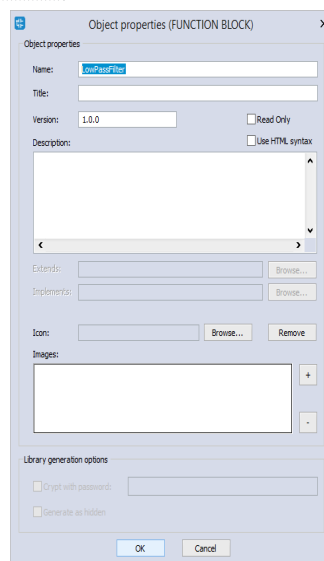
#### Deleting POU s

Select a POU from the project tree then choose the appropriate voice of the menu *Project>Delete Object*.

Confirm the operation to delete the POU.

#### Editing POU properties

Select a POU from the project tree then open its context menu by right-clicking on its icon, choose *Edit Program Properties*.



A window will appear allowing you to edit information such as version number, description and, if the POU is a function or a function block, also the icon and the image.

The icon must have a maximum size of 100x100 pixels; if added, the icon will be displayed overlaying to the block when it is used in a graphic language (LD or FBD)

The images must have a maximum weight of 30 KB per image. You can display imported images in the description windows by linking them with HTML syntax; check the *use HTML syntax* box and then use ``; to add the image to the description.

#### **View POU properties:**

Select a POU from the project tree then choose the appropriate voice of the menu `Project > View PLC object properties`.

#### **Exclude from build:**

Select a POU and right-click on it to open its context menu, from there select `Exclude from build`. Doing so the POU will be ignored when compiling the project, even if the POU is used in the project.

For example: if a program as been already assigned to a task, you can avoid to compile it by selecting "exclude from build" instead of removing the program from the task; this will allow you to change the project behaviour without changing the project tree.

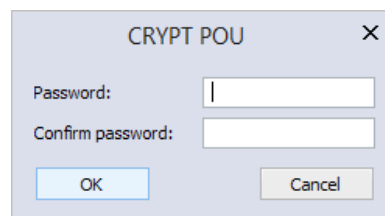
Be careful, excluding from build a POU that is required by another element will generate an error message. For example: if you exclude from build a function which is called by a program, when compiling the program you'll obtain an error message.

### **5.1.3 SOURCE CODE ENCRYPTION/DECRYPTION**

LogicLab can encrypt POU's and protect them with a password, hiding the source code of the POU.

#### **Encrypting a POU:**

Select a POU from the project tree then choose the `[Crypt]` voice of the context menu  
Double enter the password and confirm the operation.



LogicLab shows in the project tree a special marker icon that overlays the standard POU icon in order to inform the user that the POU is encrypted.

#### **Decrypting a POU:**

Select a POU from the project tree then choose the `[Decrypt]` voice of the context menu

#### **Encrypting all POU's:**

Select the root element from the project tree then choose the `[Crypt all objects]` voice of the context menu.

All POU's will be encrypted with the same password.

#### **Decrypt all POU's:**

Select the root element from the project tree then choose the `[Decrypt all objects]` voice of the context menu.

## 5.2 VARIABLES

There are two classes of variables in LogicLab: global variables and local variables.

This paragraph shows you how to add to the project, edit, and eventually remove both global and local variables.

### 5.2.1 GLOBAL VARIABLES

Global variables can be seen and referenced by any module of the project.

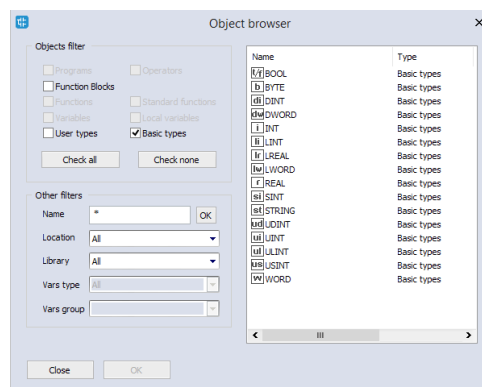
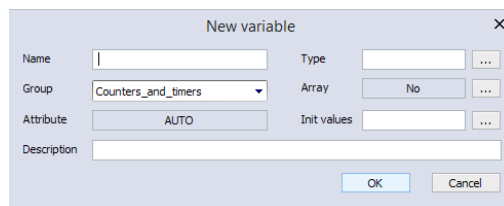
#### 5.2.1.1 CLASSES OF GLOBAL VARIABLES

Global variables are organized in special folders of the project tree called *Global\_vars*. Those variables are classified according to their properties as:

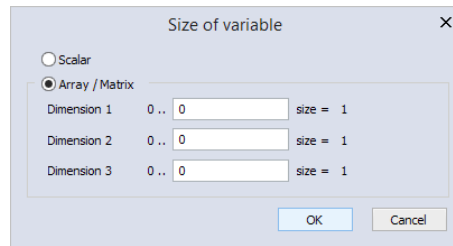
- Automatics: the compiler automatically allocates them to an appropriate location in the target device memory.
- Mapped: they have an assigned address in the target device logical addressing system, which shall be specified by the developer.
- Constants: are declared having the CONSTANT attribute; They cannot be written.
- Retains: they are declared having the RETAIN attribute; Their values are stored in a persistent memory area of the target device.

#### 5.2.1.2 CREATING A NEW GLOBAL VARIABLE

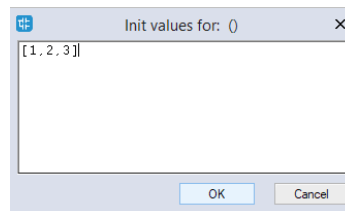
- 1) In order to create a new global variable select a *Global variables group* from the project tree then choose the appropriate voice from the menu **Project>New Object>New variable** (see Paragraph 5.7.4).
- 2) LogicLab will show you a dialog box where you must enter the name of the variable (remember that some characters, such as '?', ':', '\', and so on, cannot be used: the variable name must be a valid IEC 61131-3 identifier).
- 3) Specify the type of the variable either by typing it or by selecting it from the list that LogicLab displays when you click on the *Browse* button.



If you want to declare an array, you must specify its size.

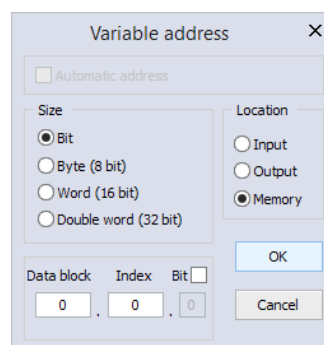
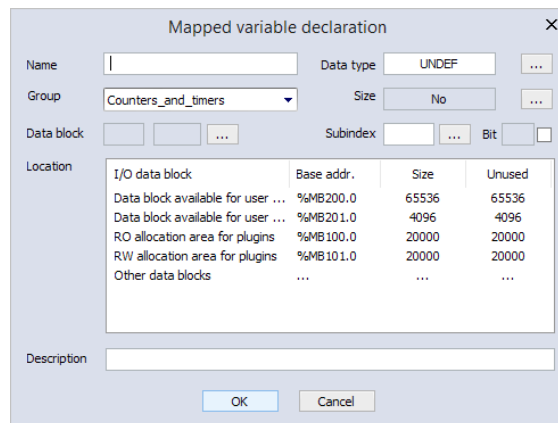


- 4) You may optionally assign the initial value to the variable or to the single elements of the array.



If you create a new mapped variable, you are required to specify the address of the variable during its definition. In order to do so, you may do one of the following actions:

- Click on the button to open the editor of the address, then enter the desired value.



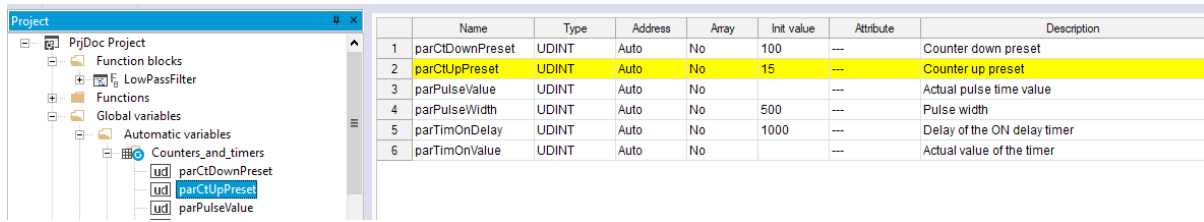
- Select from the list that LogicLab shows you the memory area you want to use: the tool automatically calculates the address of the first free memory location of that area.

### 5.2.1.3 EDITING A GLOBAL VARIABLE

To edit the definition of an existing global variable, open it by double-clicking it, or the folder that it belongs to, from the project tree. The global variables editor opens and lets you modify its definition.







**Changing the name of the variable:**

Select the variable you want to rename from the project tree then right-click on it and select **Rename Variable** from the context menu.

**Duplicating a variable:**

Select the variable you want to duplicate from the project tree then right-click on it and choose **Duplicate Variable** from the context menu.

Enter the name of the new duplicated variable and confirm.

**Deleting a variable:**

Select the variable you want to delete from the project tree then right-click on it and choose **Delete Variable** from the context menu.

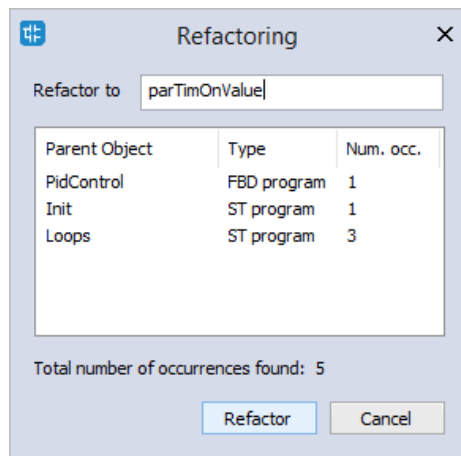
Confirm the operation to delete the variable.

**Refactoring a variable:**

Refactoring will allow you to change the name of a global variable and consequently adjust all of its occurrences in the project.

Select the desired variable from the project tree, then right-click on it and choose **Refactoring** from the context menu.

A window will appear allowing you to insert a new variable name and showing you all the occurrences found for this symbol; if you change the name and press OK, all the shown occurrences will be modified.

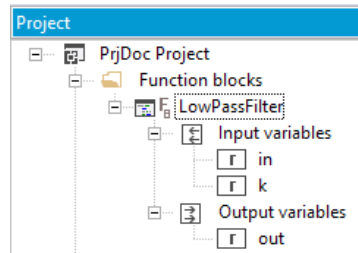


**5.2.2 LOCAL VARIABLES**

Local variables are declared within a POU (either program, or function, or function block), the module itself is the only project element that can refer to and access them.

Local variables are listed in the project tree under the POU which declares them (only when that POU is open for editing), where they are further classified according to their class (e.g., as input or inout variables).





In order to create, edit, and delete local variables, you have to open the POU for editing and use the local variables editor. The project needs to be saved in order to update the POU branch structure of the project tree, including the changes applied to the local variables.

	Class	Pin	Name	Type	Array	Init value	Attribute	Description
1	VAR_INPUT	0	in	REAL	No			
2	VAR_INPUT	1	k	REAL	No			
3	VAR_OUTPUT	0	out	REAL	No			

```

0001
0002
0003   out := k * ( in - out ) + out;
0004
0005
0006
    
```

Refer to the corresponding section in this manual for details (see Paragraph 6.6.1.2).

### 5.2.3 CREATING MULTIPLE VARIABLES

LogicLab allows you to create multiple variables in one shot.

Open the POU for editing then choose the appropriate voice of the menu **Variables > Create multiple**.

LogicLab will show you a dialog box where you must specify the prefix and the suffix to name of the new variables.

- 1) Select the type of the variables.
- 2) Choose type and attribute for the variables.
- 3) Insert the number of the variables you want to create specifying the start index, the end index and the step value. You can see an example of the generated variable names at the bottom of the dialog.



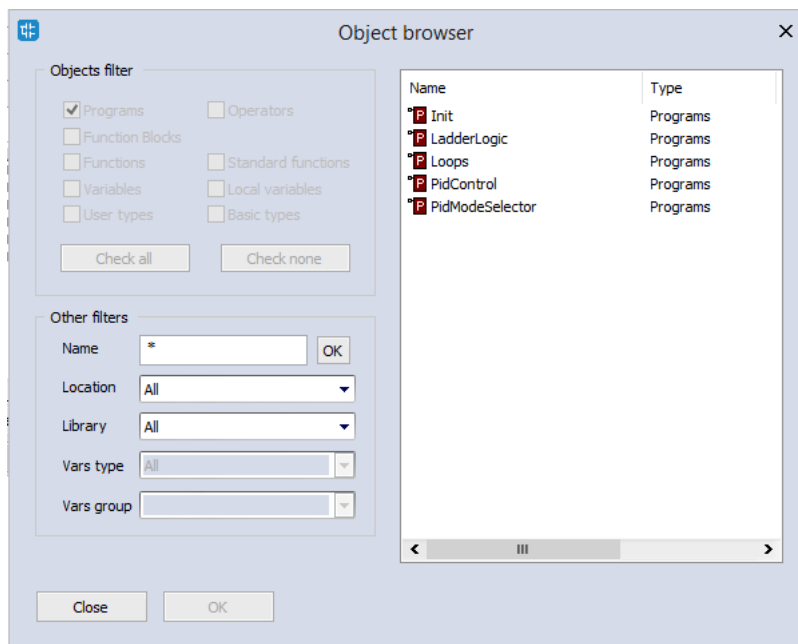
### 5.2.4 TEXTUAL EDITOR FOR VARIABLES

Both global and local variables can be created and edited also from a textual editor; refer to the corresponding section in this manual for details (see Paragraph 6.6)

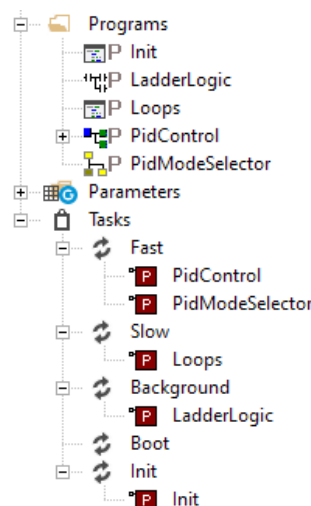
## 5.3 TASKS

### 5.3.1 ASSIGNING A PROGRAM TO A TASK

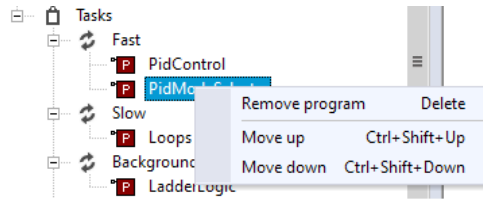
- 1) Select the task where you want to add the program from the project tree then choose the **[Add program]** voice of the context menu.
- 2) Select the program you want to be executed by the task from the list which shows up and confirm your choice.



- 3) The program has been assigned to the task, as you can see in the project tree.



Note that you can assign more than a program to a task. From the contextual menu you can sort and, eventually, remove program assignments to tasks.



### 5.3.2 TASK CONFIGURATION

Depending on the target device you are interfacing with, you may have the chance to configure some of the PLC tasks' settings.

Select the tasks element from the project tree then choose the **[Task configuration]** voice of the context menu.

In the *Task configuration* window you can edit the task execution period.

ID	Name	Type	Set period	Period (ms)	Description
0	Fast	Cyclic	No	10	Fast task
1	Slow	Cyclic	No	20	Slow task
2	Background	Cyclic	No	100	Background task
3	Boot	Single	No	0	Boot task
4	Init	Single	No	0	Init task

## 5.4 DERIVED DATA TYPES

The derived data type is a complex classification that identifies one or various data types and is made up of primitive data types.

User has the flexibility to create those own types that have advanced properties and uses far beyond those of the basic primitive data types.

In order to create a new data type you have to select the root element of the project tree, then you can either open the context menu and choose the appropriate voice from **Add > New definition**, or you can choose the appropriate voice from **Project > New Object > New definition**.

In both cases you will have to choose from six different derived data types, which will be described in the following paragraphs

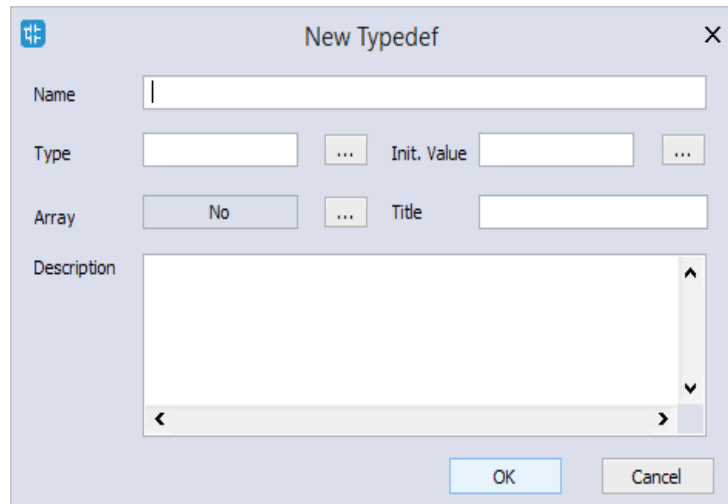
### 5.4.1 TYPEDEFS

#### 5.4.1.1 CREATING A NEW TYPEDEF

In order to create a Typedef select **Typedef** voice when creating a new definition.

LogicLab will show you a dialog box where you must specify the name of the new typedef and select the type you are defining an alias for:

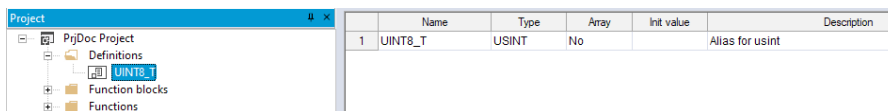




(if you want to define an alias for an array type, you shall choose the array size). Enter a meaningful description (optional) and confirm the operation.

**5.4.1.2 EDITING A TYPEDEF**

In order to edit an existing typedef you have to double-click it from the Project tree. The associated editor opens and lets you modify its definition.



**5.4.1.3 OTHER ACTIONS FOR TYPEDEFS**

If you open the context menu of a Typedef, by selecting it from the project tree and right-clicking it, you will see different actions that can be done with Typedefs:

**Deleting a Typedef:**

In order to delete a Typedef, select it from the Project tree then choose the **Delete** voice of the context menu.

**Duplicating a Typedef:**

In order to duplicate a Typedef, select it from the Project tree then choose the **Duplicate** voice of the context menu.

A window will appear asking you a new name for the duplicated Typedef.

**Refactoring a Typedef:**

Refactoring will allow you to change the name of a Typedef and consequently adjust all of its occurrences in the project.

Select the desired typedef from the project tree, then right-click on it and choose the **Refactoring** voice from the context menu.

A window will appear allowing you to insert a new name and showing you all the occurrences found for this symbol; if you change the name and press **OK**, all the shown occurrences will be modified.

**Typedef properties:**

In order to edit the properties of a Typedef, select it from the Project tree then choose **Edit properties**. A window will open allowing you to insert information such as version number, description, icon and images. Once imported, images can be used in the description by checking the *Use HTML syntax* checkbox and linking the imported image with HTML syntax, like ``.

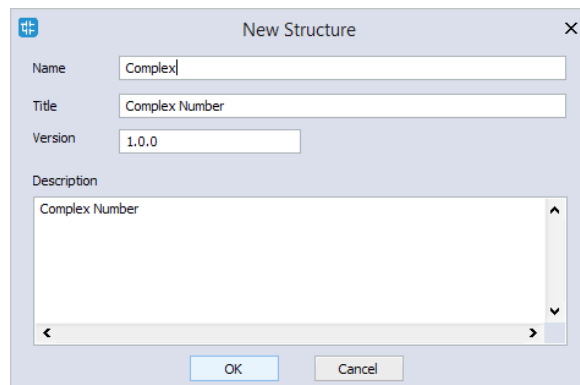


To view the properties of a Typedef, select it from the project tree then choose [View properties](#).

## 5.4.2 STRUCTURES

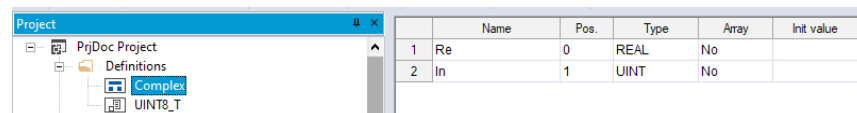
### 5.4.2.1 CREATING A NEW STRUCTURE

In order to create a structure select [Structure](#) voice when creating a new definition. LogicLab will show you a dialog box where you must specify the name of the new structure and a meaningful description, then confirm the operation.



### 5.4.2.2 EDITING A STRUCTURE

In order to edit an existing structure, open it by double-clicking it from the Project tree. The associated editor opens and lets you modify its definition and fields.



### 5.4.2.3 OTHER ACTIONS FOR STRUCTURES

If you open the context menu of a Structure, by selecting it from the project tree and right-clicking it, you will see different actions that can be done with Structures:

#### **Deleting a Structure:**

In order to delete a Structure, select it from the Project tree then choose the [Delete](#) voice of the context menu.

#### **Duplicating a Structure:**

In order to duplicate a Structure, select it from the Project tree then choose the [Duplicate](#) voice of the context menu.

A window will appear asking you a new name for the duplicated Structure.

#### **Refactoring a Structure:**

Refactoring will allow you to change the name of a Structure and consequently adjust all of its occurrences in the project.

Select the desired Structure from the project tree, then right-click on it and choose the [Refactoring](#) voice from the context menu.

A window will appear allowing you to insert a new name and showing you all the occurrences found for this symbol; if you change the name and press *OK*, all the shown occurrences will be modified.



### Structure properties:

In order to edit the properties of a Structure, select it from the Project tree then choose **Edit properties**. A window will open allowing you to insert information such as version number, description, icon and images. Once imported, images can be used in the description by checking the *Use HTML syntax* checkbox and linking the imported image with HTML syntax, like ``.

To view the properties of a Structure, select it from the project tree then choose **View properties**.

## 5.4.3 ENUMERATIONS

### 5.4.3.1 CREATING A NEW ENUMERATION

In order to create a new Enumerations select **Enumeration** voice when creating a new definition.

LogicLab will show you a dialog box where you must specify the name of the new enumeration and a meaningful description, then confirm the operation.

The dialog box titled "New Enumeration" contains three input fields: "Name", "Title", and "Description". The "Description" field is a large text area with scrollbars. At the bottom right, there are "OK" and "Cancel" buttons.

### 5.4.3.2 EDITING AN ENUMERATION

In order to edit an existing structure, open it by double-clicking it from the Project tree. The associated editor opens and lets you modify its definition and the initialization values of its elements.

The image shows a project tree on the left with "HYDROCARBON" selected. To the right is a table representing the enumeration data.

	Name	Init value	Description
1	Methane	1	
2	Butane	4	
3	Octane	8	

### 5.4.3.3 OTHER ACTIONS FOR ENUMERATIONS

If you open the context menu of an enumeration, by selecting it from the project tree and right-clicking it, you will see different actions that can be done with enumerations:

#### Deleting an Enumeration:

In order to delete an enumeration, select it from the Project tree then choose the **Delete** voice of the context menu.

#### Duplicating an Enumeration:

In order to duplicate an enumeration, select it from the Project tree then choose the **Duplicate** voice of the context menu.

A window will appear asking you a new name for the duplicated enumeration.



### Refactoring an Enumeration:

Refactoring will allow you to change the name of an enumeration and consequently adjust all of its occurrences in the project.

Select the desired enumeration from the project tree, then right-click on it and choose the **Refactoring** voice from the context menu.

A window will appear allowing you to insert a new name and showing you all the occurrences found for this symbol; if you change the name and press **OK**, all the shown occurrences will be modified.

### Enumeration properties:

In order to edit the properties of an enumeration, select it from the Project tree then choose **Edit properties**. A window will open allowing you to insert information such as version number, description, icon and images. Once imported, images can be used in the description by checking the *Use HTML syntax* checkbox and linking the imported image with HTML syntax, like ``.

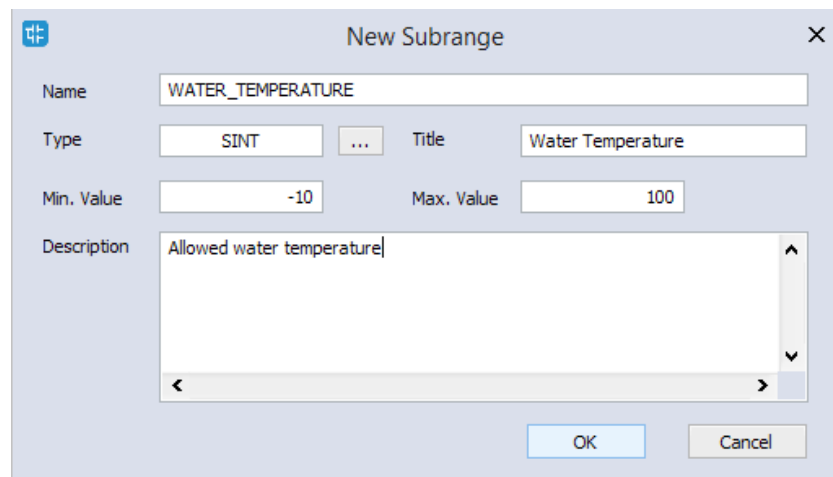
To view the properties of an enumeration, select it from the project tree then choose **View properties**.

## 5.4.4 SUBRANGES

### 5.4.4.1 CREATING A NEW SUBRANGE

In order to create a new subrange select **Subrange** voice when creating a new definition.

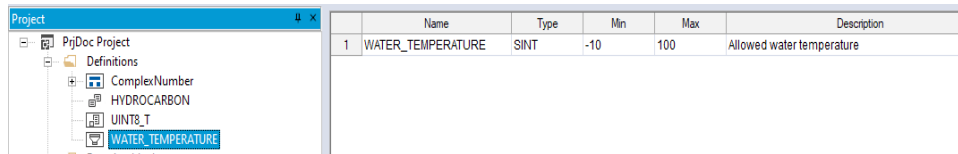
LogicLab will show you a dialog box where you must specify the name of the new subrange, select its basic type and enter the minimum and the maximum values for the subrange; optionally you can enter a meaningful description.



### 5.4.4.2 EDITING A SUBRANGE

In order to edit an existing subrange, open it by double-clicking it from the Project tree. The associated editor opens and lets you modify its definition.





### 5.4.4.3 OTHER ACTIONS FOR SUBRANGE

If you open the context menu of a subrange, by selecting it from the project tree and right-clicking it, you will see different actions that can be done with subranges:

#### Deleting a Subrange:

In order to delete a subrange, select it from the Project tree then choose the **Delete** voice of the context menu.

#### Duplicating a Subrange:

In order to duplicate a subrange, select it from the Project tree then choose the **Duplicate** voice of the context menu.

A window will appear asking you a new name for the duplicated subrange.

#### Refactoring a Subrange:

Refactoring will allow you to change the name of a subrange and consequently adjust all of its occurrences in the project.

Select the desired subrange from the project tree, then right-click on it and choose the **Refactoring** voice from the context menu.

A window will appear allowing you to insert a new name and showing you all the occurrences found for this symbol; if you change the name and press **OK**, all the shown occurrences will be modified.

#### Subrange properties:

In order to edit the properties of a subrange, select it from the Project tree then choose **Edit properties**. A window will open allowing you to insert information such as version number, description, icon and images. Once imported, images can be used in the description by checking the *Use HTML syntax* checkbox and linking the imported image with HTML syntax, like ``.

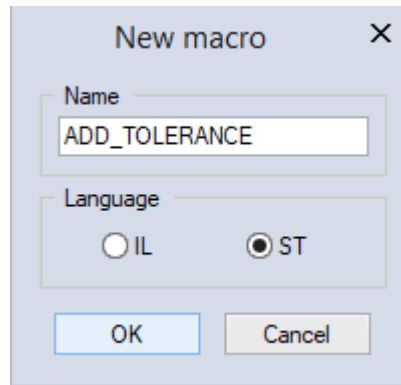
To view the properties of a subrange, select it from the project tree then choose **View properties**.

## 5.4.5 MACROS

### 5.4.5.1 CREATING A NEW MACRO

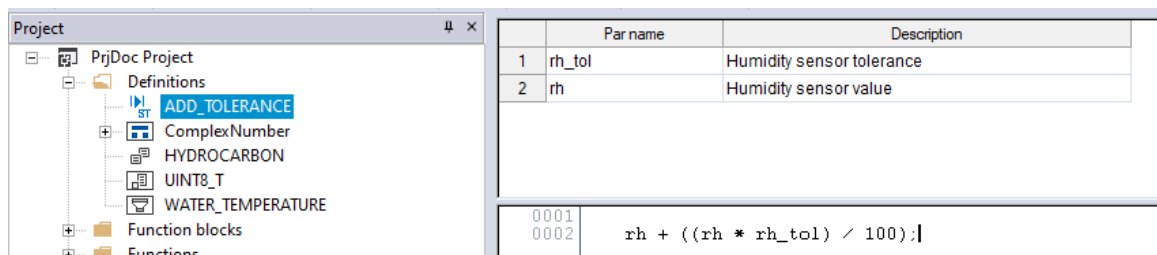
In order to create a new macro select **Macro** voice when creating a new definition.

LogicLab will show you a dialog box where you must specify the name of the new macro and its language, only IL and ST are allowed (see chapter 11.7.1 for further details).



### 5.4.5.2 EDITING A MACRO

In order to edit an existing macro, open it by double-clicking it from the Project tree. The associated editor opens and lets you modify its definition, the parameters and its code.



### 5.4.5.3 OTHER ACTIONS FOR MACROS

If you open the context menu of a macro, by selecting it from the project tree and right-clicking it, you will see different actions that can be done with macros:

#### Deleting a macro:

In order to delete a macro, select it from the Project tree then choose the **Delete** voice of the context menu.

#### Duplicating a macro:

In order to duplicate a macro, select it from the Project tree then choose the **Duplicate** voice of the context menu.

A window will appear asking you a new name for the duplicated macro.

#### Refactoring a macro:

Refactoring will allow you to change the name of a macro and consequently adjust all of its occurrences in the project.

Select the desired macro from the project tree, then right-click on it and choose the **Refactoring** voice from the context menu.

A window will appear allowing you to insert a new name and showing you all the occurrences found for this symbol; if you change the name and press **OK**, all the shown occurrences will be modified.

#### Macro properties:

In order to edit the properties of a macro, select it from the Project tree then choose **Edit properties**. A window will open allowing you to insert information such as version number, description, icon and images. Once imported, images can be used in the description by checking the *Use HTML syntax* checkbox and linking the imported image with HTML syntax, like ``.

To view the properties of a macro, select it from the project tree then choose **View properties**.

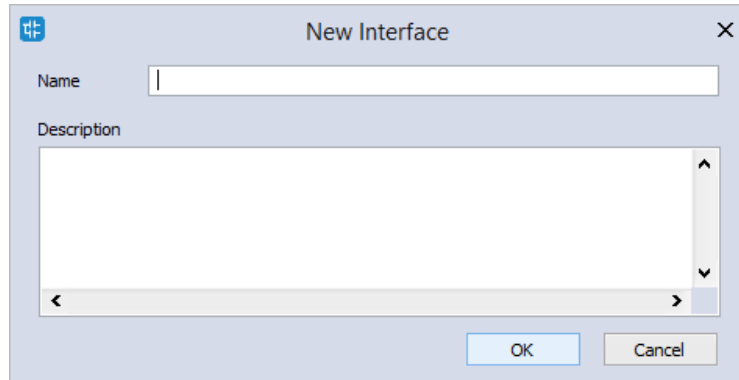


## 5.4.6 INTERFACES

### 5.4.6.1 CREATING A NEW INTERFACE

In order to create a new interface select **Interface** voice when creating a new definition; note that interfaces can be created only if Object Oriented Programming is enabled (see *paragraph 4.6.1*), and that depends on target implementation.

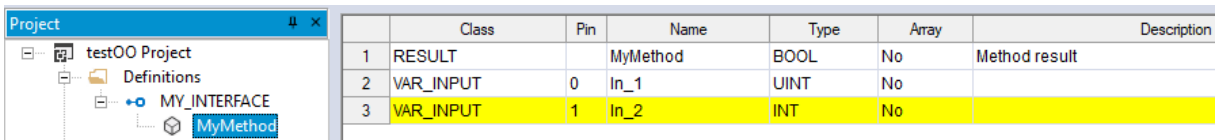
LogicLab will show you a dialog box where you must specify the name of the new macro and its language, only IL and ST are allowed (see *chapter 11.7.1* for further details).



### 5.4.6.2 EDITING AN INTRFACE

Interfaces cannot be edited, but you can add method to an interface and edit the method. To add a new method, open the context menu of the interface, by selecting it in the Project tree and right-clicking on it, then choose **Add method prototype**; a window will appear asking you to insert name and description of the method prototype.

The method prototype will be added to the Project tree; now double-clicking it will open its editor, allowing you to modify the method prototype.



### 5.4.6.3 OTHER ACTIONS FOR INTERFACES

If you open the context menu of an interface, by selecting it from the project tree and right-clicking it, you will see different actions that can be done with interfaces; same things can be done with the method prototypes in the same way

#### **Deleting an interface or a method prototype:**

In order to delete the element, select it from the Project tree then choose the **Delete** voice of the context menu.

#### **Duplicating an interface or a method prototype:**

In order to duplicate the element, select it from the Project tree then choose the **Duplicate** voice of the context menu.

A window will appear asking you a new name for the duplicated element.

#### **Refactoring an interface or a method prototype:**

Refactoring will allow you to change the name of the element and consequently adjust



all of its occurrences in the project.

Select the desired element from the project tree, then right-click on it and choose the **Refactoring** voice from the context menu.

A window will appear allowing you to insert a new name and showing you all the occurrences found for this symbol; if you change the name and press **OK**, all the shown occurrences will be modified.

### Interface and method prototype properties:

In order to edit the properties of an element, select it from the Project tree then choose **Edit properties**. A window will open allowing you to insert information such as version number, description, icon and images. Once imported, images can be used in the description by checking the *Use HTML syntax* checkbox and linking the imported image with HTML syntax, like ``.

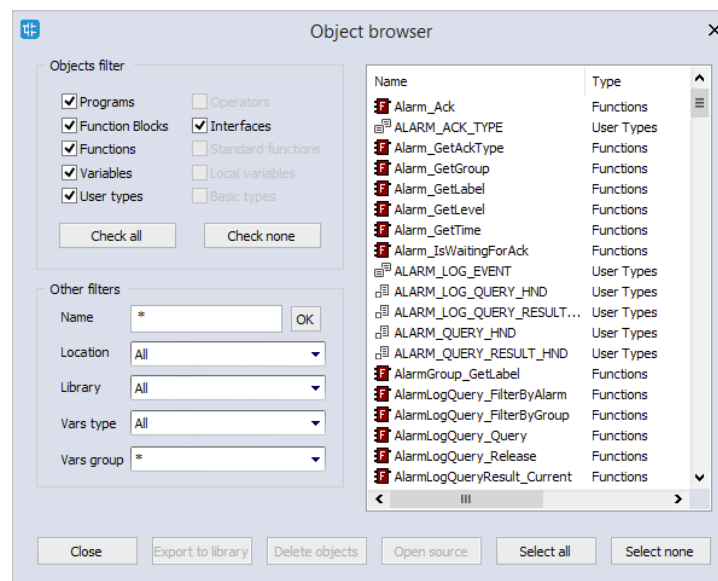
To view the properties of an element, select it from the project tree then choose **View properties**.

## 5.5 BROWSE THE PROJECT

Projects may grow huge, hence LogicLab provides two tools to search for an object within a project: the *Object browser* and the *Find in project* feature.

### 5.5.1 OBJECT BROWSER

LogicLab provides a useful tool for browsing the objects of your project: the *Object Browser*.



This tool is context dependent, this implies that the kind of objects that can be selected and that the available operations on the objects in the different contexts are not the same.

*Object browser* can be opened in these three main ways:

- *Browser mode*.
- *Import object mode*.
- *Select object mode*.

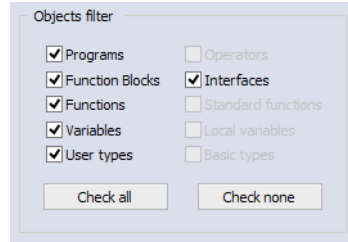
User interaction with *Object browser* is mainly the same for all the three modes and is described in the next paragraph.



### 5.5.1.1 COMMON FEATURES AND USAGE OF OBJECT BROWSER

This section describes the features and the usage of the *Object browser* that are common to every mode in which *Object browser* can be used.

#### Objects filter

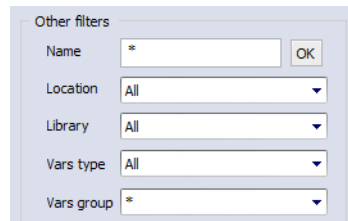


This is the main filter of the *Object browser*. User can check one of the available (enabled) object items.

In this example, *Programs*, *Function Blocks*, *Functions* are selected, so objects of this type are shown in the object list. *Variables* and *User types* objects can be selected by user but objects of that type are not currently shown in the object list. *Operators*, *Standard functions*, *Local variables*, and *Basic types* cannot be checked by user (because of the context) so cannot be browsed.

User can also click *Check all* button to select all available objects at one time or can click *Check none* button to deselect all objects at one time.

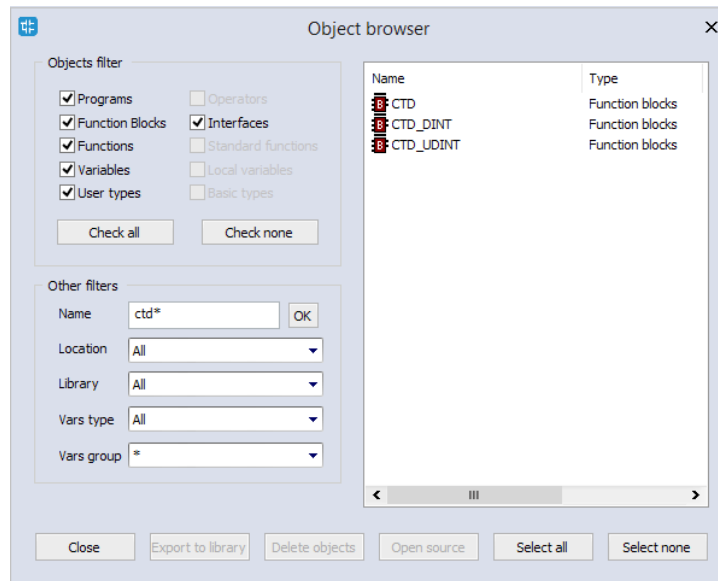
#### Other filters



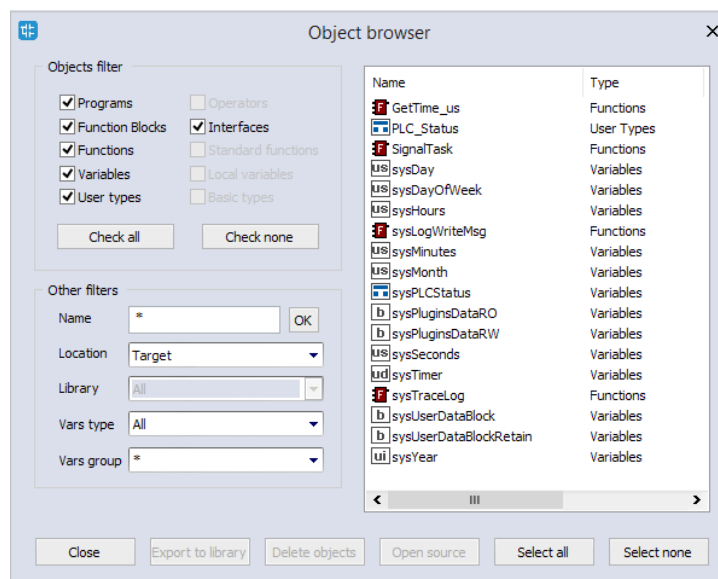
Selected objects can be also filtered by name, symbol location, specific library and var type.

Filters are all additive and are immediately applied after setting.

Name	
<b>Function</b>	Filters objects on the base of their name.
<b>Set of legal values</b>	All the strings of characters.
<b>Use</b>	Type a string to display the specific object whose name matches the string. Use the * wildcard if you want to display all the objects whose name contains the string in the <i>Name</i> text box. Type * if you want to disable this filter. Press <i>Enter</i> when edit box is focused or click on the <i>OK</i> button near the edit box to apply the filter.
<b>Applies to</b>	All object types.



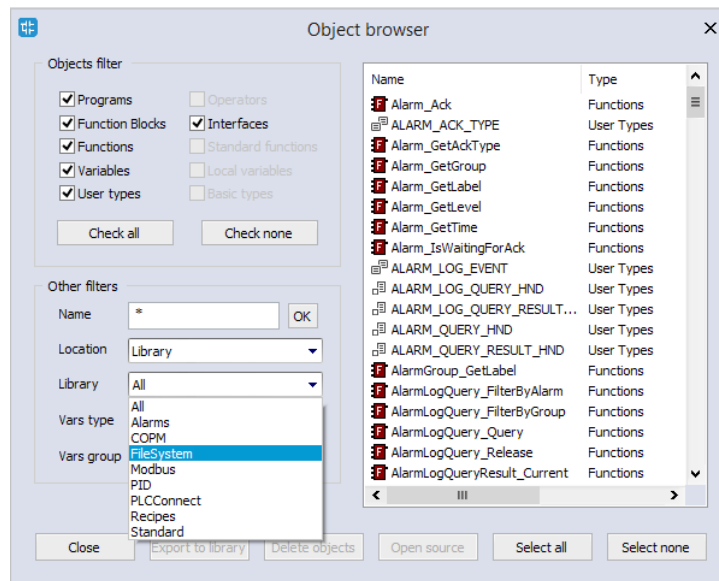
Symbol location	
<b>Function</b>	Filters objects on the base of their location.
<b>Set of legal values</b>	All, Project, Target, Library, Aux. Sources.
<b>Use</b>	All= Disables this filter. Project= Objects declared in the LogicLab project. Target= Firmware objects. Library= Objects contained in a library. In this case, use simultaneously also the <i>Library</i> filter, described below. Aux sources= Shows aux sources only.
<b>Applies to</b>	All objects types.



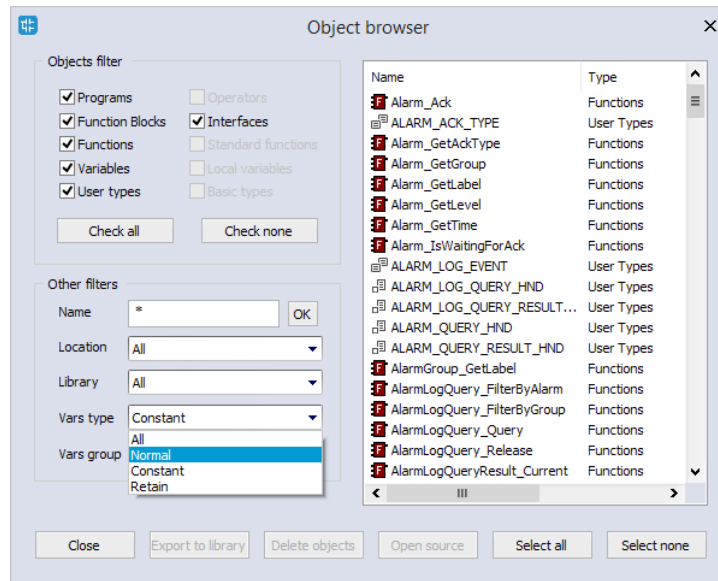
**Library**



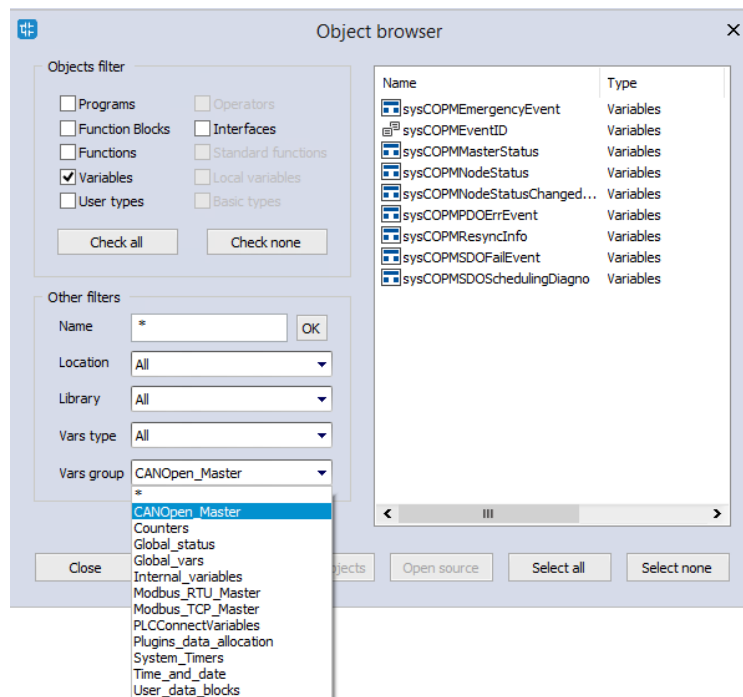
<b>Function</b>	Completes the specification of a query on objects contained in libraries. The value of this control is relevant only if the <i>Symbol location</i> filter is set to <i>Library</i> .
<b>Set of legal values</b>	All, libraryname1, libraryname2, ...
<b>Use</b>	All= Shows objects contained in whatever library. LibrarynameN= Shows only the objects contained in the library named librarynameN.
<b>Applies to</b>	All objects types.



<b>Vars Type</b>	
<b>Function</b>	Filters global variables and system variables (also known as firmware variables) according to their type.
<b>Set of legal values</b>	All, Normal, Constant, Retain
<b>Use</b>	All= Shows all the global and system variables. Normal= Shows normal global variables only. Constant= Shows constants only. Retain= Shows retain variables only.
<b>Applies to</b>	Variables.



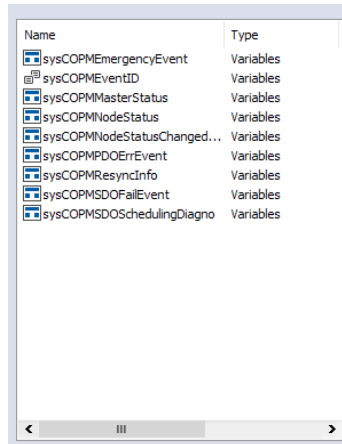
<b>Vars Group</b>	
<b>Function</b>	Filters objects on the base of their group.
<b>Set of legal values</b>	Every group known to the project
<b>Use</b>	Shows only variables belonging to the selected group.
<b>Applies to</b>	Variables.



**Object list**







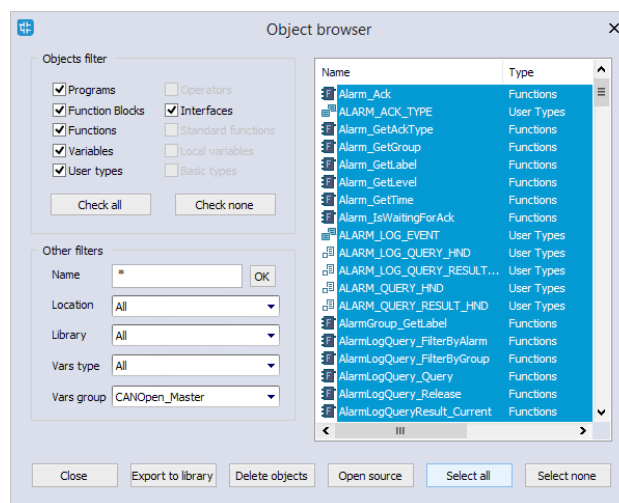
*Object list* shows all the filtered objects. List can be ordered in ascending or descending way by clicking on the header of the column. So it is possible to order items by *Name*, *Type*, or *Description*.

Double-clicking on an item allows the user to perform the default associated operation (the action is the same of the *OK*, *Import object*, or *Open source* button actions).

When item multiselection is allowed, *Select all* and *Select none* buttons are visible.

It is possible to select all objects by clicking on *Select all* button. *Select none* deselects all objects.

If at least one item is selected on the list operation, buttons are enabled.



**Resize**

Window can be resized, the cursor changes along the border of the dialog and allows the user to resize window. When reopened, *Object browser* dialog takes the same size and position of the previous usage.

**5.5.1.2 USING OBJECT BROWSER AS A BROWSER**

In order to use the object browser to simply look over through the element of the project choose the appropriate voice of the menu [Project > Object Browser](#).

**Available objects**

In this mode you can list objects of these types:

- Programs.
- Function Blocks.



- Functions.
- Variables.
- User types.

These items can be checked or unchecked in *Objects filter* section to show or to hide the objects of the chosen type in the list.

Other types of objects (Operators, Standard functions, Local variables, Basic types) cannot be browsed in this context so they are unchecked and disabled).

### Available operations

Allowed operations in this mode are:

Open source, default operation for double-click on an item	
<b>Function</b>	Opens the editor by which the selected object was created and displays the relevant source code.
<b>Use</b>	If the object is a program, or a function, or a function block, this button opens the relevant source code editor. If the object is a variable, then this button opens the variable editor. Select the object whose editor you want to open, then click on the <i>Open source</i> button.

Export to library	
<b>Function</b>	Exports an object to a library.
<b>Use</b>	Select the objects you want to export, then press the <i>Export to library</i> button.

Delete objects	
<b>Function</b>	Allows you to delete an object.
<b>Use</b>	Select the object you want to delete, then press the <i>Delete object</i> button.

### Multi selection

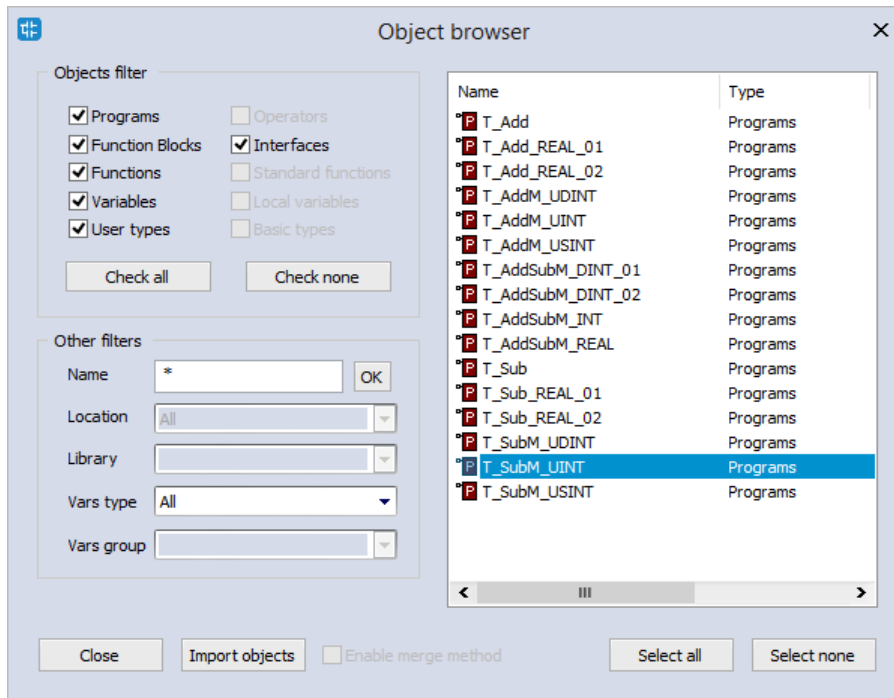
Multi selection is allowed for this mode, *Select all* and *Select none* buttons are visible.

#### 5.5.1.3 USING OBJECT BROWSER FOR IMPORT

Object browser is also used to support objects importation in the project from a desired external library.

In order to use the object browser to import external library to the project, select *Project>Import object*, then an explorer dialog will be prompted to allow you to choose the source library; once selected a library, an *Object Browser Window* will appear, allowing you to select the objects you wish to import.





### Available objects

In this mode you can list objects of these types:

- Programs.
- Function blocks.
- Functions.
- Variables.
- User types.

These items can be checked or unchecked in *Objects filter* section to show or to hide the objects of the chosen type in the list.

Other types of objects (Operators, Standard functions, Local variables, Basic types) cannot be imported so they are unchecked and disabled.

### Available operations

*Import objects* is the only operation supported in this mode. It is possible to import selected objects by clicking on *Import objects* button or by double-clicking on one of the objects in the list.

### Multi selection

Multi selection is allowed for this mode, *Select all* and *Select none* buttons are visible.

## 5.5.1.4 USING OBJECT BROWSER FOR OBJECT SELECTION

Object browser dialog is useful for many operations that requires the selection of a single PLC object. So Object browser can be used to select the program to add to a task, to select the type of a variable, to select an item to find in the project, etc..

### Available objects

Available objects are strictly dependent on the context, for example in the program assignment to a task operation the only available objects are programs objects.

It is possible that not all available objects are selected by default.

### Available operations



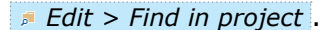
In this mode it is possible to select a single object by double-clicking on the list or by clicking on the *OK* button, then the dialog is automatically closed.

### Multi selection

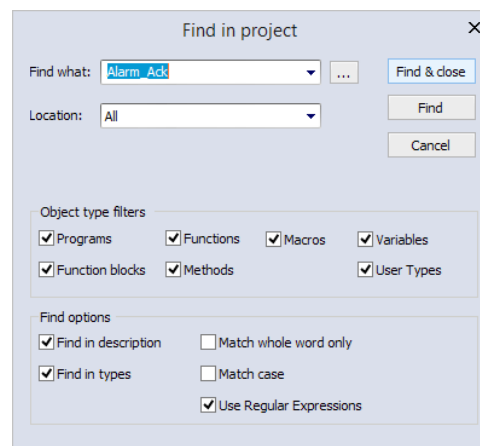
Multi selection is not allowed for this mode, *Select all* and *Select none* buttons are not visible.

## 5.5.2 SEARCH WITH THE FIND IN PROJECT COMMAND

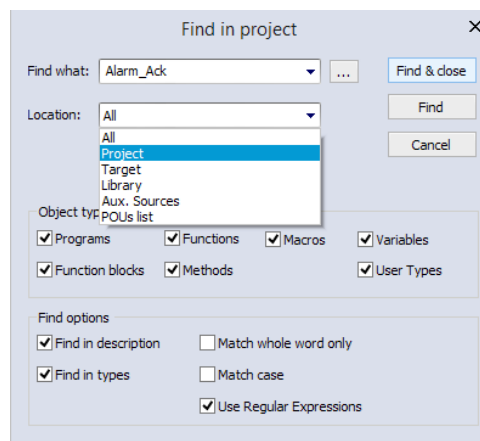
The *Find in project* command retrieves all the instances of a specified character string in the project.

In order to use this functionality choose the appropriate voice of the menu .

LogicLab will show you the following dialog box:

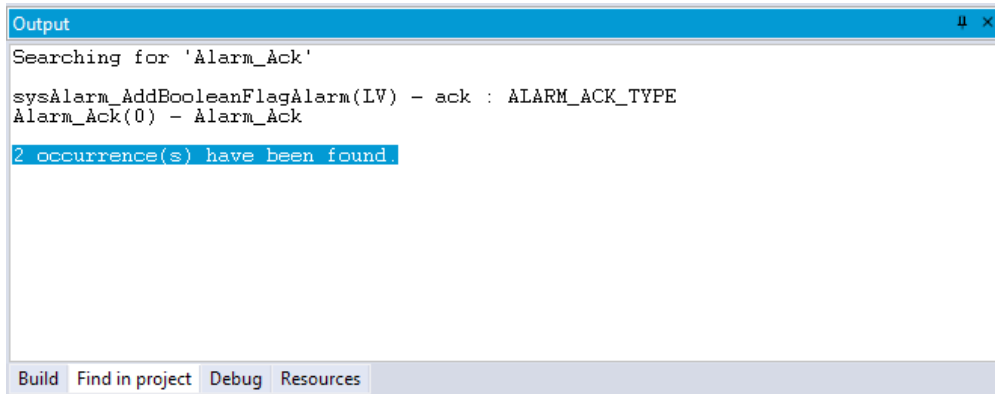


- 1) In the *Find what* text box, type the name of the object you want to search. Otherwise, click the *Browse* button to the right of the text box, and select the name of the object from the list of all the existing items.
- 2) Select one of the values listed in the *Location* combo box, so as to specify a constraint on the location of the objects to be inspected.



- 3) The frame named *Object type* contains a set of checkboxes, each of which, if ticked, enables research of the string among the object it refers to.
- 4) The frame named *Find options* contains a set of checkboxes, each of which, if ticked, modify the way the research is done. For example, check *Match case* if you want your research to be case-sensitive.
- 5) Press *Find* (or *Find & close*) to start the search, otherwise click *Cancel* to abandon.

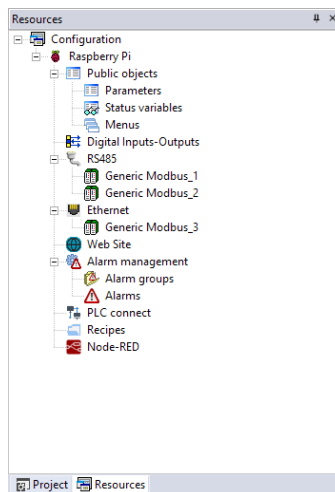
The results will be printed in the *Find in project* tab of the *Output* window.



## 5.6 WORKING WITH LOGICLAB EXTENSIONS

LogicLab's *Workspace* window may include a section whose contents completely depend on the target device the IDE is interfacing with: the *Resources* panel.

If the *Resources* panel is visible, you can access some additional features related to the target device (configuration elements, schemes, wizards, and so on).

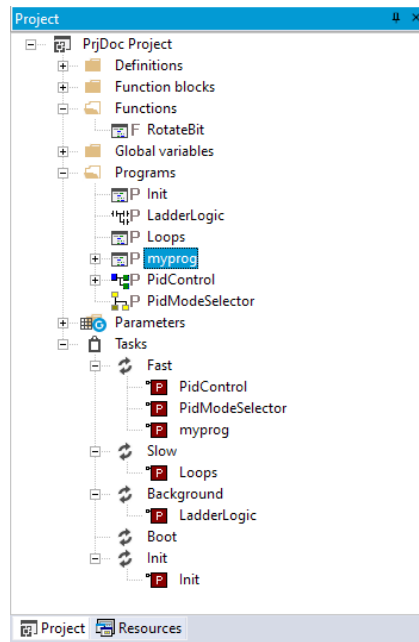


Information about these features may be found in a separate document: refer to your hardware supplier for details.

## 5.7 PROJECT CUSTOM WORKSPACE

The custom workspace functionalities allow you to organize your project tree according to your needs, in order to obtain more efficiency in the management of the project.

All organizations units of the custom workspace are logical: creating and editing those units will no triggers any effects on the PLC code.



### 5.7.1 ENABLE THE CUSTOM WORKSPACE INTO AN EXISTING PROJECT

To enable this feature see the [Project>Options...](#) (see Paragraph 4.6.1), once enabled the project needs to be reloaded.

By default this features is enabled depending on targets.

### 5.7.2 WORKSPACES MIGRATION

Whenever this feature is switched, LogicLab tries to reorder the workspace maintaining the user customization by this logic:

#### Static (old) workspace to custom (new)

Fixed logic units (Ex. Function blocks folder) are converted into new dynamic folders with the same names. Fixed global group units (Ex. Mapped variables) are converted into new global dynamic groups with the same names. All global variables that do not belong to any group will be grouped into a new group called *Ungrouped global vars*.

#### Custom (new) workspace to static (old)

All custom units will be destroyed and all POUs and global variables will be grouped into the default fixed units (Ex. Function blocks folder and Mapped Variables).

### 5.7.3 CUSTOM WORKSPACE BASIC UNITS

In the new custom workspace you can work using two different main logic units:

- *Folder*: this is an optional logical unit that can contain POUs, folders (you can nest folders into another one) and global variables group.



- *Global variables group*: this logical unit can only contain global variables. In order to create a global variable you need to have at least one global variables group defined into your custom workspace.





## 5.7.4 CUSTOM WORKSPACE OPERATIONS

Different useful operations can be performed in order to give a better organization of your project.

### Creating a folder

In order to create a folder select the root item of the project tree or, if you want to nest it, an existing folder then choose the `[Add>New folder]` voice of the context menu.

This operation adds a new customizable folder (by default named *New folder*) unit ready to be renamed.

### Editing folder properties - Set folder as read-only

You can edit the folder properties choosing `Edit folder properties` from its context menu. Once inside the folder properties window, you can choose to mark the folder as read only; doing so you'll be asked to set a password. The content of the folder is now not modifiable until someone remove the read only attribute by inserting the correct password.

### Editing folder properties - Set folder as a library folder

Still from the folder properties window, you can mark the folder as a library folder (select *Enable* from the `Library generation options` section). Doing so you'll be allowed to set other informations about the library folder (like the version number or a description).

If your folder is a library folder, it'll be represented with a different icon in the tree:



In the context menu of the folder, now, you'll find a new voice: `Generate library`; selecting it will create a new `.plcprj` library in the project folder (inside `AutoGeneratedLibraries` sub folder) with the name of the library folder and all of its content.

This is an easy and efficient way to create complex libraries.

### Creating a Global variables Group

In order to create a global variables group select the root item of the project tree or an existing folder then choose the `[Add>New global variables group]` voice of the context menu.

This operation adds a new customizable folder (by default named *New\_var\_group1*) unit ready to be renamed.

### Rename a unit (folder or Global variables group)

In order to rename a global variables group or a folder select it than choose `[Rename]` voice of the context menu.

This operations makes the name of the unit ready to be renamed.

### Deleting a unit (folder or Global variables group)

In order to delete a global variables group or a folder select it than choose `[Delete]` voice of the context menu.

If the units contains any child you will be prompted for three possibilities:

- 1) Delete all child elements too (this may impact the PLC).
- 2) Do not delete child elements, they will be moved upwards following the project tree.
- 3) Cancel the operations and do nothing.

### Export objects to library



In order to export all elements of a global variables group or a folder, select it than choose *[Export objects to library]* voice of the context menu.

This operation allows you to export recursively all child elements of the selected item into a library (see 4.8.2 for more information about new library).

### **Export objects as PLCopen**

LogicLab allows you to export all elements of a global variables group or a folder in PLCopen format. Like *Export to library* you can export the selected elements to an existing .xml file or to a new .xml file (in the latter case the file will be created), the PLCopen format will be maintained.

This operation allows you to export recursively all child elements of the selected item into a PLCopen file.

### **Moving Unit**

You can simply drag&drop units to a different location of the tree in order to organize your project workspace. All children are moved if the parent item is moved, following the original structure.

Moving variables is also possible both from project tree (single selection) and from the variable grid (single and multiple selections) (see Paragraph 6.6 for more information about variables editor).

## **5.7.5 WORKSPACE ELEMENTS WITH LIMITATIONS**

Some elements of the workspace are fixed and not customizable. They are automatically generated by LogicLab and no special custom operations are allowed on.

### **Root Project Element**

You can not move, rename or delete this element. It can contain customizable units as children.

### **POUs Children Elements**

These elements are generated following the structure of the POU they belong to. You can not move, rename or delete these elements directly from the tree. For more information about POUs (see Paragraph 5.1).

### **SFC Children Elements**

These elements follow the aforesaid rules but especially for the SFC children nodes the rename or delete operations are not allowed also on the POUs that belong to Actions or Transitions elements. For more information about SFC language (see Paragraph 6.5).

### **Aux Variables Element**

You can not move, rename or delete this element and his children. They are automatically generated by LogicLab.

### **Tasks Element**

You can not move, rename or delete these elements. They are automatically generated by LogicLab. For more information about SFC language (see Paragraph 5.3).







## 6. EDITING THE SOURCE CODE

### PLC editors

LogicLab includes five source code editors, which support the whole range of IEC 61131-3 programming languages: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD), Function Block Diagram (FBD), and Sequential Function Chart (SFC).

Moreover, LogicLab includes a grid-like editor to support the user in the definition of local variables.

All editors, both graphical and text one, support tooltips. By enabling them (see Paragraph 3.6.1.4), LogicLab will show some information about symbols on which the user move the mouse.

This chapter focuses on all these editors.




### 6.1 INSTRUCTION LIST (IL) EDITOR

0001	
0002	LD a
0003	ADD b
0004	ST a
0005	
0006	LD c
0007	ADD d
0008	ST c
0009	
0010	LD k
0011	ADD l
0012	ST k
0013	
0014	

The IL editor allows you to code and modify POU's using IL (Instruction List), one of the IEC-compliant languages.

#### 6.1.1 EDITING FUNCTIONS

The IL editor is endowed with functions common to most editors running on a Windows platform, namely:

- Text selection.
-  **Edit>Cut**.
-  **Edit>Copy**.
-  **Edit>Paste**.
- **Edit>Replace**.
- Drag-and-drop of selected text.

#### 6.1.2 REFERENCE TO PLC OBJECTS

If you need to add to your IL code a reference to an existing PLC object, you have two options:

- You can type directly the name of the PLC object.
- You can drag it to a suitable location. For example, global variables can be taken from the *Workspace* window, whereas standard operators and embedded functions can be dragged from the *Libraries* window, whereas local variables can be selected from the local variables editor.

### 6.1.3 AUTOMATIC ERROR LOCATION

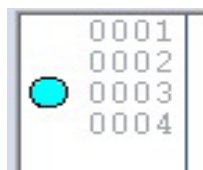
The IL editor also automatically displays the location of compiler errors. To know where a compiler error occurred, double-click the corresponding error line in the *Output* bar.

### 6.1.4 BOOKMARKS

You can set bookmarks to mark frequently accessed lines in your source file. Once a bookmark is set, you can use a keyboard command to move to it. You can remove a bookmark when you no longer need it.

#### 6.1.4.1 SETTING A BOOKMARK

Move the insertion point to the line where you want to set a bookmark, then press *Ctrl+F2*. The line is marked in the margin by a light-blue circle.



#### 6.1.4.2 JUMPING TO A BOOKMARK

Press *F2* repeatedly, until you reach the desired line

#### 6.1.4.3 REMOVING A BOOKMARK

Move the cursor to anywhere on the line containing the bookmark, then press *Ctrl+ F2*.

## 6.2 STRUCTURED TEXT (ST) EDITOR

```

0001
0002   y := SIN(x);
0003   x := x + incr;|
0004
0005   loopsValue := 0;
0006   for i := 0 to 15 do
0007     bit := (y + 0.9) > (0.125 * TO_REAL(i));
0008
0009     if bit then
0010       loopsValue := loopsValue or RotateBit(i);
0011     end_if;
0012   end_for;
0013

```




The ST editor allows you to code and modify POUs using ST (i.e. Structured Text), one of the IEC-compliant languages.

### 6.2.1 CREATING AND EDITING ST OBJECTS

See the Creating and Editing POUs section (see Paragraphs 5.1.1 and 5.1.2).

### 6.2.2 EDITING FUNCTIONS

The ST editor is endowed with functions common to most editors running on a Windows platform, namely:

- Text selection.
-  *Edit>Cut* .
-  *Edit>Copy* .
-  *Edit>Paste* .
- *Edit>Replace* .
- Drag-and-drop of selected text.

### 6.2.3 REFERENCE TO PLC OBJECTS

If you need to add to your ST code a reference to an existing PLC object, you have two options:

- You can type directly the name of the PLC object.
- You can drag it to a suitable location. For example, global variables can be taken from the *Workspace* window, whereas embedded functions can be dragged from the *Libraries* window, whereas local variables can be selected from the local variables editor.

### 6.2.4 AUTOMATIC ERROR LOCATION

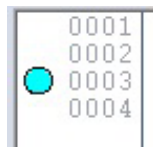
The ST editor also automatically displays the location of compiler errors. To know where a compiler error has occurred, double-click the corresponding error line in the *Output* bar.

### 6.2.5 BOOKMARKS

You can set bookmarks to mark frequently accessed lines in your source file. Once a bookmark is set, you can use a keyboard command to move to it. You can remove a bookmark when you no longer need it.

#### 6.2.5.1 SETTING A BOOKMARK

Move the insertion point to the line where you want to set a bookmark, then press *Ctrl+F2*. The line is marked in the margin by a light-blue circle.



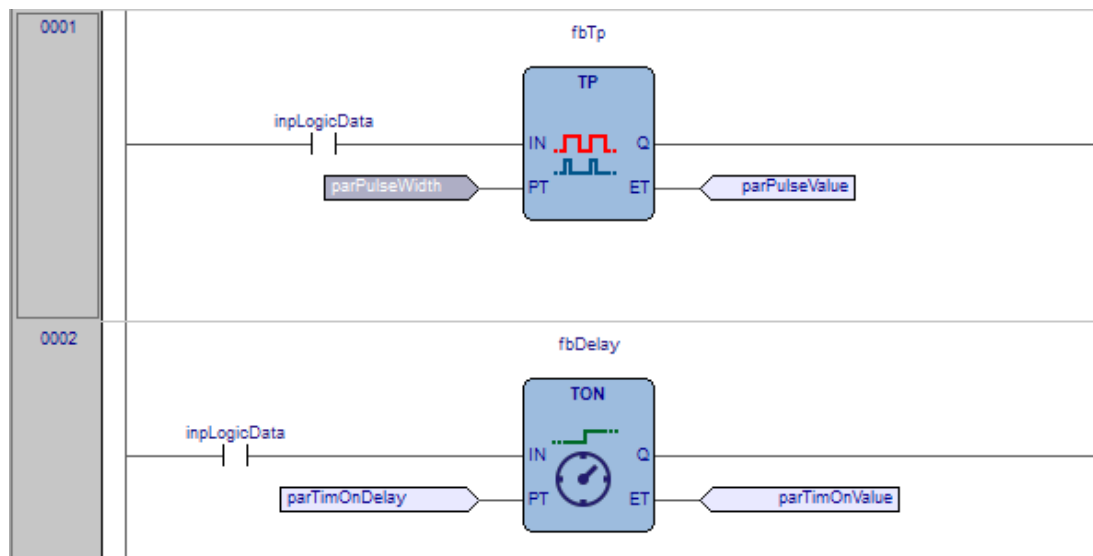
#### 6.2.5.2 JUMPING TO A BOOKMARK

Press *F2* repeatedly, until you reach the desired line.

#### 6.2.5.3 REMOVING A BOOKMARK

Move the cursor to anywhere on the line containing the bookmark, then press *Ctrl+F2*.

## 6.3 LADDER DIAGRAM (LD) EDITOR



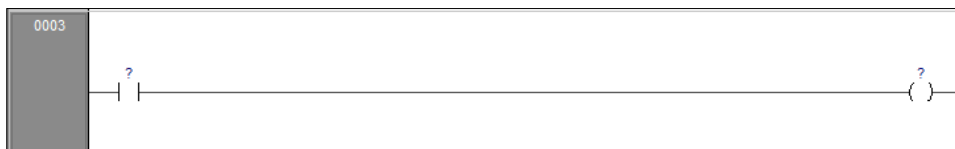
The LD editor allows you to code and modify POU's using LD (i.e. Ladder Diagram), one of the IEC-compliant languages.

### 6.3.1 CREATING A NEW LD DOCUMENT

See the Creating and Editing POU's section (see Paragraphs 5.1.1 and 5.1.2).

### 6.3.2 ADDING/REMOVING NETWORKS

Each POU coded in LD consists of a sequence of networks. A network is defined as a maximal set of interconnected graphic elements. The upper and lower bounds of every network are fixed by two straight lines, while each network is delimited on the left by a grey raised button containing the network number.



On each LD network the right and the left power rail are represented, according to the LD language indication.

On the new LD network a horizontal line links the two power rails. It is called the "power link". On this link, all the LD elements (contacts, coils and blocks) have to be placed.

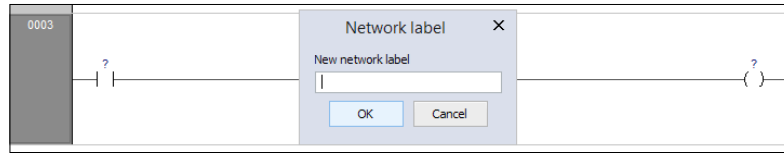
You can perform the following operations on networks:

- To add a new blank network, click [Scheme>Network>New](#), or press one of the equivalent buttons in the *Network* toolbar.
- To assign a label to a selected network, give the [Scheme>Network>Label](#). This enables jumping to the labeled network.
- To display a background grid which helps you to align objects, click [View>Grid](#).
- To add a comment, click [Scheme>Object>New>Comment](#).

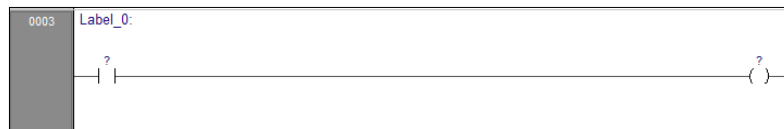
### 6.3.3 LABELING NETWORKS

You can modify the usual order of execution of networks through a jump statement, which transfers the program control to a labeled network. To assign a label to a network, double-click the raised grey button on the left, which bears the network number.

This causes a dialog box to appear, where you can type the label you want to associate with the selected network.



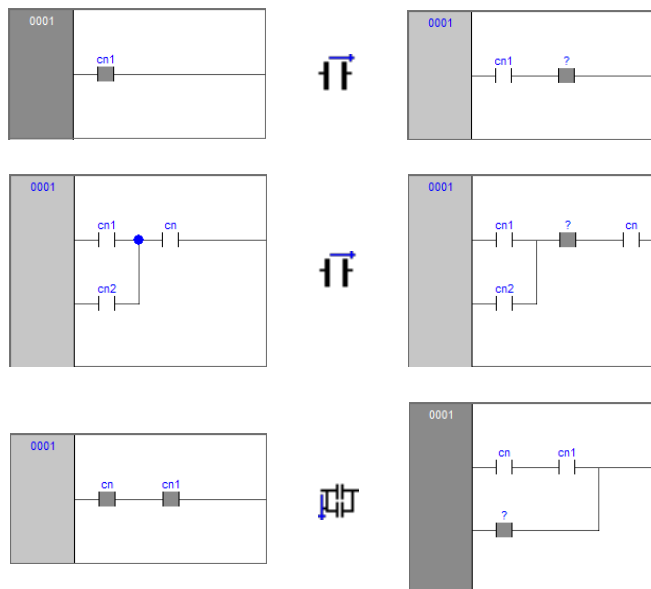
If you press *OK*, the label is printed in the top left-hand corner of the selected network.



### 6.3.4 INSERTING CONTACTS

To insert new contacts on the network apply one of the following options:

- Select a contact, a block, a pin of block, or a connection point, that will act as the insertion point. Insert the new contact choosing between the connection type (serial or parallel) and choosing the position (before or after the currently selected object) by using the *Sheme>Object>New*. For serial insertion, the new contact will be inserted on the left or right side of the selected contact/block or in the middle of the selected connection depending on the element selected before the insertion. For parallel insertions, several contacts can be selected before performing the insertion; the new contact will be inserted above or below the group of selected contacts.



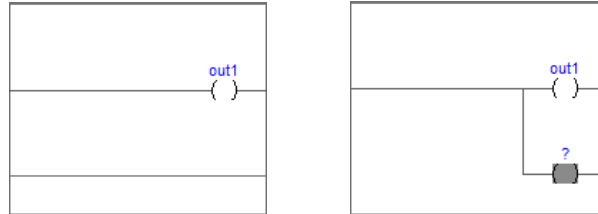
- Drag a boolean variable to the desired place over an object. For example, global variables can be taken from the *Workspace* window, whereas local variables can be selected from the local variables editor. Contacts inserted with drag and drop will always be inserted in series after the destination object.



### 6.3.5 INSERTING COILS

To insert new coils on the network apply one of the following options:

- Click **() Scheme>Object>New>Coil**. The new coil will be inserted and linked to the right power rail. If other coils, return or jumps are already present in the network, the new coil will be added in parallel with the previous ones.



- Drag a boolean variable on the network, over an existing output of the network (coil, return, jump). For example, global variables can be taken from the *Workspace* window, whereas local variables can be selected from the local variables editor.

### 6.3.6 INSERTING BLOCKS

To insert blocks on the network apply one of the following options:

- Select a contact, connection or block then click **■ Scheme>Object>New>Block**, which causes a dialog box to appear listing all the objects of the project, then choose one item from the list.
- Drag the selected object (from the *Workspace* window, the *Libraries* window or the local variables editor) over the desired connection.

If the object has at least one *BOOL* input and one *BOOL* output pins, they will be connected to the power link (and it will possible to add *EN/ENO* pins later with the provided command); otherwise the *EN/ENO* pins will be automatically added.

Operators, functions and function blocks can only be inserted into an *LD* network on the main power link, or on the power link of a branch (so they can not be inserted on the parallel of a contact); it is also not possible to create a contact in parallel of a block.

If a block has a *BOOL* input pin, it is possible to create another logical sub-network of contacts and blocks before it; otherwise, you can connect only variables, constants or expressions (that nevertheless can be connected to *BOOL* pins) to non-*BOOL* input pins.

### 6.3.7 EDITING COILS AND CONTACTS PROPERTIES

The type of a contact (normal, negated, positive, negative) or a coil (normal, negated, set, reset, positive, negative) can be changed by one of the following operations:

- Double-click on the element (contact or coil).
- Select the element and then press the *Enter* key.
- Select the element, activate the pop-up menu, then select **■ [Properties]**.

An apposite dialog box will appear. Select the desired element type from the presented list and then press *OK*.

Otherwise, select the desired contact or coil, and change its type using the six provided buttons in the *LD* toolbar or the six commands in the *Scheme* menu.

### 6.3.8 EDITING NETWORKS

The *LD* editor is endowed with functions common to most graphic applications running on a Windows platform, namely:

- Selection of a block.

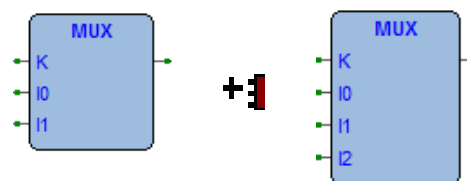


- Selection of a set of adjacent contacts by pressing *Ctrl+Left* button on each contact to select; if the selection spans across different parallel branches, more contacts will be automatically added in the selection.
- **Edit>Cut**, **Edit>Copy**, **Edit>Paste** operations of a single block as well as of a set of blocks.
- *Drag-and-drop* of the selected object or group, to move it inside or outside the current network.

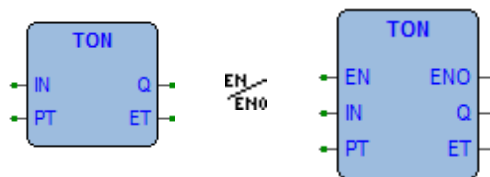
Adding, moving, deleting or copy/pasting objects will automatically recalculate the layout of the network objects; because of this, it is not possible to manually “draw” connection lines or freely placing objects without connecting them to the network.

### 6.3.9 MODIFYING PROPERTIES OF BLOCKS

- Click **Scheme>Increment pins**, to increment the number of input pins of some operators and embedded functions.



- Click **Scheme>Enable EN/ENO pins**, to display the enable input and output pins.



EN/ENO pins can be removed only if the selected block has at least one *BOOL* input and one *BOOL* output; otherwise, they will be automatically added when creating the block and it will not be possible to remove them (the *Enable EN/ENO pins* command will be disabled).

If a block has more than one *BOOL* output pin, it is possible to choose which pin will bring the *signal* out of the block and so continue the power link: select the desired output pin and click the **Scheme>Set output line** menu command.

- Click **Scheme>Object>Instance name**, to change the name of an instance of a function block.

### 6.3.10 GETTING INFORMATION ON A BLOCK

You can always get information on a block that you added to an LD document, by selecting it and then performing one of the following operations:

- click **Scheme>Object>Open source**, to open the source code of a block.
- Click **Scheme>Object properties** in the menu, to see properties and input/output pins of the selected block.

### 6.3.11 AUTOMATIC ERROR RETRIEVAL

The LD editor also automatically displays the location of compiler errors. To reach the block where a compiler error occurred, double-click the corresponding error line in the Output bar.





### 6.3.12 INSERTING VARIABLES

To connect a variable to an input or output pin of a block apply one of the following options:

- select the pin of a block, and then click the **Scheme>Object>New>Variable** menu command; then double-click the new variable object (or press *ENTER*) and enter the variable name.
- Drag the selected variable (from the *Workspace* window, the *Libraries* window or the local variables editor) over the desired pin of a block.

### 6.3.13 INSERTING CONSTANTS

To connect a numeric constant to an input pin of block, select the pin and click the **Scheme>Object>New>Constant** menu command; then double-click the new constant object (or press *ENTER*) and enter the numeric constant value.

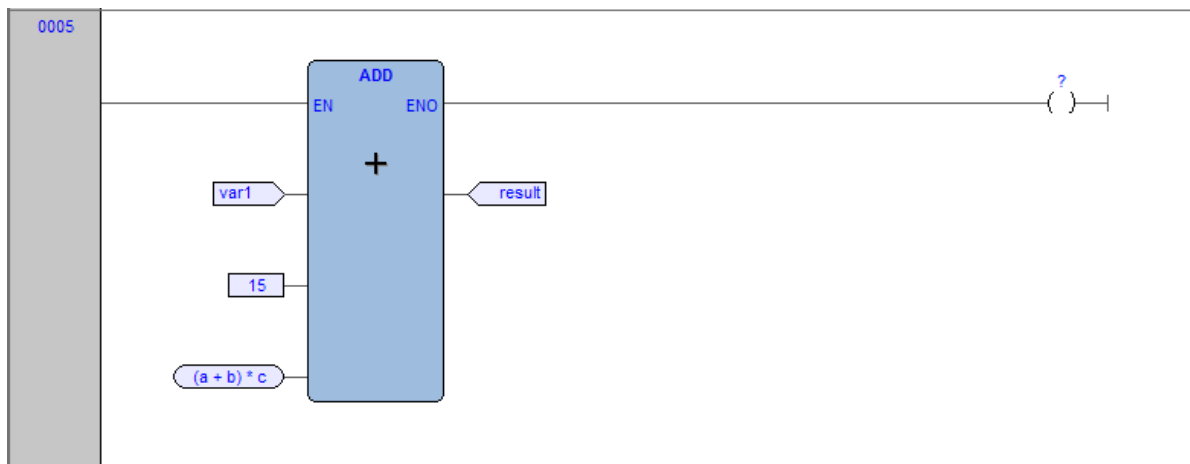
### 6.3.14 INSERTING EXPRESSION

To connect a complex expression to an input pin of block, select the pin and click the **Scheme>Object>New>Expression** menu command; then double-click the new expression object (or press *ENTER*) and enter any *ST* expression:

$(a+b) * c$

TO\_INT (n)

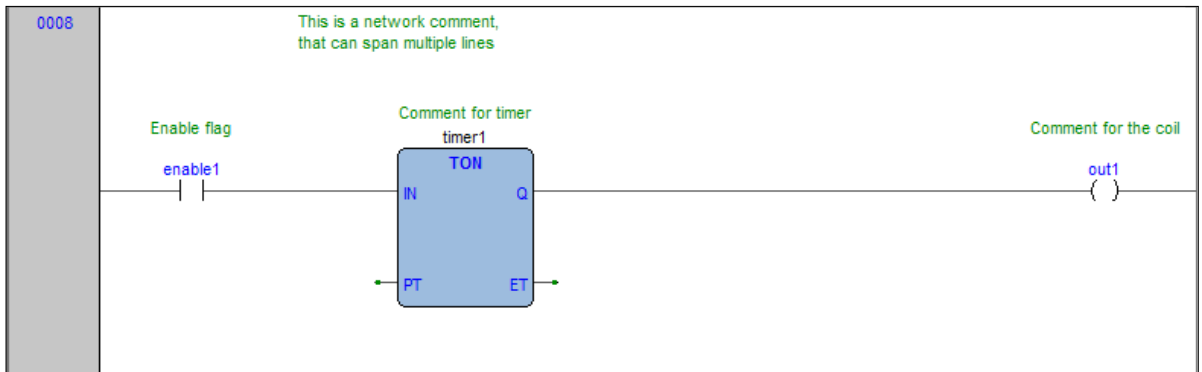
ADR (x)



### 6.3.15 COMMENTS

It is possible to insert two types of comments:

- *network* comments: activate the network by clicking on the header on the left or inside the grid (but without selecting any object), and then click the **Scheme>Object>New>Comment** menu command. The network comment will be displayed at the top of the network, and if necessary will be expanded to show all the text lines of the comment.
- *Object* comments: they are activated with the apposite menu command in **View>Show comments for objects**; above any contact, function block or coil the description of the associated PLC variable (if present) will be initially shown, but with the *Comment* command you can modify it to enter a specific object comment that will override the PLC variable description.

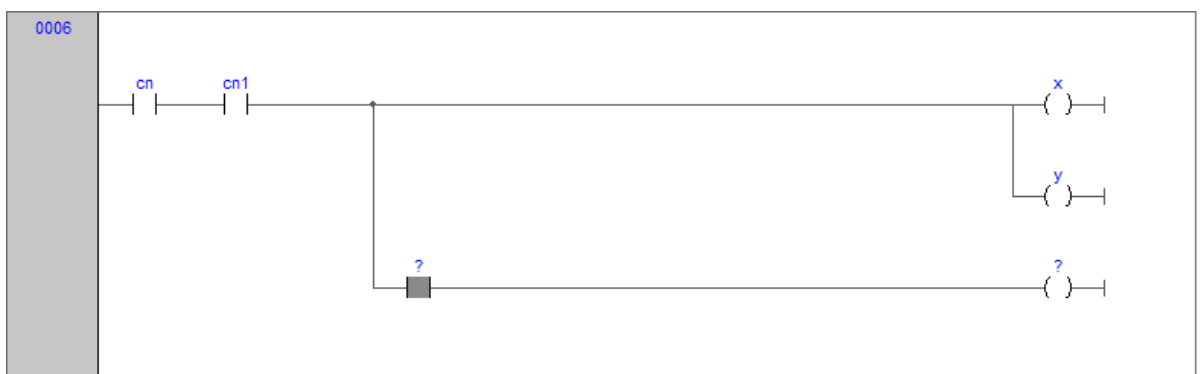
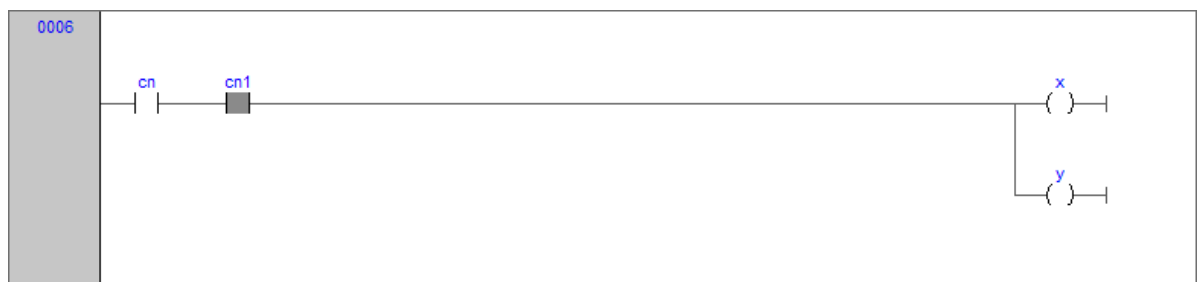


### 6.3.16 BRANCHES

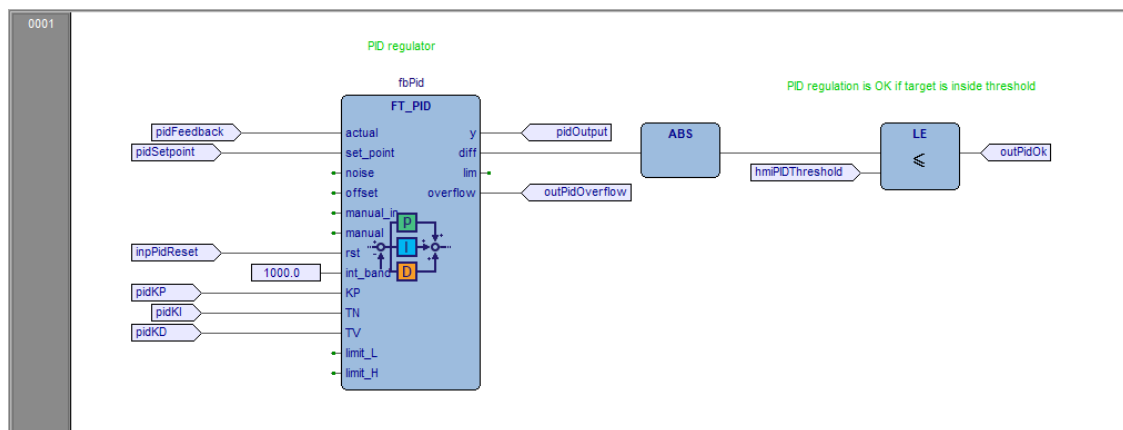
The main power line can be branched to create sub-networks, that can be further branched themselves: to add a branch, select the object after you want to create the branch and then click the `⊢ Scheme>Object>New>Branch` menu command.

The start of the new branch is marked as a big dot on the source line; deleting all objects on a branch deletes the branch itself.

Selecting an object on a branch effectively selects the branch, so for example selecting a contact on a branch and then clicking the `() Scheme>Object>New>Coil` adds the coil on the branch instead of adding it on the main power line.



## 6.4 FUNCTION BLOCK DIAGRAM (FBD) EDITOR



The FBD editor allows you to code and modify POU's using FBD (Function Block Diagram), one of the IEC-compliant languages.

### 6.4.1 CREATING A NEW FBD DOCUMENT

See the Creating and editing POU's section (see Paragraphs 5.1.1 and 5.1.2).

### 6.4.2 ADDING/REMOVING NETWORKS

Every POU coded in FBD consists of a sequence of networks. A network is defined as a maximal set of interconnected graphic elements. The upper and lower bounds of every network are fixed by two straight lines, while each network is delimited on the left by a grey raised button containing the network number.



You can perform the following operations on networks:

- To add a new blank network, click [Scheme>Network>New](#).
- To display a background grid which helps you to align objects, click [View>View grid](#).
- To add a comment, click [Scheme>Object>New>Comment](#).

### 6.4.3 LABELING NETWORKS

You can modify the usual order of execution of networks through a jump statement, which transfers the program control to a labeled network. To assign a label to a network, double-click the raised grey button on the left, that bears the network number; or, after selecting a network, choose [Scheme>Network>Label](#).



This causes a dialog box to appear, which lets you type the label you want to associate with the selected network.



If you press *OK*, the label is printed in the top left-hand corner of the selected network.



### 6.4.4 INSERTING AND CONNECTING BLOCKS

This paragraph shows you how to build a network.

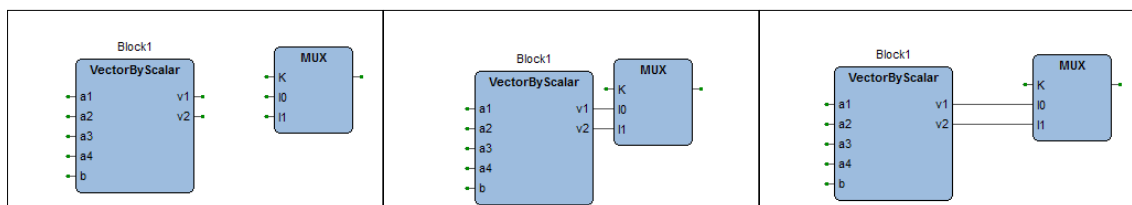
Add a block to the blank network, by applying one of the following options:

- Click **Scheme>Object>New>Function Block** which causes a dialog box to appear listing all the objects of the project, then choose one item from the list. If the block is a constant, a return statement, or a jump statement, you can directly press the relevant buttons in the *FBD* toolbar.
- Drag the selected object to the suitable location. For example, global variables can be taken from the *Workspace* window, whereas standard operators and embedded functions can be dragged from the *Libraries* window, whereas local variables can be selected from the local variables editor.

Repeat until you have added all the blocks that will make up the network.

Then connect blocks:

- Click **Edit>Connection mode**, or simply press the space bar of your keyboard. Click once the source pin, then move the mouse pointer to the destination pin: the *FBD* editor draws a logical wire from the former to the latter.
- If you want to connect two blocks having a one-to-one correspondence of pins, you can enable the auto connection mode by clicking **Scheme>Auto connect**. Then take the two blocks, drag them close to each other so as to let the corresponding pins coincide. The *FBD* editor automatically draws the logical wires.



If you delete a block, its connections are not removed automatically, but they become invalid and they are redrawn red. Click **Scheme>Delete invalid connection**.

### 6.4.5 EDITING NETWORKS

The *FBD* editor is endowed with functions common to most graphic applications running on a Windows platform, namely:

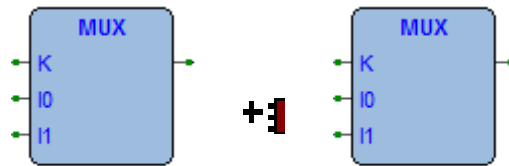
- Selection of a block.
- Selection of multiple elements: by pressing *Shift* + *left button* and drawing a frame including the blocks to select, or *Ctrl* + *left button* and select one by one the elements.
- **Edit>Cut**, **Edit>Copy**, **Edit>Paste** operations of a single block as well as of a set of blocks.



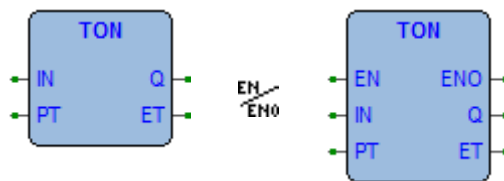
- Drag-and-drop.

### 6.4.6 MODIFYING PROPERTIES OF BLOCKS

- Click **Scheme>Increment pins**, to increment the number of input pins of some operators and embedded functions.



- Click **EN/ENO Scheme>Enable EN/ENO pins**, to display the enable input and output pins.



- Click **Scheme>Object>Instance name**, or click **Scheme>Object properties**, to change the name of an instance of a function block.

### 6.4.7 GETTING INFORMATION ON A BLOCK

You can always get information on a block that you added to an FBD document, by selecting it and then performing one of the following operations:

- Click **Scheme>Object>Open source**, or double-click the block, to open its source code.
- Click **Scheme>Object properties**, to see properties and input/output pins of the selected block.

### 6.4.8 AUTOMATIC ERROR RETRIEVAL

The FBD editor also automatically displays the location of compiler errors. To reach the block where a compiler error occurred, double-click the corresponding error line in the *Output* bar.

## 6.5 SEQUENTIAL FUNCTION CHART (SFC) EDITOR

The SFC editor allows you to code and modify POUs using SFC (i.e. Sequential Function Chart), one of the IEC-compliant languages.

### 6.5.1 CREATING A NEW SFC DOCUMENT

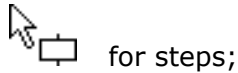
See the creating and editing POUs section (see Paragraphs 5.1.1 and 5.1.2).

### 6.5.2 INSERTING A NEW SFC ELEMENT

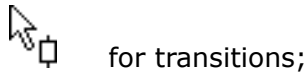
- Click **Scheme>Object>New>Step**.
- Click **Scheme>Object>New>transition**.
- Click **Scheme>Object>New>Jump**.



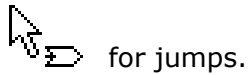
In either case, the mouse pointer changes to:



for steps;



for transitions;



for jumps.

### 6.5.3 CONNECTING SFC ELEMENTS

Follow this procedure to connect SFC blocks:

- Click `Edit>Connection mode`, or simply press the space bar on your keyboard. Click once the source pin, then move the mouse pointer to the destination pin: the SFC editor draws a logical wire from the former to the latter.
- Alternatively, you can enable the auto connection mode by clicking `Scheme>Auto connect`. Then take the two blocks, and drag them close to each other so as to let the respective pins coincide, which makes the SFC editor draw automatically the logical wire.

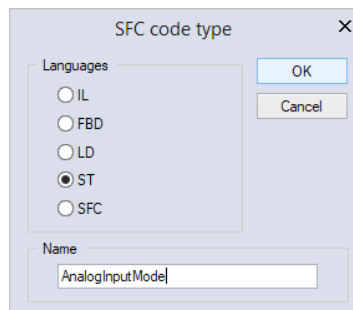
### 6.5.4 ASSIGNING AN ACTION TO A STEP

This paragraph explains how to implement an action and how to assign it to a step.

#### 6.5.4.1 CREATING AN ACTION

To create a new action: choose `Scheme>Code object>New action` or select the *Actions* folder in the *workspace* and choose `[New action]` from its context menu.

In either case, LogicLab displays a dialog box like the one shown below.



Select one of the languages and type the name of the new action in the text box at the bottom of the dialog box. Then either confirm by pressing *OK*, or quit by clicking *Cancel*.

#### 6.5.4.2 WRITING THE CODE OF AN ACTION

To modify the code of an action you can double-click it on the project tree or select it and choose `Edit source` from its context menu.

Note that you are not allowed to declare new local variables, as the action that you are now editing is a component of the original SFC module, which is the POU where local variables can be declared. The scope of local variables extends to all the actions and transitions making up the SFC diagram.

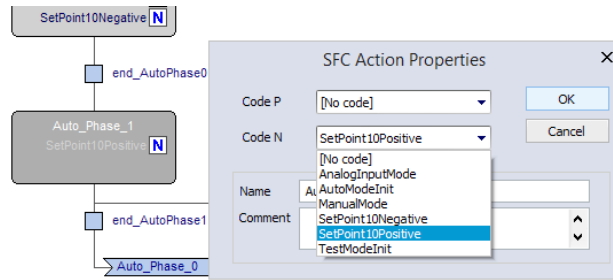
#### 6.5.4.3 ASSIGNING AN ACTION TO A STEP

You can assign or change the action associated to a step by opening its properties window; to do so, double-click the step in the SFC editor.



Be careful, if the step has already one or two action associated, they will be shown on the block as a white square with a specific letter written inside, if you click the letter you will open directly the action editor, not the step properties window.

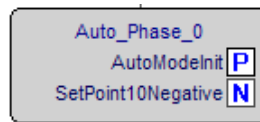
Opening the step properties window will causes the following dialog box to appear.



From the list shown in the *Code N* box, select the name of the action you want to execute when the step is active.

From the list shown in the *Code P (Pulse)* box, you may optionally choose the name of the action you want to execute each time the step becomes active (which means the action is executed only once per step activation, regardless of the number of cycles the step remains active).

Confirm the assignments by pressing *OK*.



In the SFC editor, the actions assigned to a step are represented by white square with a specific letter inside: action P with letter P in the top right corner; action N with letter N in the bottom right corner.

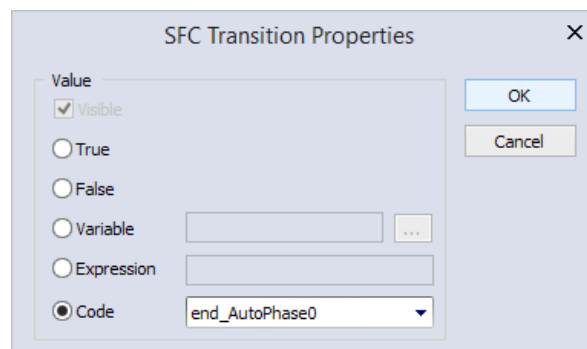
Double-clicking directly a white square works as a shortcut to open that action editor.

## 6.5.5 TRANSITIONS CONDITIONS

### 6.5.5.1 ASSIGN A CONSTANT/VARIABLE, AS A TRANSITION CONDITION

As stated in the relevant section of the language reference, a transition condition can be assigned through a constant, a variable, or a piece of code. This paragraph explains how to use the first two means, while conditional code is discussed in the next paragraph.

First of all double-click the transition you want to assign a condition to. This causes the following dialog box to appear.



Select *True* if you want this transition to be constantly cleared, *False* if you want the PLC program to keep executing the preceding block.

Instead, if you select *Variable* the transition will depend on the value of a Boolean variable. Click the corresponding bullet, to make the text box to its right available, and to specify the name of the variable.

To this purpose, you can also make use of the objects browser, that you can invoke by pressing the button its right side.

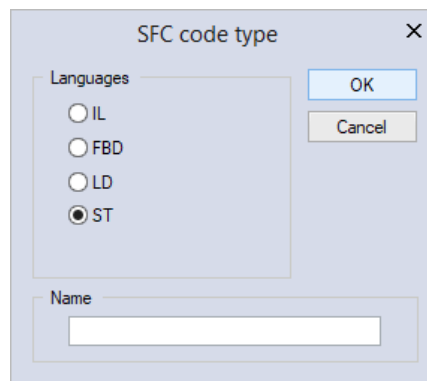
Click *OK* to confirm, or *Cancel* to quit without applying changes.

### 6.5.5.2 ASSIGN A CONDITIONAL CODE TO A TRANSITION

This paragraph explains how to specify a condition through a piece of code, and how to assign it to a transition.

Start by creating a transition code: select `Scheme>Code object>New transition code`; or select `[New transition]` from the context menu of the SFC POU in the *Workspace*

In either case, LogicLab displays a dialog box similar the one shown in the following picture.

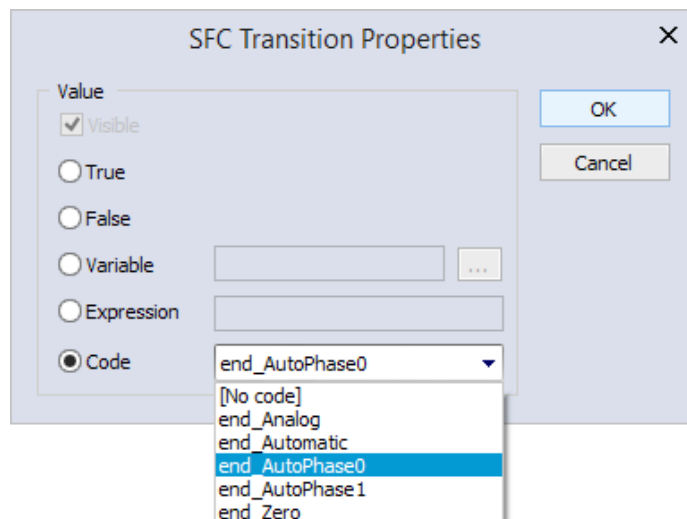


Note that you can use any language except SFC to code a condition. Select one of the languages and type the name of the new condition in the text box at the bottom of the dialog box. Then either confirm by pressing *OK*, or quit by clicking *Cancel*.

Now the transition code is added to the project tree; you can open its editor by double-clicking it, or by opening its context menu and selecting `Edit source`.

Note that you are not allowed to declare new local variables, as the module you are now editing is a component of the original SFC module, which is the POU where local variables can be declared. The scope of local variables extends to all the actions and transitions making up the SFC diagram.

To assign the transition code to the transition element, double-click the transition element in the SFC editor. This causes the following dialog box to appear.

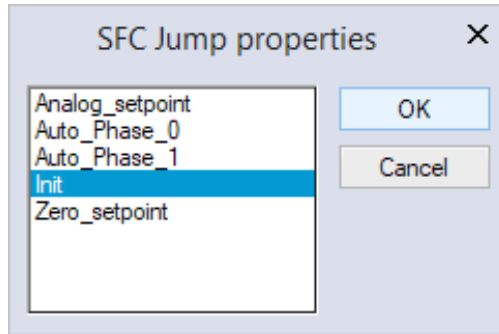




Select *Code*, and then choose the name of the condition you want to assign to this transition from the list. Then confirm by pressing *OK*.

### 6.5.6 SPECIFYING THE DESTINATION OF A JUMP

To specify the destination step of a jump, double-click the jump element in the SFC editor. This causes the dialog box shown below to appear, listing the name of all the existing steps.



Select the destination step, then either press *OK* to confirm or *Cancel* to quit. The name of the target step will be shown inside the jump element.

### 6.5.7 EDITING SFC NETWORKS

The SFC editor is endowed with functions common to most graphic applications running on a Windows platform, namely:

- Selection of a block.
- Selection of a set of blocks by pressing *Ctrl* + left button.
- *Edit>Cut*, *Edit>Copy*, *Edit>Paste* operations of a single block as well as of a set of blocks.
- Drag-and-drop.

## 6.6 VARIABLES EDITOR

LogicLab includes a graphical editor for both global and local variables that supplies a user-friendly interface for declaring and editing variables: the tool takes care of the translation of the contents of these editors into syntactically correct IEC 61131-3 source code.

As an example, consider the contents of the Global variables editor represented in the following figure.

	Name	Type	Address	Group	Array	Init value	Attribute	Description
1	pidKP	REAL	%MD1.0	PID	No		---	PID proportional gain
2	pidKI	REAL	%MD1.4	PID	No		---	PID integral time
3	pidSetpoint	REAL	%MB1.8	PID	No		---	PID setpoint (from -1 to +1)
4	pidOutput	REAL	%MD1.12	PID	No		---	PID output value

The corresponding source code will look like this:

```
VAR_GLOBAL
  gA : BOOL := TRUE;
  gB : ARRAY[ 0..4 ] OF REAL;
  gC AT %MD60.20 : REAL := 1.0;
END_VAR
VAR_GLOBAL CONSTANT
```



```

gD : INT := -74;
END_VAR

```

Alternatively, LogicLab includes also a textual editor for both global and local variables; using this editor will allow you to declare and edit the source code of the variables.

## 6.6.1 OPENING A VARIABLES EDITOR

### 6.6.1.1 OPENING THE GLOBAL VARIABLES EDITOR

In order to open the Global variables editor, expand the Global variables folder and any subfolder until you reach the desired Global variables group, then double-click on it.

	Name	Type	Address	Array	Init value	Attribute	Description
1	parCtDownPreset	UDINT	Auto	No	100	---	Counter down preset
2	parCtUpPreset	UDINT	Auto	No	15	---	Counter up preset
3	parPulseValue	UDINT	Auto	No		---	Actual pulse time value
4	parPulseWidth	UDINT	Auto	No	500	---	Pulse width
5	parTimOnDelay	UDINT	Auto	No	1000	---	Delay of the ON delay timer
6	parTimOnValue	UDINT	Auto	No		---	Actual value of the timer

### 6.6.1.2 OPENING A LOCAL VARIABLES EDITOR

To open a local variables editor, just open the relative Program Organization Unit.

	Class	Pin	Name	Type	Array	Init value	Attribute	Description
1	VAR_INPUT	0	in	REAL	No		..	
2	VAR_INPUT	1	k	REAL	No		..	
3	VAR_OUTPUT	0	out	REAL	No		..	

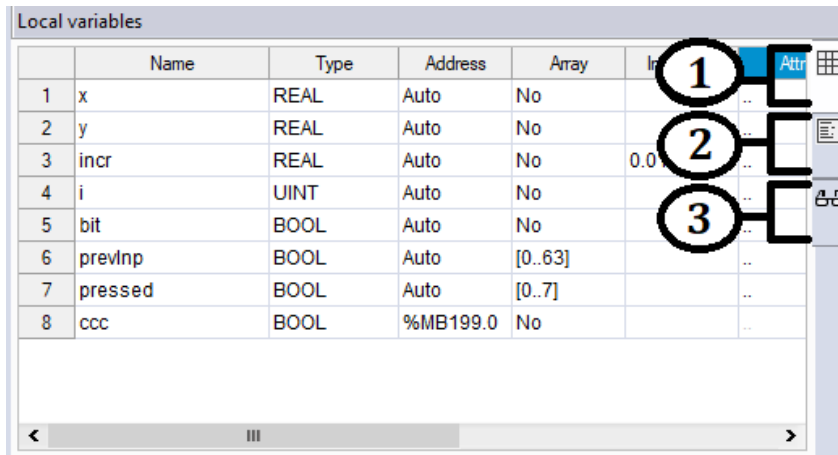
```

0001
0002
0003   out := k * ( in - out ) + out;
0004
0005
0006

```

### 6.6.1.3 SWITCH FROM GRAPHIC TO TEXTUAL EDITOR

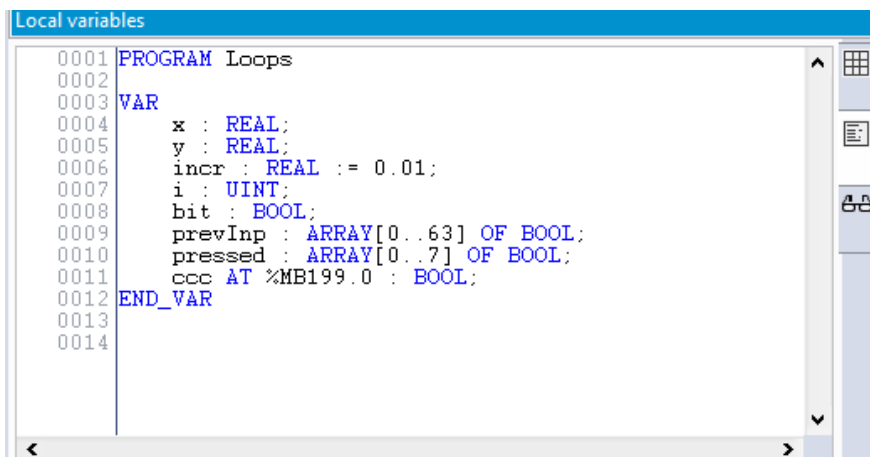
For both global and local variables, when you open their graphic editor, you'll see on the far right border three tabs:



The first, selected by default, is the graphic variables editor; this editor provides a user-friendly interface.

The second is the textual variables editor; this editor allows the user to declare and edit variables by modifying directly the source code. The textual editor can be really useful if you need to paste some variables declaration code, taken from another source.

Switching from the graphic editor to the textual editor, will show you the actual source code of the defined variables:



The third tab is the watch editor, refer to the appropriate chapter of this manual for details (see Paragraph 9.1.1.1).

### 6.6.2 CREATING A NEW VARIABLE

In order to create a new variable, you may click **Variables>Insert**.

### 6.6.3 EDITING VARIABLES

Follow this procedure to edit the declaration of a variable in a variables editor (all the following steps are optional and you will typically skip most of them when editing a variable):

- 1) Edit the name of the variable by entering the new name in the corresponding cell.

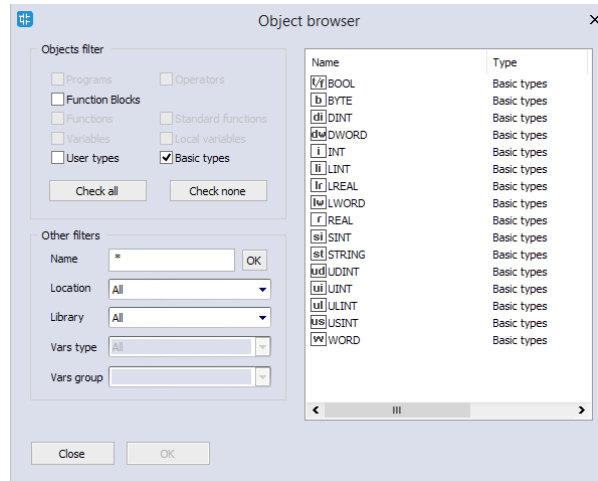
	Name	Type	Address	Array	Init value	Attribute	Description
1	fbDelay	TON	Auto	No		..	
2	fbCtu	CTU_UDINT	Auto	No		..	
3	fbCtd	CTD_UDINT	Auto	No		..	
4	fbTp	TP	Auto	No		..	
5	ettp	UDINT	Auto	No		..	

- 2) Change the variable type, either by editing the type name in the corresponding cell or by clicking on the button in its right and select the desired type from the *object*

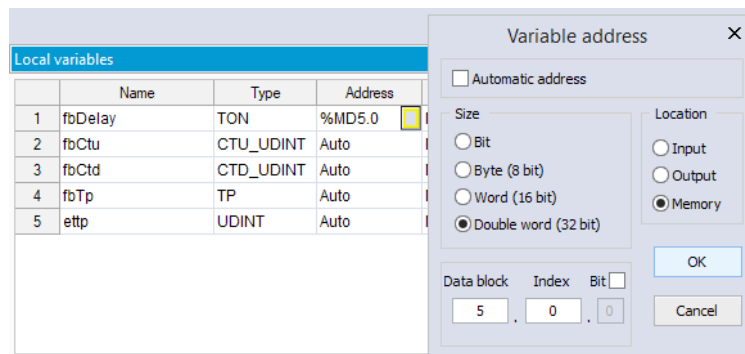


browser.

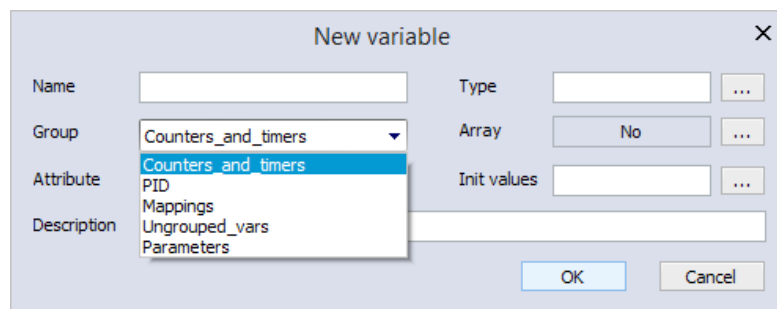
	Name	Type	Address	Array	Init value	Attribute	Description
1	fbDelay	TON	Auto	No		..	
2	fbCtu	CTU_UDINT	Auto	No		..	
3	fbCtd	CTD_UDINT	Auto	No		..	
4	fbTtp	TP	Auto	No		..	
5	ettp	UDINT	Auto	No		..	



- 3) Edit the address of the variable by clicking on the button in the corresponding cell and entering the required information in the window that shows up. Note that, in the case of global variables, this operation may change the position of the variable in the project tree.



- 4) In the case of global variables, you can assign the variable to a group. You have three ways to do so:
  - First create the group and then create the variable from inside the group.
  - You can drag&drop existing variable into a new group.
  - Specify the group the variable belongs to when creating the variable; to do so, choose the appropriate voice from the menu **Variables > Add**; the following window will appear, allowing you to specify also the variable group.



- 5) Choose whether a variable is an array or not; if it is, edit the size of the variable.

Local variables				
	Name	Type	Address	Array
1	base	UDINT	Auto	[0..4, 0..5, 0..5]
2	increment	USINT	Auto	No

Size of variable ✕

Scalar

Array / Matrix

Dimension 1 0 ..  size = 5

Dimension 2 0 ..  size = 6

Dimension 3 0 ..  size = 6

- 6) Edit the initial values of the variable: click on the button in the corresponding cell and enter the values in the window that pops up.

Init values for: () ✕

[ 1 , 2 , 3 ]

- 7) Assign an attribute to the variable (for example, `CONSTANT` or `RETAIN`), by selecting it from the list which opens when you click on the corresponding cell.

	Name	Type	Address	Array	Init value	Attribute	Description
1	base	UDINT	Auto	No	100	..	
2	increment	USINT	Auto	No		..	

**CONSTANT**

- 8) Type a description for the variable in the corresponding cell. Note that, in the case of global variables, this operation may change the position of the variable in the project tree.

	Name	Type	Address	Array	Init value	Attribute	Description
1	base	UDINT	Auto	No	100	..	Base element
2	increment	USINT	Auto	No	1	CONSTANT	Constant increment value

- 9) Save the project to persist the changes you made to the declaration of the variable.

## 6.6.4 DELETING VARIABLES

In order to delete one or more variables, select them in the editor: you may use the `CTRL` or the `SHIFT` keys to select multiple elements.

	Name	Type	Address	Array	Init value	Attribute	Description
1	fbDelay	TON	Auto	No		..	
2	fbCtu	CTU_UDINT	Auto	No		..	
3	fbCtd	CTD_UDINT	Auto	No		..	
4	fbTp	TP	Auto	No		..	
5	ettp	UDINT	Auto	No		..	

Then, click `Variables>Delete`.

Notice that you cannot delete the `RESULT` of an IEC61131-3 FUNCTION.

## 6.6.5 SORTING VARIABLES

You can sort the variables in the editor by clicking on the column header of the field you want to use as the sorting criterion.

	Name	Type	Address
1	aux	INT	Auto
2	base	UDINT	Auto
3	exponent_1	INT	Auto
4	exponent_2	REAL	Auto
5	increment	USINT	Auto
6	multiplier	USINT	Auto

	Name	Type	Address
1	increment	USINT	Auto
2	multiplier	USINT	Auto
3	aux	INT	Auto
4	exponent_1	INT	Auto
5	base	UDINT	Auto
6	exponent_2	REAL	Auto

## 6.6.6 COPYING VARIABLES

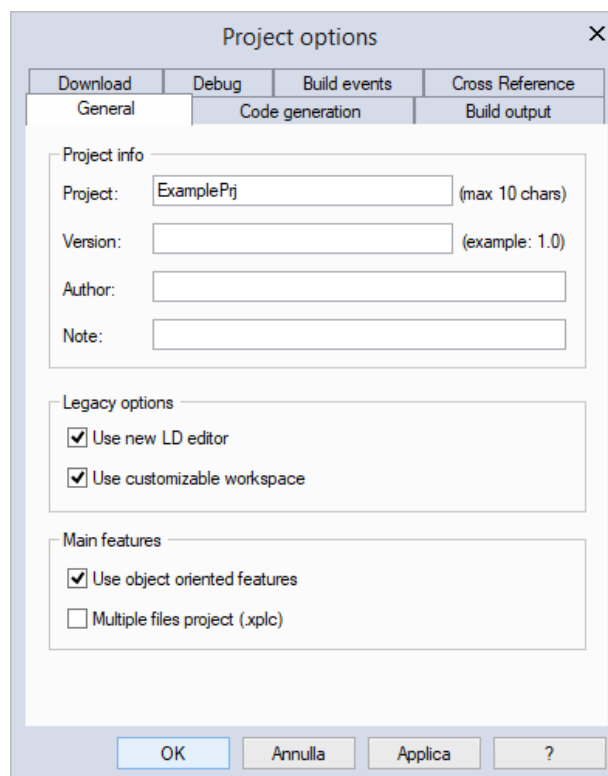
The variables editor allows you to quickly copy and paste elements. You can either use keyboard shortcuts or the *Edit>Copy* , *Edit>Paste* menu.

Note: overlapping addresses problems may occur by copying mapped variables. LogicLab can automatically assign new free address to the new pasted variable and fix the overlap. In order to enable this functionality please refer to paragraph 3.6 and 4.8.3.2 for further details.

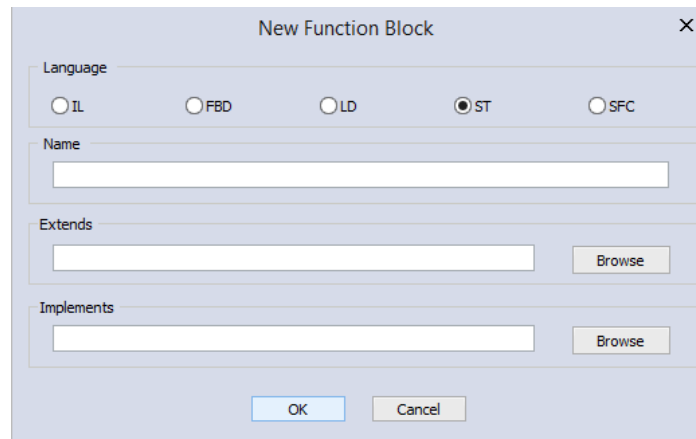
## 6.7 OBJECT ORIENTED

### 6.7.1 ENABLE OBJECT ORIENTED PROGRAMMING

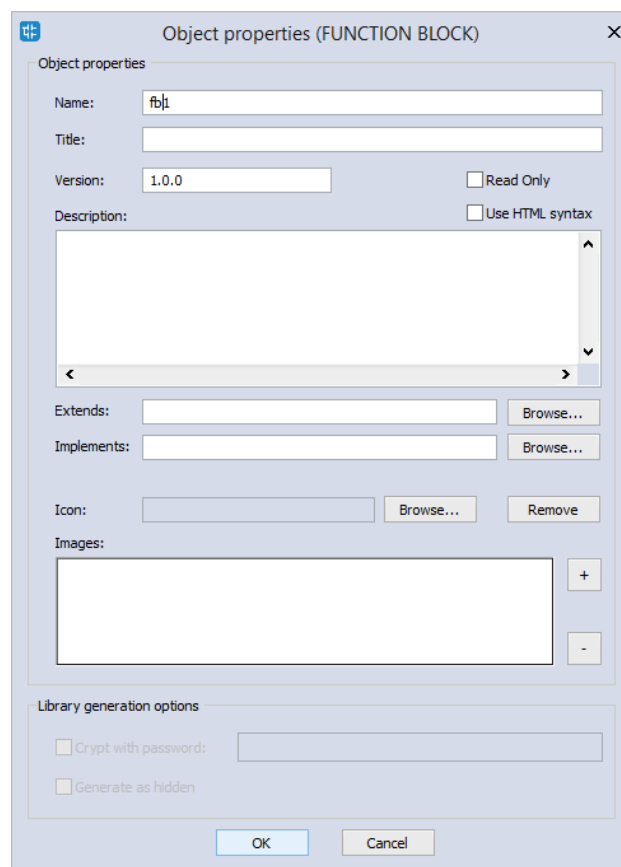
LogicLab allow the users to work with the object oriented programming, enhancing the function blocks and treating them as classes; since this feature needs to be supported by the target device implementation, it is disabled by default; in order to enable it you have to select *Project > Options > Tab general* and check the *Use object oriented features* checkbox.



Now the dialog that shows up when creating a new Function Block (as well as the Object Properties dialog), will show additional content:



The 'New Function Block' dialog features a 'Language' section with radio buttons for IL, FBD, LD, ST (selected), and SFC. Below this is a 'Name' text field. The 'Extends' section contains a text field and a 'Browse' button. The 'Implements' section also contains a text field and a 'Browse' button. At the bottom are 'OK' and 'Cancel' buttons.



The 'Object properties (FUNCTION BLOCK)' dialog includes fields for 'Name' (containing 'fb|'), 'Title', 'Version' (1.0.0), and 'Description'. It has checkboxes for 'Read Only' and 'Use HTML syntax'. The 'Extends' and 'Implements' fields each have a 'Browse...' button. The 'Icon' field has a 'Browse...' button and a 'Remove' button. The 'Images' section has a large empty box with '+' and '-' buttons. At the bottom, there are 'Library generation options' with checkboxes for 'Crypt with password' and 'Generate as hidden', and 'OK' and 'Cancel' buttons.

In the *Extends* field, you can specify another Function Block that will be treated as a father class for the current function block; only one Function Block can be specified as a father. The current Function Block will inherit every object of the father.

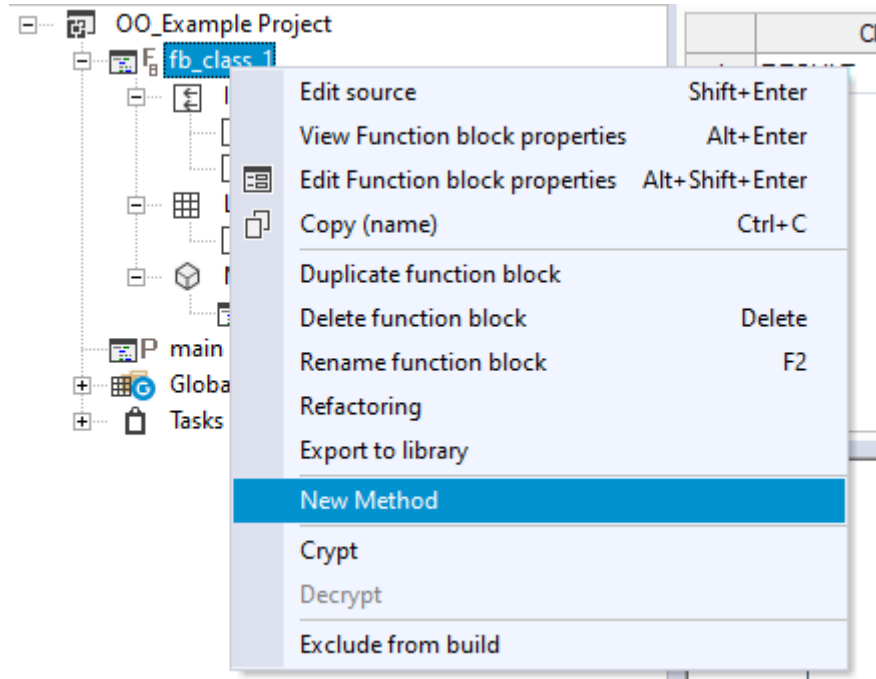
In the *Implements* field, you can specify one or more interfaces.

Please note that even if the Object Oriented feature is enabled, you can keep using the function blocks as normal function blocks.

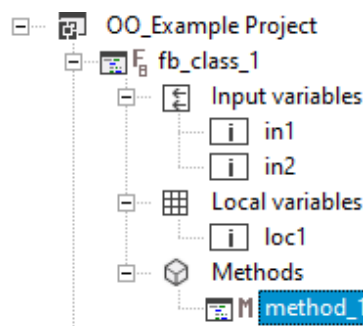
## 6.7.2 METHODS

### 6.7.2.1 CREATE NEW METHODS

If the Object Oriented feature is enabled, you can consider a Function Block as a class; in order to create a new method: open the context menu of the Function Block by right-clicking on it's instance in the project tree, then choose *New Methods*.



A dialog will appear asking you which language you wish to use to implement the method body, SFC cannot be used. Once created, the new method will be shown in the project tree:



You can define any number of methods for the same class, but you cannot define more methods with the same name (even if their prototype is different).

### 6.7.2.2 INVOKING METHODS

Methods work like functions, except that they have access to the class context (which means, for example, the *Input variables* of the function block).

Methods can be invoked in a POU (for example in a program), like extensions to the Func-





tion Block instance:

	Class	Pin	Name	Type	Array	Init value	Attribute	Description
1	VAR_INPUT	0	x	INT	No		..	
2	RESULT		method_1	DINT	No		..	Method result

```

main
0001
0002   myClass_1.in1 := 10;
0003   myClass_1.in2 := 10;
0004
0005   res := myClass_1.method_1(v1);
0006
0007

method_1
0001   method_1 := fb_in1 + fb_in2 + fb_loc1 + x;
0002
0003
    
```

Methods can be called from other methods of the same class, to do so use the *this.* specifier:

	Class	Pin	Name	Type
1	RESULT		method_2	DINT

```

0001
0002   method_2 := this.method_1(123);
    
```

Methods can be called also from inside the relative Function Block body, but the Function Block body cannot be called from inside a method.

### 6.7.2.3 INHERITED METHODS

If in the *Extends* field, when creating a new Function Block, or inside its Object properties window, you specify the name of another Function Block, the current one will inherit all of its methods.

Inherited methods are always considered as **virtual methods**, which means that if the children Function Block implements a method with the same prototype of a method implemented by the father, no error is raised, as the children method will override the father method.

If you wish to explicitly call the father method, you can do it by using the *super.* specifier:

	Class	Pin	Name	Type
1	RESULT		method_1	DINT
2	VAR_INPUT	0	x	INT

```

0001
0002   method_1 := super.method_1(x);
0003
0004
    
```

You can also call the father Function Block body, instead of a specific method, using `super();` alone.

You can see if a Function Block extends another function block, by looking at its property window.

### 6.7.2.4 METHODS POLIMORPHYSM

LogicLab can handle polimorphysm with classes, to do so you'll have to use the reference data type (see *paragraph 11.1.2.1*).

Create a variable of type reference to the father class; now valorize it usign the reference of a child class; now you can use the reference variable to call a method (or to access the class properties), LogicLab will be able to recognize which is the correct method to call.

	Name	Type	Address
1	myClass_1	fb_class_1	Auto
2	v1	INT	Auto
3	res	DINT	Auto
4	fRef	fb_class_1^	Auto
5	myClass_2	fb_class_2	Auto

```

0001
0002 myClass_1.fb_in1 := 10;
0003 myClass_1.fb_in2 := 10;
0004
0005 res := myClass_1.method_1(v1);
0006
0007
0008 fRef := REF(myClass_1);
0009
0010 res := fRef^.method_1(v1);
0011
~~~~

```

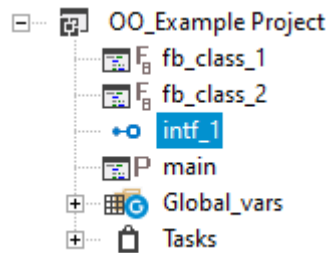
In this example, `fRef^.method_1()` will call `method_1` of `fb_Class_2` even if `fRef` is a reference to `fb_Class_1`.

## 6.7.3 INTERFACES

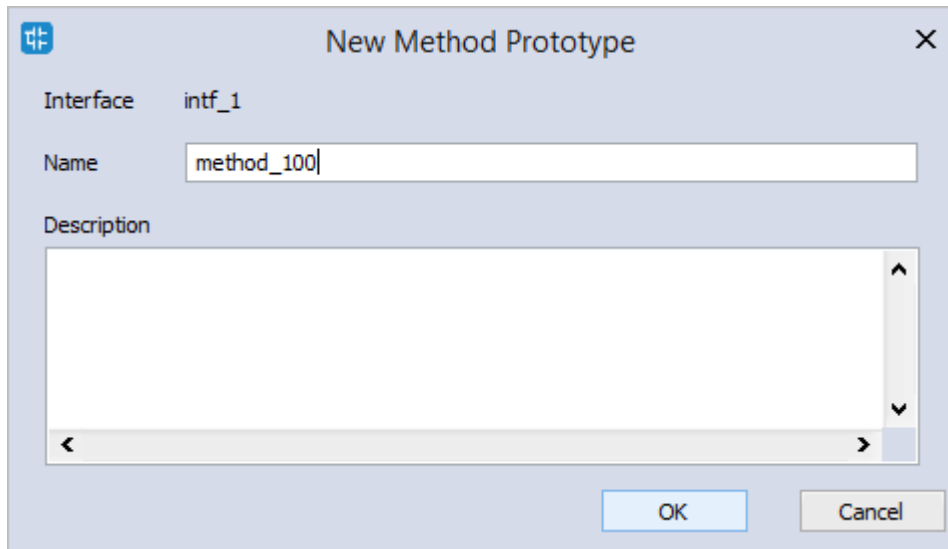
### 6.7.3.1 CREATE NEW INTERFACES

In order to create a new interface, after having enabled the Object Oriented programming, select **Project > New object > New definition > Interface**. A dialog will be shown asking you to specify the name of the interface and (optionally) a description.

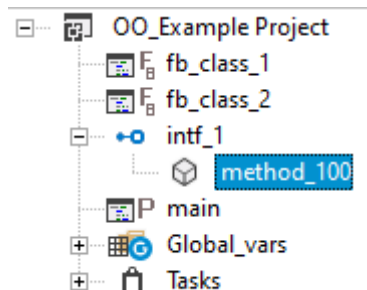
After creation, the interface will appear as a new element in the project tree.



Now you can add a method prototype: from the interface context menu, select *Add method prototype*; a dialog will appear asking you to insert the name of the method and (optionally) a description.



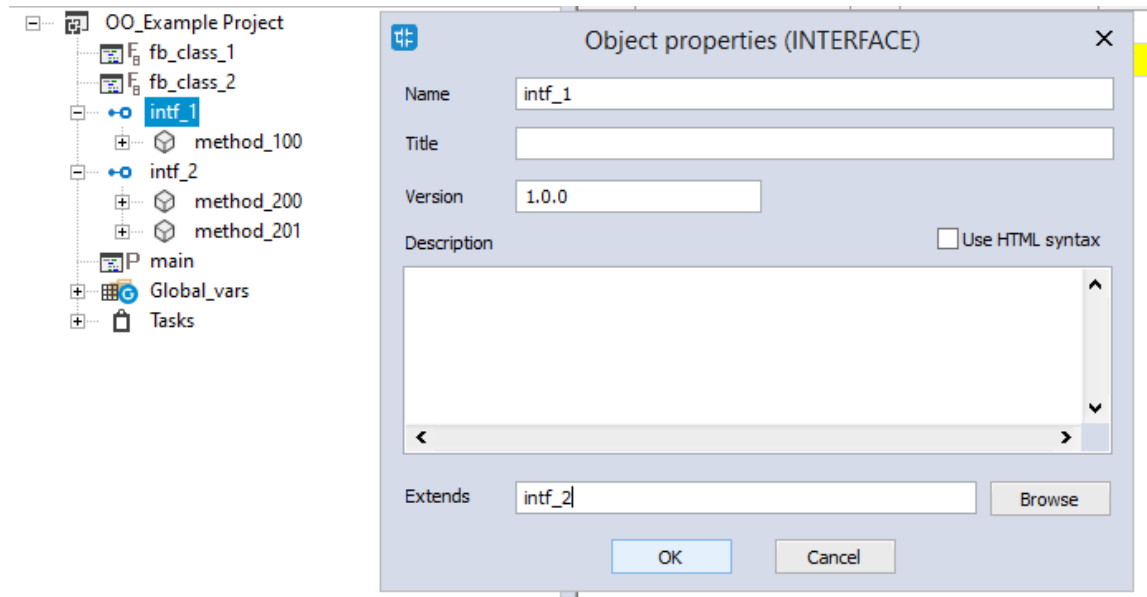
After creation, the method prototype will appear in the project tree as a subelement of the interface element.



By double clicking on the method prototype element, a new editor will be opened in the editor window, allowing you specify the method prototype (which means the required input variables).

An interface can extend another interface (only one), in a father-child hierarchy; doing so, the children interface will inherit all the method prototypes of the father interface.

To extend an interface select *Edit properties* from the interface context menu, a new dialog will be shown allowing you to modify several properties, included the extend field.

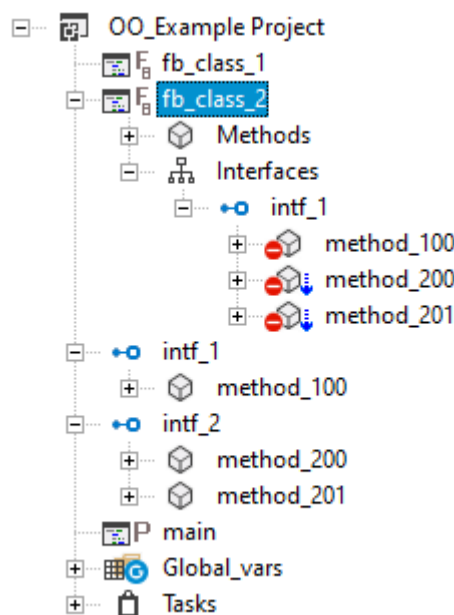


You can see if an interface extends another interface by looking at its property window.

### 6.7.3.2 IMPLEMENTING INTERFACES

The methods prototype of an interface can be implemented by a function block.

Upon creation of the function block, or editing its properties, you can specify one or more interfaces in the *Implements* field; the interface and its methods will appear in the project tree as subelements of the Function Block that implements them.



In the example of the above picture, `fb_class_2` implements `intf_1`; and `intf_1` extends `intf_2`.





In order to implement a method prototype of an interface, you need to create a method which has the same prototype (name and input variables) of the method prototype you wish to implement. Compiling a project with method prototypes not correctly implemented, will raise errors.

To implement a method prototype you can create a new method (see paragraph 6.7.2.1) which has the same prototype of the method you wish to implement, or you can right-click on the method prototype and select *Implement method*; this way a new method will be



created, already with the correct prototype.

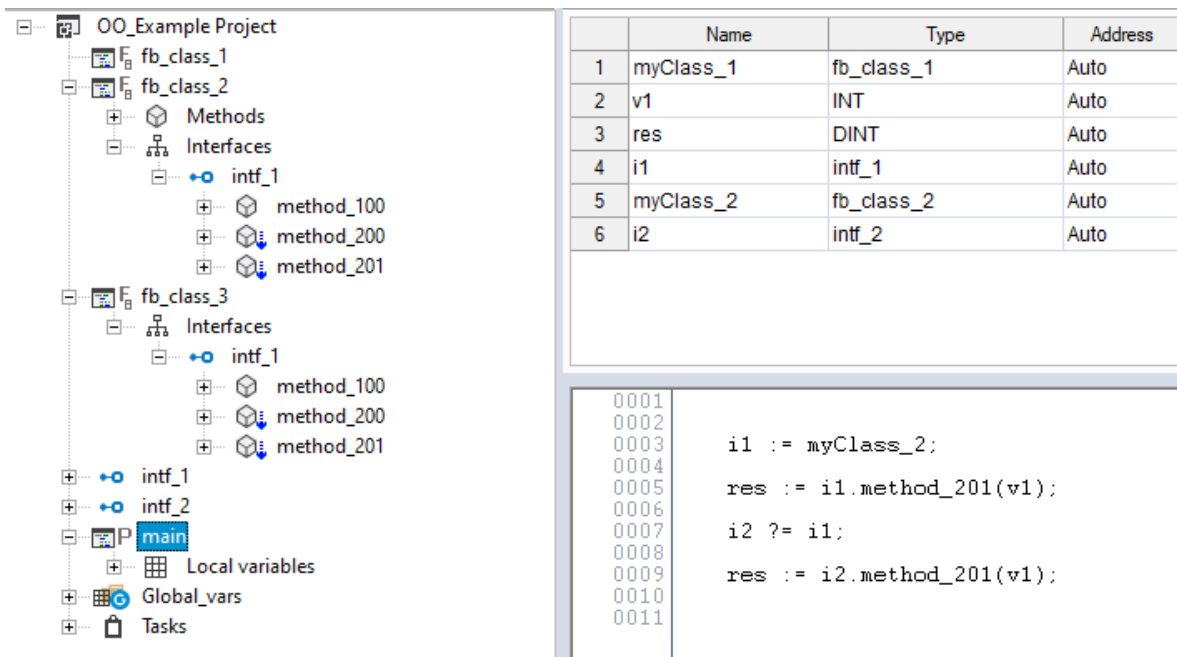
Method prototypes can have different icons, with different meaning:

-  method prototype correctly implemented.
-  there is a method with the same name but different prototype from the one you wish to implement.
-  no method has been found with the same name of the method prototype you wish to implement.
-  the previous icons can also have a blue arrow on their right side, that means the method prototype is inherited from a different interface

### 6.7.3.3 INTERFACES POLIMORPHYSM

LogicLab can handle polimorphysm with interfaces, create a variable of type interface and valorize it usign the instance of a class that implements the method prototypes of that interface; now you can use the interface variable to call methods, LogicLab will be able to recognize which is the correct method to call, even if there may be more classes that implement the same interface.

Also, using the query interface operator `IMOVE ( ?= )`, you can use a variable to an interface to valorize a variable to a second interface which is extended by the first one; LogicLab will be able to analyze the object behind the source interface, and will use the correct method to valorize the destination interface variable.



	Name	Type	Address
1	myClass_1	fb_class_1	Auto
2	v1	INT	Auto
3	res	DINT	Auto
4	i1	intf_1	Auto
5	myClass_2	fb_class_2	Auto
6	i2	intf_2	Auto

```

0001
0002
0003     i1 := myClass_2;
0004
0005     res := i1.method_201(v1);
0006
0007     i2 ?= i1;
0008
0009     res := i2.method_201(v1);
0010
0011
    
```

In the baove example: the interface `intf_1` extends the interface `intf_2`; the interface `intf_1` is implemented by both `fb_class_1` and `fb_class_2`.

`i1`, variable of type `intf_1`, is valorized using `myClass_2`, an instance of `fb_class_2`, which is a Function Block that implements `intf_1`; now calling `i1.method_201` will correctly call `method_201` of `fb_class_2`.

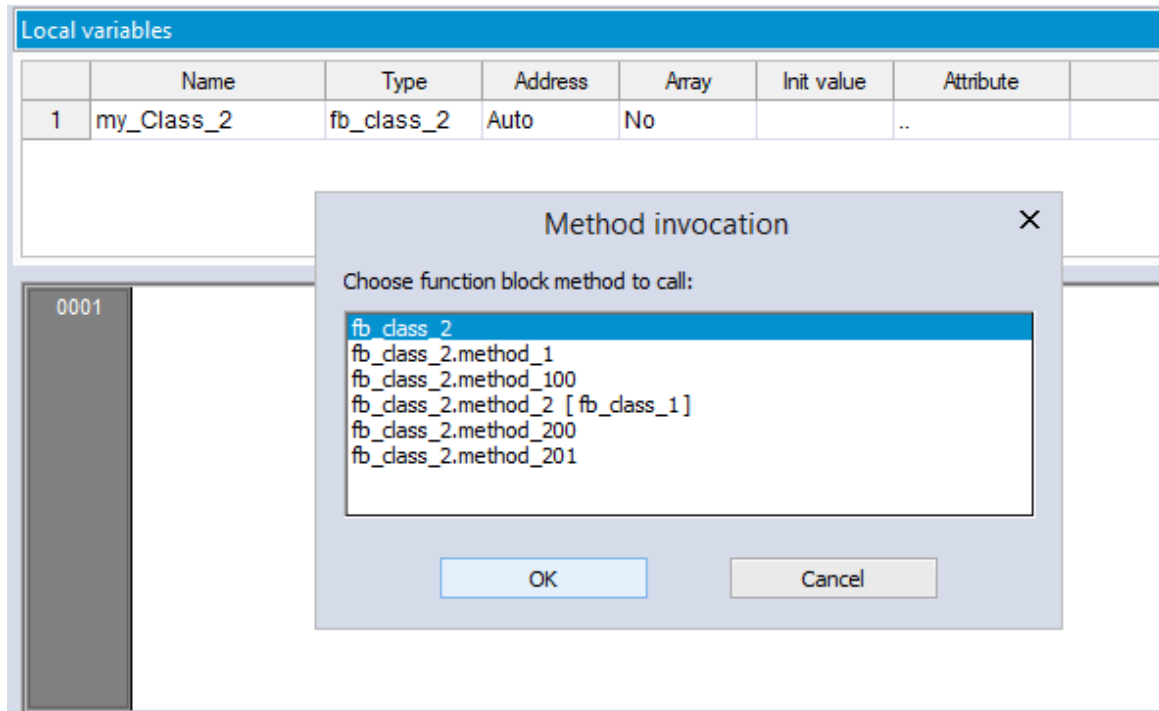
`i2`, variable of type `intf_2`, which is an interface extended by `intf_1`, is valorized with `i1` using the `IMOVE` operator; thanks to this operator there's no error even if `i1` and `i2` are not of the same type (because LogicLab understand that `i1` extends `i2`) and `i2` is valorized with a reference to the instance of `myClass_2`; so if around your project you call `i2.method201`, you will be calling `method_201` of `fb_class_2`.



### 6.7.4 OBJECT ORIENTED IN GRAPHIC LANGUAGES

In graphic languages, in case the object oriented programming is enabled, LogicLab will prompt a dialog to ask the user which element he wish to select in case of ambiguity.

For example: if you have a function block with methods and you drag it into an FBD program source editor, LogicLab will ask you if you want to call the body of the function block or a specific method.



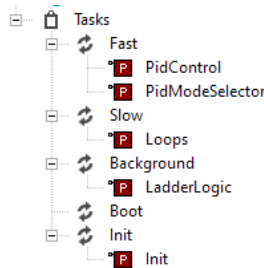


## 7. COMPILING

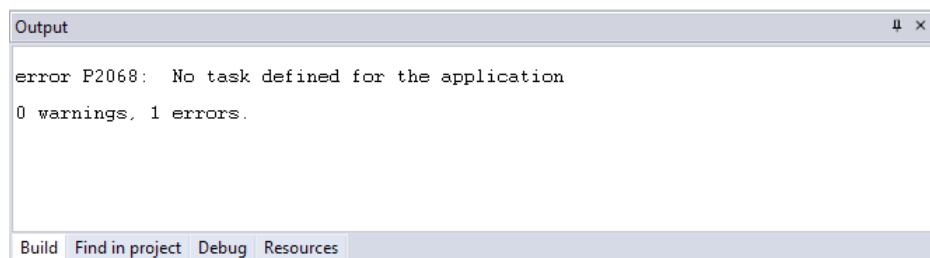
Compilation consists of taking the PLC source code and automatically translating it into binary code, which can be executed by the processor on the target device.

### 7.1 COMPILING THE PROJECT

Before starting actual compilation, make sure that at least one program has been assigned to a task.



When this pre-condition does not hold, compilation aborts with a meaningful error message.

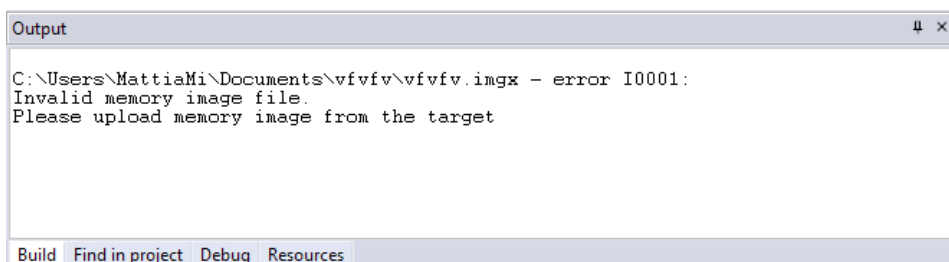


In order to start compilation, click **Project > Compile**.

Note that LogicLab automatically saves all changes to the project before starting the compilation.

#### 7.1.1 IMAGE FILE LOADING

Before performing the actual compilation, the compiler needs to load the image file (*.imgx file*), which contains the memory map of the target device. If the target is connected when compilation is started, the compiler retrieve the image file directly from the target. Otherwise, it loads the local copy of the image file from the working folder. If the target device is disconnected and there is no local copy of the image file, compilation cannot be carried out: you are then required to connect to a working target device or to refresh your working folder, loading the image file from the installed catalog (select **Project > Refresh current target**).





## 7.2 COMPILER OUTPUT

If the previous step was accomplished, the compiler performs the actual compilation, then prints a report in the *Output* window. The last string of the report has the following format:

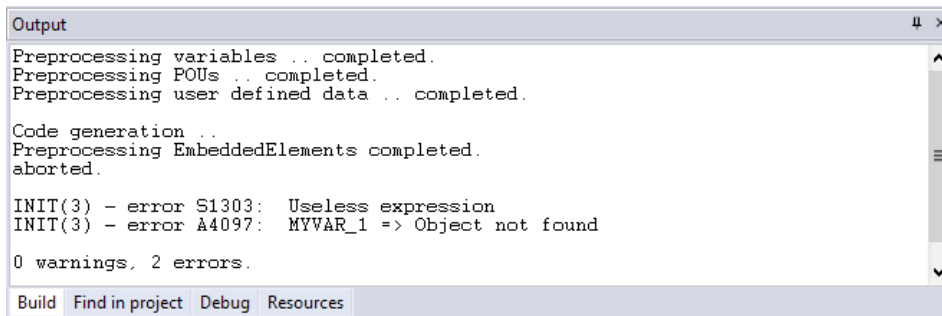
```
m warnings, n errors
```

It tells the user the outcome of compilation.

Condition	Description
n>0	Compiler error(s). The PLC code contains one or more serious errors, which cannot be worked around by the compiler; the binary output file is not generated, the project cannot be downloaded.
m>0	Emission of warning(s). The PLC code contains one or more minor errors, which the compiler automatically spotted and worked around. However, you are informed that the PLC program may act in a different way from what you expected: you are encouraged to get rid of these warnings by editing and re-compiling the application until no warning messages are emitted.  Even if you do not correct the warnings, the binary output file is generated and you can download the project.
n=m=0	PLC code entirely correct, compilation accomplished. You should always work with 0 warnings, 0 errors.  The binary output file is generated and you can download the project.

### 7.2.1 COMPILER ERRORS

When your application contains one or more errors, some useful information is printed in the *Output* window for each of those errors.



```
Output
Preprocessing variables .. completed.
Preprocessing POUs .. completed.
Preprocessing user defined data .. completed.

Code generation ..
Preprocessing EmbeddedElements completed.
aborted.

INIT(3) - error S1303: Useless expression
INIT(3) - error A4097: MYVAR_1 => Object not found

0 warnings, 2 errors.
```

As you can see, the information includes:

- the name of the Program Organization Unit affected by the error;
- the number of the source code line which procured the error;
- whether it is a fatal error (*error*) or one that the compiler could work around (*warning*);
- the error code;
- the error description.

Refer to the appropriate section for the compiler error reference.

If you double-click the error message in the *Output* bar, LogicLab opens the source code and highlights the line containing the error.

You can then fix the problem and re-compile.

## 7.3 COMMAND-LINE COMPILER

LogicLab's compiler can be used independently from the IDE: in LogicLab's directory, you can find an executable file, named *LLC.exe*, which is a command-line compiler that can be invoked (for example, in a batch file) with a number of options.

In order to get information about the syntax and the options of this command-line tool, just launch the executable without parameters.

```

Prompt dei comandi
flags: compiler options
Compiler options:
/D          - Download compiled project
            (if not already compiled will compile it)
/C          - Compile the project (trying to connect)
/c         - Compile the project without connecting
/A         - Rebuild and download the project
/GC[:<file>] - Generate C headers in the destination file.
            (default 'Default.h')
/GT[:<file>] - Generate the target variable track file.
            (default 'Default.osc')
/GG[:<file>] - Generate the global variable track file.
            (default 'Default.osc')
/F:<comm>   - Force the communication properties
/T:<target> - Force the target board type
            (used to download code without opening the project)
/DR        - Perform download acknowledge request
/R         - Rebuild the project (trying to connect)
/r        - Rebuild the project without connecting
/FU        - Perform download without checking for updated source s
tatus
/Q         - Append the output to the destination file.
            (valid only with /GT e /GG flags)
/M[:file[,pwd]] - Generate redistributable source file.
/cfg:<file> - Use additional configuration file
/tgsx:<file> - Generate TGSX target definition file for simulator
/NORELOADPLC - Do not reload PLC after download
/U[:file[,pwd]] - Decode RSM file and save as indicated in file paramete
r
/x         - If specified llc does not upload image file for target
            or get download address from target
/WRU       - Verify target identity
/uP:<procname> - Force target processor
            (used in combination with /WRU and /T to verify target
identity without opening the project)
/b[:path[,name]] - Generate target binaries (bin, tsc, tds) files.
                Optional absolute or relative path and name can be spe
cified (Default: relative folder "Download", name: project name)
/SAVEAS:<file> - Save project with different folder and name, changing
project format if necessary
/PERC      - Print progress percentage while downloading.

```





## 8. LAUNCHING THE APPLICATION

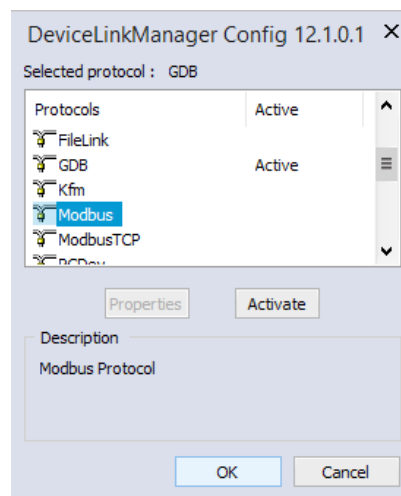
In order to download and debug the application, you have to establish a connection with the target device. This chapter focuses on the operations required to connect to the target and to download the application, while the wide range of LogicLab's debugging tools deserves a separate chapter (see Chapter 9).

### 8.1 SETTING UP THE COMMUNICATION

In order to establish the connection with the target device, make sure the physical link is up (all the cables are plugged in, the network is properly configured, and so on).

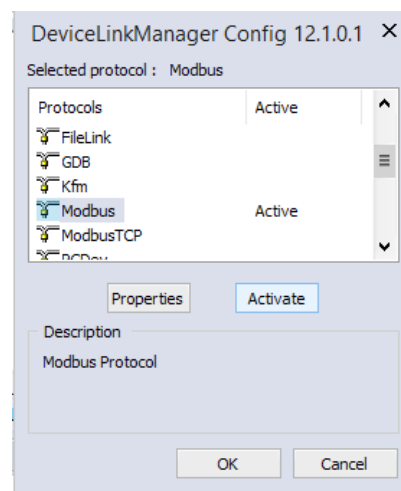
Follow this procedure to set up and establish the connection to the target device:

- 1) Click **On-line>Set up communication...** menu of the LogicLab main window. This causes the following dialog box to appear.

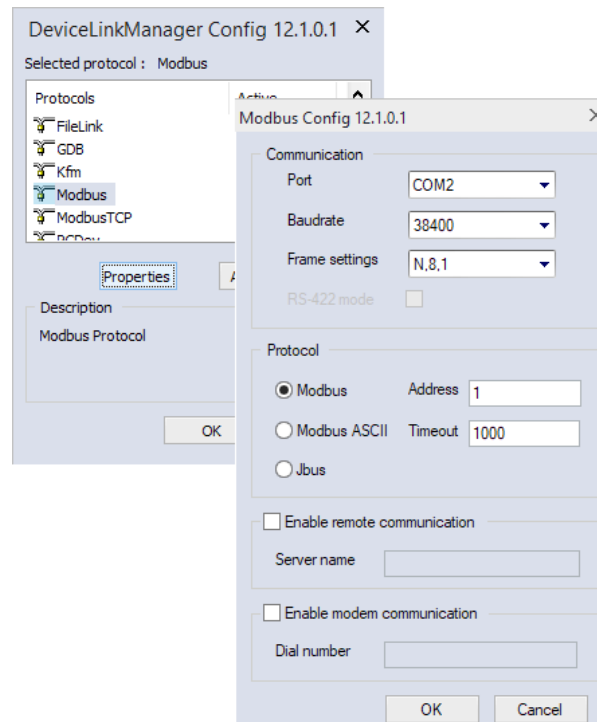


The elements in the list of communication protocols you can select from depend on the setup executable(s) you have run on your PC (refer to your hardware provider if a protocol you expect to appear in the list is missing).

- 2) Choose the appropriate protocol and make it the active protocol by clicking on the *activate* button.



- 3) Now, by clicking on the *Properties* button, you can fill in all the protocol-specific settings (for example, the target address and baudrate or the communication timeout - that is how long LogicLab must wait for an answer from the target before displaying a communication error message).



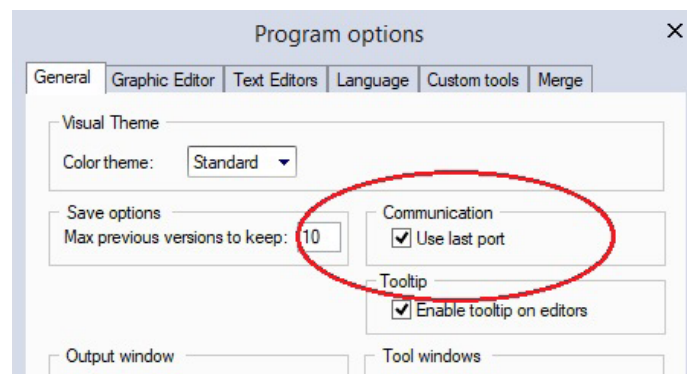
- 4) Apply the changes you made to the communication settings by pressing the *OK* button; otherwise, if you press the *Cancel* button, all changes will be undone.

Now you can establish a communication by clicking [On-line>Connect](#) menu.

### 8.1.1 SAVING THE LAST USED COMMUNICATION PORT

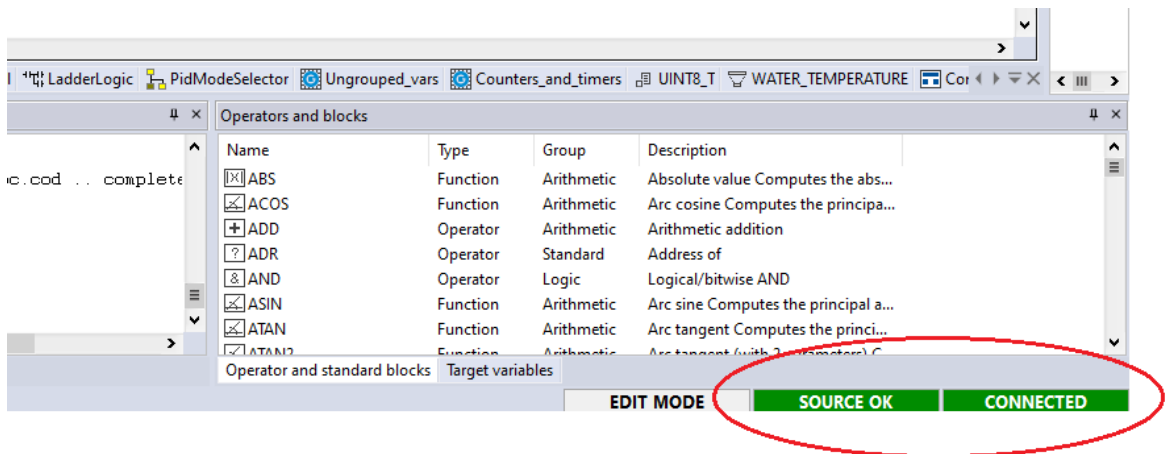
Given the fact that many modern PCs have only one COM port, you will usually use the same port when connecting to target devices using a serial protocol; LogicLab allows you to save the last used COM port and configuration, in order to override the project setting procedure. This feature proves especially useful when you share projects with other developers, which may use a different COM port to connect to the target device.

In order to save your COM port settings, enable the *Use last port* option in [File>Options...](#) menu.



## 8.2 ON-LINE STATUS

At the bottom-right of LogicLab window, next to the right border of the *Status Bar*, there are other two boxes: the first is the *application status*, the second is the *connection status*.



### 8.2.1 APPLICATION STATUS

The application status box gives information about the status of the application currently executing on the target device; such information are available only when you're connected with the target device.

The application can have on of the following status:

- No code: no application is executing on the target device.

**NO CODE**

- Diff. code: the application currently executing on the target device is not the same as the one currently open in the IDE; moreover, no debug information consistent with the running application is available: thus, the values shown in the watch window or in the oscilloscope are not reliable and the debug mode cannot be activated.

**DIFF. CODE**

- Diff. code, Symbols OK: the application currently executing on the target device is not the same as the one currently open in the IDE; however, some debug information consistent with the running application is available (for example, because that application has been previously downloaded to the target device from the same PC): the values shown in the watch window or in the oscilloscope are reliable, but the debug mode still cannot be activated.

**DIFF. CODE (SYM)**

- Source OK: the application currently executing on the target device is the same as the one currently open in the IDE: the debug mode can be activated.

**SOURCE OK**

### 8.2.2 CONNECTION STATUS

The communication status gives you information about the status of the actual communication with the target device.

If you have not yet attempted to connect to the target, the status of communication is



set to *Not connected*.

**NOT CONNECTED**

When you try to connect to the target device, the state of communication becomes one of the following:

- **Error**: the communication cannot be established. You should check both the physical link and the communication settings.

**ERROR**

- **Connected**: the communication has been correctly established.

**CONNECTED**

### 8.3 DOWNLOADING THE APPLICATION

A compiled PLC application must be downloaded to the target device in order to have the processor execute it. This paragraph shows you how to send a PLC code to a target device. Note that LogicLab can download the code to the target device only if the latter is connected to the PC where LogicLab is running. See the related section for details.

To download the application, click **On-line>Download code**.

LogicLab checks whether the project has unsaved changes. If this is the case, it automatically starts the compilation of the application. The binary code is eventually sent to the target device, which then undergoes automatic reset at the end of transmission. Now the code you sent is actually executed by the processor on the target device.

#### 8.3.1 CONTROLLING SOURCE CODE DOWNLOAD

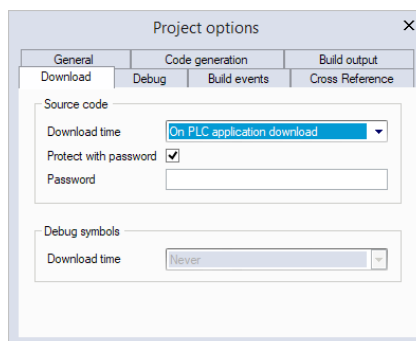
Whether the source code of the application is downloaded along with the binary code or not, depends on the target device you are interfacing with: some devices host the application source code in their storage, in order to allow the developer to upload the project in a later moment.

If this is the case, you can control some aspects of the source code download process, as explained in the following paragraphs.

##### 8.3.1.1 PROTECTING THE SOURCE CODE WITH A PASSWORD

You may want to protect the source code downloaded to the target device with a password, so that LogicLab will not open the uploaded project unless the correct password is entered.

Click the **Project>Options...** menu and set the password.



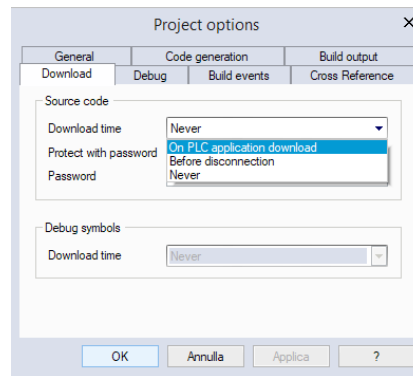
You may opt to disable the password, instead.

### 8.3.1.2 SOURCE CODE AND DEBUG SYMBOLS DOWNLOAD TIME

As stated before, LogicLab allows you to download also the source code on the target device; this way you can retrieve the source code in a later time having the certainty to work with the exact project that is running on the target device.

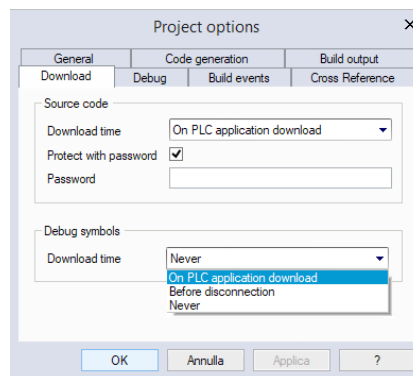
The availability of this feature depends on the device implementation.

LogicLab allows you to choose when the download of the source code must be done; from the menu **Project > Options > Download** you can choose one of the following voices:



- *On PLC application download*: the Source code will be downloaded to the target together with PLC application.
- *Before disconnection*: the Source code will be downloaded before target disconnection.
- *Never*: the Source code will be never downloaded to the target.

As well as the Source code, also the Debug symbols download time can be set using the following select menu with the same options.



## 8.4 SIMULATION

Depending on the target device you are interfacing with, you may be able to simulate the execution of the PLC application with LogicLab's integrated simulation environment: LogicLab.

In order to start the simulation, just click **Debug > Simulation mode**.

Refer to LogicLab's manual to gain information on how to control the simulation.

## 8.5 CONTROL THE PLC EXECUTION

The PLC application execution can be controlled using the related functions in the project bar or by the command presents in the On-line menu.





### 8.5.1 HALT

You can stop the PLC execution by clicking **On-line>Halt**.

### 8.5.2 COLD RESTART

The PLC application execution will be restarted and both retain and non-retain variables will be reset.

You can cold restart the PLC execution by clicking **On-line>Cold restart**.

### 8.5.3 WARM RESTART

The PLC application execution will be restarted and only non-retain variables will be reset.

You can warm restart the PLC execution by clicking **On-line>Warm restart**.

### 8.5.4 HOT RESTART

The PLC application execution will be restarted and no variables will be reset.

You can hot restart the PLC execution by clicking **On-line>Hot restart**.

### 8.5.5 REBOOT TARGET

You can reboot the target by clicking **On-line>Reboot target**.

## 9. DEBUGGING

LogicLab provides several debugging tools, which help the developer to check whether the application behaves as expected or not.

All these debugging tools basically allow the developer to watch the value of selected variables while the PLC application is running.

LogicLab debugging tools can be gathered in two classes:

- Asynchronous debuggers. They read the values of the variables selected by the developer with successive queries issued to the target device. Both the manager of the debugging tool (that runs on the PC) and, potentially, the task which is responsible to answer those queries (on the target device) run independently from the PLC application. Thus, there is no guarantee about the values of two distinct variables being sampled in the same moment, with respect to the PLC application execution (one or more cycles may have occurred); for the same reason, the evolution of the value of a single variable is not reliable, especially when it changes fast.
- Synchronous debuggers. They require the definition of a trigger in the PLC code. They refresh simultaneously all the variables they have been assigned every time the processor reaches the trigger, as no further instruction can be executed until the value of all the variables is refreshed. As a result, synchronous debuggers obviate the limitations affecting asynchronous ones.

This chapter shows you how to debug your application using both asynchronous and synchronous tools.

### 9.1 WATCH WINDOW

The *Watch* window allows you to monitor the current values of a set of variables. Being an asynchronous tool, the *Watch* window does not guarantee synchronization of values. Therefore, when reading the values of the variables in the *Watch* window, be aware of the possibility that they may refer to different execution cycles of the corresponding task.

The *Watch* window contains an item for each variable that you added to it. The information shown in the *Watch* window includes the name of the variable, its value, its type, and its location in the PLC application.

Symbol	Value	Type	Location	Descript
▲ X	67.439	REAL	@SLOW:LOOPS	
▼ Y	-0.993	REAL	@SLOW:LOOPS	
— INCR	0.010	REAL	@SLOW:LOOPS	
— I	8	UINT	@SLOW:LOOPS	
■ BIT	TRUE	BOOL	@SLOW:LOOPS	
[-] [ ] PRESSED	-	BOOL[]	@SLOW:LOOPS	
[0]	FALSE	BOOL	@SLOW:LOOPS	
[1]	FALSE	BOOL	@SLOW:LOOPS	
[2]	FALSE	BOOL	@SLOW:LOOPS	
[3]	FALSE	BOOL	@SLOW:LOOPS	
[4]	FALSE	BOOL	@SLOW:LOOPS	
[5]	FALSE	BOOL	@SLOW:LOOPS	
[6]	FALSE	BOOL	@SLOW:LOOPS	



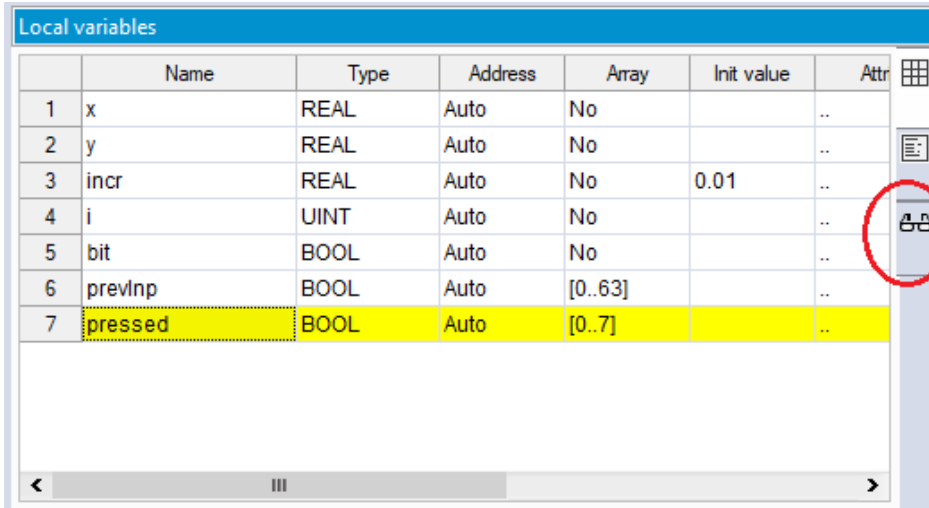
### 9.1.1 OPENING AND CLOSING THE WATCH WINDOW

To open, close the *Watch* window, click `View>Tool windows>Watch`.

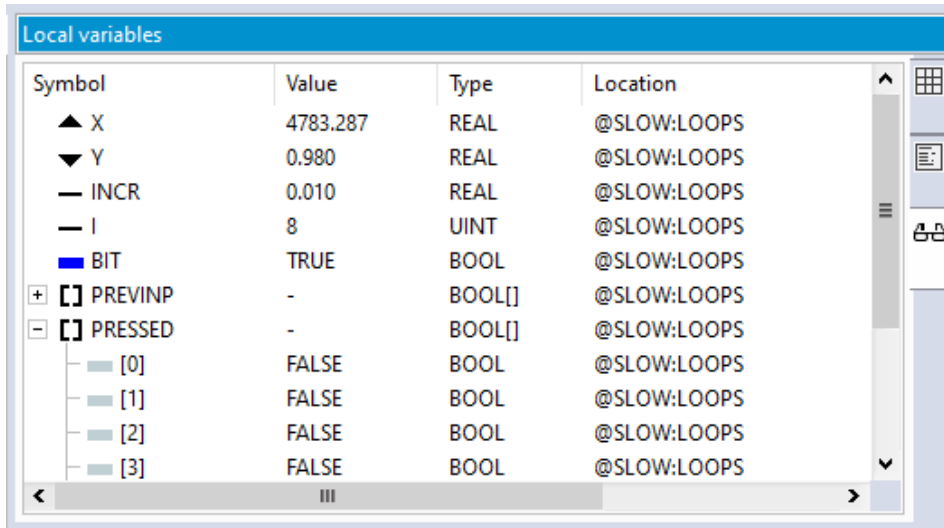
Closing the *Watch* window means simply hiding it, not resetting it. As a matter of fact, if you close the *Watch* window and then open it again, you will see that it still contains all the variables you added to it.

#### 9.1.1.1 WATCH VARIABLES EDITOR

A particular implementation of the watch window is available from the variables editor (both global and local); on the right side of the variables editor you'll see three tabs:



The third one, when pressed, will turn the variables editor into a watch window, where all the variables of the editor are already added to the list and their values are shown:



No other variables can be added to this list and the variables in this list cannot be removed.

### 9.1.2 ADDING ITEMS TO THE WATCH WINDOW

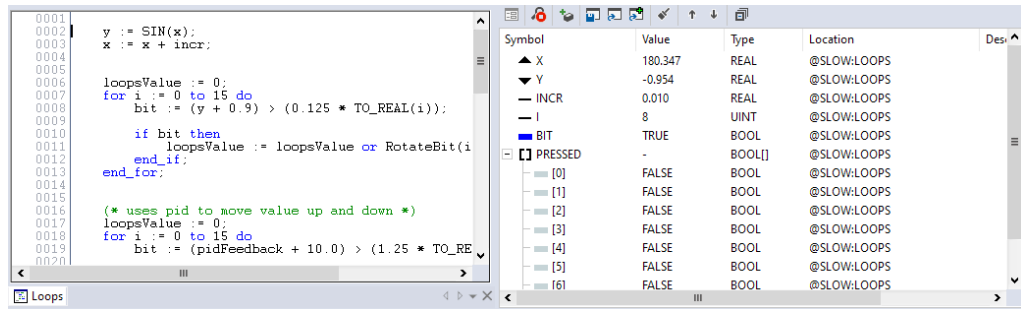
To watch a variable, you need to add it to the watch list.

Note that, unlike trigger windows and the *Graphic trigger* window, you can add to the *Watch* window all the variables of the project, regardless of where they were declared.



### 9.1.2.1 ADDING A VARIABLE FROM A TEXTUAL SOURCE CODE EDITOR

Follow this procedure to add a variable to the *Watch* window from a textual (that is, IL or ST) source code editor: select a variable, by double-clicking on it, and then drag it into the watch window.

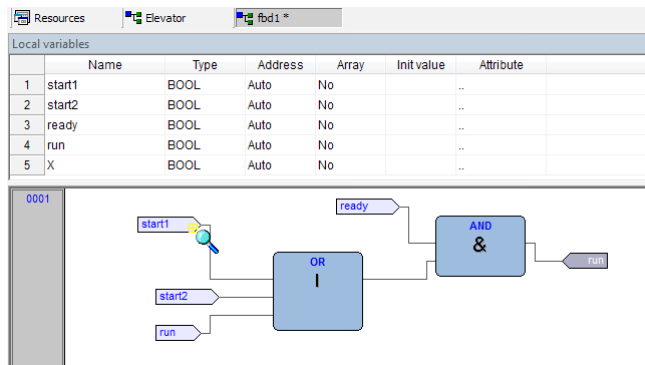


The same procedure applies to all the variables you wish to inspect.

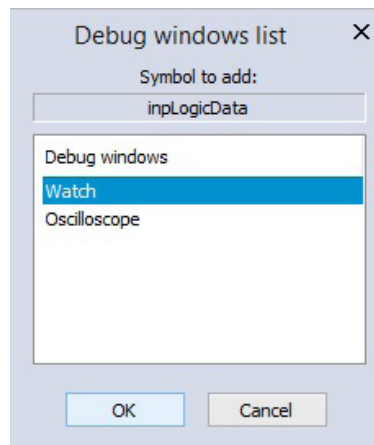
### 9.1.2.2 ADDING A VARIABLE FROM A GRAPHICAL SOURCE CODE EDITOR

Follow this procedure to add a variable to the *Watch* window from a graphical (that is, LD, FBD, or SFC) source code editor:

- 1) Click **Edit>Watch mode**.
- 2) Click on the block representing the variable you wish to be shown in the *Watch* window.



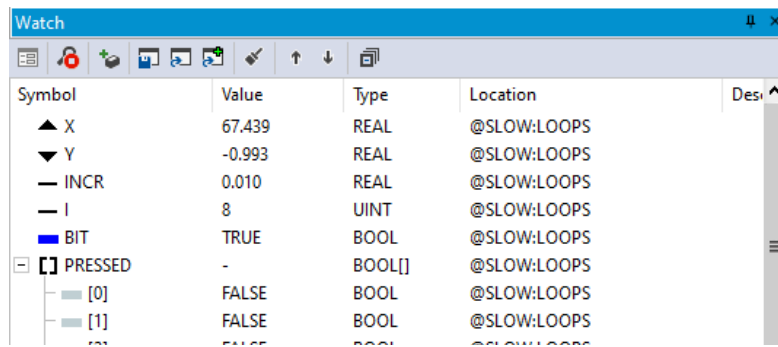
A dialog box appears listing all the currently existing instances of debug windows, and asking you which one is to receive the object you have just clicked on.



In order to display the variable in the *Watch* window, select *Watch*, then press *OK*. The variable name, value, and location are now displayed in a new row of the *Watch* window.



dow.

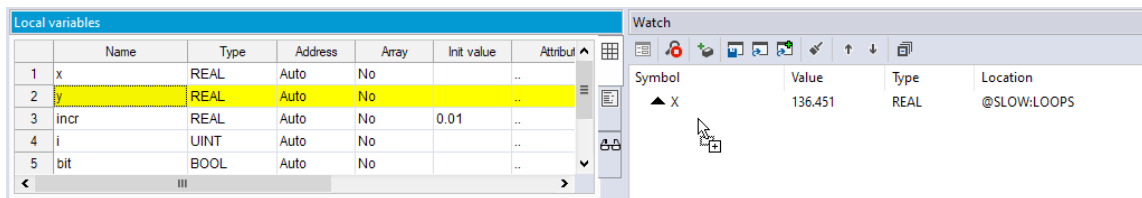


The same procedure applies to all the variables you wish to inspect.

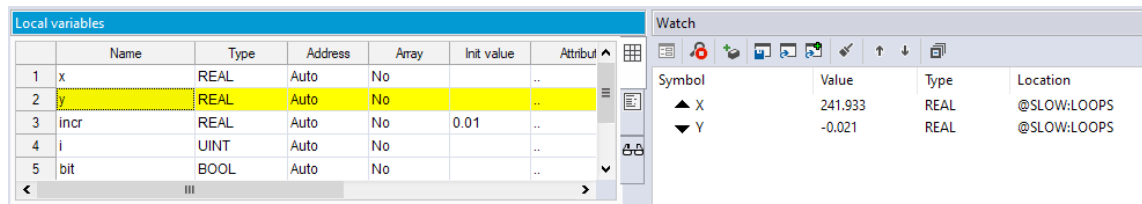
Once you have added to the *Watch* window all the variables you want to observe, you should click **Edit>Insert/Move mode**: the mouse cursor turns to its original shape.

### 9.1.2.3 ADDING A VARIABLE FROM A VARIABLES EDITOR

In order to add a variable to the *Watch* window, you can select the corresponding record in the variables editor and then either drag-and-drop it in the *Watch* window

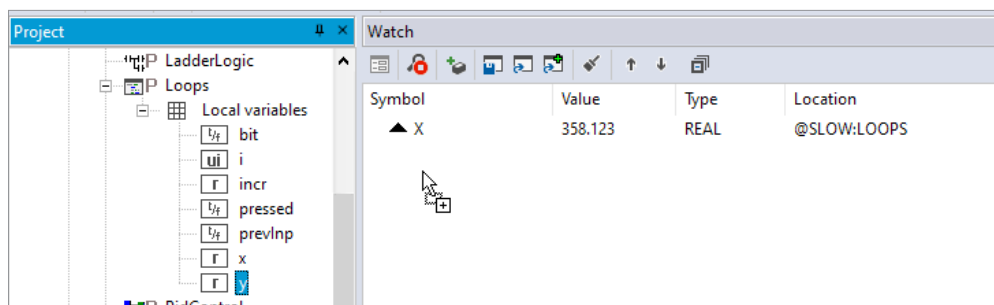


or press the *F8* key.



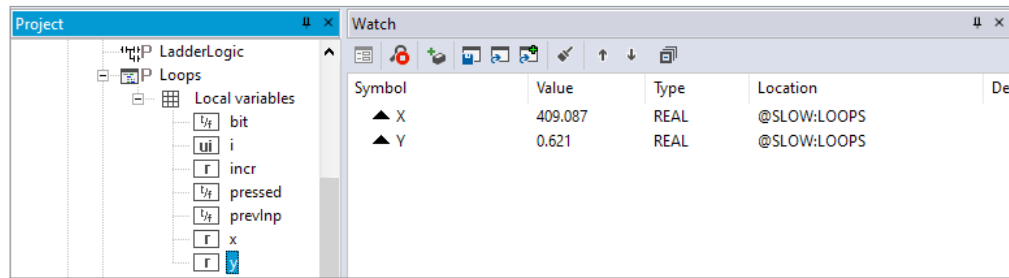
### 9.1.2.4 ADDING A VARIABLE FROM THE PROJECT TREE

In order to add a variable to the *Watch* window, you can select it in the project tree and then either drag-and-drop it in the *Watch* window



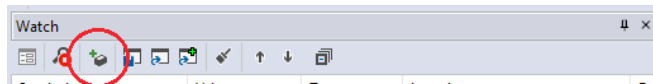
or press the *F8* key.



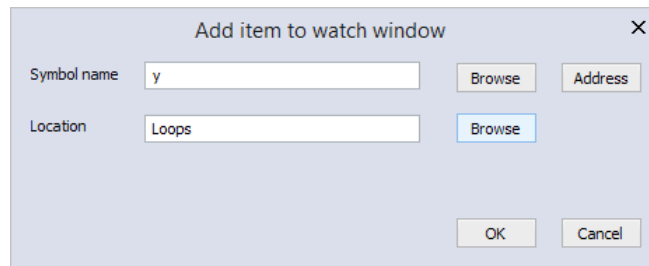


### 9.1.2.5 ADDING A VARIABLE FROM THE WATCH WINDOW TOOLBAR

You can also click on the appropriate item of the Watch window inner toolbar, in order to add a variable to it.



You shall type (or select by browsing the project symbols) the name of the variable and its location (where it has been declared).



### 9.1.3 REMOVING A VARIABLE

If you want a variable not to be displayed any more in the *Watch* window, select it by clicking on its name once, then press the *Del* key.

If you want to remove all item from the *Watch* window, select the following icon:



### 9.1.4 REFRESHMENT OF VALUES

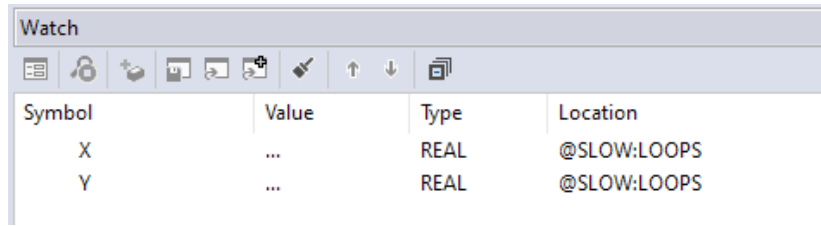
#### 9.1.4.1 NORMAL OPERATION

The watch window manager reads periodically from memory the value of the variables. However, this action is carried out asynchronously, so it may happen that a higher-priority task modifies the value of some of the variables while they are being read. Thus, at the end of a refreshment process, the values displayed in the window may refer to different execution states of the PLC code.

#### 9.1.4.2 TARGET DISCONNECTED

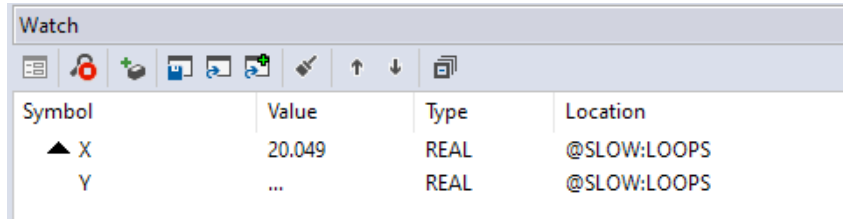
If the target device is disconnected, the *Value* column contains three dots.





**9.1.4.3 OBJECT NOT FOUND**

If the PLC code changes and LogicLab cannot retrieve the memory location of an object in the *Watch* window, then the *Value* column contains three dots.



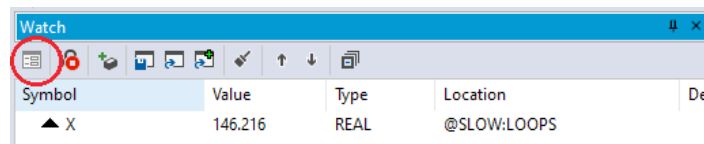
LogicLab does not allow you to add to the *Watch* window a symbol which has not been allocated, any attempt will be ignored.

**9.1.5 CHANGING THE FORMAT OF DATA**

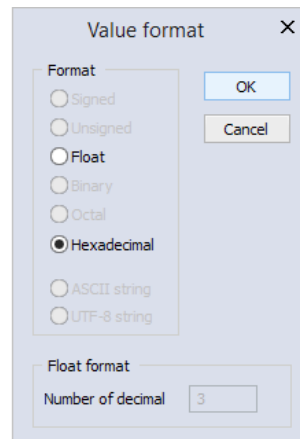
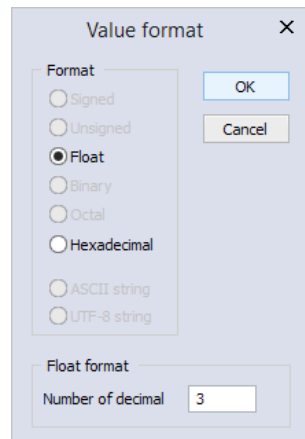
When you add a variable to the *Watch* window, LogicLab automatically recognizes its type (unsigned integer, signed integer, floating point, hexadecimal...), and displays its value consistently. Also, if the variable is floating point, LogicLab assigns it a default number of decimal figures.

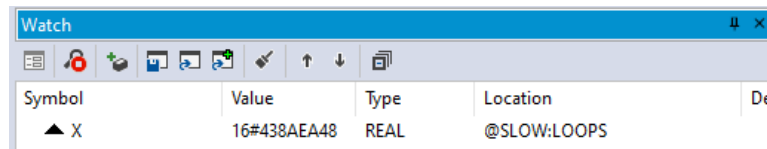
However, you may need the variable to be printed in a different format.

To impose another format than the one assigned by LogicLab, press the *Format value* button in the toolbar.



Choose the format and confirm your choice.



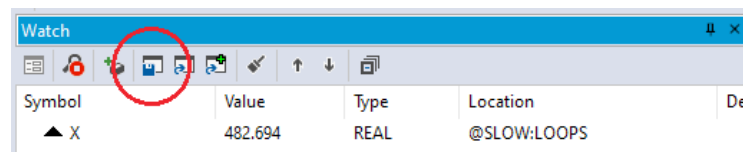


## 9.1.6 WORKING WITH WATCH LISTS

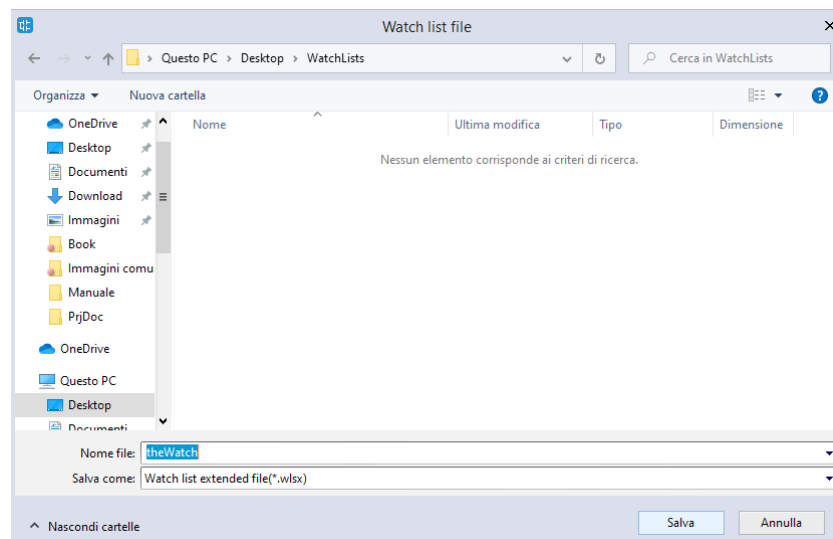
You can store to file the set of all the items in the *Watch* window, in order to easily restore the status of this debugging tools in a successive working session.

Follow this procedure to save a watch list:

- 1) Click on the corresponding item in the *Watch window* toolbar.

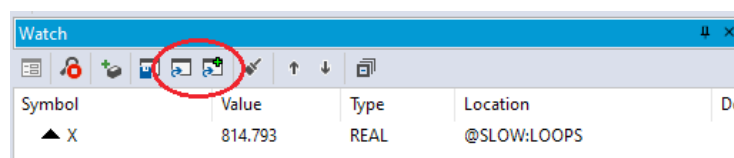


- 2) Enter the file name and choose its destination in the file system.



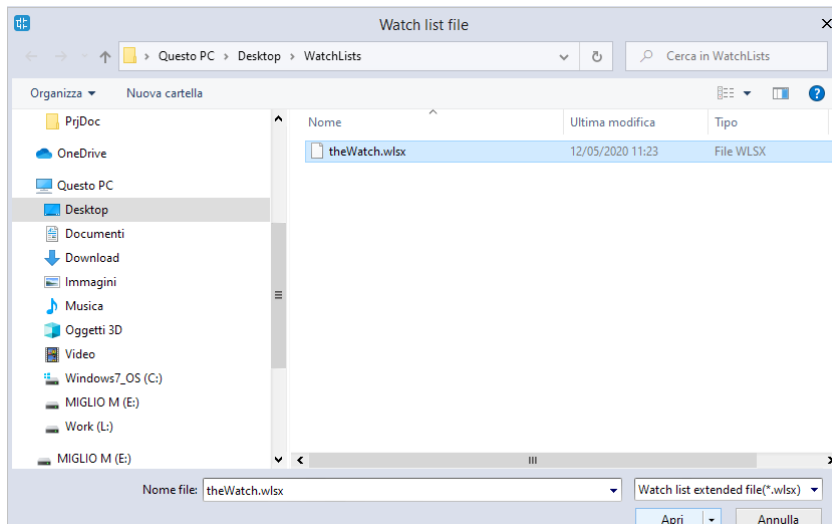
When loading a watch list from file, you have two options: you can load the items from file and append them to the items currently shown in the watch window; or you can automatically clear the watch window and display only the items loaded from file. Either way, follow this procedure and choose the desired option:

- 1) Click on the corresponding icon in the *Watch* window toolbar. The one labeled *Load (no appends) watch list* will remove all currently displayed elements and show only the ones loaded from file; the icon labeled *Load watch list* will append them instead.



- 2) Browse the file system and select the watch list file.





The set of symbols in the watch list is added to the *Watch* window.

### 9.1.7 AUTOSAVE WATCH LIST

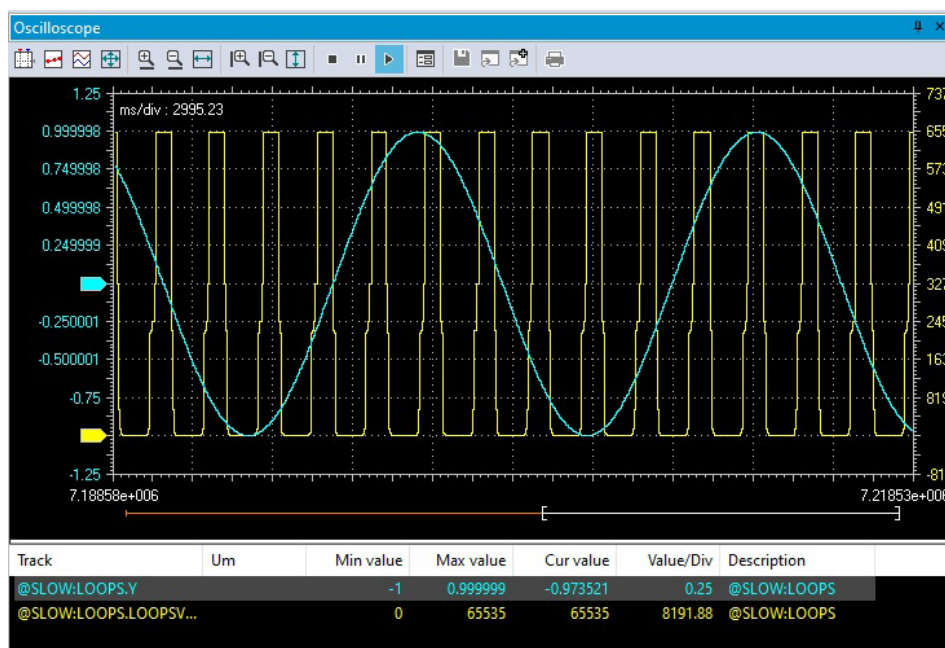
By selecting the associated option in the project options dialog (see Paragraph 4.6.5 for more info) the watch list will be automatically saved on the project closing.

The saved watch list will be automatically loaded (with no append option) on the first connection to target when the project will be re-opened.

## 9.2 OSCILLOSCOPE

The Oscilloscope allows you to plot the evolution of the values of a set of variables. Being an asynchronous tool, the Oscilloscope cannot guarantee synchronization of samples.

Opening the Oscilloscope causes a new window to appear next to the right-hand border of the LogicLab frame. This is the interface for accessing the debugging functions that the Oscilloscope makes available. The Oscilloscope consists of three elements, as shown in the following picture.



The toolbar allows you to better control the Oscilloscope. A detailed description of the



function of each control is given later in this chapter.

The Chart area includes several items:

- Plot: area containing the curve of the variables.
- Vertical cursors: cursors identifying two distinct vertical lines. The values of each variable at the intersection with these lines are reported in the corresponding columns.
- Scroll bar: if the scale of the x-axis is too large to display all the samples in the Plot area, the scroll bar allows you to slide back and forth along the horizontal axis.

The lower section of the Oscilloscope is a table consisting of a row for each variable.

## 9.2.1 OPENING AND CLOSING THE OSCILLOSCOPE

To open, close the Oscilloscope, click `View>Tool windows>Oscilloscope`.

Closing the Oscilloscope means simply hiding it, not resetting it. As a matter of fact, if you open again the Oscilloscope after closing it, you will see that plotting of the curve of all the variables you added to it starts again.

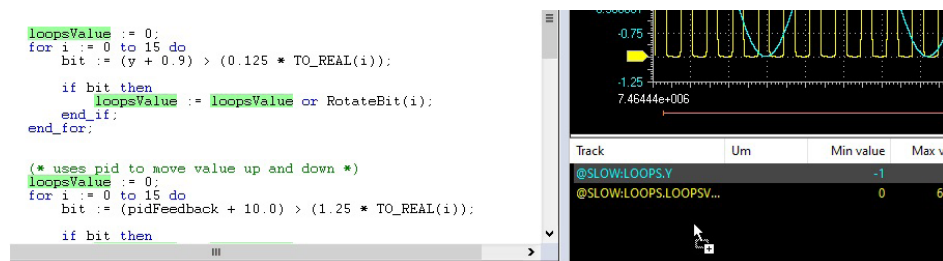
## 9.2.2 ADDING ITEMS TO THE OSCILLOSCOPE

In order to plot the evolution of the value of a variable, you need to add it to the Oscilloscope.

Note that unlike trigger windows and the *Graphic trigger* window, you can add to the Oscilloscope all the variables of the project, regardless of where they were declared.

### 9.2.2.1 ADDING A VARIABLE FROM A TEXTUAL SOURCE CODE EDITOR

Follow this procedure to add a variable to the Oscilloscope from a textual (that is, IL or ST) source code editor: select a variable by double-clicking on it, and then drag it into the *Oscilloscope* window.

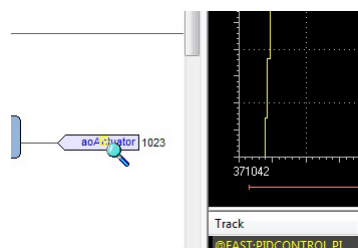


The same procedure applies to all the variables you wish to inspect.

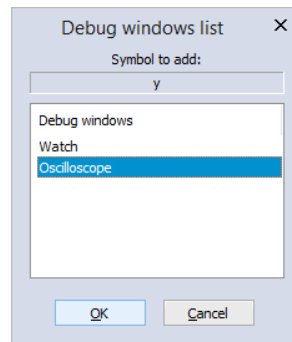
### 9.2.2.2 ADDING A VARIABLE FROM A GRAPHICAL SOURCE CODE EDITOR

Follow this procedure to add a variable to the Oscilloscope from a graphical (that is, LD, FBD, or SFC) source code editor:

- 1) Click `Edit>Watch mode`.
- 2) Click on the block representing the variable you wish to be shown in the Oscilloscope.



- 3) A dialog box appears listing all the currently existing instances of debug windows, and asking you which one is to receive the object you have just clicked on.



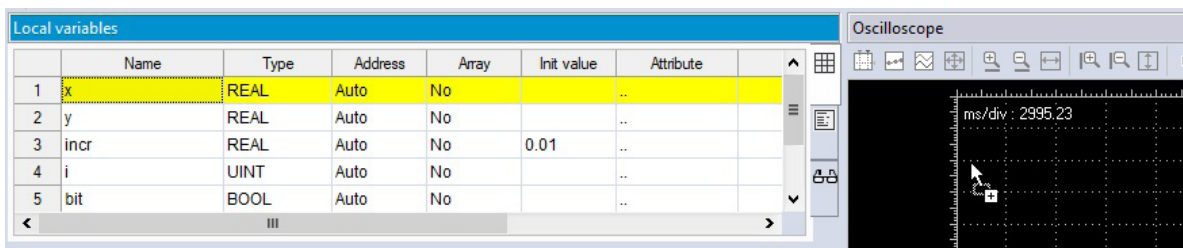
Select *Oscilloscope*, then press *OK*. The name of the variable is now displayed in the *Track* column.

The same procedure applies to all the variables you wish to inspect.

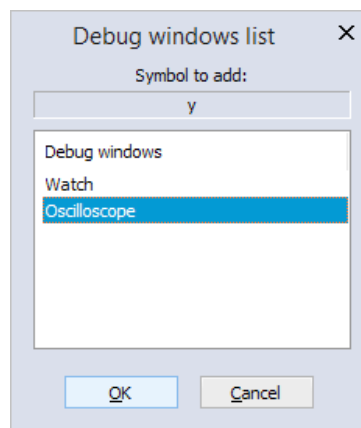
Once you have added to the Oscilloscope all the variables you want to observe, you should click **Edit>Insert/Move mode**: the mouse cursor turns to its original shape.

### 9.2.2.3 ADDING A VARIABLE FROM A VARIABLES EDITOR

In order to add a variable to the Oscilloscope, you can select the corresponding record in the variables editor and then either drag-and-drop it in the Oscilloscope

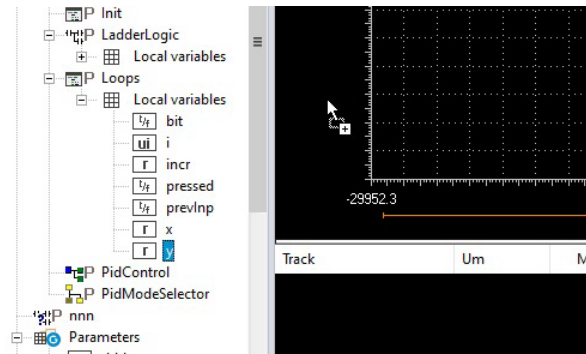


or press the *F10* key and choose *Oscilloscope* from the list of debug windows which pops up.

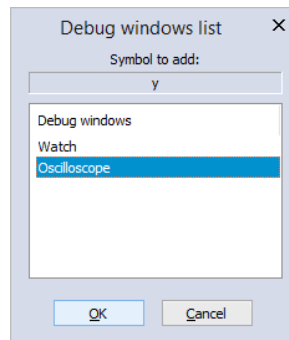


### 9.2.2.4 ADDING A VARIABLE FROM THE PROJECT TREE

In order to add a variable to the Oscilloscope, you can select it in the project tree and then either drag-and-drop it in the Oscilloscope



or press the *F10* key and choose *Oscilloscope* from the list of debug windows which pops up.



### 9.2.3 REMOVING A VARIABLE

If you want to remove a variable from the Oscilloscope, select it by clicking on its name once, then press the *Del* key.

### 9.2.4 VARIABLES SAMPLING

#### 9.2.4.1 NORMAL OPERATION

The Oscilloscope manager periodically reads from memory the value of the variables. However, this action is carried out asynchronously, that is it may happen that a higher-priority task modifies the value of some of the variables while they are being read. Thus, at the end of a sampling process, data associated with the same value of the x-axis may actually refer to different execution states of the PLC code.

#### 9.2.4.2 TARGET DISCONNECTED

If the target device is disconnected, the curves of the dragged-in variables get frozen, until communication is restored.

### 9.2.5 CONTROLLING DATA ACQUISITION AND DISPLAY

The Oscilloscope includes a toolbar with several commands, which can be used to control the acquisition process and the way data are displayed. This paragraph focuses on these commands.

Note that all the commands in the toolbar are disabled if no variable has been added to the Oscilloscope.

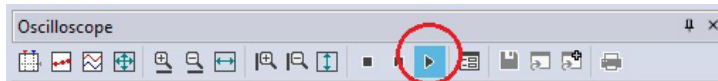


**9.2.5.1 STARTING AND STOPPING DATA ACQUISITION**

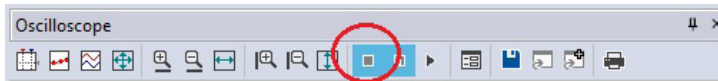
When you add a variable to the Oscilloscope, data acquisition begins immediately. However, you can suspend the acquisition by clicking on *Pause acquisition*.



The curve freezes (while the process of data acquisition is still running in background), until you click on *Restart acquisition*.



In order to stop the acquisition you may click on *Stop acquisition*.



In this case, when you click on *Restart acquisition*, the evolution of the value of the variable is plotted from scratch.

**9.2.5.2 SETTING THE SCALE OF THE AXES**

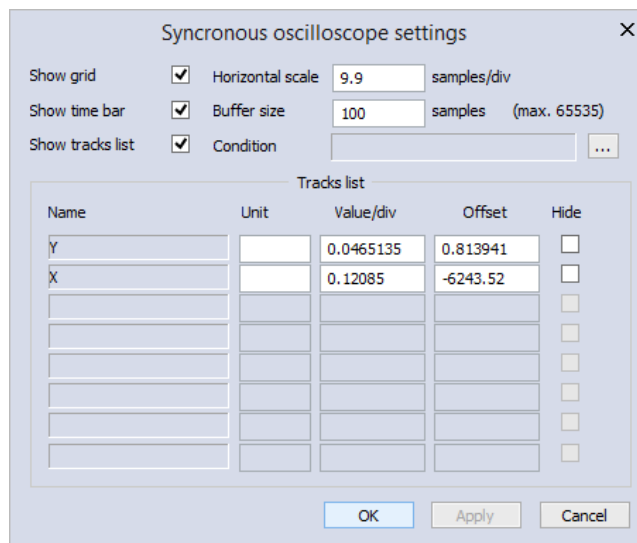
When you open the Oscilloscope, LogicLab applies a default scale to the axes. However, if you want to set a different scale, you can do that by opening the graph properties clicking on the corresponding icon in the toolbar:



The graph settings window will open, allowing you to change both the vertical and the horizontal axis.

The horizontal scale is the same for every track.

The vertical scale can be different for every variable, you can specify the desired scale in the edit box labeled *Value/div*.



### 9.2.5.3 CHANGE THE ZOOM

In the Oscilloscope toolbar you can also find icons for zoom in and out, and also to force the Oscilloscope to display all the sample.

These options are axis-related, so you can zoom in, zoom out and show all samples for the horizontal axis alone:



as well as you can zoom in, zoom out and show all samples for the vertical axis alone:

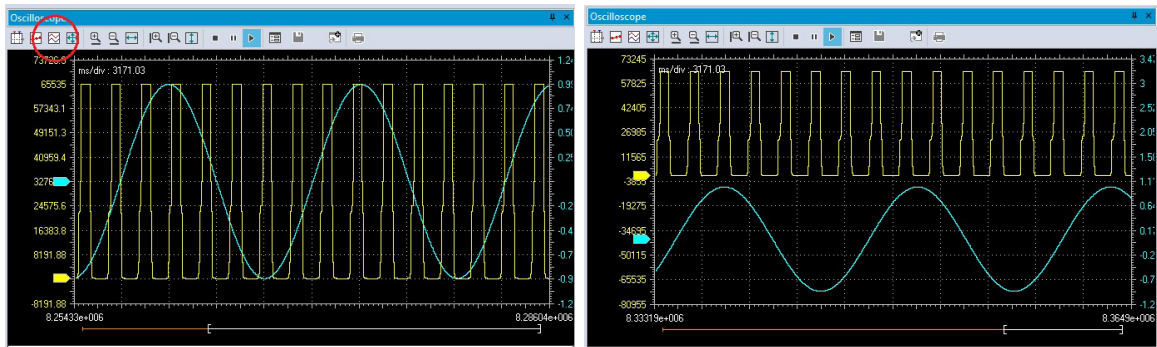


You can also quickly force the oscilloscope to display all samples for both horizontal and vertical axis (which means show all values) with a specific icon:

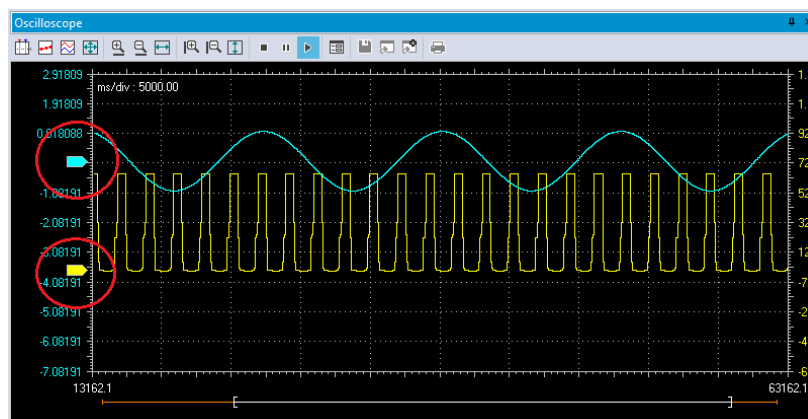


### 9.2.5.4 VERTICAL SPLIT

When you are watching the evolution of two or more variables, you may want to split the respective tracks. For this purpose, click on the *Vertical split* item in the Oscilloscope toolbar.

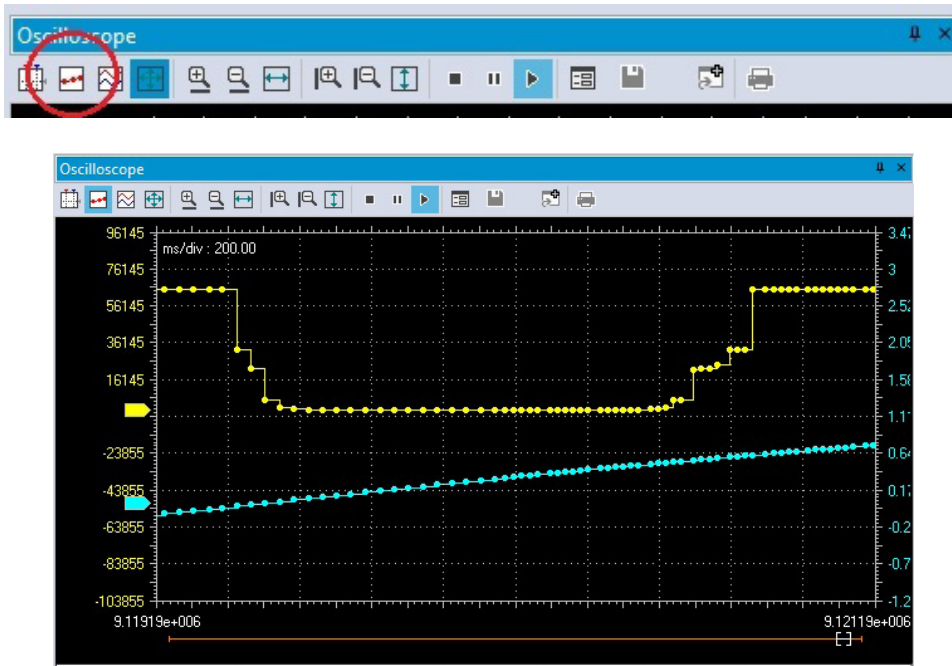


To separate traces you can also manually move them; to do so grab the corresponding coloured flag on the left of the chart and drag it to the desired location.



**9.2.5.5 VIEWING SAMPLES**

If you click on the *Show samples* item in the *Oscilloscope* toolbar, the tool highlights the single values detected during data acquisition.



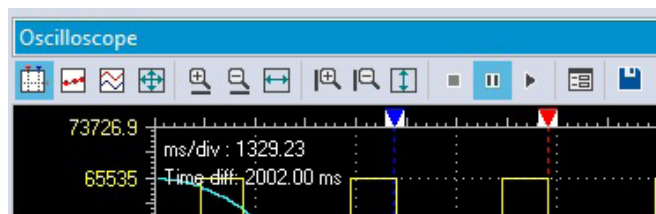
You can click on the same item again, in order to go back to the default view mode.

**9.2.5.6 TAKING MEASURES**

The Oscilloscope includes two measure bars, which can be exploited to take some measures on the chart; in order to show and hide them, click on the *Show measure bars* item in the Oscilloscope toolbar.



If you want to measure a time interval between two events, you just have to move one bar to the point in the graph that corresponds to the first event and the other to the point that corresponds to the second one.



The time interval between the two bars is shown in the top left corner of the chart.

You can use a measure bar also to read the value of all the variables in the Oscilloscope at a particular moment: move the bar to the point in the graph which corresponds to the instant you want to observe.

In the table below the chart, you can now read the values of all the variables at that particular moment.

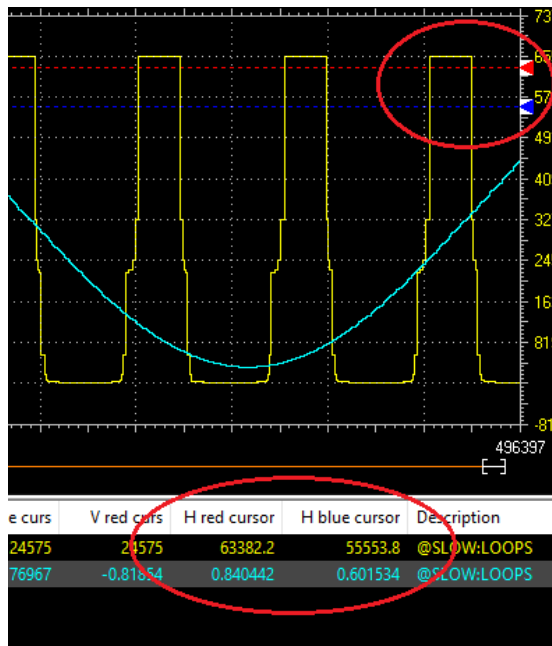




Track	Min value	Max value	Cur value	Value/Di	V blue curs	V red curs	H red cursor	H blue cur...	Descripti
@SLOW:LOOPS.LOOPSV...	0	65535	22015	8191.88	24575	24575	53247.2	12287.8	@SLOW:L
@SLOW:LOOPS.Y	-1	1	0.977755	0.25	0.0176967	-0.81854	0.66766	-0.582338	@SLOW:L

In addition to the vertical bar, by enabling the measure bars, you will also find two horizontal bars on the right of the grid. They work the same way of the vertical bars, but they take values on the vertical axis.

Like the vertical bars, also the horizontal bars have their values displayed for every variable in the table below the chart.



### 9.2.5.7 OSCILLOSCOPE SETTINGS

You can further customize the appearance of the Oscilloscope by clicking on the *Graph properties* item in the toolbar.



In the window that pops up you can choose whether to display or not the *Background grid*, the *Time slide bar*, and the *Track list*.

### 9.2.5.8 CHANGING THE POLLING RATE

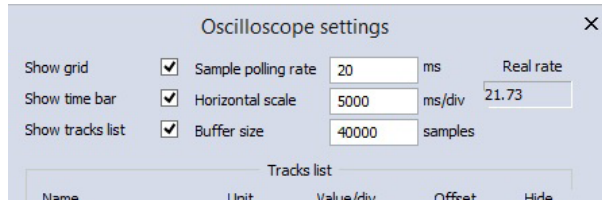
LogicLab periodically sends queries to the target device, in order to read the data to be plotted in the Oscilloscope.

The polling rate can be configured by modifying the *Sample polling rate* voice in the Oscilloscope settings window.

Note that the actual rate depends on the performance of the target device, in particular on the performance of its communication task.







### 9.2.6 SAVING, RESTORING AND PRINTING THE GRAPH

LogicLab allows you to persist the acquisition either by saving the data to a file or by printing a view of the data plotted in the Oscilloscope.

#### 9.2.6.1 SAVING DATA TO A FILE

You can save the samples acquired by the Oscilloscope to a file, in order to further analyze the data with other tools.

- 1) You may want to stop acquisition before saving data to a file.
- 2) Click on the *Save tracks data into file* in the Oscilloscope toolbar.



- 3) Choose between the available output file format: `osc` is a simple plain-text file, containing time and value of each sample; `oscx` is an XML file, that includes more complete information, which can be further analyzed with another tool, provided separately from LogicLab.
- 4) Choose a file name and a destination directory, then confirm the operation.

#### 9.2.6.2 RESTORING DATA INTO THE GRAPH (CURRENTLY NOT IMPLEMENTED)

The oscilloscope allows you to restore data previously saved on file; you have two options to do so: *Load and append graph data* or *Load (no append) graph data*.



In both cases you'll have to select the `OSC` (or `OSCX`) file to load; then if you've chosen to load and append, the data taken from the file will be added to the list of those already inside the oscilloscope; if you've chosen to load without append, the variables list of the oscilloscope will be cleared before adding the data taken from the file.

#### 9.2.6.3 PRINTING THE GRAPH

Follow this procedure to print a view of the data plotted in the Oscilloscope:

- 1) Either suspend or stop the acquisition.
- 2) Only the elements included in the view will be printed, so move the time slide bar and adjust the zoom, in order to include in the view the elements you want to print.
- 3) Click on the *Print graph* item.



### 9.3 EDIT AND DEBUG MODE

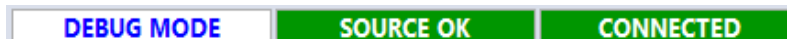
While both the *Watch* window and the Oscilloscope do not make use of the source code, all the other debuggers do: when debug mode is on, changes to the source code are inhibited and debug tools become active.

LogicLab automatically enables debug mode when at least one of the following conditions are met:

- at least one breakpoint is correctly set.
- At least one trigger (graphic or textual) is correctly set.
- Live debug mode is on.

When all the conditions above are not met, the debug mode automatically switches off and LogicLab enters in edit mode.

The status bar shows whether the debug mode is active or not.



Note that you cannot enter the debug mode if the connection status differs from *Connected*.

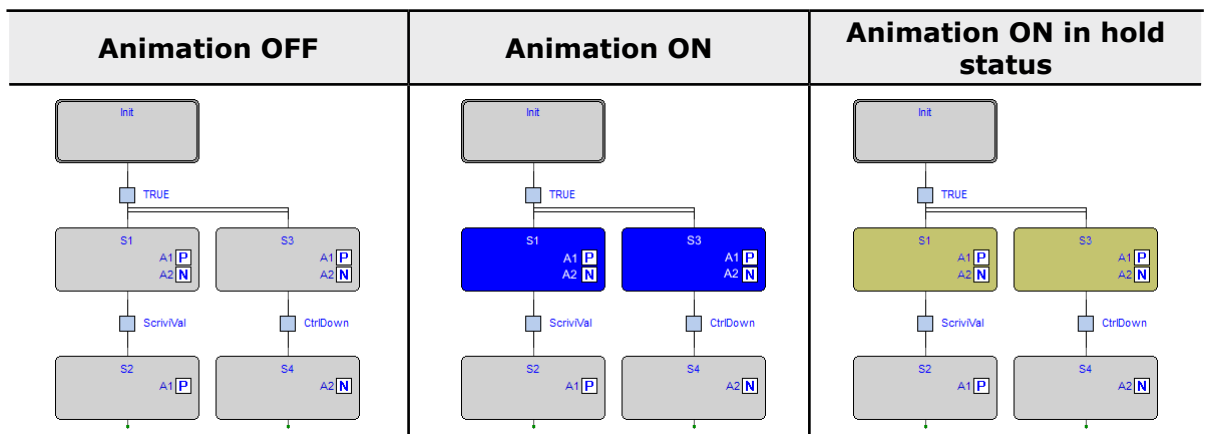
### 9.4 LIVE DEBUG

LogicLab can display meaningful animation of the current and changing state of execution over time of a Program Organization Unit (POU) coded in any IEC 61131-3 programming language.

To switch on and off the live debug mode, you may click [Debug > Live debug mode](#).

#### 9.4.1 SFC ANIMATION

As explained in the relevant section of the language reference, an SFC POU is structured in a set of steps, each of which is either active or inactive at any given moment. Once started up, this SFC-specific debugging tool animates the SFC documents by highlighting the active steps.



In the left column, a portion of an SFC network is shown, diagram animation being off.

In the middle column the same portion of network is displayed when the live debug mode is active. The picture in the middle column shows that steps *S1* and *S3* are currently active, whereas *Init*, *S2*, and *S4* are inactive.

In the right column the same portion of network is displayed with steps *S1* and *S3* that are currently active but in hold status.



This may occur in SFC blocks when they are children of a parent in inactive status.

Note that the SFC animation manager tests periodically the state of all steps, the user not being allowed to edit the sampling period. Therefore, it may happen that a step remains active for a slot of time too short to be displayed.

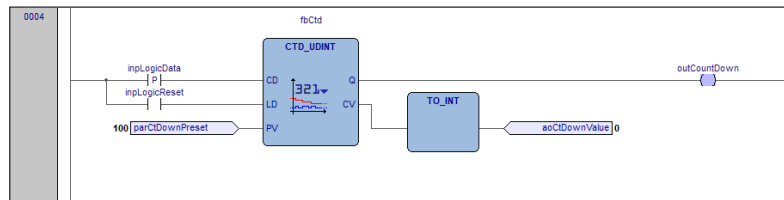
The fact that a step is never highlighted does not imply that its action is not executed, it may simply mean that the sampling rate is too slow to detect the execution.

**9.4.1.1 DEBUGGING ACTIONS AND CONDITIONS**

As explained in the SFC language reference, an action can be assigned to a step, and a transition can be associated with a transition code (condition). Actions and transition codes can be coded in any of the IEC 61131-3 languages (except SFC for transition code). General-purpose debugging tools can be used within each action/condition, as if it was a stand-alone POU.

**9.4.2 LD ANIMATION**

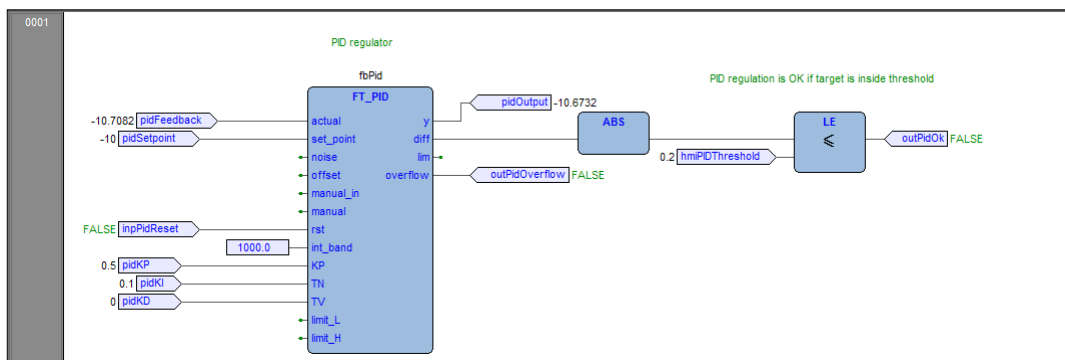
In live debug mode, LogicLab displays the values of all the visible variables directly in the graphical source code editor.



Note that the LD animation manager tests periodically the state of all the elements. It may happen that an element remains true for a slot of time too short to be displayed on the video. The fact that an element is never highlighted does not imply that its value never becomes true (the sampling rate may be too slow).

**9.4.3 FBD ANIMATION**

In live debug mode, LogicLab displays the values of all the visible variables directly in the graphical source code editor.



Note that the LD animation manager tests periodically the state of all the elements. It may happen that an element remains true for a slot of time too short to be displayed on the video. The fact that an element is never highlighted does not imply that its value never becomes true (the sampling rate may be too slow).



## 9.4.4 IL AND ST ANIMATION

The live debug mode also applies to textual source code editors (the ones for IL and ST). The values of a variable is directly displayed in a green box just at the right of the variable.

```

0001
0002 y -0.481671 := SIN(x 462.327 );
0003 x 462.327 := x 462.327 + incr 0.01 ;
0004
0005
0006 loopsValue 16#005F := 0;
0007 for i 8 := 0 to 15 do
0008   bit FALSE := (y -0.481671 + 0.9) > (0.125 * TO_REAL(i 8 ));
0009
0010   if bit FALSE then
0011     loopsValue 16#005F := loopsValue 16#005F or RotateBit(i 8 );
0012   end_if;
0013 end_for;
0014
0015
0016 (* uses pid to move value up and down *)
0017 loopsValue 16#005F := 0;
0018 for i 8 := 0 to 15 do
0019   bit FALSE := (pidFeedback -2.66368 + 10.0) > (1.25 * TO_REAL(i 8 ));
0020
0021   if bit FALSE then

```

## 9.5 TRIGGERS

### 9.5.1 TRIGGER WINDOW

The *Trigger window* tool allows you to watch the value of a set of variables and to have them updated synchronously in a special pop-up window.

#### 9.5.1.1 PRE-CONDITIONS TO OPEN A TRIGGER WINDOW

##### No need for special compilation

LogicLab debugging tools operate at run-time. Thus, unlike other programming languages such as C++, the compiler does not need to be told whether or not to support trigger windows: given a PLC code, the compiler's output is unique, and there is no distinction between debug and release version.

##### Memory availability

A trigger window takes a segment in the application code sector, having a well-defined length. Obviously, in order to start up a trigger window, it is necessary that a sufficient amount of memory is available, otherwise an error message appears.

##### Incompatibility with graphic trigger windows

A graphic trigger window takes the whole free space of the application code sector. Therefore, once such a debugging tool has been started, it is not possible to add any trigger window, and an error message appears if you attempt to start a new window. Once the graphic trigger window is eventually closed, trigger windows are enabled again.

Note that all the trigger windows existing before the starting of a graphic trigger window keep working normally. You are simply not allowed to add new ones.





#### 9.5.1.2 SET AND REMOVE TRIGGERS

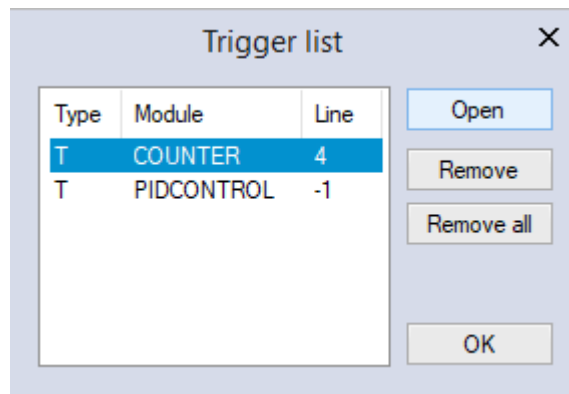
From the *Debug* menu you can select the appropriate voice for work with triggers.

Triggers can be palced only if a valid connection is established and the PLC is currently executing.

From the *Debug* menu you can choose the following voices:



Command	Icon in debug toolbar	Description
<i>Add/Remove text trigger</i>		In order to actually start a trigger window, select the point of the PLC code where to insert the relative trigger and then press this button or use the shortcut pressing <i>F9</i> . Do the same to remove the trigger.
<i>Add/Remove graphic trigger</i>		This button operates exactly as the above <i>Set/Remove trigger</i> , except for that it opens a graphic trigger window. It can be used likewise also to remove a graphic trigger window. Shortcut key: pressing <i>Shift + F9</i> .
<i>Remove all triggers</i>		Pressing this key causes all the existing trigger windows and the graphic trigger window to be removed simultaneously. Shortcut key: pressing <i>Ctrl+Shift+F9</i> is equivalent to clicking on this button.
<i>Trigger list</i>		This key opens a dialog listing all the existing trigger windows. Shortcut key: pressing <i>Ctrl+I</i> is equivalent to clicking on this button.

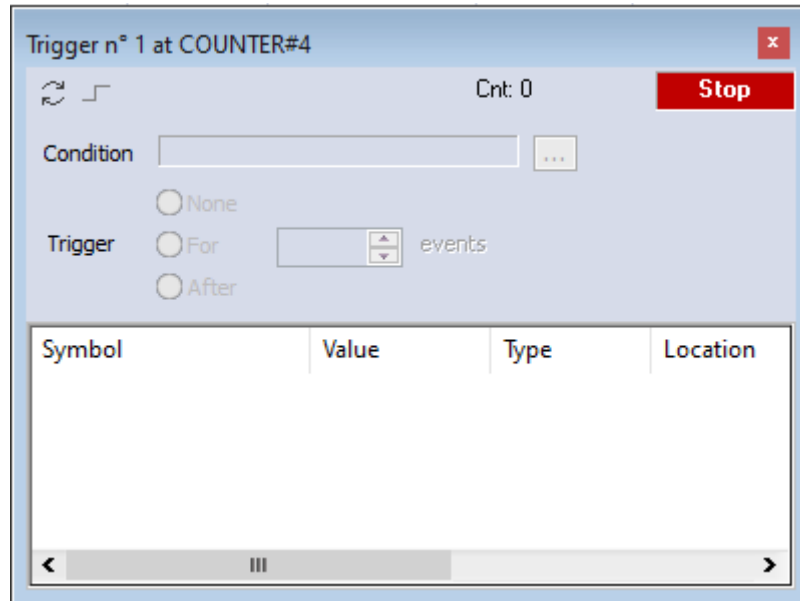


Each record refers to a trigger window, either graphic or textual. The following table explains the meaning of each field.

Field	Description
<i>Type</i>	<i>T</i> : trigger window. <i>G</i> : graphic trigger window.
<i>Module</i>	Name of the program, function, or function block where the trigger is placed. If the module is a function block, this field contains its name, not the name of its instance where you actually put the trigger.
<i>Line</i>	For the textual languages (IL, ST) indicates the line in which the trigger is placed. For the other languages the value is always <i>-1</i> .

### 9.5.1.3 TRIGGER WINDOW INTERFACE

Setting a trigger causes a pop-up window to appear, which is called *Interface* window: this is the interface to access the debugging functions that the trigger window makes available. It consists of three elements, as shown below.



#### Caption bar

The *Caption* bar of the pop-up window shows information on the location of the trigger which causes the refresh of the *Variables* window, when reached by the processor.

The text in the *Caption* bar has the following format:

Trigger n° X at ModuleName#Location

where

<i>X</i>	Trigger identifier.
<i>ModuleName</i>	Name of the program, function, or function block where the trigger was placed.
<i>Location</i>	Exact location of the trigger, within module <i>ModuleName</i> . If <i>ModuleName</i> is in IL, <i>Location</i> has the following format: N1 Otherwise, if <i>ModuleName</i> is in FBD, it becomes: N2\$BT: BID where: N1 = instruction line number N2 = network number BT = block type (operand, function, function block, etc.) BID = block identifier

#### Controls section

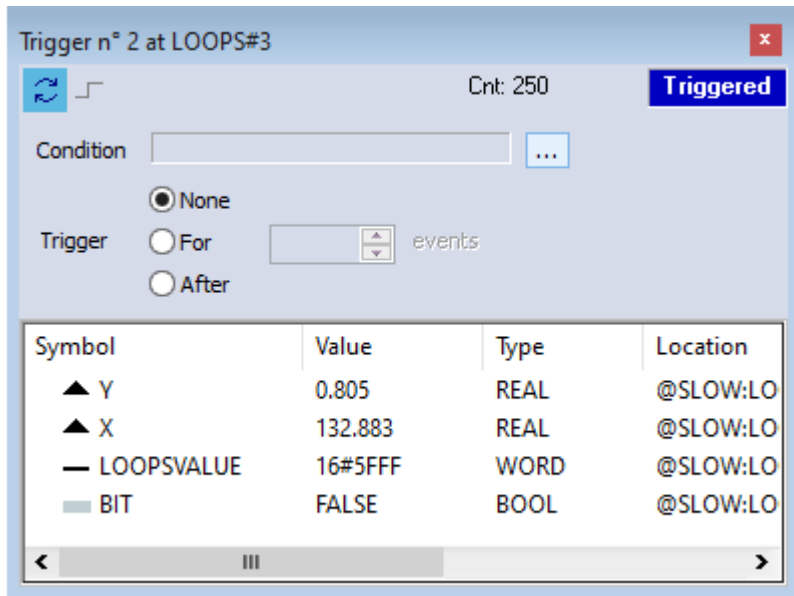
This dialog box allows the user to better control the refresh of the trigger window to get more information on the code under scope. A detailed description of the function of each control is given in the *Trigger window* controls section (see Paragraph 9.5.2.11).

All controls are not accessible until at least one variable is dragged into the debug window.



**The Variables section**

This lower section of the *Debug* window is a table consisting of a row for each variable that you dragged in. Each row has four fields: the name of the variable, its value, its type, and its location (@task:ModuleName) read from memory during the last refresh.



**9.5.1.4 TRIGGER WINDOW: DRAG AND DROP INFORMATION**

To watch a variable, you need to copy it to the lower section of the *Debug* window. This section is a table consisting of a row for each variable you dragged in. You can drag into the trigger window only variables local to the module where you placed the relative trigger, or global variables, or parameters. You cannot drag variables declared in another program, or function, or function block.

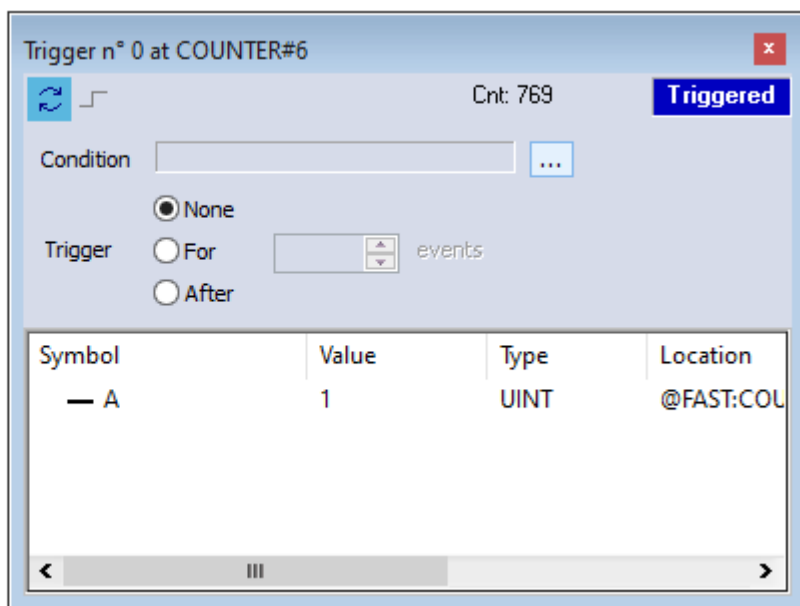
**9.5.1.5 REFRESH OF THE VALUES**

Let us consider the following example.

```

0001
0002 ID 1
0003 ST a
0004
0005 ID 2
0006 ST a
0007
0008 ID 3
0009 ST a
0010
0011
0012
0013
0014
0015
0016
0017
0018
0019
0020
0021
0022
0023
0024

```



The value of variables is refreshed every time the window manager is triggered, that is every time the processor executes the instruction marked by the green arrowhead. However, you can set controls in order to have variables refreshed only when triggers satisfy the more limiting conditions you define.

Note that the value of the variables in column *Symbol* is read from memory just before the marked instruction (in this case: the instruction at line 5) and immediately after the previous instruction (the one at line 4) has been performed.



Thus, in the above example the second ST statement has not been executed yet when the new value of *a* is read from memory and displayed in the trigger window. Thus the result of the second ST *a* is 1.

**9.5.1.6 TRIGGER WINDOW CONTROLS**

This paragraph deals with the trigger window controls, which allows you to better supervise the working of this debugging tool, to get more information on the code under scope. Trigger window controls act in a well-defined way on the behaviour of the window, regardless for the type of the module (either IL or FBD) where the related trigger has been inserted.

All controls are not accessible until at least one variable is dragged into the *Variables* window.

Window controls are made accessible to users through the grey top half of the debug window.

Button	Command	Description
	<i>Start/Stop</i>	This control is used to start a triggering session. If system is triggering you can click this button to force stop. Otherwise session automatically stops when conditions are reached. At this point you can press this button to start another triggering session.
	<i>Single step execution</i>	This control is used to execute a single step trigger. It is enabled only when there is no active triggering session and <i>None</i> is selected. Specified condition is considered. After the single step trigger is done, triggering session automatically stops.

**Trigger counter**

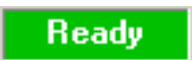
Cnt: 4647

This read-only control counts how many times the debug window manager has been triggered, since the window was installed.

The window manager automatically resets this counter every time a new triggering session is started.

**Trigger state**

This read-only control shows the user the state of the *Debug* window. It can assume the following values.

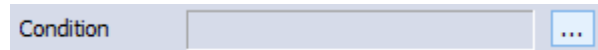
	The trigger has not occurred during the current task execution.
---	---





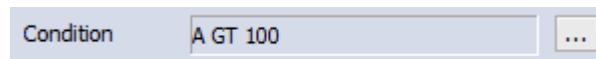
<b>Triggered</b>	The trigger has occurred during the current task execution.
<b>Stop</b>	System is not triggering. Triggering has not been started yet or it has been stopped by user or an halt condition has been reached.
<b>Error</b>	Communication with target interrupted, the state of the trigger window cannot be determined.

**User-defined condition**

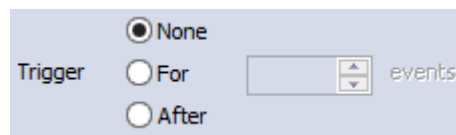


If you define a condition by using this control, the values in the *Debug* window are re-freshed every time the window manager is triggered and the user-defined condition is true.

After you have entered a condition, the control displays its simplified expression.



**Counters**



These controls allow the user to define conditions on the trigger counter.

The trigger window can be in one of the following three states.

- *None*: no counter has been started up, thus no condition has been specified upon the trigger.
- *For*: assuming that you gave the counter limit the value  $N$ , the window manager adds  $1$  to the current value of the counter and refreshes the value of its variables, each time the debug window is triggered. However, when the counter equals  $N$ , the window stops refreshing the values, and it changes to the *Stop* state.
- *After*: assuming that you gave the counter limit the value  $N$ , the window manager re-sets the counter and adds  $1$  to its current value each time it is triggered. The window remains in the *Ready* state and does not update the value of its variables until the counter reaches  $N$ .

**9.5.2 DEBUGGING WITH TRIGGER WINDOWS**

**9.5.2.1 INTRODUCTION**

The trigger window tool allows the user to select a set of variables and to have their values displayed and updated synchronously in a pop-up window. Unlike the *Watch* window, trigger windows refresh simultaneously all the variables they contain, every time they are triggered.

**9.5.2.2 OPENING A TRIGGER WINDOW FROM AN IL MODULE**

Let us assume that you have an IL module, also containing the following instructions.



```

0001
0002 LD a
0003 ADD b
0004 ST a
0005
0006 LD c
0007 ADD d
0008 ST c
0009
0010 LD k
0011 ADD 1
0012 ST k
0013
0014

```

Let us also assume that you want to know the value of *b*, *d*, and *k*, just before the *ST k* instruction is executed. To do so, move the cursor to line 12, then click **Debug>Add/Remove text trigger**.

A green arrowhead appears next to the line number, and the related trigger window pops up.

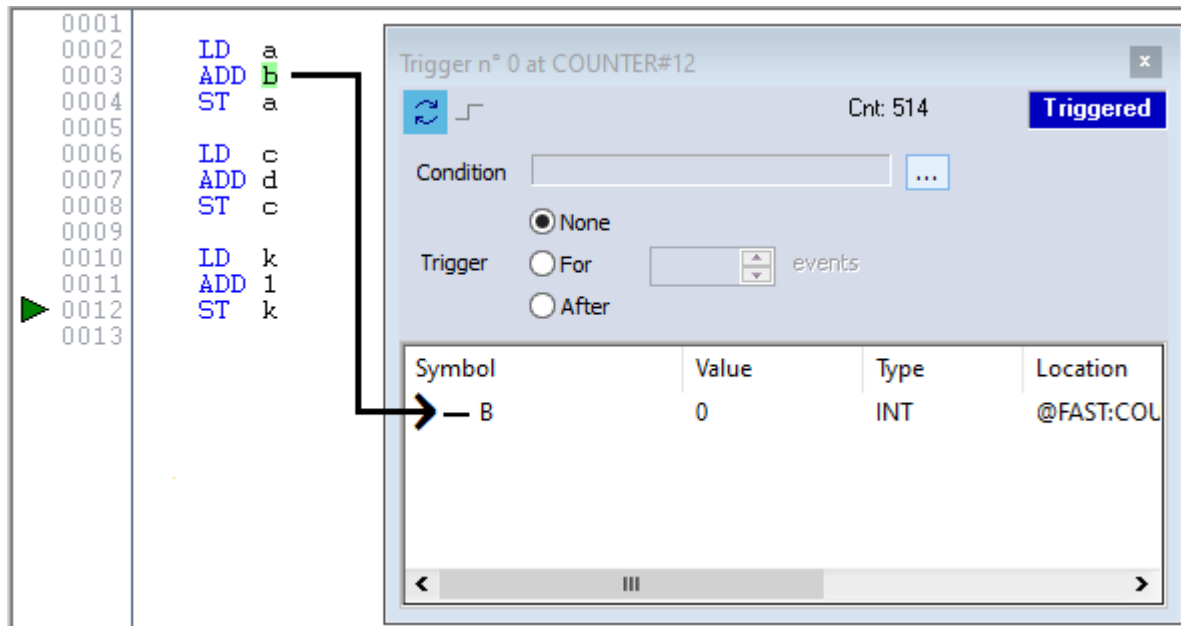
The screenshot shows the assembly code from the previous block. A green arrowhead points to line 12. A dialog box titled "Trigger n° 0 at COUNTER#12" is open. It contains a "Condition" field, a "Trigger" section with radio buttons for "None", "For", and "After", and a table with columns "Symbol", "Value", "Type", and "Location". The "For" radio button is selected. The "Cnt: 0" and "Stop" buttons are also visible.

Not all the IL instructions support triggers. For example, it is not possible to place a trigger at the beginning of a line containing a *JMP* statement.

### 9.5.2.3 ADDING A VARIABLE TO A TRIGGER WINDOW FROM AN IL MODULE

In order to watch the value of a variable, you need to add it to the trigger window. To this purpose, select a variable by double-clicking it, and then drag it into the *Variables* window, that is the lower white box in the pop-up window. The variable's name now appears in the *Symbol* column.

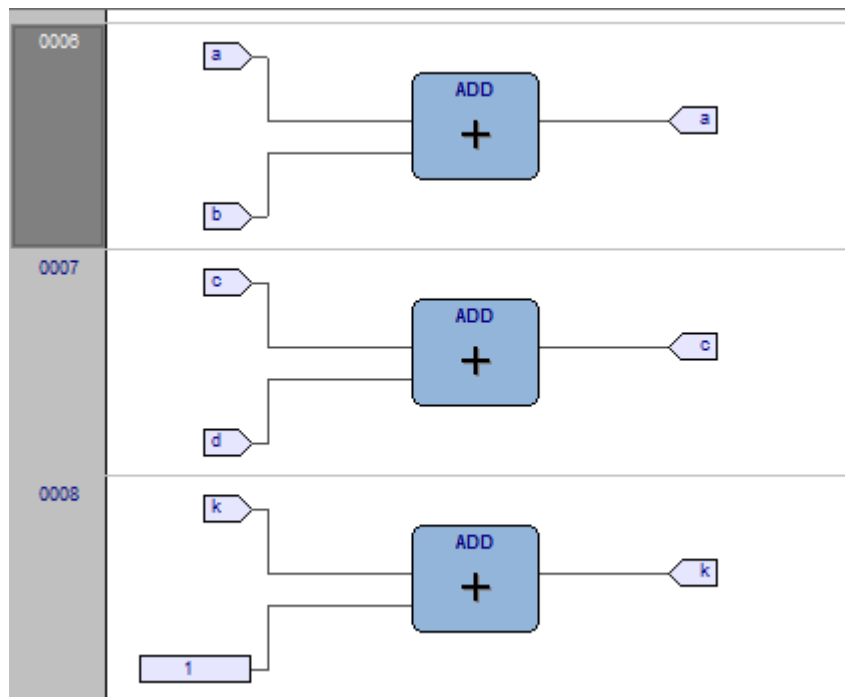




The same procedure applies to all the variables you wish to inspect.

**9.5.2.4 OPENING A TRIGGER WINDOW FROM AN FBD MODULE**

Let us assume that you have an FBD module, also containing the following instructions.



Let us also assume that you want to know the values of c, d, and k, just before the last instruction is executed.

Provided that you can never place a trigger in a block representing a variable such as



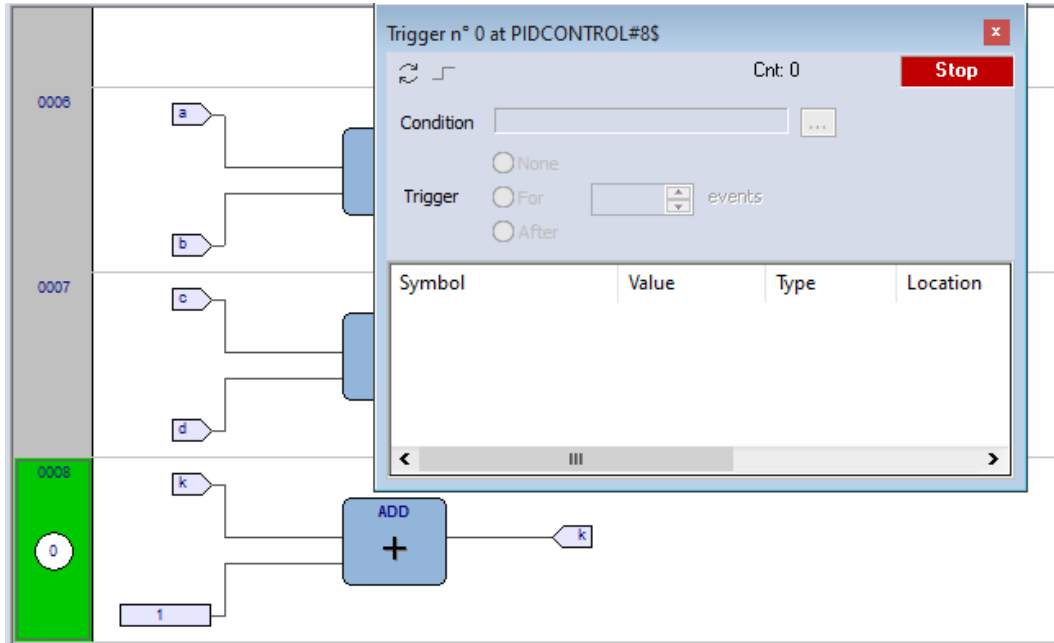
you must select the first available block preceding the selected variable. In the example of the above figure, you must move the cursor to network 8, and click the ADD block.



You can click [Debug>Add/Remove text trigger](#).

Else, you can also insert trigger on the whole line; this means you'll hit the trigger just before the first instruction of that line is executed. To do so, select the row by clicking the gray button on the left (the one with the row number inside) and add the trigger.

In both cases, the color of the selected block turns to green, a white circle with a number inside appears in the middle of the block, and the related trigger window pops up.



When preprocessing FBD source code, the compiler translates it into IL instructions. The instruction in the selected network will be expanded to:

```
LD k
ADD 1
ST k
```

When you add a trigger to an FBD block, you actually place the trigger on the first statement of its IL equivalent code.

### 9.5.2.5 ADDING A VARIABLE TO A TRIGGER WINDOW FROM AN FBD MODULE

In order to watch the value of a variable, you need to add it to the trigger window. Let us assume that you want to inspect the value of variable *k* of the FBD code in the figure below.

To this purpose, click [Edit>Watch mode](#).

The cursor will become as follows.



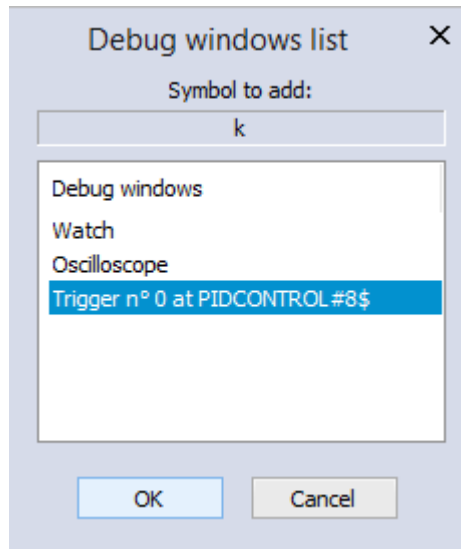
Now you can click the block representing the variable you wish to be shown in the trigger window.

Now select the element you wish to inspect, in our example let's suppose the variable *K*.

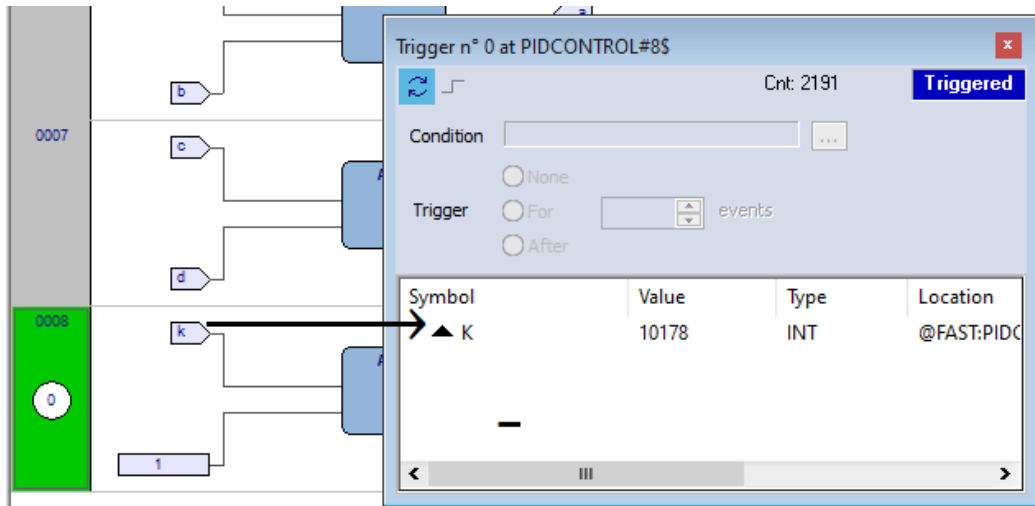


A dialog box appears listing all the currently existing instances of debug windows, and asking you which one is to receive the object you have just clicked.





In order to display the variable `k` in the trigger window, select its reference in the *Debug windows* column, then press *OK*. The name of the variable is now printed in the *Symbol* column.

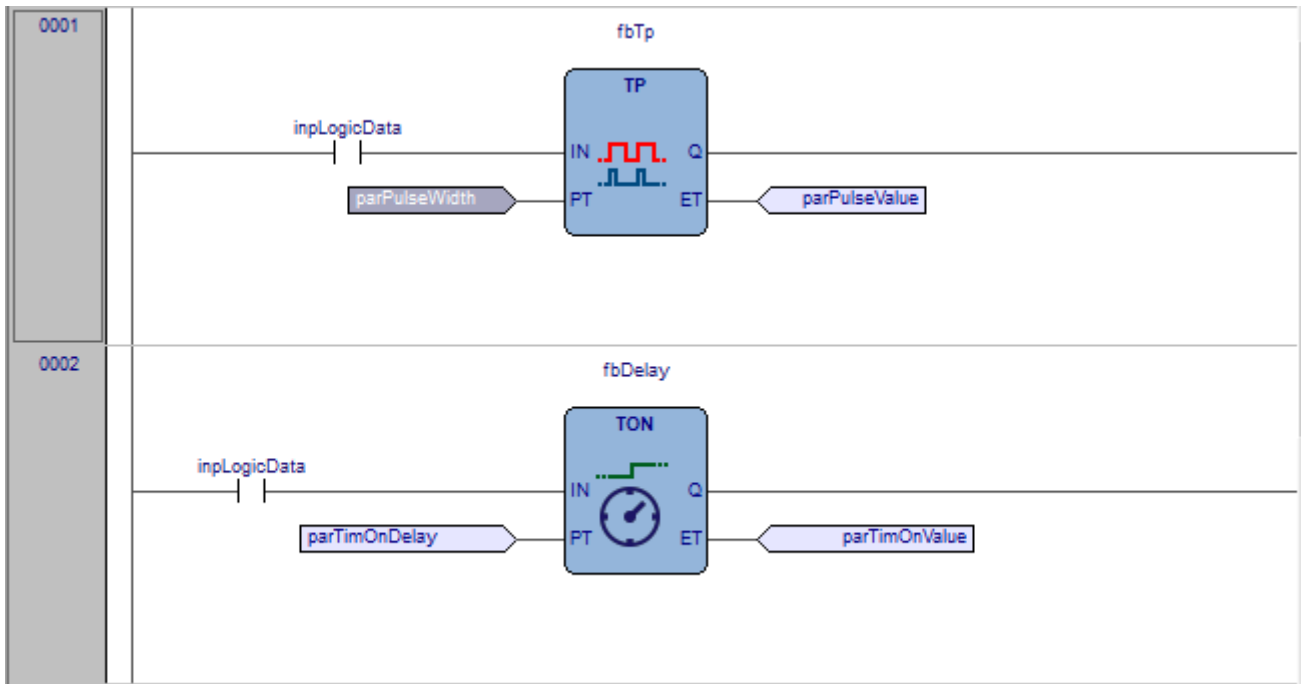


The same procedure applies to all the variables you wish to inspect.

Once you have added to the *Graphic watch* window all the variables you want to observe, you can click `Edit>Insert/Move mode`, so as to let the cursor take back its original shape.

### 9.5.2.6 OPENING A TRIGGER WINDOW FROM AN LD MODULE

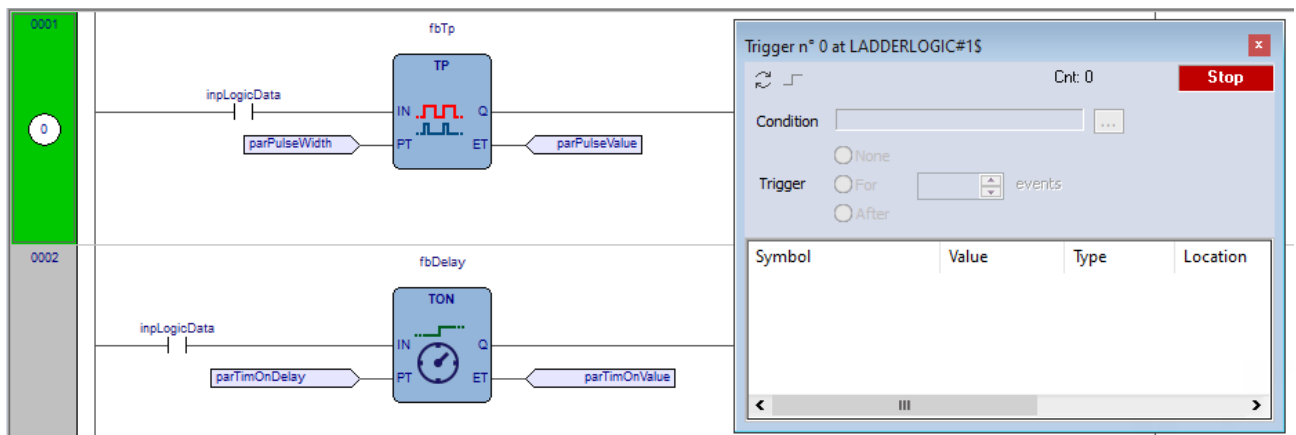
Let us assume that you have an LD module containing the following instructions:



You can place a trigger directly on a block, a contact, a coil or the entire row.

Let us assume that you want to know the value of some variables every time the processor reaches network number 1. First select the network number 1 by clicking on the grey area on its left, the one containing the network number. Now you can click [Debug>Add/Remove text trigger](#).

In both cases, the grey raised button containing the network number turns to green, and a white circle with the number of the trigger inside appears in the middle of the button, while the related trigger window pops up.



Unlike the other languages supported by LogicLab, LD does not allow you to insert a trigger into a single contact or coil, as it lets you select only an entire network. Thus the variables in the trigger window will be refreshed every time the processor reaches the beginning of the selected network.

### 9.5.2.7 ADDING A VARIABLE TO A TRIGGER WINDOW FROM AN LD MODULE

In order to watch the value of a variable, you need to add it to the trigger window. Let us assume that you want to inspect the value of variable *b* in the LD code represented in the figure below.

To this purpose, click [Edit>Watch mode](#).

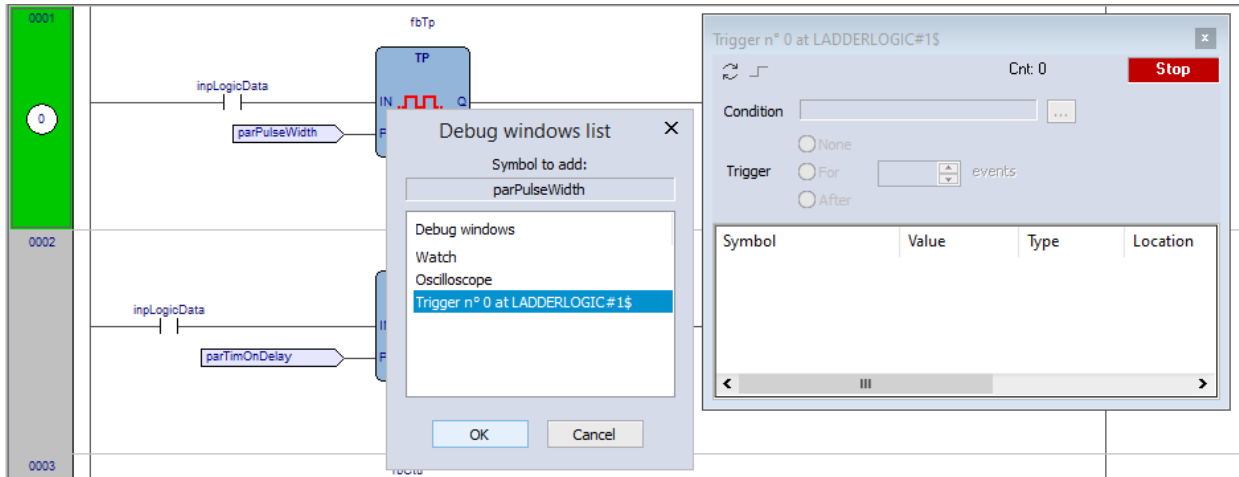
The cursor will become as follows.





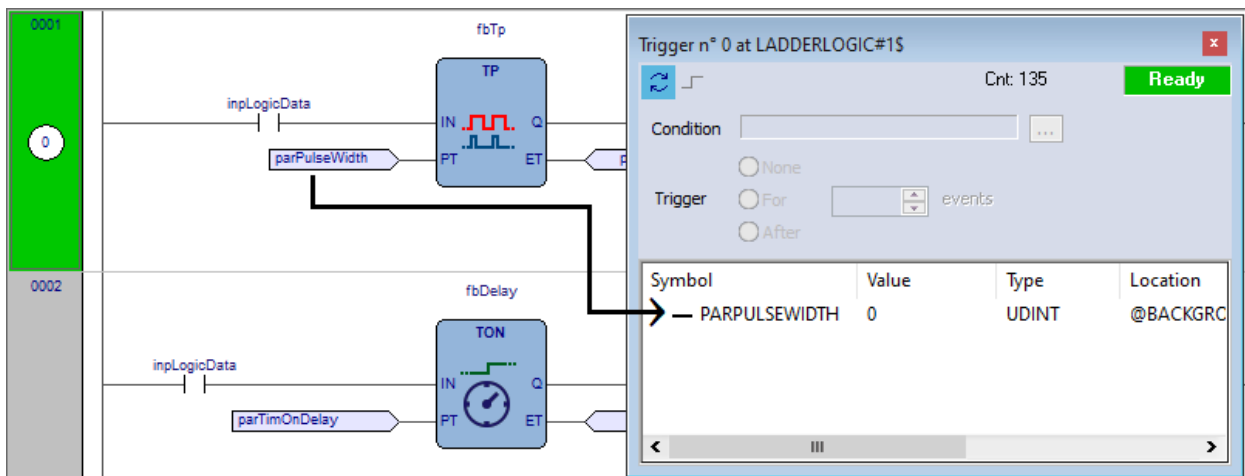
Now you can click the item representing the variable you wish to be shown in the trigger window.

A dialog box appears listing all the currently existing instances of debug windows, and asking you which one is to receive the object you have just clicked.



In order to display variable *parPulseWidth* in the trigger window, select its reference in the *Debug window* column, then press *OK*.

The name of the variable is now printed in the *Symbol* column.



The same procedure applies to all the variables you wish to inspect.

Once you have added to the *Graphic watch* window all the variables you want to observe, you can click **Edit>Insert/Move mode**, so as to restore the original shape of the cursor.

### 9.5.2.8 OPENING A TRIGGER WINDOW FROM AN ST MODULE

Let us assume that you have an ST module containing the following instructions.

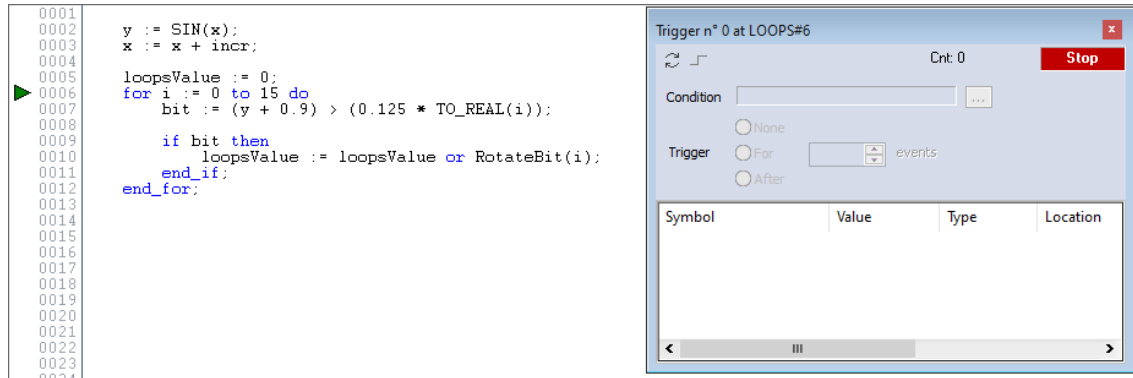
```

0001
0002   y := SIN(x);
0003   x := x + incr;|
0004
0005   loopsValue := 0;
0006   for i := 0 to 15 do
0007     bit := (y + 0.9) > (0.125 * TO_REAL(i));
0008
0009     if bit then
0010       loopsValue := loopsValue or RotateBit(i);
0011     end_if;
0012   end_for;
0013

```



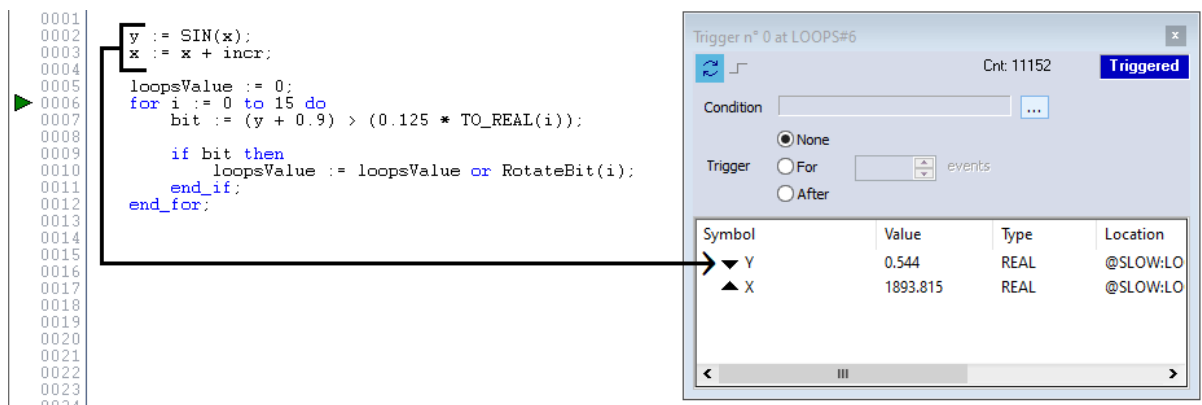
Let us assume that you want to know the value of  $x$  and  $y$ , just before the *for* cycle is executed. To do so, move the cursor to line 6 and click **Debug>Add/Remove text trigger**. A green arrowhead appears next to the line number, and the related trigger window pops up.



Not all the ST instructions support triggers. For example, it is not possible to place a trigger on a line containing a terminator such as `END_IF`, `END_FOR`, `END_WHILE`, etc..

### 9.5.2.9 ADDING A VARIABLE TO A TRIGGER WINDOW FROM AN ST MODULE

In order to watch the value of a variable, you need to add it to the trigger window. To this purpose, select a variable, by double clicking it, and then drag it into the *Variables* window, that is the lower white box in the pop-up window. The variable name now appears in the *Symbol* column.



The same procedure applies to all the variables you wish to inspect.

### 9.5.2.10 REMOVING A VARIABLE FROM THE TRIGGER WINDOW

If you want a variable not to be displayed any more in the trigger window, select it by clicking its name once, then press the *Del* key.

### 9.5.2.11 CLOSING A TRIGGER WINDOW AND REMOVING A TRIGGER

This page deals with what you can do when you finish a debug session with a trigger window. You can choose between the following options.

- Closing the trigger window.
- Removing the trigger.
- Removing all the triggers.





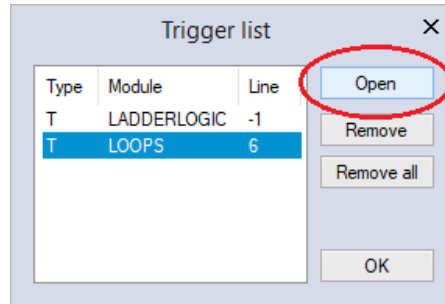
Notice that the actions listed above produce very different results.

**Closing the trigger window**

If you close a trigger window you’re just hiding its interface window, the trigger is still active. You will still see the green arrowhead (or the network highlighted).

The fact that the trigger is active, means it is still working, if you reopen the interface you won’t find it at the same status it was when you closed it; for example the counter may have raised.

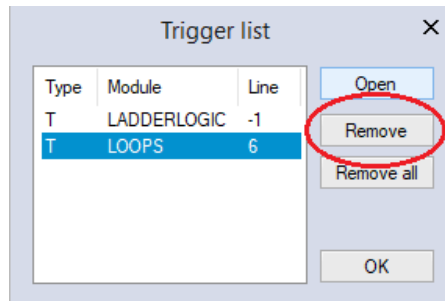
You can reopen the interface window, to resume working with the same trigger, by opening the *Trigger list* window, select the record referred to that trigger, and click the *Open* button.



The interface window appears with value of variables and trigger counter updated, as if it had not been closed.

**Removing a trigger**

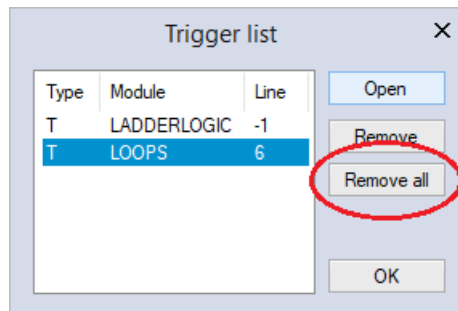
If you choose this option, you completely remove the code both of the window manager and of its trigger. To this purpose, just open the *Trigger list* window, select the record referred to the trigger window you want to eliminate, and click the *Remove* button.



Alternatively, you can move the cursor to the line (if the module is in IL or ST), or click the block (if the module is in FBD or LD) where you placed the trigger. Now press `Debug>Add/Remove trigger`.

**Removing all the triggers**

If you wish, you can remove all the existing triggers at once, regardless for which records are selected, by clicking on the *Remove all* button from the *Trigger list* window.



Alternatively you can select `Debug>Remove all triggers`.

## 9.6 GRAPHIC TRIGGERS

### 9.6.1 GRAPHIC TRIGGER WINDOW

The graphic trigger window tool allows you to select a set of variables and to have them sampled synchronously and to have their curve displayed in a special pop-up window.

Sampling of the dragged-in variables occurs every time the processor reaches the position (i.e. the instruction - if IL, ST - or the block - if FBD, LD) where you placed the trigger.

#### 9.6.1.1 PRE-CONDITIONS TO OPEN A GRAPHIC TRIGGER WINDOW

##### No need for special compilation

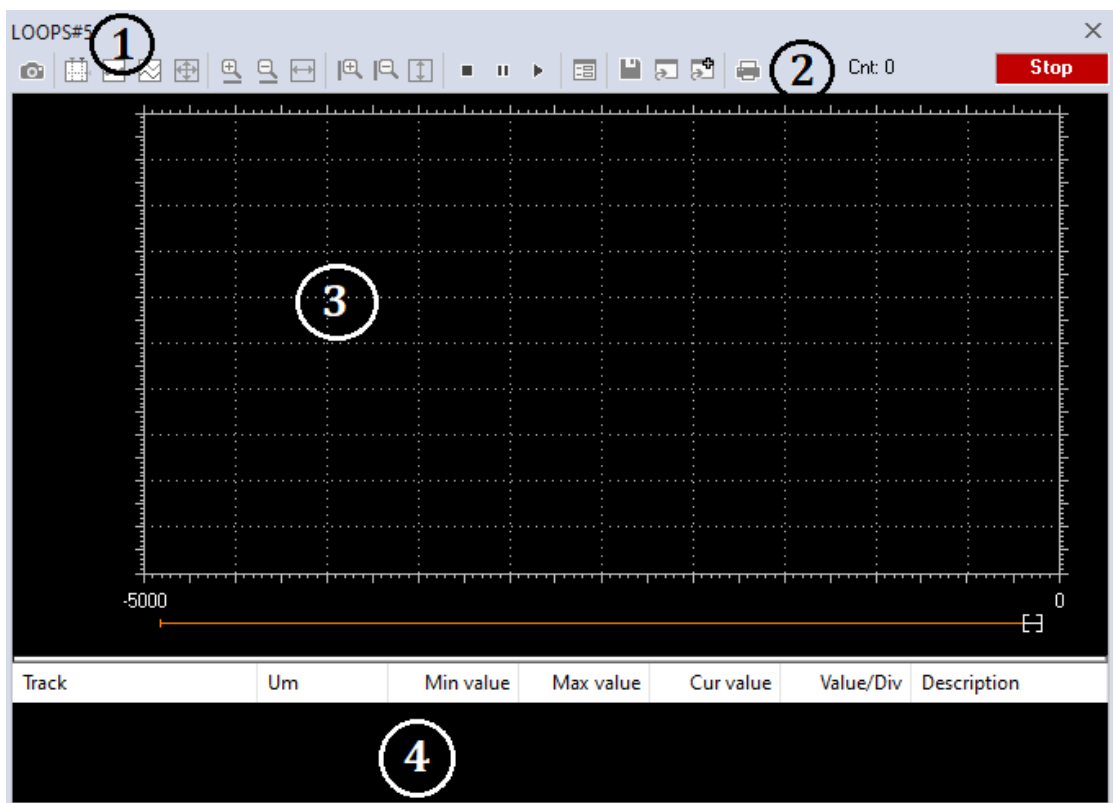
All the LogicLab debugging tools operate at run-time. Thus, unlike other programming languages such as C++, the compiler does not need to be told whether or not to support trigger windows: given a PLC code, the compiler's output is unique, and there is no distinction between debug and release version.

##### Memory availability

A graphic trigger window takes all the free memory space in the application code sector. Obviously, in order to start up a trigger window, it is necessary that a sufficient amount of memory is available, otherwise an error message appears.

#### 9.6.1.2 GRAPHIC TRIGGER WINDOW INTERFACE

Setting a graphic trigger causes a pop-up window to appear, which is called *Interface* window. This is the main interface for accessing the debugging functions that the graphic trigger window makes available. It consists of several elements, as shown below.



1. Caption bar 2. Controls bar 3. Chart area 4. Variables window



### The caption bar

The *Caption* bar at the top of the pop-up window shows information on the location of the trigger which causes the variables listed in the *Variables* window to be sampled.

The text in the caption has the following format:

ModuleName#Location

Where

ModuleName	Name of program, function, or function block where the trigger was placed.
Location	<p>Exact location of the trigger, within module <code>ModuleName</code>.</p> <p>If <code>ModuleName</code> is in IL, ST, <code>Location</code> has the format:</p> <p>N1</p> <p>Otherwise, if <code>ModuleName</code> is in FBD, LD, it becomes:</p> <p>N2\$BT:PID</p> <p>N1 = instruction line number  N2 = network number  BT = block type (operand, function, function block, etc.)  PID = block identifier</p>

### The Controls bar

This dialog box allows you to better control the working of the graphic trigger window. A detailed description of the function of each control is given in the *Graphic trigger* window controls section (see Paragraph 9.6.1.5).

### The Chart area

The *Chart* area includes six items:

- 1) Plot: area containing the actual plot of the curve of the dragged-in variables.
- 2) Samples to acquire: number of samples to be collected by the graphic trigger window manager.
- 3) Horizontal cursor: cursor identifying a horizontal line. The value of each variable at the intersection with this line is reported in the column *horz cursor*.
- 4) Blue cursor: cursor identifying a vertical line. The value of each variable at the intersection with this line is reported in the column *left cursor*.
- 5) Red cursor: same as blue cursor.
- 6) Scroll bar: if the scale of the x-axis is too large to display all the samples in the *Plot* area, the scroll bar allows you to slide back and forth along the horizontal axis.

### The Variables window

This lower section of the *Debug* window is a table consisting of a row for each variable that you have dragged in. Every row has several fields, which are described in detail in the Drag and drop information section.

## 9.6.1.3 GRAPHIC TRIGGER WINDOW: DRAG AND DROP INFORMATION

To watch a variable, you need to copy it to the lower section of the *Debug* window.

This lower section of the *Debug* window is a table consisting of a row for each variable that you dragged in. Each row has several fields, as shown in the picture below.



Track	Um	Min value	Max value	Cur value	Value/Div	V blue curs	V red curs	H red cursor	H blue cursor	Description
Y		0.665679	0.999997	0.992272	0.0417897	0.665679	0.993436	0.937312	0.728363	@SLOW:LOOPS
X		4782.24	4783.21	4783.21	0.12085	4782.24	4783.2	4783.03	4782.42	@SLOW:LOOPS

Field	Description
<i>Track</i>	Name of the variable.
<i>Um</i>	Unit of measurement.
<i>Min value</i>	Minimum value in the record set.
<i>Max value</i>	Maximum value in the record set.
<i>Cur value</i>	Current value of the variable.
<i>v/div</i>	How many engineering units are represented by a unit of the y-axis (i.e. the space between two ticks on the vertical axis).
<i>V Blue cursor</i>	Value of the variable at the intersection with the line identified by the vertical blue cursor.
<i>V Red cursor</i>	Value of the variable at the intersection with the line identified by the vertical red cursor.
<i>H Red cursor</i>	Value of the variable at the intersection with the line identified by the horizontal red cursor.
<i>H Blue cursor</i>	Value of the variable at the intersection with the line identified by the horizontal red cursor.
<i>Description</i>	Information about the variable in watch, like its parent POU and the execution task

Note that you can drag into the graphic trigger window only variables local to the module where you placed the relative trigger, or global variables, or parameters. You cannot drag variables declared in another program, or function, or function block.

#### 9.6.1.4 SAMPLING OF VARIABLES

Let us consider the following example.

The value of the variables is sampled every time the window manager is triggered, that is every time the processor executes the instruction marked by the green arrowhead. However, you can set controls in order to have variables sampled when triggers also satisfy further limiting conditions that you define.

The value of the variables in the column *Track* is read from memory just before the marked instruction and immediately after the previous instruction.



### 9.6.1.5 GRAPHIC TRIGGER WINDOW CONTROLS

This paragraph deals with controls of the *Graphic trigger* window. Controls allow you to specify in detail when LogicLab is supposed to sample the variables added to the *Variables* window.









Graphic trigger window controls act in a well-defined way on the behaviour of the window, regardless for the type of the module (IL, ST, FBD or LD) where the related trigger has been inserted.

Window controls are made accessible to users through the *Controls* bar of the debug window.



Button	Command	Description
	<i>Start graphic trace</i>	When you push this button down, you let acquisition start. Now, if acquisition is running and you release this button, you stop the sample collection process, and you reset all the data you have acquired so far.
	<i>Enable/Disable cursors</i>	The two cursors (red cursor, blue cursor) may be seen and moved along their axis as long as this button is pressed. Release this button if you want to hide simultaneously all the cursors.
	<i>Show samples</i>	This control is used to put in evidence the exact point in which the variables are triggered at each sample.
	<i>Split tracks</i>	When pressed, this control splits the y-axis into as many segments as the dragged-in variables, so that the diagram of each variable is drawn in a separate band.
	<i>Show all values</i>	It is used to fill in the graph window all the values sampled for the selected variables in the current record set.
	<i>Horizontal Zoom In and Zoom Out</i>	Zooming in is an operation that makes the curves in the <i>Chart</i> area appear larger on the screen, so that greater detail may be viewed. Zooming out is an operation that makes the curves appear smaller on the screen, so that it may be viewed in its entirety. Horizontal zoom acts only on the horizontal axis.
	<i>Horizontal show all</i>	This control is used to horizontally center record set samples. So first sample will be placed on the left margin, and last will be placed on the right margin of the graphic window.
	<i>Vertical Zoom In and Zoom Out</i>	<i>Vertical Zoom</i> acts only on the vertical axis.
	<i>Vertical show all</i>	This control is used to vertically center record set samples. So max value sample will be placed near top margin and low value sample will be placed on the bottom margin of the graphic window.



Button	Command	Description
	<i>Stop acquisition</i>	Not implemented.
	<i>Pause acquisition</i>	Not implemented.
	<i>Restart acquisition</i>	Not implemented.
	<i>Graphic trigger window properties</i>	Pushing this button causes a tabs dialog box to appear, which allows you to set general user options affecting the action of the graphic trigger window. Since the options you can set are quite numerous, they are dealt with in a section apart. Click here to access this section.
	<i>Save chart</i>	Press this button to save the chart.
	<i>Load data (no append)</i>	Not implemented.
	<i>Load data and append</i>	Not implemented.
	<i>Print chart</i>	Push this button to print both the <i>Chart area</i> and the <i>Variables</i> window.

## Trigger counter

Cnt: 107/1000





This read-only control displays two numbers with the following format:  $x/y$ .

$x$  indicates how many times the debug window manager has been triggered, since the graphic trigger was installed.

$y$  represents the number of samples the graphic window has to collect before stopping data acquisition and drawing the curves.

## Trigger state

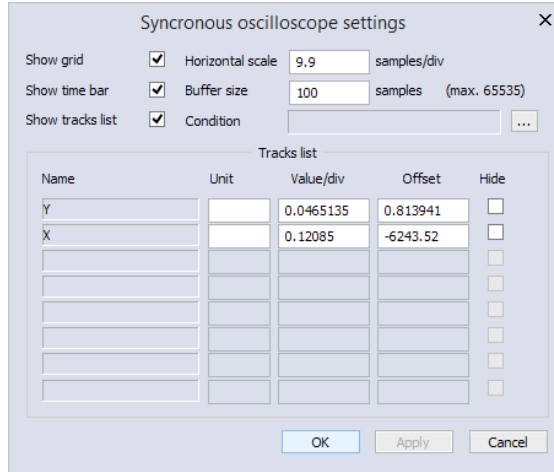
This read-only control shows you the state of the *Debug* window. It can assume the following values.

	No sample(s) taken, as the trigger has not occurred during the current task execution.
	Sample(s) collected, as the trigger has occurred during the current task execution.
	The trigger counter indicates that a number of samples has been collected satisfying the user request or memory constraints, thus the acquisition process is stopped.
	Communication with target interrupted, the state of the trigger window cannot be determined.



**9.6.1.6 GRAPHIC TRIGGER WINDOW OPTIONS**

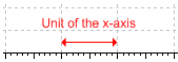
In order to open the options tab, you must click the *Properties* button in the *Controls* bar. When you do this, the following dialog box appears.



**Control**

Control	Description
<i>Show grid</i>	Tick this control to display a grid in the <i>Chart area</i> background.
<i>Show time bar</i>	The scroll bar at the bottom of the <i>Chart area</i> is available as long as this box is checked.
<i>Show tracks list</i>	The <i>Variables</i> window is shown as long as this box is checked, otherwise the <i>Chart area</i> extends to the bottom of the graphic trigger window.

**Values**

Control	Description
<i>Horizontal scale</i> 	Number of samples per unit of the x-axis. By unit of the x-axis the space is meant between two vertical lines of the background grid.
<i>Buffer size</i>	Number of samples to acquire. When you open the option tab, after having dragged-in all the variables you want to watch, you can read a default number in this field, representing the maximum number of samples you can collect for each variable. You can therefore type a number which is less or equal to the default one.

**Tracks**

This tab allows you to define some graphic properties of the plot of each variable. To select a variable, click its name in the *Track list* column.

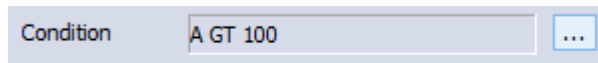
Control	Description
<i>Unit</i>	Unit of measurement, printed in the table of the <i>Variables</i> window.
<i>Value/div</i>	$\Delta$ value per unit of the y-axis. By unit of the y-axis is meant the space between two horizontal lines of the background grid.
<i>Hide</i>	Check this flag to hide selected track on the graph.

Push *Apply* to make your changes effective, or push *OK* to apply your changes and to close the options tab.

**User-defined condition**

If you define a condition by using this control, the sampling process does not start until that condition is satisfied. Note that, unlike trigger windows, once data acquisition begins, samples are taken every time the window manager is triggered, regardless of the user condition being still true or not.

After you enter a condition, the control displays its simplified expression.



**9.6.2 DEBUGGING WITH THE GRAPHIC TRIGGER WINDOW**

The graphic trigger window tool allows you to select a set of variables and to have them sampled synchronously and their curve displayed in a special pop-up window.

**9.6.2.1 OPENING THE GRAPHIC TRIGGER WINDOW FROM AN IL MODULE**

Let us assume that you have an IL module, also containing the following instructions.

```

0001
0002      LD  a
0003      ADD b
0004      ST  a
0005
0006      LD  c
0007      ADD d
0008      ST  c
0009
0010      LD  k
0011      ADD 1
0012      ST  k
0013
0014

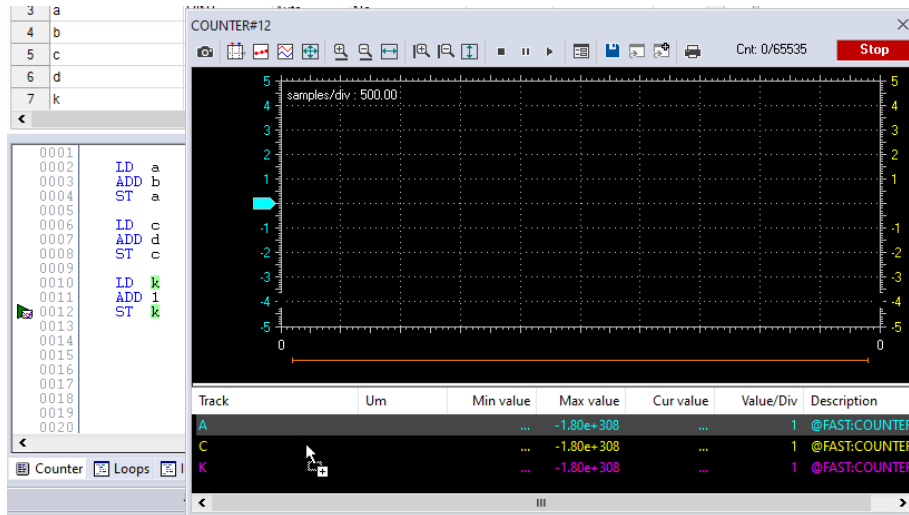
```

Let us also assume that you want to know the value of *a*, *c*, and *k*, just before the *ST k* instruction is executed. To do so, move the cursor to line 12 and click **Debug>Add/Remove graphic trigger**.

A green arrowhead appears next to the line number, and the graphic trigger window pops up.



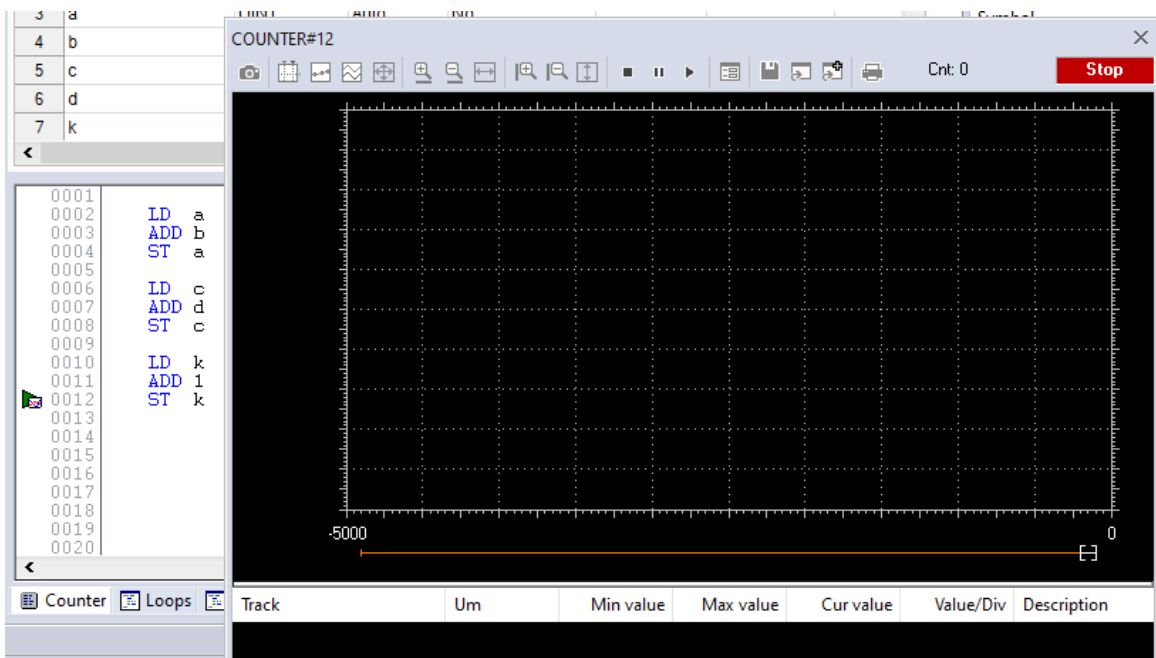




Not all the IL instructions support triggers. For example, it is not possible to place a trigger at the beginning of a line containing a `JMP` statement.

### 9.6.2.2 ADDING A VARIABLE TO THE GRAPHIC TRIGGER WINDOW FROM AN IL MODULE

In order to get the diagram of a variable plotted, you need to add it to the graphic trigger window. To this purpose, select a variable, by double clicking it, and then drag it into the *Variables* window. The variable now appears in the *Track* column.



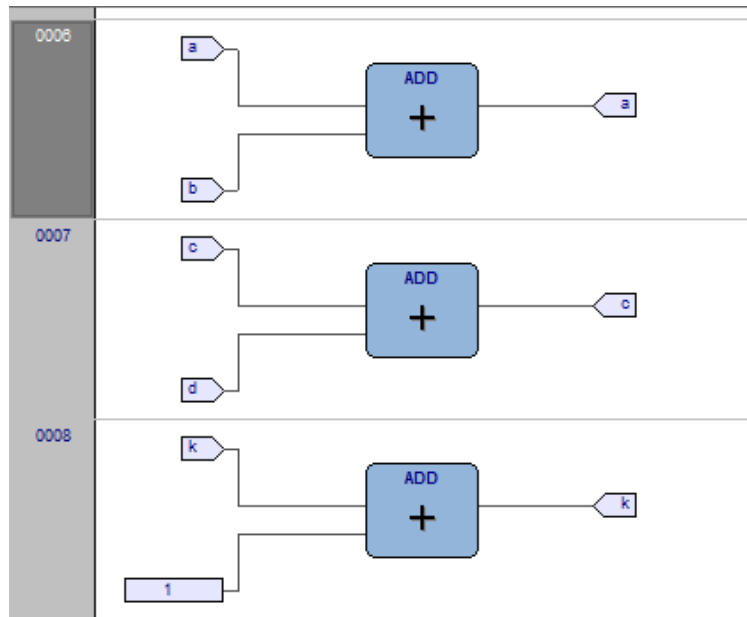
The same procedure applies to all the variables you wish to inspect.

Once the first variable is dropped into a graphic trace, the *Graphic properties* window is automatically shown and allows the user to setup sampling and visualization properties.

### 9.6.2.3 OPENING THE GRAPHIC TRIGGER WINDOW FORM AN FBD MODULE

Let us assume that you have an FBD module, also containing the following instructions.





Let us also assume that you want to know the values of *a*, *c*, and *k*, just before the last instruction is executed.

Provided that you can never place a trigger in a block representing a variable such as

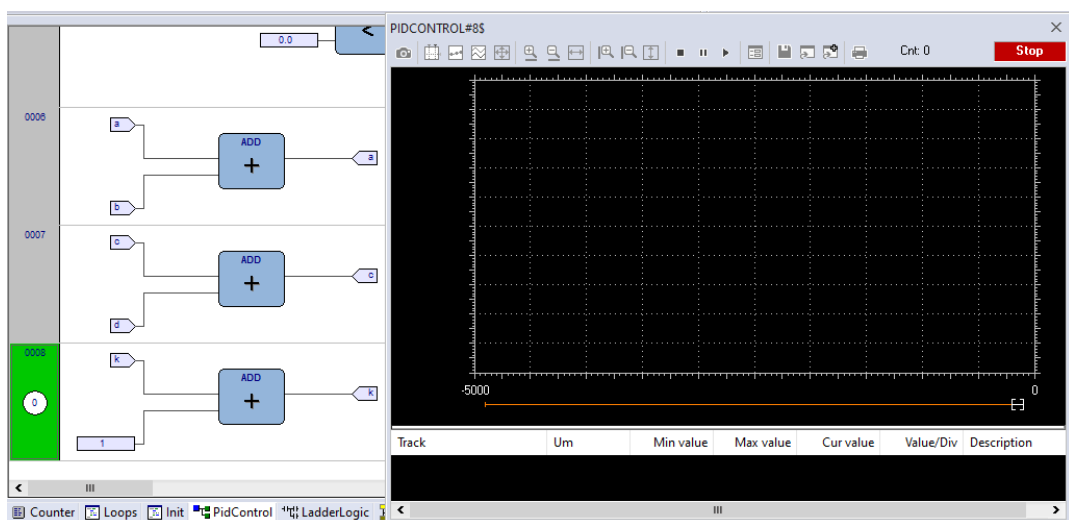


you must select the first available block preceding the selected variable. In the example of the above figure, you must move the cursor to network 8, and click the ADD block.

Now click **Debug > Add/Remove graphic trigger**.

Else, you can also insert trigger on the whole line; this means you'll hit the trigger just before the first instruction of that line is executed. To do so, select the row by clicking the gray button on the left (the one with the row number inside) and add the trigger.

In both cases the colour of the selected block will turn to green, a white circle with the trigger ID number inside will appear in the middle of the block, and the related trigger interface window will pop up.



When preprocessing the FBD source code, compiler translates it into IL instructions. The ADD instruction in network 8 is expanded to:

```
LD k
```



```
ADD 1
ST k
```

When you add a trigger to an FBD block, you actually place the trigger before the first statement of its IL equivalent code.

**9.6.2.4 ADDING A VARIABLE TO THE GRAPHIC TRIGGER WINDOW FROM AN FBD MODULE**

In order to watch the diagram of a variable, you need to add it to the trigger window. Let us assume that you want to see the plot of the variable *k* of the FBD code in the figure below.

To this purpose, click `Edit>Watch mode`.

The cursor will become as follows.

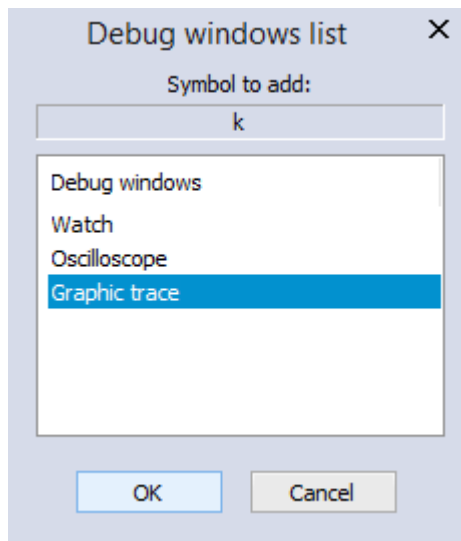


Now you can click the block representing the variable you wish to be shown in the graphic trigger window.

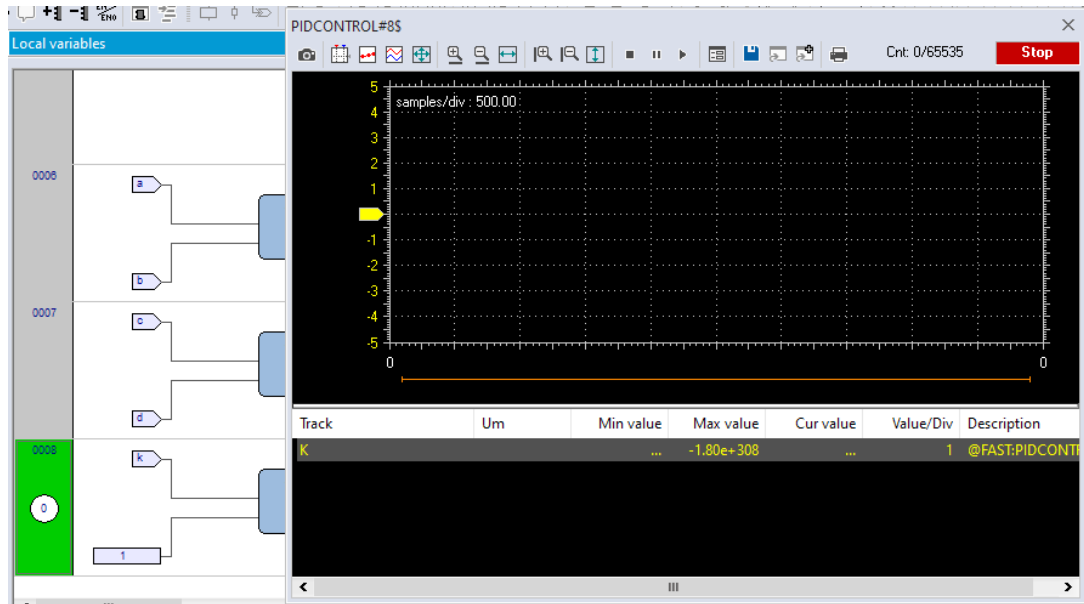
In the example we are considering, click the button block.



A dialog box appears listing all the currently existing instances of debug windows, and asking you which one is to receive the object you have just clicked.



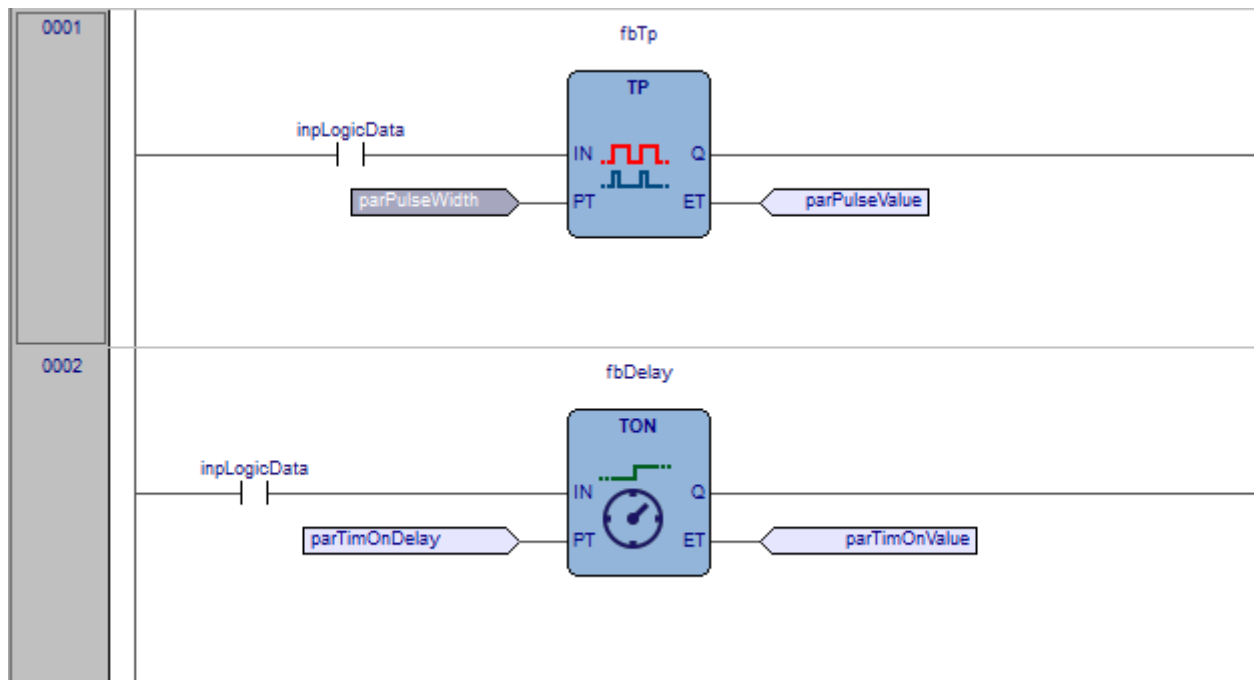
In order to plot the curve of variable *k*, select *Graphic Trace* in the *Debug windows* column, then press *OK*. The name of the variable is now printed in the *Track* column.



The same procedure applies to all the variables you wish to inspect. Once you have added to the *Graphic watch* window all the variables you want to observe, you can click `Edit>Insert/Move mode`, in order to restore the original cursor. Once the first variable is dropped into a graphic trace, the *Graphic properties* window is automatically shown and allows the user to setup sampling and visualization properties.

### 9.6.2.5 OPENING THE GRAPHIC TRIGGER WINDOW FROM AN LD MODULE

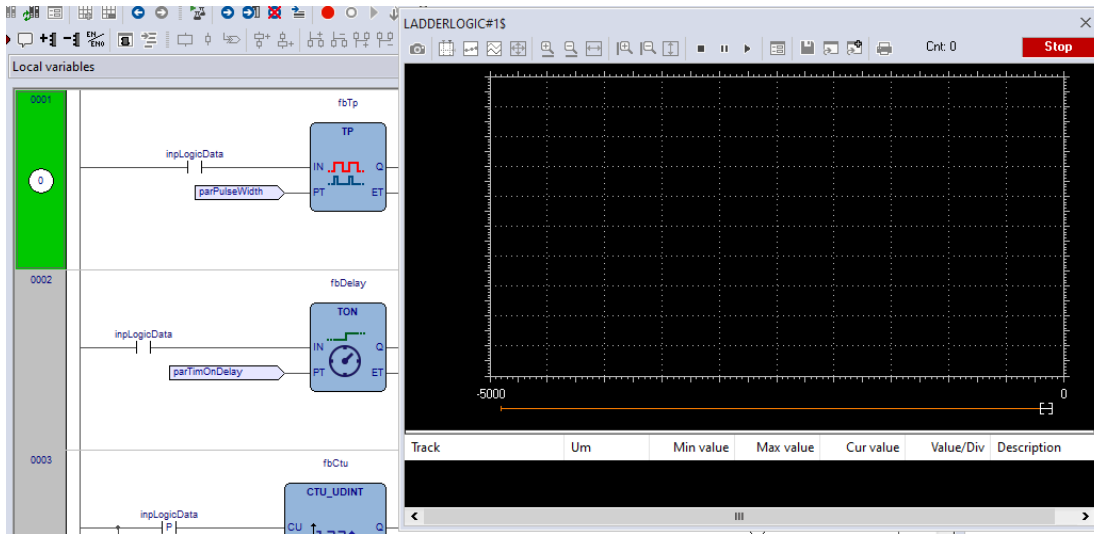
Let us assume that you have an LD module, also containing the following instructions.



You can place a graphic trigger directly on a block, a contact, a coil or the entire row. Let us assume that you want to know the value of some variables every time the processor reaches network number 1. First select the network number 1 by clicking on the grey area on its left, the one containing the network number. Now you can click `Debug>Add/Remove graphic trigger`.



In both cases, the grey raised button containing the network number turns to green, and a white circle with the number of the trigger inside appears in the middle of the button, while the related trigger interface window pops up.



Note that unlike the other languages supported by LogicLab, LD does not allow you to insert a trigger before a single contact or coil, as it lets you select only an entire network. Thus the variables in the *Graphic trigger* window will be sampled every time the processor reaches the beginning of the selected network.

**9.6.2.6 ADDING A VARIABLE TO THE GRAPHIC TRIGGER FROM AN LD MODULE**

In order to watch the diagram of a variable, you need to add it to the *Graphic trigger* window. Let us assume that you want to see the plot of the variable *b* in the LD code represented in the figure below.

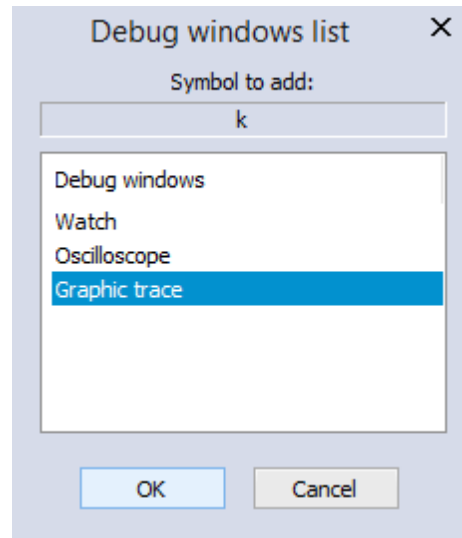
To this purpose, click **Edit>Watch mode**.

The cursor will become as follows.

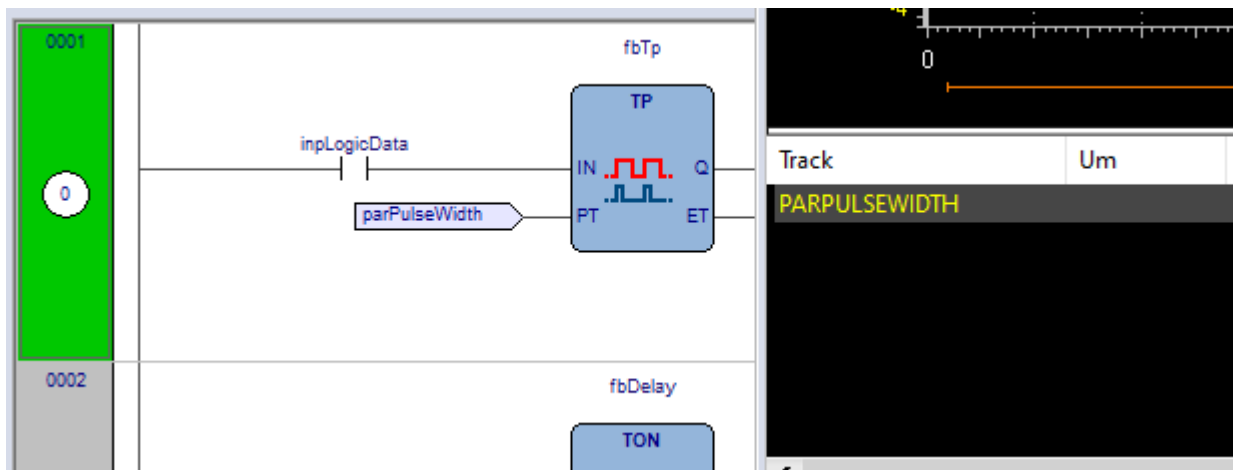


Now you can click the item representing the variable you wish to be shown in the *Graphic trigger* window.

A dialog box appears listing all the currently existing instances of debug windows, and asking you which one is to receive the object you have just clicked.



In order to plot the curve of variable *b*, select *Graphic trace* in the *Debug windows* column, then press *OK*. The name of the variable is now printed in the *Track* column.



The same procedure applies to all the variables you wish to inspect.

Once you have added to the *Graphic watch* window all the variables you want to observe, you can click [Edit>Insert/Move mode](#), so as to restore the original shape of the cursor.

Once the first variable is dropped into a graphic trace, the *Graphic properties* window is automatically shown and allows the user to setup sampling and visualization properties.

### 9.6.2.7 OPENING THE GRAPHIC TRIGGER WINDOW FROM AN ST MODULE

Let us assume that you have an ST module, also containing the following instructions.

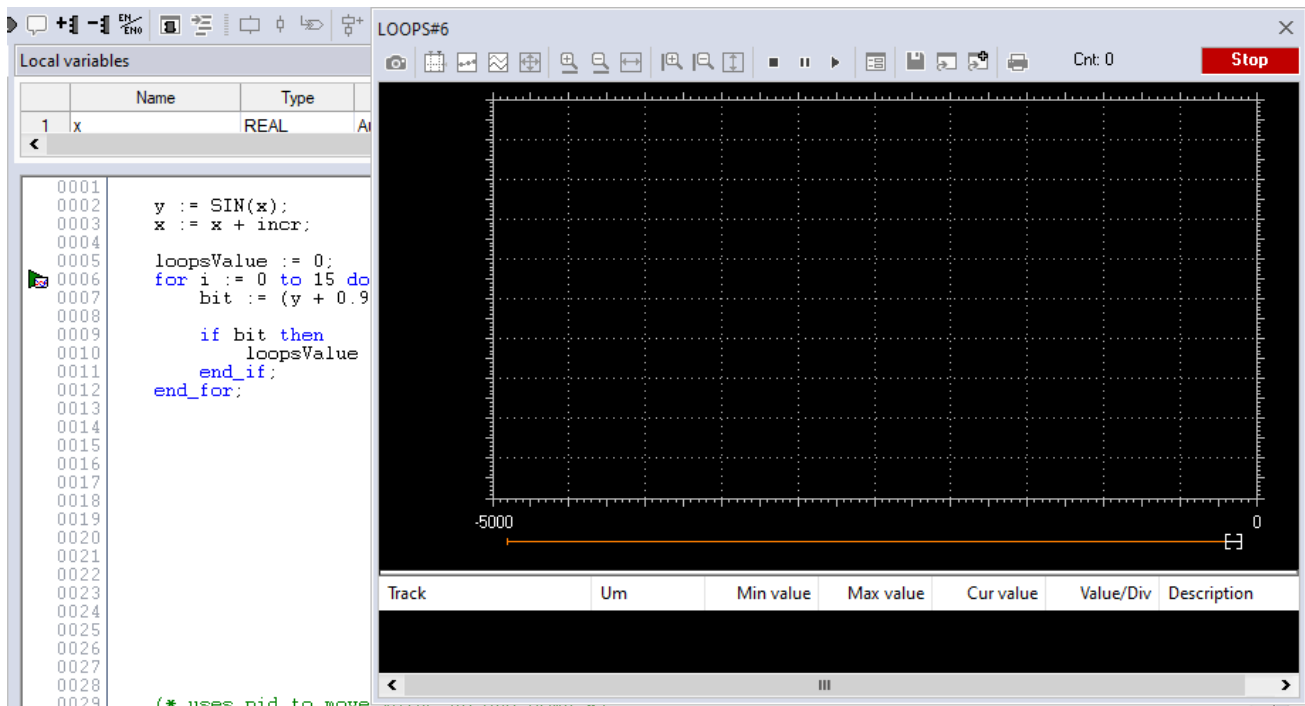


```

0001
0002   y := SIN(x);
0003   x := x + incr;|
0004
0005   loopsValue := 0;
0006   for i := 0 to 15 do
0007       bit := (y + 0.9) > (0.125 * TO_REAL(i));
0008
0009       if bit then
0010           loopsValue := loopsValue or RotateBit(i);
0011       end_if;
0012   end_for;
0013

```

Let us also assume that you want to know the value of *x* and *y*, just before the *for* cycle is executed. To do so, move the cursor to line 6 and click **Debug>Add/Remove graphic trigger** ; a green arrowhead appears next to the line number, and the *Graphic trigger* window pops up.



Not all the ST instructions support triggers. For example, it is not possible to place a trigger on a line containing a terminator such as `END_IF`, `END_FOR`, `END_WHILE`, etc.

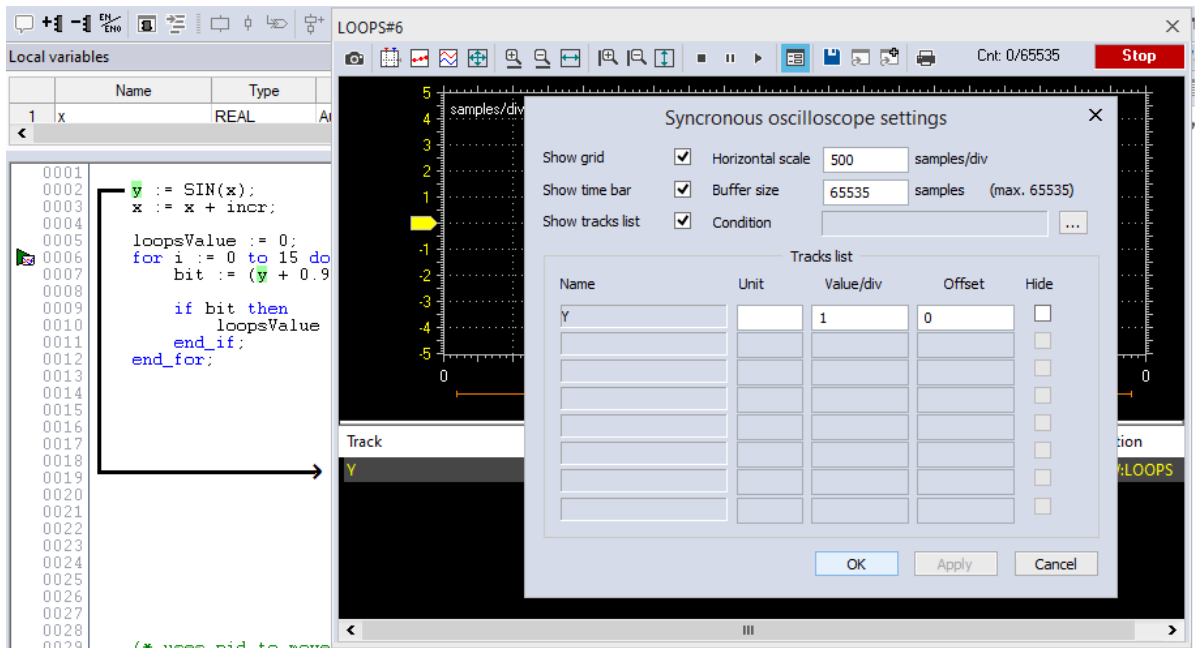
### 9.6.2.8 ADDING A VARIABLE TO THE GRAPHIC TRIGGER WINDOW FROM AN ST MODULE

In order to get the diagram of a variable plotted, you need to add it to the *Graphic trigger* window. To this purpose, select a variable, by double clicking it, and then drag it into the *Variables* window, that is the lower white box in the pop-up window. The variable now appears in the *Track* column.

The same procedure applies to all the variables you wish to inspect.

Once the first variable is dropped into a graphic trace, the *Graphic properties* window is automatically shown and allows the user to setup sampling and visualization properties.





**9.6.2.9 REMOVING A VARIABLE FROM THE GRAPHIC TRIGGER WINDOW**

If you want to remove a variable from the Graphic trigger window, select it by clicking its name once, then press the *Del* key.

**9.6.2.10 CLOSING THE GRAPHIC TRIGGER WINDOW AND REMOVING THE TRIGGER**

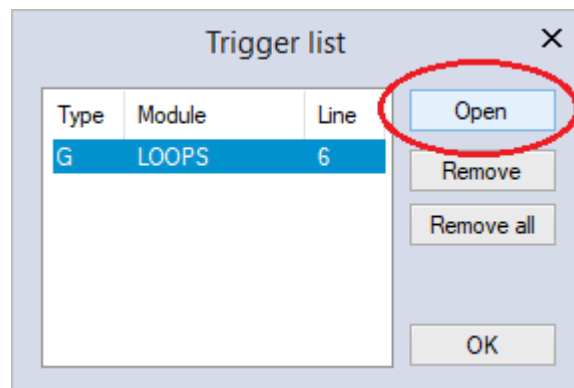
At the end of a debug session with the graphic trigger window you can choose between the following options:

- Closing the *Graphic trigger* window.
- Removing the trigger.
- Removing all the triggers.

**Closing the graphic trigger window**

If you close a graphic trigger window you’re just hiding its interface window, the trigger is still active. You will still see the green arrowhead (or the network highlighted).

You can reopen the interface window, to resume working with the same graphic trigger, by opening the *Trigger list* window, select the record referred to that trigger, and click the *Open* button.



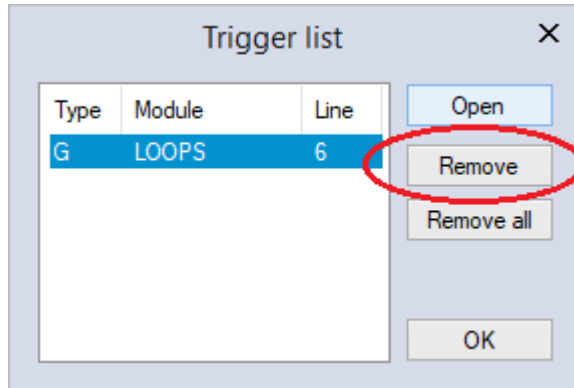
**Removing the trigger**

If you choose this option, you’ll completely remove the code both of the window manager and of its trigger. To do so: open the *Trigger list* window, select the record (having type





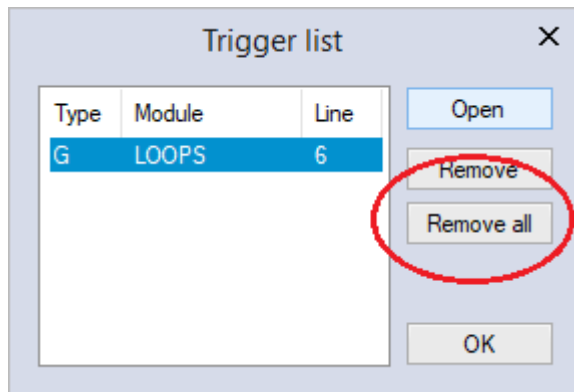
g) and click the *Remove* button.



Alternatively, you can move the cursor to the line (if the module is in IL or ST), or click the block (if the module is in FBD or LD) where you placed the trigger. Now select [Debug>Add/Remove graphic trigger](#)

**Removing all the triggers**

This way you'll remove all the existing triggers at once, regardless for which records are selected. To do so: open the *Trigger list* window and click on the *Remove all* button.



**9.7 BREAKPOINTS**

**9.7.1 THE BREAKPOINT TOOL**

The Breakpoint tool allows you to halt the program execution when it reaches a certain location. When a breakpoint is hit you can take all the time you want to watch other variables values or check every condition you need to verify before resuming the program execution; with breakpoints you also have the option of executing only the next instruction, running your program step by step.

Breakpoints cannot be used with SFC modules.

**9.7.1.1 PRE-CONDITIONS TO SET A BREAKPOINT**

**No need for special compilation**

LogicLab debugging tools operate at run-time. Thus, unlike other programming languages such as C++, the compiler does not need to be told whether or not to support breakpoints: given a PLC code, the compiler's output is unique, and there is no distinction between debug and release version.

**Memory availability**



A breakpoint takes a segment in the application code sector, having a well-defined length. Obviously, in order to set a breakpoint, it is necessary that a sufficient amount of memory is available, otherwise an error message appears.

### Incompatibility with graphic trigger windows






A graphic trigger window takes the whole free space of the application code sector. Therefore, once such a debugging tool has been started, it is not possible to add any breakpoint, and an error message appears if you attempt to set one. Once the graphic trigger window is eventually closed, breakpoints are enabled again.

Note that all the breakpoints existing before the starting of a graphic trigger window will keep working normally. You are simply not allowed to add new ones.

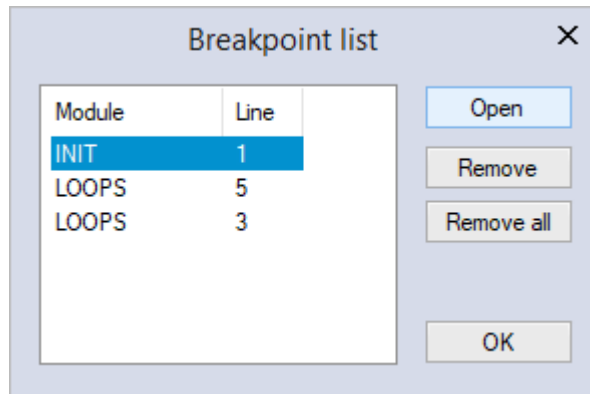
## 9.7.2 SET AND REMOVE A BREAKPOINT

From the *Debug* menu you can select the appropriate voice for work with breakpoints. Breakpoints can be palced only if a valid connection is established and the PLC is currently executing.

From the Debug menu you can choose the following voices:

Command	Icon in debug toolbar	Description
<i>Set/Remove breakpoint</i>		In order to actually set a breakpoint, select the point of the PLC code where to insert the relative breakpoint and then select this voice or use the shortcut pressing <i>F12</i> . Do the same to remove the breakpoint.
<i>Run</i>		One a breakpoint is hit, the program execution will be stopped and its state will become <i>HALTED</i> ; in order to resume the normal program execution select this voice.
<i>Step</i>		Once a breakpoint is hit, the program execution will be stopped and its state will become <i>HALTED</i> ; select this voice to execute only the next instruction.
<i>Remove all breakpoints</i>		Selecting this voice will cause all the existing breakpoints to be removed simultaneously.
<i>Breakpoint list</i>		Select this voice to open a dialog listing all the existing breakpoints.

If you have set several breakpoint, you may want to keep control over them using the *Breakpoints list* window. Selecting the *Breakpoints list* voice the following window will be opened:



In this window you can see the list of all active breakpoints, the program name and the relative code line where they are set is shown in the column *Module* and *Line* respectively. Once selected a breakpoint from the list, you can:

- *press the Open button*: the relative program will be opened and the line with the selected breakpoint will be highlighted; this feature allows you to fast navigate to the desired breakpoint. You can obtain the same result by double clicking the breakpoint instead of selecting it and then pressing the *Open* button
- *press Remove button*: this will actually remove the selected breakpoint.
- *press Remove all button*: this will remove all breakpoints in every program.

### 9.7.3 WORKING WITH BREAKPOINTS

#### 9.7.3.1 USE BREAKPOINTS WITH IL AND ST LANGUAGES

Breakpoints work the same way in both IL and ST languages; in order to set a new breakpoint, move to the code line where you want to halt the program execution, then choose

• *Debug>Set/Remove breakpoint* .

At the far left side of the code line, left to the line number, a red circle will appear, meaning that there's a breakpoint on that line.

```

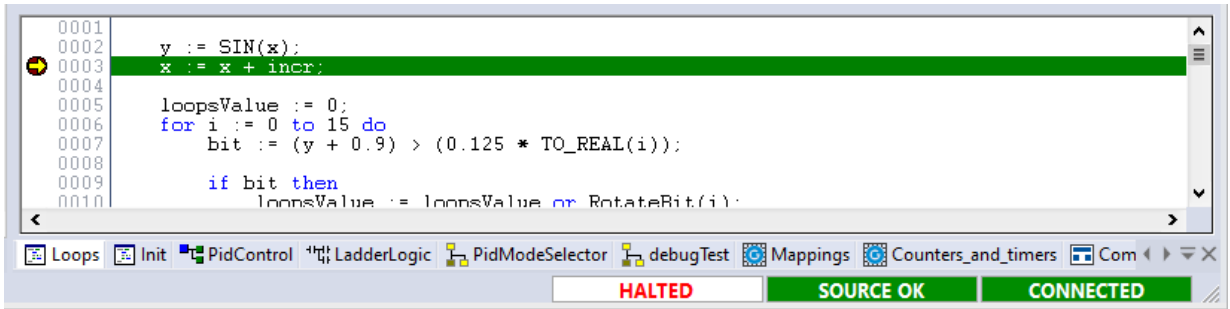
0004
● 0005   loopsValue := 0;]
0006   for i := 0 to 15 do
0007       bit := (y + 0.9) > (0.125 * TO_REAL(i));
0008
0009       if bit then
0010           loopsValue := loopsValue or RotateBit(i);
0011       end_if;
0012   end_for;
0013

```

Note that you cannot set breakpoints on every instruction, for example you cannot set a breakpoint on an *end\_if* statement and similar.

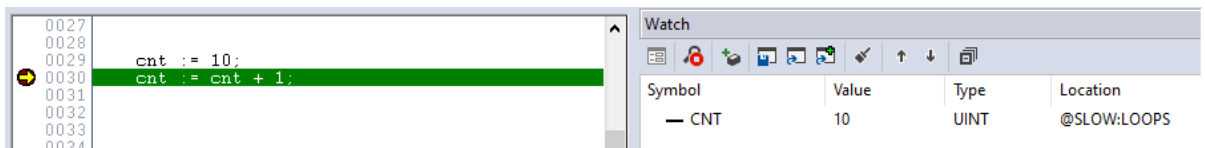
When a breakpoint is hit (which means that the code execution has reached the line where the breakpoint is set), the whole line is highlighted and a yellow arrow appears inside the red circle, to indicate that the program is waiting to execute that instruction.

Also the program execution state will change, in the far bottom-right corner, it will change from running to halted.



Note that when a breakpoint is hit, the program execution is halted *before* that instruction took place.

In the following image you can see that if we halt the program execution on an addition instruction, the addition has not yet taken place.



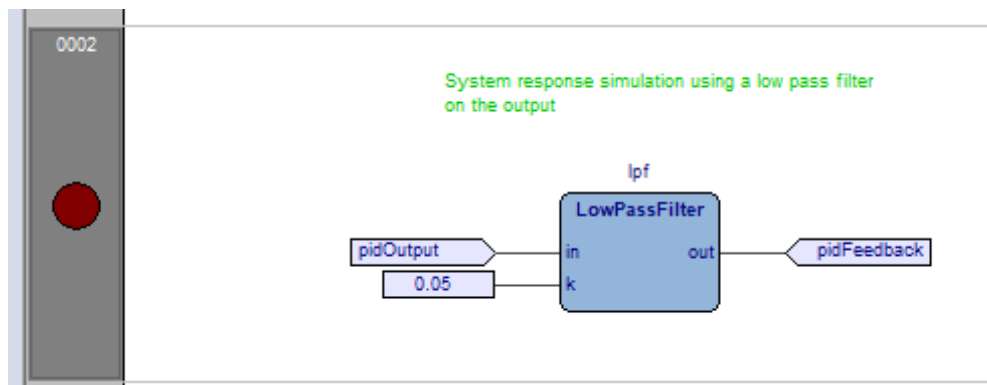
Selecting **Debug > Run** the program execution will be resumed until another breakpoint is hit, or the same breakpoint will be hit again after an entire cycle has taken place.

Selecting **Debug > Step** the code line where the program execution has halted, will be executed, but the program will halt on the very next instruction. In other words, the result is the same as setting a new virtual breakpoint to the next instruction and then resume the execution.

### 9.7.3.2 USE BREAKPOINTS WITH LD AND FBD LANGUAGES

Breakpoints work the same way in both LD and FBD languages; in order to set a new breakpoint, move to the line where you want to halt the program execution, then choose **Debug > Set/remove breakpoint**.

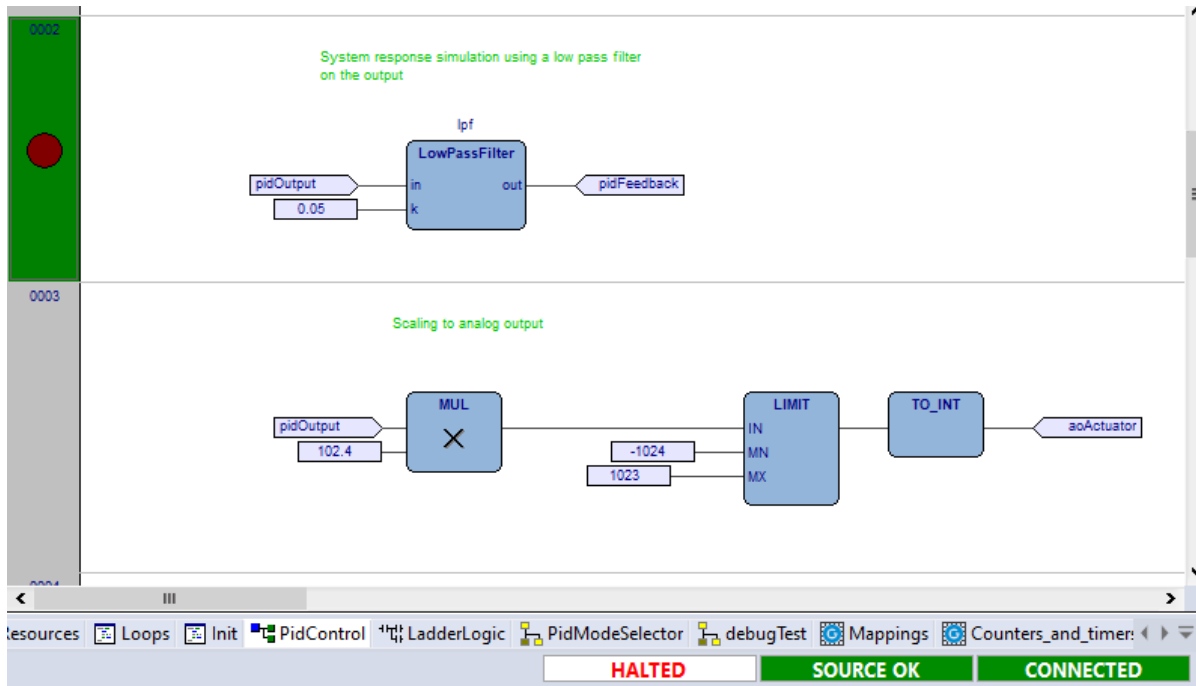
On the header of the line, the gray raised button with the line number on it, will now appear a red circle, meaning that there's a breakpoint on that line.



When a breakpoint is hit (which means that the code execution has reached the line where the breakpoint is set), the header is highlighted to indicate that the program is waiting to execute that instruction.

Also the program execution state will change, in the far bottom-right corner, it will change from running to halted.





Note that, as shown with the textual languages, when a breakpoint is hit, the program execution is halted *before* that instruction took place.

Selecting **Debug > Run** the program execution will be resumed until another breakpoint is hit, or the same breakpoint will be hit again after an entire cycle has took place. Selecting **Debug > Step** the line where the program execution has halted, will be executed, but the program will halt on the very next instruction. In other words, the result is the same as setting a new virtual breakpoint to the next instruction and then resume the execution.

### 9.7.4 REMOVING BREAKPOINTS

To remove a breakpoint you can either select it from the *breakpoints list* window and press the *Remove* button, or you can move to the line where the breakpoint is and select **Debug > Set/remove breakpoint**.

If you have done using breakpoints and you wish to remove them all with a single command, you can either open the *breakpoints list* window and press the *Remove all* button, or you can select the **Debug > Remove all breakpoints**.








## 10. LOGICLAB REFERENCE


### 10.1 MENUS REFERENCE

In the following tables you can see the list of all LogicLab's commands. However, since LogicLab has a multi-document interface (MDI), you may find some disabled commands or even some unavailable menus, depending on what kind of document is currently active.














#### 10.1.1 FILE MENU

Command	Icon	Key	Description
<i>New project</i>			Creates a new LogicLab project.
<i>Open project</i>			Opens an existing LogicLab project.
<i>Import project from target</i>			Imports sources project from target device.
<i>View project (read only)</i>			Opens an existing LogicLab project in read-only mode.
<i>Save project</i>			Saves the current open project.
<i>Save project As</i>			Saves the current open project specifying new name, location and extension.
<i>Close project</i>			Closes the open project.
<i>New text file</i>			Opens a blank new generic text file.
<i>Open file</i>		<i>Ctrl+O</i>	Opens an existing file, whatever its extension. The file is displayed in the text editor. Anyway, if you open a project file, you actually open the LogicLab project it refers to.
<i>Save</i>		<i>Ctrl+S</i>	Saves the document of the currently active window.
<i>Close</i>			Closes the document of the currently active window.
<i>Options</i>			Opens the LogicLab options dialog box.
<i>Print</i>		<i>Ctrl+P</i>	Prints the document of the currently active window.
<i>Print preview</i>			Creates a preview of the document of the currently active window, ready to be printed.
<i>Print project</i>			Prints all the documents making up the project.
<i>Printer setup</i>			Opens the Printer setup dialog box.
<i>..recent..</i>			Lists a set of project file recently opened.
<i>Exit</i>			Closes LogicLab.












#### 10.1.2 EDIT MENU

Command	Icon	Key	Description
<i>Undo</i>		<i>Ctrl+Z</i>	Cancels last action made in the document.



Command	Icon	Key	Description
<i>Redo</i>		<i>Ctrl+Y</i>	Restores the last action cancelled by Undo.
<i>Cut</i>		<i>Ctrl+X</i>	Removes the selected items from the active document and stores them in a system buffer.
<i>Copy</i>		<i>Ctrl+C</i>	Copies the selected items to a system buffer.
<i>Paste</i>		<i>Ctrl+V</i>	Pastes in the active document the contents of the system buffer.
<i>Delete</i>		<i>Del</i>	Deletes the selected item.
<i>Delete line</i>		<i>Ctrl+E</i>	Deletes the whole source code line.
<i>Go to symbol</i>		<i>Shift+F12</i>	Allows you to move through the results of a symbol search
<i>Find in project</i>		<i>Ctrl+Shift+F</i>	Opens the Find in project dialog box.
<i>Bookmarks...</i>			
<i>Add/Toggle</i>		<i>Ctrl+F2</i>	Adds a bookmark to mark lines. If a bookmark is already defined, removes it.
<i>Next</i>		<i>F2</i>	Goes to next defined bookmark
<i>Prev</i>		<i>Shift+F2</i>	Goes to previous defined bookmark
<i>Remove all</i>			Removes all defined bookmarks
<i>Go to line</i>		<i>Ctrl+G</i>	Allows you to quickly move to a specific line in the source code editor.
<i>Find</i>		<i>Ctrl+F</i>	Asks you to type a string and searches for its first instance within the active document from the current location of the cursor.
<i>Find next</i>		<i>F3</i>	Iterates between the results of the research, found by the <i>Find</i> command.
<i>Replace</i>		<i>Ctrl+H</i>	Allows you to automatically replace one or all the instances of a string with another string.
<i>Insert/Move mode</i>		<i>Spacebar</i>	Toggle between those two editing modes, used to insert or move blocks. You can switch from Insert/Move to Connection mode with the spacebar
<i>Connection mode</i>		<i>Spacebar</i>	Editing mode which allows you to draw logical wires to connect pins. You can switch from Insert/Move to Connection mode with the spacebar
<i>Watch mode</i>			Editing mode which allows you to add variables to any debugging tool.
<i>Zoom in</i>		<i>Ctrl + mouse wheel</i>	Increase zoom level inside the active POU.
<i>Zoom out</i>		<i>Ctrl + mouse wheel</i>	Decrease zoom level inside the active POU.

### 10.1.3 VIEW MENU




Command	Icon	Key	Description
<i>Toolbar</i>			
<i>Main</i>			Shows or hides the <i>Main</i> toolbar.
<i>Project</i>		<i>Ctrl+J</i>	Shows or hides the <i>Project</i> toolbar.
<i>Status bar</i>			Shows or hides the <i>Status</i> bar.
<i>Debug</i>		<i>Ctrl+B</i>	Shows or hides the <i>Debug</i> toolbar.
<i>FBD bar</i>		<i>Ctrl+D</i>	Shows or hides the <i>FBD</i> toolbar.
<i>LD bar</i>		<i>Ctrl+A</i>	Shows or hides the <i>LD</i> toolbar.
<i>SFC bar</i>		<i>Ctrl+Q</i>	Shows or hides the <i>SFC</i> toolbar.
<i>Network</i>		<i>Ctrl+N</i>	Shows or hides the <i>Network</i> toolbar.
<i>Tool windows</i>			
<i>Local variables</i>			Shows or hides the local variables window for the active POU.
<i>Project</i>		<i>Ctrl+W</i>	Shows or hides the <i>Workspace</i> window (also called <i>Workspace</i> window).
<i>Watch</i>		<i>Ctrl+T</i>	Shows or hides the <i>Watch</i> window.
<i>Properties window</i>			Shows or hide the properties window.
<i>Oscilloscope</i>		<i>Ctrl+K</i>	Shows or hides the <i>Oscilloscope</i> window.
<i>PLC run-time status</i>			Shows or hides the <i>PLC run-time</i> window.
<i>Operators and blocks</i>		<i>Ctrl+L</i>	Shows or hides the Operators and Blocks window
<i>Library Tree</i>			Shows or hides the <i>Libraries</i> window.
<i>Output</i>		<i>Ctrl+R</i>	Shows or hides the <i>Output</i> window.
<i>Cross Reference</i>			Shows or hides the cross reference window.
<i>Resources</i>			Add or remove the resources panel from the workspace.
<i>Catalog</i>			Shows or hides the catalog window.
<i>Full screen</i>		<i>Ctrl+U</i>	Expands the currently active document window to fill entire screen. ( <i>Esc</i> to exit from this mode).
<i>Grid</i>			Shows or hides a dotted grid in the background of graphical source code editors.


### 10.1.4 PROJECT MENU

Command	Icon	Key	Description
<i>New object</i>			














Command	Icon	Key	Description
<i>New program</i>			Creates a new program. A dialog is prompted in order to specify the new program properties.
<i>New function block</i>			Creates a new function block. A dialog is prompted in order to specify the new function block properties.
<i>New function</i>			Creates a new function block. A dialog is prompted in order to specify the new function properties.
<i>New variable</i>			
<i>Automatic</i>			Creates a new automatic variable. A dialog is prompted in order to specify the new variable properties.
<i>Mapped variable</i>		<i>Ctrl + Shift + M</i>	Creates a new mapped variable. A dialog is prompted in order to specify the new variable properties.
<i>Constant</i>			Creates a new constant. A dialog is prompted in order to specify the new constant properties.
<i>Retain</i>			Creates a new retain variable. A dialog is prompted in order to specify the new variable properties.
<i>New definition</i>			
<i>Enumeration</i>			Creates a new user defined type, of type enumeration.
<i>Interface</i>			Creates a new user defined type, of type interface.
<i>Macro</i>			Creates a new user defined type, of type macro.
<i>Structure</i>			Creates a new user defined type, of type structure.
<i>Subrange</i>			Creates a new user defined type, of type subrange.
<i>TypeDef</i>			Creates a new user defined type, of type typedef.
<i>Copy object</i>			Copies the object currently selected in the <i>Workspace</i> .
<i>Paste object</i>			Pastes the previously copied object.
<i>Duplicate object</i>			Duplicates the object currently selected in the <i>Workspace</i> , and asks you to type the name of the copy.
<i>Delete object</i>		<i>Del</i>	Deletes the currently selected object.
<i>View PLC object properties</i>		<i>Alt+Enter</i>	Shows properties and description of the currently selected object.
<i>Object browser</i>			Opens the <i>Object</i> browser, which lets you navigate between objects.
<i>Compile</i>		<i>F7</i>	Launches the LogicLab compiler.
<i>Recompile all</i>		<i>Ctrl+ Alt+F7</i>	Recompiles the project.
<i>Generate redistributable source module</i>			Generates an RSM file.
<i>Import objects</i>			Lets you import a LogicLab object from a library.
<i>Export object to library</i>			Lets you export a LogicLab object to a library.
<i>Library manager</i>			Opens the <i>Library</i> manager.

Command	Icon	Key	Description
<i>Refresh all libraries</i>			Reloads all libraries linked to the project.
<i>Select target...</i>			Lets you to select a new target for the project.
<i>Refresh current target</i>			Lets you update the target file for the same version of the target.
<i>Options...</i>			Opens the project options dialog.











### 10.1.5 ONLINE MENU

Command	Icon	Key	Description
<i>Set up communication...</i>			Lets you set the properties of the connection to the target.
<i>Connect</i>			LogicLab tries to establish a connection to the target.
<i>Download code</i>		F5	LogicLab checks if any changes have been applied since last compilation, if so compiles the project and then downloads the source code to the target.
<i>Download options</i>			Lets you set the properties of the source code downloaded to the target.
<i>Force target image upload</i>			If the target device is connected, lets you upload the img file.
<i>Force debug symbols upload</i>			If the target device is connected, lets you upload the debug symbols file.
<i>Halt</i>			Stops the PLC execution.
<i>Cold restart</i>			Restarts the PLC execution and both retain and non-retain variables will be reset.
<i>Warm restart</i>			Restarts the PLC execution and non-retain variables will be reset.
<i>Hot restart</i>			Restarts the PLC execution without any reset on variables.
<i>Reboot target</i>			Reboots the target.
<i>Read all logs again</i>			Reloads all remote logs from target.





### 10.1.6 DEBUG MENU

Command	Icon	Key	Description
<i>Simulation mode</i>			Open/close the integrated simulation environment.
<i>Start/Stop watch value</i>			Starts or stops (toggle) the evaluation of the symbols added in the watch window.
<i>Add symbol to watch</i>		F8	Adds a symbol to the <i>Watch</i> window.














Command	Icon	Key	Description
Add symbol to a debug window		F10	Adds a symbol to a <i>debug</i> window.
Insert new item into a debug window		Shift+F10	Inserts a new item into a <i>debug</i> window.
Quick watch symbol		Shift+F8	Opens a new window which allows you to see the current value of the selected symbol (not refreshing) and to force a new value.
Live debug mode			If debug mode is running, starts or stops (toggle) the live debug mode.
Add/remove text trigger		F9	Adds/removes a text trigger.
Add/remove graphic trigger		Shift+F9	Adds/removes a graphic trigger.
Remove all triggers		Ctrl+Shift+F9	Removes all the active triggers.
Trigger list		Ctrl+I	Lists all the active triggers.
Run			Restarts program after a breakpoint is hit.
Step			Restarts program executing only one instruction, after a breakpoint is hit.
Add/Remove breakpoint		F12	Adds or removes a breakpoint.
Remove all breakpoints			Removes all the active breakpoints.
Breakpoint list			Lists all the active breakpoints.

### 10.1.7 SCHEME MENU FOR FBD

















Command	Icon	Key	Description
Network			
New			
Top			Adds a blank network at the top of the active document.
Bottom			Adds a blank network at the bottom of the active document.
Before			Adds a blank network before the selected network in the active document.
After			Adds a blank network after the selected network in the active document.
Label			Assigns a label to the selected network, so that it can be indicated as the target of a jump instruction.
Object			
New			



Command	Icon	Key	Description
<i>Function</i>			Opens the object browser in order to choose a function to be added to the current active document.
<i>Function block</i>			Opens the object browser in order to choose a function block to be added to the current active document.
<i>Variable</i>		<i>shift+V</i>	Opens the object browser in order to choose a variable to be added to the current active document.
<i>Constant</i>		<i>shift+K</i>	Opens the object browser in order to choose a constant to be added to the current active document.
<i>Return</i>		<i>shift+R</i>	Adds a return statement into the selected network.
<i>Jump to label</i>		<i>shift+J</i>	Adds a jump statement into the selected network.
<i>Operator</i>			Opens the object browser in order to choose an operator to be added to the current active document.
<i>Comment</i>		<i>shift+M</i>	Adds a comment into the selected network.
<i>Instance name</i>			Opens the object browser in order to choose an operator to be added to the current active document.
<i>Open source</i>			<p>Opens the editor by which the selected object was created, and displays the relevant source code:</p> <ul style="list-style-type: none"> <li>- if the object is a program, or a function, or a function block, this command opens its source code;</li> <li>- if the object is a variable or a parameter, this command opens the corresponding variable editor;</li> <li>- if the object is a standard function or an operator, this command has no functionality.</li> </ul>
<i>Auto connect</i>			Toggle auto-connection mode, in order to connect automatically two blocks when they are close enough.
<i>Delete invalid connection</i>		<i>Ctrl+M</i>	Removes all invalid connections, represented by a red line in the active scheme.
<i>Increment pins</i>		<i>Ctrl+'+'</i>	Adds additional pins to the selected block in order to increase standard ones.
<i>Decrement pins</i>		<i>Ctrl+'-'</i>	Removes pins added by the <i>Increment pins</i> command.
<i>Enable EN/ENO pins</i>			Adds the <i>enable in/enable out</i> pins to the selected block. The code implementing the selected block will be executed only when the <i>enable in</i> signal is true. The <i>enable out</i> signal simply repeats the value of <i>enable in</i> , allowing you either to enable or to disable a set of blocks in cascade.
<i>Object properties</i>			Shows some properties of the selected block.



## 10.1.8 SCHEME MENU FOR LD










Command	Icon	Key	Description
<i>Network</i>			
<i>New</i>			
<i>Top</i>			Adds a blank network at the top of the active document.
<i>Bottom</i>			Adds a blank network at the bottom of the active document.
<i>Before</i>			Adds a blank network before the selected network in the active document.
<i>After</i>			Adds a blank network after the selected network in the active document.
<i>Label</i>			Assigns a label to the selected network in order to be used as target of a jump instruction.
<i>Object</i>			
<i>New</i>			
<i>Parallel contact</i>		<i>Shift+P</i>	Adds a contact parallel to the selected one.
<i>Serie contact</i>		<i>Shift+C</i>	Adds a contact in series to the selected one.
<i>Coil</i>		<i>Shift+O</i>	Adds a Coil into the selected network.
<i>Block</i>		<i>Shift+B</i>	Opens the object browser in order to choose which block should be added to the current active document.
<i>Constant</i>		<i>Shift+K</i>	Opens the object browser in order to choose a constant to be added to the current active document.
<i>Return</i>		<i>Shift+R</i>	Adds a Return statement into the selected network.
<i>Jump</i>		<i>Shift+J</i>	Adds a jump statement into the selected network.
<i>Variable</i>		<i>Shift+V</i>	Opens the object browser in order to choose a variable to be added to the current active document.
<i>Expression</i>		<i>Shift+E</i>	Adds an expression into the selected network.
<i>New branch</i>			Creates new branch after the current position.
<i>Comment</i>		<i>Shift+M</i>	Adds a comment into the selected network.
<i>Instance name</i>			Lets you assign a name to an instance of the selected function block.
<i>Open source</i>			Opens the editor by which the selected object was created, and displays the relevant source code: <ul style="list-style-type: none"> <li>- if the object is a program, or a function, or a function block, this command opens its source code;</li> <li>- if the object is a variable or a parameter, this command opens the corresponding variable editor;</li> <li>- if the object is a standard function or an operator, this command has no functionality.</li> </ul>

Command	Icon	Key	Description
<i>Open object</i>		<i>O</i>	Changes the selected object into an open contact object.
<i>Negated object</i>		<i>C</i>	Changes the selected object into a negated contact object.
<i>Positive object</i>		<i>P</i>	Changes the selected object into a positive contact object.
<i>Negative object</i>		<i>N</i>	Changes the selected object into a negative contact object.
<i>Set coil</i>		<i>S</i>	Changes the selected coil into a set coil.
<i>Reset coil</i>		<i>R</i>	Changes the selected coil into a reset coil.
<i>Increment pins</i>		<i>Ctrl+'+',</i>	Adds additional pins to the selected block in order to increase standard ones.
<i>Decrement pins</i>		<i>Ctrl+'-',</i>	Removes pins added by the <i>Increment pins</i> command.
<i>Enable EN/ENO pins</i>		<i>E</i>	Adds the <i>enable in/enable out</i> pins to the selected block. The code implementing the selected block will be executed only when the <i>enable in</i> signal is true. The <i>enable out</i> signal simply repeats the value of <i>enable in</i> , allowing you either to enable or to disable a set of blocks in cascade.
<i>Set output line</i>			Set selected pin as the output line of the block.
<i>Object properties</i>			Shows some properties of the selected block.



### 10.1.9 SCHEME SFC MENU

Command	Icon	Key	Description
<i>Object</i>			
<i>New</i>			
<i>Step</i>			Adds new step into the selected network.
<i>Transition</i>			Adds new transition into the selected network.
<i>Jump</i>			Adds new jump into the selected network.
<i>Modify</i>			
<i>Add pin to divergent transition</i>			Adds a divergent pin to the selected transition.
<i>Remove pin from divergent transition</i>			Removes a divergent pin to the selected transition.
<i>Add pin to convergent transition</i>			Adds a convergent pin to the selected transition.
<i>Remove pin from convergent transition</i>			Removes a convergent pin to the selected transition.



Command	Icon	Key	Description
<i>Add pin to simultaneous divergent transition</i>			Adds a simultaneous divergent pin to the selected transition.
<i>Remove pin from simultaneous divergent transition</i>			Removes a simultaneous divergent pin to the selected transition.
<i>Add pin to simultaneous convergent transition</i>			Adds a simultaneous convergent pin to the selected transition.
<i>Remove pin from simultaneous convergent transition</i>			Removes a simultaneous convergent pin to the selected transition.
<i>Add space before rightmost pin</i>			Adds a space before the rightmost pin.
<i>Remove space before rightmost pin</i>			Removes a space before the rightmost pin.
<i>Code Object</i>			
<i>New Action</i>			Adds an action in the active document.
<i>New Transition code</i>			Adds a transition in the active document.
<i>Auto connect</i>			Toggle auto-connection mode, in order to connect automatically two blocks when they are close enough.
<i>Delete invalid connection</i>		<i>Ctrl+M</i>	Removes all invalid connections, represented by a red line in the active scheme.

### 10.1.10 VARIABLES MENU

Command	Icon	Key	Description
<i>Insert</i>		<i>Ctrl+Shift+ins</i>	Adds a new row to the grid in the currently active editor.
<i>Delete</i>		<i>Del</i>	Deletes the variable in the selected row of the currently active table.
<i>Create multiple</i>			Lets you to create a set of multiple variables.

### 10.1.11 WINDOW MENU

Command	Icon	Key	Description
<i>Cascade</i>			Displaces all open documents in cascade, so that they completely overlap except for the caption.
<i>Tile</i>			The PLC editors area is split into frames having the same dimensions, depending on the number of currently open documents. Each frame is automatically assigned to one of such documents.



Command	Icon	Key	Description
<i>Arrange Icons</i>			Displaces the icons of the minimized documents in the bottom left-hand corner of the PLC editors area.
<i>Close all</i>			Closes all open documents.
<i>Windows...</i>		<i>Alt+W</i>	Shows the list of active open windows, allowing you to close or bring to focus specific window. Useful when you have a lot of open window.

### 10.1.12 TOOLS MENU

Command	Icon	Key	Description
<i>Custom tools</i>			Allows to see and execute previously loaded custom tools.

### 10.1.13 HELP MENU

Command	Icon	Key	Description
<i>Index</i>			Lists all the <i>Help keywords</i> and opens the related topic.
<i>Context</i>		<i>F1</i>	Context-sensitive help. Opens the topic related to the currently active window.
<i>About...</i>			Credits and version information.

## 10.2 TOOLBARS REFERENCE

In the following tables you can see the list of all LogicLab's toolbars. The buttons making up each toolbar are always the same, whatever the currently active document. However, some of them may produce no effect, if there is no logical relation to the active document.

### 10.2.1 MAIN TOOLBAR



### 10.2.2 FBD TOOLBAR



### 10.2.3 LD TOOLBAR





**10.2.4 SFC TOOLBAR****10.2.5 PROJECT TOOLBAR****10.2.6 NETWORK TOOLBAR****10.2.7 DEBUG TOOLBAR**

# 11. LANGUAGE REFERENCE

All LogicLab languages are IEC 61131-3 standard-compliant.

- Common elements
- Instruction list (IL)
- Function block diagram (FBD)
- Ladder diagram (LD)
- Structured text (ST)
- Sequential Function Chart (SFC).

Moreover, LogicLab implements some extensions:

- Pointers
- Macros.
- Object Oriented programming.

## 11.1 COMMON ELEMENTS

By common elements are intended those elements which are common to all programming languages of the IEC 61131-3 standard.

Note: definition and editing of most of the common elements (variables, structured elements, function blocks definitions etc.) are managed by LogicLab through specific editors, forms and tables.

The following paragraphs are meant to be a language specification. To correctly manage common elements refer to the LogicLab user guide.

### 11.1.1 BASIC ELEMENTS

#### 11.1.1.1 CHARACTER SET

Textual documents and textual elements of graphic languages are written by using the standard ASCII character set.

#### 11.1.1.2 COMMENTS

User comments are delimited at the beginning and at the end by special character combinations. For the textual languages (IL and ST) are allowed this format of comments:

- *(\* multi-lines comment \*)*
- *// single line comment*
- */\* multi-line comment \*/*

Same format comments cannot be nested.

Different format comments can be nested but it is a strongly not recommended practice.

```

0014      (*
0015          multiple
0016          line
0017          comment
0018      *)
0019
0020      // single line comment
0021
0022      /*
0023          multiple
0024          line
0025          comment
0026      */

```

For graphic languages, comments are inserted using dedicated commands; they are then translated, into the project file, using the first comment format *(\* comment \*)*.

Comments are permitted anywhere in the program, and they have no syntactic or semantic significance in any of the languages defined in this standard.



## 11.1.2 ELEMENTARY DATA TYPES

A number of elementary (i.e. pre-defined) data types is made available by LogicLab, all compliant with IEC 61131-3 standard.

Elementary data types, keyword for each data type, number of bits per data element, and range of values for each elementary data type are described in the following table.

Keyword	Description	Bits	Range / Notes
BOOL	Boolean	See note	0 to 1 - size of the BOOL data type depends on the target device processor. e.g. it is 1 bit long for devices that have a bit-addressable area.
SINT	Short integer	8	-128 to 127
USINT	Unsigned short integer	8	0 to 255
INT	Integer	16	-32768 to 32767
UINT	Unsigned integer	16	0 to 65536
DINT	Double integer	32	-2147483648 to 2147483647
UDINT	Unsigned Double integer	32	0 to 4294967295
LINT	Long integer	64	-9223372036854775808 to 9223372036854775807
ULINT	Unsigned Long integer	64	0 to 18446744073709551615
BYTE	Bit string of length 8	8	—
WORD	Bit string of length 16	16	—
DWORD	Bit string of length 32	32	—
LWORD	Bit string of length 64	64	—
REAL	Real number	32	-3.40E+38 to +3.40E+38
LREAL	Long Real Number	64	-1.7E+308 to +1.7E+308
STRING	String of characters encoded with UTF-8	—	Characters are delimited by single quotes ( 'abc' )
WSTRING	String of characters encoded with UTF-16	—	Characters are delimited by double quotes ( "abc" )
DATE	Date expressed in seconds, represented with format YYYY-MM-DD	32	1970-01-01 to 2038-01-19
LDATE	Date expressed in nanoseconds, represented with format YYYY-MM-DD	64	1970-01-01 to 2262-04-11
TIME	Time expressed in milliseconds represented with format dd_hh_mm_ss_ms	32	-24d_20h_31m_23s_648ms to 24d_20h_31m_23s_647ms

Keyword	Description	Bits	Range / Notes
LTIME	Time expressed in nanoseconds represented with format dd_hh_mm_ss_ms_us_ns	64	-106751d_23h_47m_16s_854ms_775us_808ns to 106751d_23h_47m_16s_854ms_775us_807ns
DATE_AND_TIME	Date expressed in seconds, represented with format YYYY-MM-DD-hh:mm:ss	32	1970-01-01-00:00:00 to 2038-01-19-03:14:07
LDATE_AND_TIME	Date expressed in nanoseconds, represented with format YYYY-MM-DD-hh:mm:ss.us	64	1970-01-01-00:00:00 to 2262-04-11-23:47:16.854
TIME_OF_DAY	Time of the day expressed in milliseconds, represented with format hh:mm:ss.ms	32	00:00:00 to 23:59:59.999
LTIME_OF_DAY	Time of the day expressed in nanoseconds, represented with format hh:mm:ss.ns	64	00:00:00 to 23:59:59.999999999
^ANY_TYPE	Referece to a variable of any type (IEC standard)	32	See 11.1.2.1 for further information

### 11.1.2.1 REFERENCES

References are a standard IEC type that act as a pointer to another variable (the referenced variable). Reference type works like the pointer type (see paragraph 11.7.2), but the pointer type is not a standard IEC type and has several less restriction, which make the pointer type more flexible but also more dangerous than the reference type.

The value of the reference is the address of the referenced variable; in order to access the data stored at the referenced address, a reference can be dereferenced.

Reference declaration requires the same syntax used in variable declaration, where the type name is the type name of the referenced variable with ^ sign after:

```
VAR
    <reference_var_name> : <referenced_var_type>^;
END_VAR
```

For example, the declaration of a reference to an INT shall be as follows:

```
VAR
    rInt : INT^;
END_VAR
```

Reference can be assigned with another reference or with an address; the special operator REF is available to retrieve the reference address of a variable.

```
rx := ry;      (* where rx and ry are reference of the same type *)
rx := REF(x); (* where rx is a reference to the same type of x *)
```



Accessing to the reference variable followed by the ^ sign, will dereference the variable:

```
rx := REF(x);      (* rx i a reference to x *)
rx^ := 10;        (* x vaule is now 10 *)
rx^ := rx^ + 1;   (* x value now is 11 *)
y := rx^;         (* y value now is 11, y is of the same type of x *)
```

For further information about references, please refer to IEC standard reference.

### 11.1.3 DERIVED DATA TYPES

Derived data types can be declared using the `TYPE...END_TYPE` construct. They can be used in variable declarations, in addition to the elementary data types.

Both single-element variables and elements of a multi-element variable, which are declared to be of derived data types, can be used anywhere where a variable of its parent type can be used.

#### 11.1.3.1 TYPEDEFS

The purpose of typedefs is to assign alternative names to existing types. There are not any differences between a typedef and its parent type, except the name.

Typedefs can be declared using the following syntax:

```
TYPE
  <enumerated data type name> : <parent type name>;
END_TYPE
```

For example, consider the following declaration, mapping the name `LONGWORD` to the IEC 61131-3 standard type `DWORD`:

```
TYPE
  longword : DWORD;
END_TYPE
```

#### 11.1.3.2 ENUMERATED DATA TYPES

An enumerated data type declaration specifies that the value of any data element of that type can only be one of the values given in the associated list of identifiers. The enumeration list defines an ordered set of enumerated values, starting with the first identifier of the list, and ending with the last.

Enumerated data types can be declared using the following syntax:

```
TYPE
  <enumerated data type name> : ( <enumeration list> );
END_TYPE
```

For example, consider the following declaration of two enumerated data types. Note that, when no explicit value is given to an identifier in the enumeration list, its value equals the value assigned to the previous identifier augmented by one.

```
TYPE
  enum1: (
    val1, (* the value of val1 is 0 *)
    val2,      (* the value of val2 is 1 *)
    val3 (* the value of val3 is 2 *)
  );
```



```

enum2: (
    k := -11,
    i := 0,
    j,          (* the value of j is ( i + 1 ) = 1 *)
    l := 5
);
END_TYPE

```

Different enumerated data types may use the same identifiers for enumerated values. In order to be uniquely identified when used in a particular context, enumerated literals may be qualified by a prefix consisting of their associated data type name and the # sign.

### 11.1.3.3 SUBRANGES

A subrange declaration specifies that the value of any data element of that type is restricted between and including the specified upper and lower limits.

Subranges can be declared using the following syntax:

```

TYPE
    <subrange name> : <parent type name> ( <lower limit>..<upper limit>
);
END_TYPE

```

For a concrete example consider the following declaration:

```

TYPE
    int_0_to_100 : INT (0..100);
END_TYPE

```

### 11.1.3.4 STRUCTURES

A STRUCT declaration specifies that data elements of that type shall contain sub-elements of specified types which can be accessed by the specified names.

Structures can be declared using the following syntax:

```

TYPE
    <structured type name> : STRUCT
        <declaration of structurestructure elements>
END_STRUCT;
END_TYPE

```

For example, consider the following declaration:

```

TYPE
    structure1 : STRUCT
        elem1 : USINT;
        elem2 : USINT;
        elem3 : INT;
        elem3 : REAL;
END_STRUCT;
END_TYPE

```

## 11.1.4 LITERALS

### 11.1.4.1 NUMERIC LITERALS

External representation of data in the various programmable controller programming languages consists of numeric literals.

There are two classes of numeric literals: integer literals and real literals. A numeric literal is defined as a decimal number or a based number.

Decimal literals are represented in conventional decimal notation. Real literals are distinguished by the presence of a decimal point. An exponent indicates the integer power of ten by which the preceding number needs to be multiplied to obtain the represented value. Decimal literals and their exponents can contain a preceding sign (+ or -).

Integer literals can also be represented in base 2, 8 or 16. The base is in decimal notation. For base 16, an extended set of digits consisting of letters A through F is used, with the conventional significance of decimal 10 through 15, respectively. Based numbers do not contain any leading sign (+ or -).

Boolean data are represented by the keywords `FALSE` or `TRUE`.

Numerical literal features and examples are shown in the table below.

Feature description	Examples
Integer literals	-12 0 123 +986
Real literals	-12.0 0.0 0.4560
Real literals with exponents	-1.34E-12 or -1.34e-12 1.0E+6 or 1.0e+6 1.234E6 or 1.234e6
Base 2 literals	2#11111111 (256 decimal) 2#11100000 (240 decimal)
Base 8 literals	8#377 (256 decimal) 8#340 (240 decimal)
Base 16 literals	16#FF or 16#ff (256 decimal) 16#E0 or 16#e0 (240 decimal)
Boolean <code>FALSE</code> and <code>TRUE</code>	<code>FALSE</code> <code>TRUE</code>

### 11.1.4.2 CHARACTER STRING LITERALS

A character string literal is a sequence of zero or more characters prefixed and terminated by the single quote character (`'`).

The three-character combination of the dollar sign (`$`) followed by two hexadecimal digits shall be interpreted as the hexadecimal representation of the eight-bit character code.

Example	Explanation
<code>''</code>	Empty string (length zero)
<code>'A'</code>	String of length one containing the single character A
<code>' '</code>	String of length one containing the <i>space</i> character
<code>'\$''</code>	String of length one containing the <i>single quote</i> character
<code>'"\$'</code>	String of length one containing the <i>double quote</i> character
<code>'\$R\$L'</code>	String of length two containing CR and LF characters
<code>'\$0A'</code>	String of length one containing the <code>LF</code> character

Two-character combinations beginning with the dollar sign shall be interpreted as shown in the following table when they occur in character strings.

Combination	Interpretation when printed
\$\$	Dollar sign
\$'	Single quote
\$L or \$l	Line feed
\$N or \$n	Newline
\$P or \$p	Form feed (page)
\$R or \$r	Carriage return
\$T or \$t	Tab

## 11.1.5 VARIABLES

### 11.1.5.1 FOREWORD

Variables provide a means of identifying data objects whose contents may change, e.g., data associated with the inputs, outputs, or memory of the programmable controller. A variable must be declared to be one of the elementary types. Variables can be represented symbolically, or alternatively in a manner which directly represents the association of the data element with physical or logical locations in the programmable controller's input, output, or memory structure.

Each program organization unit (POU) (i.e., each program, function, or function block) contains at its beginning at least one declaration part, consisting of one or more structuring elements, which specify the types (and, if necessary, the physical or logical location) of the variables used in the organization unit. This declaration part has the textual form of one of the keywords `VAR`, `VAR_INPUT`, or `VAR_OUTPUT` as defined in the keywords section, followed in the case of `VAR` by zero or one occurrence of the qualifiers `RETAIN`, `NON_RETAIN` or the qualifier `CONSTANT`, and in the case of `VAR_INPUT` or `VAR_OUTPUT` by zero or one occurrence of the qualifier `RETAIN` or `NON_RETAIN`, followed by one or more declarations separated by semicolons and terminated by the keyword `END_VAR`. A declaration may also specify an initialization for the declared variable, when a programmable controller supports the declaration by the user of initial values for variables.

### 11.1.5.2 STRUCTURING ELEMENT

The declaration of a variable must be performed within the following program structuring element:

```

KEYWORD [RETAIN] [CONSTANT]
  Declaration 1
  Declaration 2
  ...
  Declaration N
END_VAR

```

### 11.1.5.3 KEYWORDS AND SCOPE

Keyword	Variable usage
<code>VAR</code>	Internal to organization unit.
<code>VAR_INPUT</code>	Externally supplied.





Keyword	Variable usage
VAR_OUTPUT	Supplied by organization unit to external entities.
VAR_IN_OUT	Supplied by external entities, can be modified within organization unit.
VAR_EXTERNAL	Supplied by configuration via VAR_GLOBAL, can be modified within organization unit.
VAR_GLOBAL	Global variable declaration.

The scope (range of validity) of the declarations contained in structuring elements is local to the program organization unit (POU) in which the declaration part is contained. That is, the declared variables are accessible to other program organization units except by explicit argument passing via variables which have been declared as inputs or outputs of those units. The one exception to this rule is the case of variables which have been declared to be global.

Such variables are accessible to programs in any case, or via a VAR\_EXTERNAL declaration to function blocks. The type of a variable declared in a VAR\_EXTERNAL must agree with the type declared in the VAR\_GLOBAL block.

To give access to this variables to all type of POU, without using any keyword, you must enable this option in the code generation tab of the project options (see Paragraph 4.6.2).

There is an error if:

- any program organization unit attempts to modify the value of a variable that has been declared with the CONSTANT qualifier;
- a variable declared as VAR\_GLOBAL CONSTANT in a configuration element or program organization unit (the "containing element") is used in a VAR\_EXTERNAL declaration (without the CONSTANT qualifier) of any element contained within the containing element.

#### 11.1.5.4 QUALIFIERS

Qualifier	Description
CONST	The attribute CONST indicates that the variables within the structuring elements are constants, i.e. they have a constant value, which cannot be modified once the PLC project has been compiled.
RETAIN	The attribute RETAIN indicates that the variables within the structuring elements are retentive, i.e. they keep their value even after the target device is reset or switched off.

#### 11.1.5.5 SINGLE-ELEMENT VARIABLES AND ARRAYS

A single-element variable represents a single data element of either one of the elementary types or one of the derived data types.

An array is a collection of data elements of the same data type; in order to access a single element of the array, a subscript (or index) enclosed in square brackets has to be used. Subscripts can be either integer literals or single-element variables.

To easily represent data matrices, arrays can be multi-dimensional; in this case, a composite subscript is required, one index per dimension, separated by commas. The maximum number of dimensions allowed in the definition of an array is three.



### 11.1.5.6 DECLARATION SYNTAX

Variables must be declared within structuring elements, using the following syntax:

```
VarName1 : Typename1 [ := InitialVal1 ];
VarName2 AT Location2 : Typename2 [ := InitialVal2 ];
VarName3 : ARRAY [ 0..N ] OF Typename3;
```

where:

Keyword	Description
VarNameX	Variable identifier, consisting of a string of alphanumeric characters, of length 1 or more. It is used for symbolic representation of variables.
TypenameX	Data type of the variable, selected from elementary data types.
InitialValX	The value the variable assumes after reset of the target.
LocationX	See the next paragraph.
N	Index of the last element, the array having length N + 1.

### 11.1.5.7 LOCATION

Variables can be represented symbolically, i.e. accessed through their identifier, or alternatively in a manner which directly represents the association of the data element with physical or logical locations in the programmable controller's input, output, or memory structure.

Direct representation of a single-element variable is provided by a special symbol formed by the concatenation of the percent sign "%", a location prefix and a size prefix, and one or two unsigned integers, separated by periods (.).

```
%location size index.subindex
```

#### 1) location

The location prefix may be one of the following:

Location prefix	Description
I	Input location
Q	Output location
M	Memory location

#### 2) size

The size prefix may be one of the following:

Size prefix	Description
X	Single bit size
B	Byte (8 bits) size
W	Word (16 bits) size
D	Double word (32 bits) size
L	Long word (64 bits) size
R	Real (32 bits) size
Q	Lorg real (64 bits) size



### 3) index.index

This sequence of unsigned integers, separated by dots, specifies the actual position of the variable in the area specified by the location prefix.

#### Example:

Direct representation	Description
%MW4.6	Word starting from the first byte of the 7 <sup>th</sup> element of memory datablock 4.
%IX0.4	First bit of the first byte of the 5 <sup>th</sup> element of input set 0.

Note that the absolute position depends on the size of the datablock elements, not on the size prefix. As a matter of fact, %MW4.6 and %MD4.6 begin from the same byte in memory, but the former points to an area which is 16 bits shorter than the latter.

For advanced users only: if the index consists of one integer only (no dots), then it loses any reference to data blocks, and it points directly to the byte in memory having the index value as its absolute address.

Direct representation	Description
%MW4.6	Word starting from the first byte of the 7 <sup>th</sup> element of datablock 4 in memory.
%MW4	Word starting from byte 4 of memory.

#### Example

```
VAR [RETAIN] [CONSTANT]
  XQuote : DINT;      Enabling : BOOL := FALSE;
  TorqueCurrent AT %MW4.32 : INT;
  Counters : ARRAY [ 0 .. 9 ] OF UINT;
  Limits: ARRAY [0..3, 0..9]
END_VAR
```

- Variable `XQuote` is 32 bits long, and it is automatically allocated by the LogicLab compiler.
- Variable `Enabling` is initialized to `FALSE` after target reset.
- Variable `TorqueCurrent` is allocated in the memory area of the target device, and it takes 16 bits starting from the first byte of the 33<sup>rd</sup> element of datablock 4.
- Variable `Counters` is an array of 10 independent variables of type unsigned integer.

### 11.1.5.8 DECLARING VARIABLES IN LOGICLAB

Whatever the PLC language you are using, LogicLab allows you to disregard the syntax above, as it supplies the Local variables editor, the Global variables editor, and the Parameters editor, which provide a friendly interface to declare all kinds of variables.

### 11.1.6 PROGRAM ORGANIZATION UNITS

Program organization units are functions, function blocks, and programs. Program Organization Units can be delivered by the manufacturer, or programmed by the user through the means defined in this part of the standard

Program organization units are not recursive; that is, the invocation of a program organization unit cannot cause the invocation of another program organization unit of the same type.



### 11.1.6.1 FUNCTIONS

#### Introduction

For the purposes of programmable controller programming languages, a function is defined as a program organization unit (POU) which, when executed, yields exactly one data element, which is considered to be the function result.

Functions contain no internal state information, i.e., invocation of a function with the same arguments (input variables `VAR_INPUT` and in-out variables `VAR_IN_OUT`) always yields the same values (output variables `VAR_OUTPUT`, in-out variables `VAR_IN_OUT` and function result).

#### Declaration syntax

The declaration of a function must be performed as follows:

```
FUNCTION FunctionName : RetDataType
  VAR_INPUT
    declaration of input variables (see the relevant section)
  END_VAR
  VAR
    declaration of local variables (see the relevant section)
  END_VAR
  Function body
END_FUNCTION
```

Keyword	Description
FunctionName	Name of the function being declared.
RetDataType	Data type of the value to be returned by the function.
Function body	Specifies the operations to be performed upon the input variables in order to assign values dependent on the function's semantics to a variable with the same name as the function, which represents the function result. It can be written in any of the languages supported by LogicLab.

#### Declaring functions in LogicLab

Whatever the PLC language you are using, LogicLab allows you to disregard the syntax above, as it supplies a friendly interface for using functions.

### 11.1.6.2 FUNCTION BLOCKS

#### Introduction

For the purposes of programmable controller programming languages, a function block is a program organization unit which, when executed, yields one or more values. Multiple, named instances (copies) of a function block can be created. Each instance has an associated identifier (the instance name), and a data structure containing its input, output and internal variables. All the values of the output variables and the necessary internal variables of this data structure persist from one execution of the function block to the next; therefore, invocation of a function block with the same arguments (input variables) does not always yield the same output values.

Only the input and output variables are accessible outside of an instance of a function block, i.e., the function block's internal variables are hidden from the user of the function block.

In order to execute its operations, a function block needs to be invoked by another POU.



Invocation depends on the specific language of the module calling the function block. The scope of an instance of a function block is local to the program organization unit in which it is instantiated.

### Declaration syntax

The declaration of a function must be performed as follows:

```

FUNCTION_BLOCK FunctionBlockName
  VAR_INPUT
    declaration of input variables (see the relevant section)
  END_VAR
  VAR_OUTPUT
    declaration of output variables
  END_VAR
  VAR_EXTERNAL
    declaration of external variables
  END_VAR
  VAR
    declaration of local variables
  END_VAR
  Function block body
END_FUNCTION_BLOCK
    
```

Keyword	Description
FunctionBlockName	Name of the function block being declared (note: name of the template, not of its instances).
VAR_EXTERNAL .. END_VAR	A function block can access global variables only if they are listed in a VAR_EXTERNAL structuring element. Variables passed to the FB via a VAR_EXTERNAL construct can be modified from within the FB.
Function block body	Specifies the operations to be performed upon the input variables in order to assign values to the output variables - dependent on the function block's semantics and on the value of the internal variables. It can be written in any of the languages supported by LogicLab.

### Declaring functions in LogicLab

Whatever the PLC language you are using, LogicLab allows you to disregard the syntax above, as it supplies a friendly interface for using function blocks.

#### 11.1.6.3 PROGRAMS

##### Introduction

A program is defined in IEC 61131-1 as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system".

##### Declaration syntax



The declaration of a program must be performed as follows:

```
PROGRAM < program name>
    Declaration of variables (see the relevant section)
    Program body
END_PROGRAM
```

Keyword	Description
Program Name	Name of the program being declared.
Program body	Specifies the operations to be performed to get the intended signal processing. It can be written in any of the languages supported by LogicLab.

### Writing programs in LogicLab

Whatever the PLC language you are using, LogicLab allows you to disregard the syntax above, as it supplies a friendly interface for writing programs.

## 11.1.7 OBJECT ORIENTED REFERENCE

Object Oriented feature is achieved by enhancing the function block POU, there is not a specific element of type "class".

This way a function block can have methods, which are handled like functions except that they can see the relative function block context.

A function block can extends another function block (only one) in a father-child hierarchy; and can implements any number of interfaces.

Interfaces are handled like new type definitions and allow the user to specify a list of method prototypes composed of name and expected input variables; also interfaces can extends other interfaces (max one) in a father-child hierarchy.

If a function block implements one or more interfaces, it must correctly implements all of their methods in order to succesfully complete a project compilation.

LogicLab can handle polimorphysm on both function blocks and interfaces.

## 11.1.8 IEC 61131-3 STANDARD FUNCTIONS

This paragraph is a reference of all IEC 61131-3 standard functions available in LogicLab, along with a few others, which may be considered as LogicLab's extensions to the standard.

These functions are common to the whole set of programming languages and can therefore be used in any Programmable Organization Unit (POU).

A function specified in this paragraph to be extensible (Ext.) is allowed to have a variable number of inputs.

### Type conversion functions

According to the IEC 61131-3 standard, type conversion functions shall have the form `*_TO_**`, where "\*" is the type of the input variable, and "\*\*" the type of the output variable (for example, `INT_TO_REAL`). LogicLab provides a more convenient set of overloaded type conversion functions, relieving the developer to specify the input variable type.

TO_BOOL	
Description	Conversion to BOOL (boolean)
Number of operands	1



<b>TO_BOOL</b>	
<b>Input data type</b>	Any numerical type
<b>Output data type</b>	BOOL
<b>Examples</b>	<pre>out := TO_BOOL( 0 ); (* out = FALSE *) out := TO_BOOL( 1 ); (* out = TRUE *) out := TO_BOOL( 1000 ); (* out = TRUE *)</pre>

<b>TO_BYTE</b>	
<b>Description</b>	Conversion to BYTE (8-bit string)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	BYTE
<b>Examples</b>	<pre>out := TO_BYTE( -1 ); (* out = 16#FF *) out := TO_BYTE( 16#100 ); (* out = 16#00 *)</pre>

<b>TO_DATE</b>	
<b>Description</b>	Conversion to DATE (32-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	DATE_AND_TIME, LDATE_AND_TIME, LDATE
<b>Output data type</b>	DATE
<b>Examples</b>	

<b>TO_DATE_AND_TIME</b>	
<b>Description</b>	Conversion to DATE_AND_TIME (32-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	LDATE_AND_TIME
<b>Output data type</b>	DATE_AND_TIME
<b>Examples</b>	

<b>TO_DINT</b>	
<b>Description</b>	Conversion to DINT (32-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	DINT
<b>Examples</b>	<pre>out := TO_DINT( 10.0 ); (* out = 10 *) out := TO_DINT( 16#FFFFFFFF ); (* out = -1 *)</pre>

<b>TO_DWORD</b>	
<b>Description</b>	Conversion to DWORD (32-bit string)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	DWORD



<b>TO_DWORD</b>	
<b>Examples</b>	<pre>out := TO_DWORD( 10.0 ); (* out = 16#0000000A *) out := TO_DWORD( -1 ); (* out = 16#FFFFFFFF *)</pre>

<b>TO_INT</b>	
<b>Description</b>	Conversion to INT (16-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	INT
<b>Examples</b>	<pre>out := TO_INT( -1000.0 ); (* out = -1000 *) out := TO_INT( 16#8000 ); (* out = -32768 *)</pre>

<b>TO_LDATE</b>	
<b>Description</b>	Conversion to LDATE (64-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	DATE_AND_TIME, LDATE_AND_TIME, DATE
<b>Output data type</b>	LDATE
<b>Examples</b>	

<b>TO_LDATE_AND_TIME</b>	
<b>Description</b>	Conversion to LDATE_AND_TIME (64-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	DATE_AND_TIME
<b>Output data type</b>	LDATE_AND_TIME
<b>Examples</b>	

<b>TO_LINT</b>	
<b>Description</b>	Conversion to LINT (64-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	LINT
<b>Examples</b>	<pre>out := TO_LINT( -1 ); (* out = -1 *) out := TO_LINT(16#FFFFFFFFFFFFFFFF); (* out = -1 *)</pre>

<b>TO_LREAL</b>	
<b>Description</b>	Conversion to LREAL (64-bit floating point)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	LREAL
<b>Examples</b>	<pre>out := TO_LREAL( -1000 ); (* out = -1000.0 *) out := TO_LREAL( 16#8000 ); (* out = -32768.0 *)</pre>





<b>TO_LTIME</b>	
<b>Description</b>	Conversion to LTIME (64-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	TIME
<b>Output data type</b>	LTIME
<b>Examples</b>	

<b>TO_LTIME_OF_DAY</b>	
<b>Description</b>	Conversion to LTIME_OF_DAY (64-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	TIME_OF_DAY, DATE_AND_TIME, LDATE_AND_TIME
<b>Output data type</b>	LTIME_OF_DAY
<b>Examples</b>	

<b>TO_LWORD</b>	
<b>Description</b>	Conversion to LWORD (64-bit unsigned integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	LWORD
<b>Examples</b>	<pre>out := TO_LWORD( 10.0 ); (* out = 16#0000000000000000A *) out := TO_LWORD( -1 ); (* out = 16#FFFFFFFFFFFFFFF *)</pre>

<b>TO_POINTER</b>	
<b>Description</b>	Conversion to pointer
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	REAL
<b>Examples</b>	<pre>out := TO_POINTER( -1000 );</pre>

<b>TO_REAL</b>	
<b>Description</b>	Conversion to REAL (32-bit floating point)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	REAL
<b>Examples</b>	<pre>out := TO_REAL( -1000 ); (* out = -1000.0 *) out := TO_REAL( 16#8000 ); (* out = -32768.0 *)</pre>

<b>TO_SINT</b>	
<b>Description</b>	Conversion to SINT (8-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING



<b>TO_SINT</b>	
<b>Output data type</b>	SINT
<b>Examples</b>	<pre>out := TO_SINT( -1 ); (* out = -1 *) out := TO_SINT( 16#100 ); (* out = 0 *)</pre>

<b>TO_STRING</b>	
<b>Description</b>	Conversion to STRING
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type
<b>Output data type</b>	STRING
<b>Examples</b>	<pre>str := TO_STRING( 10.0 ); (* str = '10,0' *) str := TO_STRING( -1 ); (* str = '-1' *)</pre>

<b>TO_TIME</b>	
<b>Description</b>	Conversion to LTIME (32-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	LTIME
<b>Output data type</b>	TIME
<b>Examples</b>	

<b>TO_TIME_OF_DAY</b>	
<b>Description</b>	Conversion to TIME_OF_DAY (32-bit signed integer)
<b>Number of operands</b>	1
<b>Input data type</b>	LTIME_OF_DAY, DATE_AND_TIME, LDATE_AND_TIME
<b>Output data type</b>	TIME_OF_DAY
<b>Examples</b>	

<b>TO_UDINT</b>	
<b>Description</b>	Conversion to UDINT (32-bit unsigned integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	UDINT
<b>Examples</b>	<pre>out := TO_UDINT( 10.0 ); (* out = 10 *) out := TO_UDINT( 16#FFFFFFFF ); (* out = 4294967295 *)</pre>

<b>TO_UINT</b>	
<b>Description</b>	Conversion to UINT (16-bit unsigned integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	UINT
<b>Examples</b>	<pre>out := TO_UINT( 1000.0 ); (* out = 1000 *) out := TO_UINT( 16#8000 ); (* out = 32768 *)</pre>



<b>TO_ULINT</b>	
<b>Description</b>	Conversion to ULINT (64-bit unsigned integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	ULINT
<b>Examples</b>	<pre>out := TO_ULINT( 10.0 ); (* out = 10 *) out := TO_ULINT( 16#FFFFFFFFFFFFFFFF ); (* out = 18446744073709551615 *)</pre>

<b>TO_USINT</b>	
<b>Description</b>	Conversion to USINT (8-bit unsigned integer)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	USINT
<b>Examples</b>	<pre>out := TO_USINT( -1 ); (* out = 255 *) out := TO_USINT( 16#100 ); (* out = 0 *)</pre>

<b>TO_WORD</b>	
<b>Description</b>	Conversion to WORD (16-bit string)
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type or STRING
<b>Output data type</b>	WORD
<b>Examples</b>	<pre>out := TO_WORD( 1000.0 ); (* out = 16#03E8 *) out := TO_WORD( -32768 ); (* out = 16#8000 *)</pre>

<b>TO_WSTRING</b>	
<b>Description</b>	Conversion to WSTRING
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type
<b>Output data type</b>	WSTRING
<b>Examples</b>	<pre>wstr := TO_STRING( 10.0 ); (* wstr = "10,0" *) wstr := TO_STRING( -1 ); (* wstr = "-1" *)</pre>

### Numerical functions

The availability of the following functions depends on the target device. Please refer to your hardware supplier for details.

<b>ABS</b>	
<b>Description</b>	Absolute value. Computes the absolute value of the input
<b>Number of operands</b>	1
<b>Input data type</b>	Any numerical type
<b>Output data type</b>	Same as input



<b>ABS</b>	
<b>Examples</b>	<pre>OUT := ABS( -5 ); (* OUT = 5 *) OUT := ABS( -1.618 ); (* OUT = 1.618 *) OUT := ABS( 3.141592 ); (* OUT = 3.141592 *)</pre>

<b>ACOS</b>	
<b>Description</b>	Arc cosine. Computes the principal arc cosine of input #0; result is expressed in radians
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := ACOS( 1.0 ); (* OUT = 0.0 *) OUT := ACOS( -1.0 ); (* OUT = PI *)</pre>

<b>ADD</b>	
<b>Description</b>	Arithmetic addition. Computes the sum of the two inputs.
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	<pre>OUT := ADD( 20, 40 ); (* OUT = 60 *)</pre>

<b>ASIN</b>	
<b>Description</b>	Arc sine. Computes the principal arc sine of input #0; result is expressed in radians
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := ASIN( 0.0 ); (* OUT = 0.0 *) OUT := ASIN( 1.0 ); (* OUT = PI / 2 *)</pre>

<b>ATAN</b>	
<b>Description</b>	Arc tangent. Computes the principal arc tangent of input #0; result is expressed in radians
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := ATAN( 0.0 ); (* OUT = 0.0 *) OUT := ATAN( 1.0 ); (* OUT = PI / 4 *)</pre>

<b>ATAN2*</b>	
<b>Description</b>	Arc tangent (with 2 parameters). Computes the principal arc tangent of Y/X; result is expressed in radians
<b>Number of operands</b>	2



<b>ATAN2*</b>	
<b>Input data type</b>	LREAL where available, REAL otherwise; LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := ATAN2( 0.0, 1.0 ); (* OUT = 0.0 *) OUT := ATAN2( 1.0, 1.0 ); (* OUT = PI / 4 *) OUT := ATAN2( -1.0, -1.0 ); (* OUT = ( -3/4 ) * PI *) OUT := ATAN2( 1.0, 0.0 ); (* OUT = PI / 2 *)</pre>

<b>CEIL*</b>	
<b>Description</b>	Rounding up to integer. Returns the smallest integer that is greater than or equal to input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := CEIL( 1.95 ); (* OUT = 2.0 *) OUT := CEIL( -1.27 ); (* OUT = -1.0 *)</pre>

<b>COS</b>	
<b>Description</b>	Cosine. Computes the cosine function of input #0 expressed in radians
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := COS( 0.0 ); (* OUT = 1.0 *) OUT := COS( -3.141592 ); (* OUT ~ -1.0 *)</pre>

<b>COSH*</b>	
<b>Description</b>	Hyperbolic cosine. Computes the hyperbolic cosine function of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	<pre>OUT := COSH( 0.0 ); (* OUT = 1.0 *)</pre>

<b>DIV</b>	
<b>Description</b>	Arithmetic division. Divides input #0 by input #1
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	<pre>OUT := DIV( 20, 2 ); (* OUT = 10 *)</pre>



<b>EXP</b>	
<b>Description</b>	Natural exponential. Computes the exponential function of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := EXP( 1.0 ); (* OUT ~ 2.718281 *)

<b>FLOOR*</b>	
<b>Description</b>	Rounding down to integer. Returns the largest integer that is less than or equal to input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := FLOOR( 1.95 ); (* OUT = 1.0 *) OUT := FLOOR( -1.27 ); (* OUT = -2.0 *)

<b>LN</b>	
<b>Description</b>	Natural logarithm. Computes the logarithm with base e of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := LN( 2.718281 ); (* OUT = 1.0 *)

<b>LOG</b>	
<b>Description</b>	Common logarithm. Computes the logarithm with base 10 of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := LOG( 100.0 ); (* OUT = 2.0 *)

<b>MOD</b>	
<b>Description</b>	Module. Computes input #0 module input #1
<b>Number of operands</b>	2
<b>Input data type</b>	Integer type, integer type
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	OUT := MOD( 10, 3 ); (* OUT = 1 *)

<b>MUL</b>	
<b>Description</b>	Arithmetic multiplication. Multiplies the two inputs.
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type



<b>MUL</b>	
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	OUT := MUL( 10, 10 ); (* OUT = 100 *)

<b>POW</b>	
<b>Description</b>	Exponentiation. Raises Base to the power Expo
<b>Number of operands</b>	2
<b>Input data type</b>	LREAL where available, REAL otherwise; LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := POW( 2.0, 3.0 ); (* OUT = 8.0 *) OUT := POW( -1.0, 5.0 ); (* OUT = -1.0 *)

<b>SIN</b>	
<b>Description</b>	Sine. Computes the sine function of input #0 expressed in radians
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := SIN( 0.0 ); (* OUT = 0.0 *) OUT := SIN( 2.5 * 3.141592 ); (* OUT ~ 1.0 *)

<b>SINH*</b>	
<b>Description</b>	Hyperbolic sine. Computes the hyperbolic sine function of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := SINH( 0.0 ); (* OUT = 0.0 *)

<b>SQRT</b>	
<b>Description</b>	Square root. Computes the square root of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := SQRT( 4.0 ); (* OUT = 2.0 *)

<b>SUB</b>	
<b>Description</b>	Arithmetic subtraction. Subtracts input #1 from input #0
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	OUT := SUB( 10, 3 ); (* OUT = 7 *)



<b>TAN</b>	
<b>Description</b>	Tangent. Computes the tangent function of input #0 expressed in radians
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := TAN( 0.0 ); (* OUT = 0.0 *) OUT := TAN( 3.141592 / 4.0 ); (* OUT ~ 1.0 *)

<b>TANH*</b>	
<b>Description</b>	Hyperbolic tangent. Computes the hyperbolic tangent function of input #0
<b>Number of operands</b>	1
<b>Input data type</b>	LREAL where available, REAL otherwise
<b>Output data type</b>	LREAL where available, REAL otherwise
<b>Examples</b>	OUT := TANH( 0.0 ); (* OUT = 0.0 *)

\*: function provided as extension to the IEC 61131-3 standard.

### Bit string functions

<b>AND</b>	
<b>Description</b>	Logical AND if both Input #0 and Input #1 are BOOL, otherwise bitwise AND.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but STRING, Any but STRING
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	OUT := TRUE AND FALSE; (* OUT = FALSE *) OUT := 16#1234 AND 16#5678; (* OUT = 16#1230 *)

<b>NOT</b>	
<b>Description</b>	Logical NOT if Input is BOOL, otherwise bitwise NOT.
<b>Number of operands</b>	1
<b>Input data type</b>	Any but STRING
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	OUT := NOT FALSE; (* OUT = TRUE *) OUT := NOT 16#1234; (* OUT = 16#EDCB *)

<b>OR</b>	
<b>Description</b>	Logical OR if both Input #0 and Input #1 are BOOL, otherwise bitwise OR.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but STRING, Any but STRING
<b>Output data type</b>	Same as Inputs





<b>OR</b>	
<b>Examples</b>	<pre>OUT := TRUE OR FALSE; (* OUT = FALSE *) OUT := 16#1234 OR 16#5678; (* OUT = 16#567C *)</pre>

<b>R</b>	
<b>Description</b>	Reset operator: reset input to 0
<b>Number of operands</b>	1
<b>Input data type</b>	BOOL
<b>Output data type</b>	Same as Input #0
<b>Examples</b>	

<b>ROL</b>	
<b>Description</b>	Input #0 left-shifted of Input #1 bits, circular.
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Input #0
<b>Examples</b>	<pre>OUT := ROL( IN := 16#1000CAFE, 4 ); (* OUT = 16#000CAFE1 *)</pre>

<b>ROR</b>	
<b>Description</b>	Input #0 right-shifted of Input #1 bits, circular.
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Input #0
<b>Examples</b>	<pre>OUT := ROR( IN := 16#1000CAFE, 16 ); (* OUT = 16#CAFE1000 *)</pre>

<b>S</b>	
<b>Description</b>	Set operator: set input to 1
<b>Number of operands</b>	1
<b>Input data type</b>	BOOL
<b>Output data type</b>	Same as Input #0
<b>Examples</b>	

<b>SHL</b>	
<b>Description</b>	Input#0 left-shifted of Input #1 bits, zero filled on the right.
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Input #0
<b>Examples</b>	<pre>OUT := SHL( IN := 16#1000CAFE, 16 ); (* OUT = 16#CAFE0000 *)</pre>



<b>SHR</b>	
<b>Description</b>	Input #0 right-shifted of Input #1 bits, zero filled on the left.
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, Any numerical type
<b>Output data type</b>	Same as Input #0
<b>Examples</b>	<pre>OUT := SHR( IN := 16#1000CAFE, 24 ); (* OUT = 16#00000010 *)</pre>

<b>XOR</b>	
<b>Description</b>	Logical XOR if both Input #0 and Input #1 are BOOL, otherwise bitwise XOR.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but STRING, Any but STRING
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	<pre>OUT := TRUE OR FALSE; (* OUT = TRUE *) OUT := 16#1234 OR 16#5678; (* OUT = 16#444C *)</pre>

### Selection functions

<b>LIMIT</b>	
<b>Description</b>	Limits Input #0 to be equal or more than Input#1, and equal or less than Input #2.
<b>Number of operands</b>	3
<b>Input data type</b>	Any numerical type, Any numerical type, Any numerical type
<b>Output data type</b>	Same as Inputs
<b>Examples</b>	<pre>OUT := LIMIT( IN := 4, MN := 0, MX := 5 ); (* OUT = 4 *) OUT := LIMIT( IN := 88, MN := 0, MX := 5 ); (* OUT = 5 *) OUT := LIMIT( IN := -1, MN := 0, MX := 5 ); (* OUT = 0 *)</pre>

<b>MAX</b>	
<b>Description</b>	Maximum value selection
<b>Number of operands</b>	2, extensible
<b>Input data type</b>	Any numerical type, Any numerical type, .., Any numerical type
<b>Output data type</b>	Same as max Input
<b>Examples</b>	<pre>OUT := MAX( -8, 120, -1000 ); (* OUT = 120 *)</pre>

<b>MIN</b>	
<b>Description</b>	Minimum value selection
<b>Number of operands</b>	2, extensible



<b>MIN</b>	
<b>Input data type</b>	Any numerical type, Any numerical type, .., Any numerical type
<b>Output data type</b>	Same as min Input
<b>Examples</b>	OUT := MIN( -8, 120, -1000 ); (* OUT = -1000 *)

<b>MUX</b>	
<b>Description</b>	Multiplexer. Selects one of N inputs depending on input K
<b>Number of operands</b>	3, extensible
<b>Input data type</b>	Any numerical type, Any numerical type, ..., Any numerical type
<b>Output data type</b>	Same as selected Input
<b>Examples</b>	OUT := MUX( 0, A, B, C ); (* OUT = A *)

<b>SEL</b>	
<b>Description</b>	Binary selection
<b>Number of operands</b>	3
<b>Input data type</b>	BOOL, Any, Any
<b>Output data type</b>	Same as selected Input
<b>Examples</b>	OUT := SEL( G := FALSE, IN0 := X, IN1 := 5 ); (* OUT = X *)

### Comparison functions

Comparison functions can be also used to compare strings if this feature is supported by target device.

<b>EQ</b>	
<b>Description</b>	Equal to. Returns TRUE if Input #0 = Input #1, otherwise FALSE.
<b>Number of operands</b>	2
<b>Input data type</b>	Any, Any
<b>Output data type</b>	BOOL
<b>Examples</b>	OUT := EQ( TRUE, FALSE ); (* OUT = FALSE *) OUT := EQ( 'AZ', 'ABC' ); (* OUT = FALSE *)

<b>GE</b>	
<b>Description</b>	Greater than or equal to. Returns TRUE if Input #0 >= Input #1, otherwise FALSE.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but BOOL, Any but BOOL
<b>Output data type</b>	BOOL
<b>Examples</b>	OUT := GE( 20, 20 ); (* OUT = TRUE *) OUT := GE( 'AZ', 'ABC' ); (* OUT = FALSE *)



<b>GT</b>	
<b>Description</b>	Greater than. Returns TRUE if Input #0 > Input #1, otherwise FALSE.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but BOOL, Any but BOOL
<b>Output data type</b>	BOOL
<b>Examples</b>	<pre>OUT := GT( 0, 20 ); (* OUT = FALSE *) OUT := GT( 'AZ', 'ABC' ); (* OUT = TRUE *)</pre>

<b>LE</b>	
<b>Description</b>	Less than or equal to. Returns TRUE if Input #0 <= Input #1, otherwise FALSE.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but BOOL, Any but BOOL
<b>Output data type</b>	BOOL
<b>Examples</b>	<pre>OUT := LE( 20, 20 ); (* OUT = TRUE *) OUT := LE( 'AZ', 'ABC' ); (* OUT = FALSE *)</pre>

<b>LT</b>	
<b>Description</b>	Less than. Returns TRUE if Input #0 < Input #1, otherwise FALSE.
<b>Number of operands</b>	2
<b>Input data type</b>	Any but BOOL, Any but BOOL
<b>Output data type</b>	BOOL
<b>Examples</b>	<pre>OUT := LT( 0, 20 ); (* OUT = TRUE *) OUT := LT( 'AZ', 'ABC' ); (* OUT = FALSE *)</pre>

<b>NE</b>	
<b>Description</b>	Not equal to. Returns TRUE if Input #0 != Input #1, otherwise FALSE.
<b>Number of operands</b>	2
<b>Input data type</b>	Any, Any
<b>Output data type</b>	BOOL
<b>Examples</b>	<pre>OUT := NE( TRUE, FALSE ); (* OUT = TRUE *) OUT := NE( 'AZ', 'ABC' ); (* OUT = TRUE *)</pre>

### String functions

The availability of the following functions depends on the target device. Please refer to your hardware supplier for details.

<b>CONCAT</b>	
<b>Description</b>	Character string concatenation
<b>Number of operands</b>	2
<b>Input data type</b>	STRING, STRING
<b>Output data type</b>	STRING



<b>CONCAT</b>	
<b>Examples</b>	<code>OUT := CONCAT( 'AB', 'CD' ); (* OUT = 'ABCD' *)</code>

<b>DELETE</b>	
<b>Description</b>	Delete L characters of IN, beginning at the P-th character position
<b>Number of operands</b>	3
<b>Input data type</b>	STRING, UINT, UINT
<b>Output data type</b>	STRING
<b>Examples</b>	<code>OUT := DELETE( IN := 'ABXYC', L := 2, P := 3 );</code> <code>(* OUT = 'ABC' *)</code>

<b>FIND</b>	
<b>Description</b>	Find the character position of the beginning of the first occurrence of IN2 in IN1. If no occurrence of IN2 is found, then OUT := 0.
<b>Number of operands</b>	2
<b>Input data type</b>	STRING, STRING
<b>Output data type</b>	UINT
<b>Examples</b>	<code>OUT := FIND( IN1 := 'ABCBC', IN2 := 'BC' ); (* OUT = 2 *)</code>

<b>INSERT</b>	
<b>Description</b>	Insert IN2 into IN1 after the P-th character position
<b>Number of operands</b>	3
<b>Input data type</b>	STRING, STRING, UINT
<b>Output data type</b>	STRING
<b>Examples</b>	<code>OUT := INSERT( IN1 := 'ABC', IN2 := 'XY', P := 2 );</code> <code>(* OUT = 'ABXYC' *)</code>

<b>LEFT</b>	
<b>Description</b>	Leftmost L characters of IN
<b>Number of operands</b>	2
<b>Input data type</b>	STRING, UINT
<b>Output data type</b>	STRING
<b>Examples</b>	<code>OUT := LEFT( IN := 'ASTR', L := 3 ); (* OUT = 'AST' *)</code>

<b>LEN</b>	
<b>Description</b>	Return the length of a string
<b>Number of operands</b>	1
<b>Input data type</b>	STRING
<b>Output data type</b>	UINT
<b>Examples</b>	<code>OUT := LEN( 'ASTRING' ); (* OUT = 7 *)</code>



<b>MID</b>	
<b>Description</b>	L characters of IN, beginning at the P-th
<b>Number of operands</b>	3
<b>Input data type</b>	STRING, UINT, UINT
<b>Output data type</b>	STRING
<b>Examples</b>	OUT := MID( IN := 'ASTR', L := 2, P := 2 ); (* OUT = 'ST' *)

<b>REPLACE</b>	
<b>Description</b>	Replace L characters of IN1 by IN2, starting at the P-th character position
<b>Number of operands</b>	4
<b>Input data type</b>	STRING, STRING, UINT, UINT
<b>Output data type</b>	STRING
<b>Examples</b>	OUT := REPLACE( IN1 := 'ABCDE', IN2 := 'X', L := 2, P := 3 ); (* OUT = 'ABXE' *)

<b>RIGHT</b>	
<b>Description</b>	Rightmost L characters of IN
<b>Number of operands</b>	2
<b>Input data type</b>	STRING, UINT
<b>Output data type</b>	STRING
<b>Examples</b>	OUT := RIGHT( IN := 'ASTR', L := 3 ); (* OUT = 'STR' *)

<b>TO_STRINGFORMAT</b>	
<b>Description</b>	Conversion to STRING, with format specifier
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, STRING
<b>Output data type</b>	STRING
<b>Examples</b>	str := TO_STRINGFORMAT(10, '%04d'); (* str = '0010' *)

<b>TO_WSTRINGFORMAT</b>	
<b>Description</b>	Conversion to WSTRING, with format specifier
<b>Number of operands</b>	2
<b>Input data type</b>	Any numerical type, WSTRING
<b>Output data type</b>	WSTRING
<b>Examples</b>	wstr := TO_WSTRINGFORMAT(10, '%04d'); (* wstr = "0010" *)

## Standard operator



<b>ADR</b>	
<b>Description</b>	Return the address of a variable
<b>Number of operands</b>	1
<b>Input data type</b>	Any
<b>Output data type</b>	DWORD
<b>Examples</b>	

<b>IMOVE</b>	
<b>Description</b>	Valorize an interface using another interface and executing a query interface; checking if the class that implement the source interface is also implementing the destination interface
<b>Number of operands</b>	1
<b>Input data type</b>	Interface instance
<b>Output data type</b>	NULL if error, any if ok
<b>Examples</b>	

<b>JMP</b>	
<b>Description</b>	Jump to a specific label
<b>Number of operands</b>	1
<b>Input data type</b>	STRING
<b>Output data type</b>	
<b>Examples</b>	

<b>MOVE</b>	
<b>Description</b>	Assign a value to a variable, equivalent of LD and ST
<b>Number of operands</b>	2
<b>Input data type</b>	Any, Any
<b>Output data type</b>	
<b>Examples</b>	

<b>REF</b>	
<b>Description</b>	Return the reference to a variable
<b>Number of operands</b>	1
<b>Input data type</b>	Any
<b>Output data type</b>	Reference to the input variable
<b>Examples</b>	

<b>RET</b>	
<b>Description</b>	Return
<b>Number of operands</b>	0
<b>Input data type</b>	
<b>Output data type</b>	
<b>Examples</b>	



<b>SIZEOF</b>	
<b>Description</b>	Return the size of a variable
<b>Number of operands</b>	1
<b>Input data type</b>	Any
<b>Output data type</b>	UDINT
<b>Examples</b>	

### Date functions

<b>CONCAT_DATE</b>	
<b>Description</b>	Create a valid DATE value given its single components (year, month and day).
<b>Number of operands</b>	3
<b>Input data type</b>	INT, INT, INT
<b>Output data type</b>	DATE
<b>Examples</b>	<code>out := CONCAT_DATE(2020, 6, 17); (* out = 2020-06-17 *)</code>

<b>CONCAT_DATE_LTOD</b>	
<b>Description</b>	Concatenate a DATE and a LTIME_OF_DAY into a LDATE_AND_TIME
<b>Number of operands</b>	2
<b>Input data type</b>	DATE, LTIME_OF_DAY
<b>Output data type</b>	LDATE_AND_TIME
<b>Examples</b>	<code>out := CONCAT_DATE_LTOD(2020-06-17, 13:40:55.123456789); (* out = 2020-06-17-13:40:55.123456789 *)</code>

<b>CONCAT_DATE_TOD</b>	
<b>Description</b>	Concatenate a DATE and a TIME_OF_DAY into a DATE_AND_TIME
<b>Number of operands</b>	2
<b>Input data type</b>	DATE, TIME_OF_DAY
<b>Output data type</b>	DATE_AND_TIME
<b>Examples</b>	<code>out := CONCAT_DATE_TOD(2020-06-17, 13:40:55.123); (* out = 2020-06-17-13:40:55.123 *)</code>

<b>CONCAT_DT</b>	
<b>Description</b>	Create a valid DATE_AND_TIME value given its single components (year, month, day, hours, minutes, seconds).
<b>Number of operands</b>	2
<b>Input data type</b>	INT, INT, INT, INT, INT, INT
<b>Output data type</b>	DATE_AND_TIME
<b>Examples</b>	<code>out := CONCAT_DT(2020, 6, 17, 13, 40, 55); (* out = 2020-06-17-13:40:55 *)</code>





<b>CONCAT_LDATE</b>	
<b>Description</b>	Create a valid LDATE value given its single components (year, month and day).
<b>Number of operands</b>	3
<b>Input data type</b>	INT, INT, INT
<b>Output data type</b>	LDATE
<b>Examples</b>	<code>out := CONCAT_LDATE(2020, 6, 17); (* out = 2020-06-17 *)</code>

<b>CONCAT_LDT</b>	
<b>Description</b>	Create a valid LDATE_AND_TIME value given its single components (year, month, day, hours, minutes, seconds, milliseconds, microseconds, nanoseconds).
<b>Number of operands</b>	9
<b>Input data type</b>	INT, INT, INT, INT, INT, INT, INT, INT, INT
<b>Output data type</b>	LDATE_AND_TIME
<b>Examples</b>	<code>out := CONCAT_LDATE_AND_TIME(2020, 6, 17, 13, 40, 55, 123, 456, 789);</code> <code>(* out = 2020-06-17-13:40:55.123456789 *)</code>

<b>CONCAT_LTOD</b>	
<b>Description</b>	Create a valid LTIME_OF_DAY value given its single components (hours, minutes, seconds, milliseconds, microseconds and nanoseconds).
<b>Number of operands</b>	6
<b>Input data type</b>	INT, INT, INT, INT, INT, INT
<b>Output data type</b>	LTIME_OF_DAY
<b>Examples</b>	<code>out := CONCAT_LTOD(13, 40, 55, 123, 456, 789); (*</code> <code>out = 13:40:55.123456789 *)</code>

<b>CONCAT_TOD</b>	
<b>Description</b>	Create a valid TIME_OF_DAY value given its single components (hours, minutes, seconds, milliseconds).
<b>Number of operands</b>	4
<b>Input data type</b>	INT, INT, INT, INT
<b>Output data type</b>	TIME_OF_DAY
<b>Examples</b>	<code>out := CONCAT_TOD(13, 40, 55, 123); (* out =</code> <code>13:40:55.123 *)</code>

<b>DAY_OF_WEEK</b>	
<b>Description</b>	Get the day of week. It returns the day of the week represented in a range from 0 (Sunday) to 6 (Saturday).
<b>Number of operands</b>	1
<b>Input data type</b>	LDATE
<b>Output data type</b>	SINT
<b>Examples</b>	<code>out := DAY_OF_WEEK(2020-06-17); (* out = 3 *)</code>



<b>SPLIT_DATE</b>	
<b>Description</b>	Split a LDATE into year, month and day integer pointer variables. The function returns TRUE in case of no errors.
<b>Number of operands</b>	4
<b>Input data type</b>	DATE/LDATE, @INT, @INT, @INT
<b>Output data type</b>	BOOL
<b>Examples</b>	

<b>SPLIT_DT</b>	
<b>Description</b>	Split a DATE_AND_TIME into year, month, day, hours, minutes and seconds integer pointer variables. The function returns TRUE in case of no errors.
<b>Number of operands</b>	7
<b>Input data type</b>	DATE_AND_TIME, @INT, @INT, @INT, @INT, @INT, @INT
<b>Output data type</b>	BOOL
<b>Examples</b>	

<b>SPLIT_LDT</b>	
<b>Description</b>	Split a LDATE_AND_TIME into year, month, day, hours, minutes, seconds, milliseconds, microseconds and nanoseconds integer pointer variables. The function returns TRUE in case of no errors.
<b>Number of operands</b>	10
<b>Input data type</b>	DATE_AND_TIME, @INT, @INT, @INT, @INT, @INT, @INT, @INT, @INT, @INT, @INT
<b>Output data type</b>	BOOL
<b>Examples</b>	

<b>SPLIT_LTOD</b>	
<b>Description</b>	Split a LTIME_OF_DAY into hours, minutes, seconds, milliseconds, microseconds and nanoseconds integer pointer variables. The function returns TRUE in case of no errors.
<b>Number of operands</b>	7
<b>Input data type</b>	LTIME_OF_DAY, @INT, @INT, @INT, @INT, @INT, @INT
<b>Output data type</b>	BOOL
<b>Examples</b>	

<b>SPLIT_TOD</b>	
<b>Description</b>	Split a TIME_OF_DAY into hours, minutes, seconds, milliseconds integer pointer variables. The function returns TRUE in case of no errors.
<b>Number of operands</b>	5
<b>Input data type</b>	TIME_OF_DAY, @INT, @INT, @INT, @INT
<b>Output data type</b>	BOOL
<b>Examples</b>	



## 11.2 INSTRUCTION LIST (IL)

This section defines the semantics of the IL (Instruction List) language.

### 11.2.1 SYNTAX AND SEMANTICS

#### 11.2.1.1 SYNTAX OF IL INSTRUCTIONS

IL code is composed of a sequence of instructions. Each instruction begins on a new line and contains an operator with optional modifiers, and, if necessary for the particular operation, one or more operands separated by commas. Operands can be any of the data representations for literals and for variables.

The instruction can be preceded by an identifying label followed by a colon (:). Empty lines can be inserted between instructions.

#### Example

Let us parse a small piece of code:

```
START:
    LD %IX1 (* Push button *)
    ANDN %MX5.4 (* Not inhibited *)
    ST %QX2 (* Fan out *)
```

The elements making up each instruction are classified as follows:

Label	Operator [+ modifier]	Operand	Comment
START:	LD	%IX1	(* Push button *)
	ANDN	%MX5.4	(* Not inhibited *)
	ST	%QX2	(* Fan out *)

#### Semantics of IL instructions

##### - Accumulator

By accumulator a register is meant containing the value of the currently evaluated result.

##### - Operators

Unless otherwise specified, the semantics of the operators is

```
accumulator := accumulator OP operand
```

That is, the value of the accumulator is replaced by the result yielded by operation OP applied to the current value of the accumulator itself, with respect to the operand. For instance, the instruction "AND %IX1" is interpreted as

```
accumulator := accumulator AND %IX1
```

and the instruction "GT %IW10" will have the Boolean result `TRUE` if the current value of the accumulator is greater than the value of input word 10, and the Boolean result `FALSE` otherwise:

```
accumulator := accumulator GT %IW10
```

##### - Modifiers

The modifier "N" indicates bitwise negation of the operand.

The left parenthesis modifier "(" indicates that evaluation of the operator must be deferred until a right parenthesis operator ")" is encountered. The form of a parenthesized sequence of instructions is shown below, referred to the instruction

```
accumulator := accumulator AND (%MX1.3 OR %MX1.4)
```



The modifier “C” indicates that the associated instruction can be performed only if the value of the currently evaluated result is Boolean 1 (or Boolean 0 if the operator is combined with the “N” modifier).

## 11.2.2 STANDARD OPERATORS

Standard operators with their allowed modifiers and operands are as listed below.

Operator	Modifiers	Supported operand types: Acc_type, Op_type	Semantics
LD	N	Any, Any	Sets the accumulator equal to operand.
ST	N	Any, Any	Stores the accumulator into operand location.
S		BOOL, BOOL	Sets operand to TRUE if accumulator is TRUE.
R		BOOL, BOOL	Sets operand to FALSE if accumulator is TRUE.
AND	N, (	Any but REAL, Any but REAL	Logical or bitwise AND
OR	N, (	Any but REAL, Any but REAL	Logical or bitwise OR
XOR	N, (	Any but REAL, Any but REAL	Logical or bitwise XOR
NOT		Any but REAL	Logical or bitwise NOT
ADD	(	Any but BOOL	Addition
SUB	(	Any but BOOL	Subtraction
MUL	(	Any but BOOL	Multiplication
DIV	(	Any but BOOL	Division
MOD	(	Any but BOOL	Modulo-division
GT	(	Any but BOOL	Comparison:
GE	(	Any but BOOL	Comparison: =
EQ	(	Any but BOOL	Comparison: =
NE	(	Any but BOOL	Comparison:
LE	(	Any but BOOL	Comparison:
LT	(	Any but BOOL	Comparison:
JMP	C, N	Label	Jumps to label
CAL	C, N	FB instance name	Calls function block
RET	C, N		Returns from called program, function, or function block.
)			Evaluates deferred operation.



## 11.2.3 CALLING FUNCTIONS AND FUNCTION BLOCKS

### 11.2.3.1 CALLING FUNCTIONS

Functions (as defined in the relevant section) are invoked by placing the function name in the operator field. This invocation takes the following form:

```
LD 1
MUX 5, var0, -6.5, 3.14
ST vRES
```

Note that the first argument is not contained in the input list, but the accumulator is used as the first argument of the function. Additional arguments (starting with the 2<sup>nd</sup>), if required, are given in the operand field, separated by commas, in the order of their declaration. For example, operator `MUX` in the table above takes 5 operands, the first of which is loaded into the accumulator, whereas the remaining 4 arguments are orderly reported after the function name.

**The following rules apply to function invocation.**

- 1) Assignments to `VAR_INPUT` arguments may be empty, constants, or variables.
- 2) Execution of a function ends upon reaching a `RET` instruction or the physical end of the function. When this happens, the output variable of the function is copied into the accumulator.

#### Calling Function Blocks

Function blocks (as defined in the relevant section) can be invoked conditionally and unconditionally via the `CAL` operator. This invocation takes the following form:

```
LD A
ADD 5
ST INST5.IN1
LD 3.141592
ST INST5.IN2
CAL INST5
LD INST5.OUT1
ST vRES
LD INST5.OUT2
ST vVALID
```

This method of invocation is equivalent to a `CAL` with an argument list, which contains only one variable with the name of the FB instance.

Input arguments are passed to / output arguments are read from the FB instance through `ST / LD` operations performed on operands taking the following form:

`FBInstanceName.IO_var`

where

Keyword	Description
<code>FBInstanceName</code>	Name of the instance to be invoked.
<code>IO_var</code>	Input or output variable to be written / read.

## 11.3 FUNCTION BLOCK DIAGRAM (FBD)



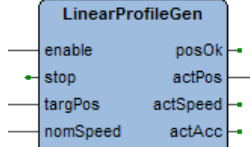
This section defines the semantics of the FBD (Function Block Diagram) language.



### 11.3.1 REPRESENTATION OF LINES AND BLOCKS

The graphic language elements are drawn using graphic or semi graphic elements, as shown in the table below.

No storage of data or association with data elements can be associated with the use of connectors; hence, to avoid ambiguity, connectors cannot be given any identifier.

Feature	Example
Lines	
Line crossing with connection	
Blocks with connecting lines and unconnected pins	

### 11.3.2 DIRECTION OF FLOW IN NETWORKS

A network is defined as a maximal set of interconnected graphic elements. A network label delimited on the right by a colon (:) can be associated with each network or group of networks. The scope of a network and its label is local to the program organization unit (POU) where the network is located.

Graphic languages are used to represent the flow of a conceptual quantity through one or more networks representing a control plan. Namely, in the case of function block diagrams (FBD), the "Signal flow" is typically used, analogous to the flow of signals between elements of a signal processing system. Signal flow in the FBD language is from the output (right-hand) side of a function or function block to the input (left-hand) side of the function or function block(s) so connected.

### 11.3.3 EVALUATION OF NETWORKS

#### 11.3.3.1 ORDER OF EVALUATION OF NETWORKS

The order in which networks and their elements are evaluated is not necessarily the same as the order in which they are labeled or displayed. When the body of a program organization unit (POU) consists of one or more networks, the results of network evaluation within the aforesaid body are functionally equivalent to the observance of the following rules:

- 1) No element of a network is evaluated until the states of all of its inputs have been evaluated.
- 2) The evaluation of a network element is not complete until the states of all of its outputs have been evaluated.
- 3) As stated when describing the FBD editor, a network number is automatically assigned to every network. Within a program organization unit (POU), networks are evaluated according to the sequence of their number: network N is evaluated before network N+1, unless otherwise specified by means of the execution control elements.



**11.3.3.2 COMBINATION OF ELEMENTS**

Elements of the FBD language must be interconnected by signal flow lines.

Outputs of blocks shall not be connected together. In particular, the “wired-OR” construct of the LD language is not allowed, as an explicit Boolean “OR” block is required.

**Feedback**

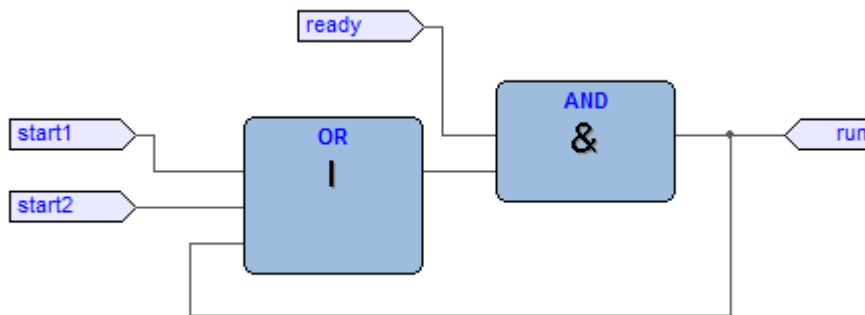
A feedback path is said to exist in a network when the output of a function or function block is used as the input to a function or function block which precedes it in the network; the associated variable is called a feedback variable.

Feedback paths can be utilized subject to the following rules:

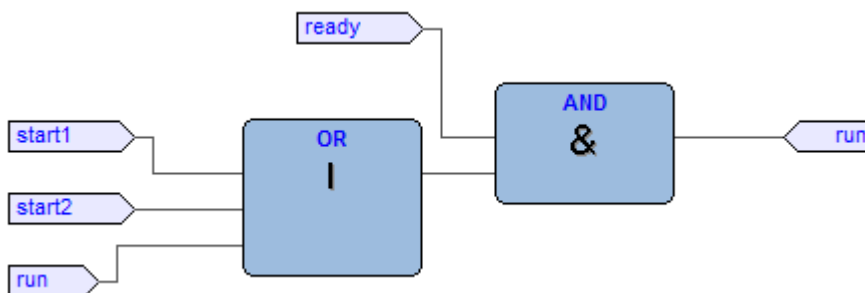
- 1) Feedback variables must be initialized, and the initial value is used during the first evaluation of the network. Look at the *Global variables* editor, the *Local variables* editor, or the *Parameters* editor to know how to initialize the respective item.
- 2) Once the element with a feedback variable as output has been evaluated, the new value of the feedback variable is used until the next evaluation of the element.

For instance, the Boolean variable `RUN` is the feedback variable in the example shown below.

**Explicit loop**



**Implicit loop**



**11.3.4 EXECUTION CONTROL ELEMENTS**

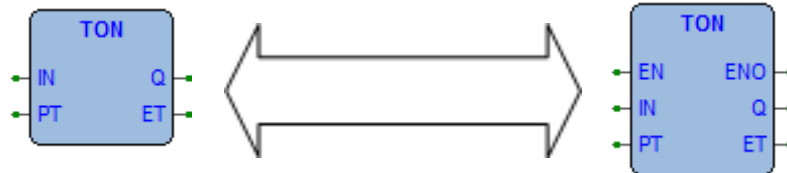
**11.3.4.1 EN/ENO SIGNALS**

Additional Boolean `EN` (Enable) input and `ENO` (Enable Out) characterize LogicLab blocks, according to the declarations



EN	ENO
VAR_INPUT	VAR_OUTPUT
EN: BOOL := 1;	ENO: BOOL;
END_VAR	END_VAR

See the *Modifying* properties of blocks section to know how to add these pins to a block.



When these variables are used, the execution of the operations defined by the block are controlled according to the following rules:

- 1) If the value of EN is FALSE when the block is invoked, the operations defined by the function body are not executed and the value of ENO is reset to FALSE by the programmable controller system.
- 2) Otherwise, the value of ENO is set to TRUE by the programmable controller system, and the operations defined by the block body are executed.

### 11.3.4.2 JUMPS

Jumps are represented by a Boolean signal line terminated in a double arrowhead. The signal line for a jump condition originates at a Boolean variable, or at a Boolean output of a function or function block. A transfer of program control to the designated network label occurs when the Boolean value of the signal line is TRUE; thus, the unconditional jump is a special case of the conditional jump.

The target of a jump is a network label within the program organization unit within which the jump occurs.


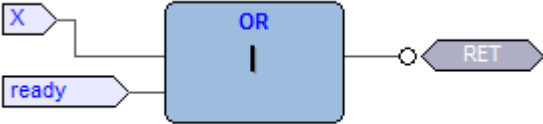
Symbol / Example	Explanation
	Unconditional Jump
	Conditional Jump
	Example: Jump Condition Network





**11.3.4.3 CONDITIONAL RETURNS**

- Conditional returns from functions and function blocks are implemented using a RETURN construction as shown in the table below. Program execution is transferred back to the invoking entity when the Boolean input is TRUE, and continues in the normal fashion when the Boolean input is FALSE.
- Unconditional returns are provided by the physical end of the function or function block.

Symbol / Example	Explanation
	Conditional Return
	Example: Return Condition Network

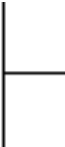

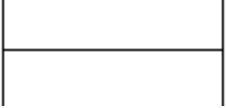
**11.4 LADDER DIAGRAM (LD)**

This section defines the semantics of the LD (Ladder Diagram) language.

**11.4.1 POWER RAILS**

The LD network is delimited on the left side by a vertical line known as the left power rail, and on the right side by a vertical line known as the right power rail. The right power rail may be explicit in the LogicLab implementation and it is always shown.

The two power rails are always connected with an horizontal line named signal link. All LD elements should be placed and connected to the signal link.

Description	Symbol
Left power rail (with attached horizontal link)	
Right power rail (with attached horizontal link)	
Power rails connected by the signal link	

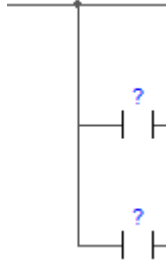


### 11.4.2 LINK ELEMENTS AND STATES

Link elements may be horizontal or vertical. The state of the link elements shall be denoted "ON" or "OFF", corresponding to the literal Boolean values 1 or 0, respectively. The term link state shall be synonymous with the term power flow.

The following properties apply to the link elements:

- The state of the left rail shall be considered ON at all times. No state is defined for the right rail.
- A horizontal link element is indicated by a horizontal line. A horizontal link element transmits the state of the element on its immediate left to the element on its immediate right.
- The vertical link element consists of a vertical line intersecting with one or more horizontal link elements on each side. The state of the vertical link represents the inclusive OR of the ON states of the horizontal links on its left side, that is, the state of the vertical link is:
  - OFF if the states of all the attached horizontal links to its left are OFF;
  - ON if the state of one or more of the attached horizontal links to its left is ON.
- The state of the vertical link is copied to all of the attached horizontal links on its right.
- The state of the vertical link is not copied to any of the attached horizontal links on its left.

Description	Symbol
Vertical link with attached horizontal links	

### 11.4.3 CONTACTS

A contact is an element which imparts a state to the horizontal link on its right side which is equal to the Boolean AND of the state of the horizontal link at its left side with an appropriate function of an associated Boolean input, output, or memory variable.

A contact does not modify the value of the associated Boolean variable. Standard contact symbols are given in the following table.

Name	Description	Symbol
Normally open contact	The state of the left link is copied to the right link if the state of the associated Boolean variable is ON. Otherwise, the state of the right link is OFF.	+
Normally closed contact	The state of the left link is copied to the right link if the state of the associated Boolean variable is OFF. Otherwise, the state of the right link is OFF.	+ /



Name	Description	Symbol
Positive transition-sensing contact	The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from OFF to ON is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.	$\{P\}$
Negative transition-sensing contact	The state of the right link is ON from one evaluation of this element to the next when a transition of the associated variable from ON to OFF is sensed at the same time that the state of the left link is ON. The state of the right link shall be OFF at all other times.	$\{N\}$


#### 11.4.4 COILS

A coil copies the state of the link on its left side to the link on its right side without modification, and stores an appropriate function of the state or transition of the left link into the associated Boolean variable.

Standard coil symbols are shown in the following table.

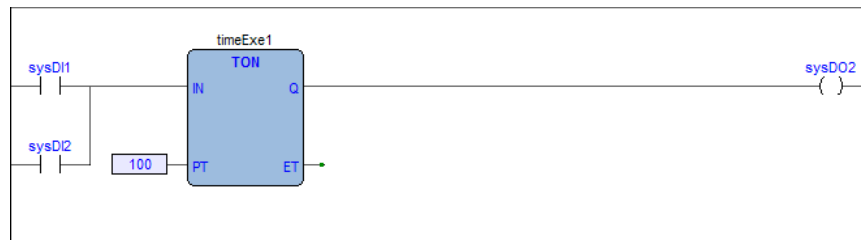
Name	Description	Symbol
Coil	The state of the left link is copied to the associated Boolean variable.	$\{ \}$
Negated coil	The inverse of the state of the left link is copied to the associated Boolean variable, that is, if the state of the left link is OFF, then the state of the associated variable is ON, and vice versa.	$\{ / \}$
SET (latch) coil	The associated Boolean variable is set to the ON state when the left link is in the ON state, and remains set until reset by a RESET coil.	$\{S\}$
RESET (unlatch) coil	The associated Boolean variable is reset to the OFF state when the left link is in the ON state, and remains reset until set by a SET coil.	$\{R\}$
Positive transition-sensing coil	The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from OFF to ON is sensed.	$\{P\}$



Name	Description	Symbol
Negative transition-sensing coil	The state of the associated Boolean variable is ON from one evaluation of this element to the next when a transition of the left link from ON to OFF is sensed.	

### 11.4.5 OPERATORS, FUNCTIONS AND FUNCTION BLOCKS

The representation of functions and function blocks in the LD language is similar to the one used for FBD. At least one Boolean input and one Boolean output shall be shown on each block to allow for power flow through the block as shown in the following figure.



## 11.5 STRUCTURED TEXT (ST)

This section defines the semantics of the ST (Structured Text) language.

### 11.5.1 EXPRESSIONS

An expression is a construct which, when evaluated, yields a value corresponding to one of the data types listed in the elementary data types table. LogicLab does not set any constraint on the maximum length of expressions.

Expressions are composed of operators and operands.

#### 11.5.1.1 OPERANDS

An operand can be a literal, a variable, a function invocation, or another expression.

#### 11.5.1.2 OPERATORS

Open the table of operators to see the list of all the operators supported by ST. The evaluation of an expression consists of applying the operators to the operands in a sequence defined by the operator precedence rules.

#### 11.5.1.3 OPERATOR PRECEDENCE RULES

Operators have different levels of precedence, as specified in the table of operators. The operator with highest precedence in an expression is applied first, followed by the operator of next lower precedence, etc., until evaluation is complete. Operators of equal precedence are applied as written in the expression from left to right.

For example if A, B, C, and D are of type INT with values 1, 2, 3, and 4, respectively, then:

$$A+B-C*ABS(D)$$

yields -9, and:

$$(A+B-C) *ABS(D)$$



yields 0.

When an operator has two operands, the leftmost operand is evaluated first. For example, in the expression

`SIN(A)*COS(B)`

the expression `SIN(A)` is evaluated first, followed by `COS(B)`, followed by evaluation of the product.

Functions are invoked as elements of expressions consisting of the function name followed by a parenthesized list of arguments, as defined in the relevant section.

#### 11.5.1.4 OPERATORS OF THE ST LANGUAGE

Operation	Symbol	Precedence
Parenthesizing	<code>(&lt;expression&gt;)</code>	HIGHEST
Function evaluation	<code>&lt;fname&gt;(&lt;arglist&gt;)</code>	.
Negation Complement	- NOT	.
Exponentiation	**	.
Multiply Divide Modulo	* / MOD	.
Add Subtract	+ -	.
Comparison	<code>&lt;, &gt;, &lt;=, &gt;=</code>	.
Equality Inequality	= <code>&lt;&gt;</code>	.
Boolean AND	AND	.
Boolean Exclusive OR	XOR	.
Boolean OR	OR	LOWEST

#### 11.5.2 STATEMENTS IN ST

All statements comply with the following rules:

- they are terminated by semicolons;
- unlike IL, a carriage return or new line character is treated the same as a space character;
- LogicLab does not set any constraint on the maximum length of statements.

ST statements can be divided into classes, according to their semantics.

##### 11.5.2.1 ASSIGNMENTS

###### Semantics

The assignment statement replaces the current value of a single or multi-element variable by the result of evaluating an expression.

The assignment statement is also used to assign the value to be returned by a function, by placing the function name to the left of an assignment operator in the body of the function declaration. The value returned by the function is the result of the most recent evaluation of such an assignment.



## Syntax

An assignment statement consists of a variable reference on the left-hand side, followed by the assignment operator ":", followed by the expression to be evaluated. For instance, the statement

```
A := B ;
```

would be used to replace the single data value of variable A by the current value of variable B if both were of type `INT`.

## Examples

```
a := b ;
```

assignment

```
pCV := pCV + 1 ;
```

assignment

```
c := SIN( x ) ;
```

assignment with function invocation

```
FUNCTION SIMPLE_FUN : REAL
variables declaration
...
function body
...
SIMPLE_FUN := a * b - c ;
END_FUNCTION
```

assigning the output value to a function

## 11.5.2.2 FUNCTION AND FUNCTION BLOCK STATEMENTS

### Semantics

- Functions are invoked as elements of expressions consisting of the function name followed by a parenthesized list of arguments. Each argument can be a literal, a variable, or an arbitrarily complex expression.
- Function blocks are invoked by a statement consisting of the name of the function block instance followed by a parenthesized list of arguments. Both invocation with formal argument list and with assignment of arguments are supported.
- RETURN: function and function block control statements consist of the mechanisms for invoking function blocks and for returning control to the invoking entity before the physical end of a function or function block. The RETURN statement provides early exit from a function or a function block (e.g., as the result of the evaluation of an IF statement).

### Syntax

1) Function:

```
dst_var := function_name( arg1, arg2 , ... , argN );
```

2) Function block with formal argument list:

```
instance_name(   var_in1 := arg1 ,
                 var_in2 := arg2 ,
                 ... ,
                 var_inN := argN );
```

3) Function block with assignment of arguments:

```
instance_name.var_in1 := arg1;
...
instance_name.var_inN := argN;
```



```
instance_name();
```

#### 4) Function and function block control statement:

```
RETURN;
```

#### Examples

```
CMD_TMR( IN := %IX5,
PT:= 300 ) ;
```

#### FB invocation with formal argument list:

```
IN := %IX5 ;
PT:= 300 ;
CMD_TMR() ;
```

#### FB invocation with assignment of arguments:

```
a := CMD_TMR.Q;
```

#### FB output usage:

```
RETURN ;
```

early exit from function or function block.

### 11.5.2.3 SELECTION STATEMENTS

#### Semantics

Selection statements include the `IF` and `CASE` statements. A selection statement selects one (or a group) of its component statements for execution based on a specified condition.

- `IF`: the `IF` statement specifies that a group of statements is to be executed only if the associated Boolean expression evaluates to the value `TRUE`. If the condition is false, then either no statement is to be executed, or the statement group following the `ELSE` keyword (or the `ELSIF` keyword if its associated Boolean condition is true) is executed.
- `CASE`: the `CASE` statement consists of an expression which evaluates to a variable of type `DINT` (the "selector"), and a list of statement groups, each group being labeled by one or more integer or ranges of integer values, as applicable. It specifies that the first group of statements, one of whose ranges contains the computed value of the selector, is to be executed. If the value of the selector does not occur in a range of any case, the statement sequence following the keyword `ELSE` (if it occurs in the `CASE` statement) is executed. Otherwise, none of the statement sequences is executed.

LogicLab does not set any constraint on the maximum allowed number of selections in `CASE` statements.

#### Syntax

Note that square brackets include optional code, while braces include repeatable portions of code.

##### 1) `IF`:

```
IF expression1 THEN
stat_list
[ { ELSIF expression2 THEN
stat_list } ]
ELSE
stat_list
END_IF ;
```

##### 2) `CASE`:

```
CASE expression1 OF
```



```

intv [ {, intv } ] :
stat_list
{ intv [ {, intv } ] :
stat_list }
[ ELSE
stat_list ]
END_CASE ;
intv being either a constant or an interval: a or a..b

```

## Examples

### IF statement:

```

IF d 0.0 THEN
nRoots := 0 ;
ELSIF d = 0.0 THEN
nRoots := 1 ;
x1 := -b / (2.0 * a) ;
ELSE
nRoots := 2 ;
x1 := (-b + SQRT(d)) / (2.0 * a) ;
x2 := (-b - SQRT(d)) / (2.0 * a) ;
END_IF ;

```

### CASE statement:

```

CASE tw OF
1, 5:
display := oven_temp ;
2:
display := motor_speed ;
3:
display := gross_tare;
4, 6..10:
display := status(tw - 4) ;
ELSE
display := 0;
tw_error := 1;
END_CASE ;

```

## 11.5.2.4 ITERATION STATEMENTS

### Semantics

Iteration statements specify that the group of associated statements are executed repeatedly. The `FOR` statement is used if the number of iterations can be determined in advance; otherwise, the `WHILE` or `REPEAT` constructs are used.

- `FOR`: the `FOR` statement indicates that a statement sequence is repeatedly executed, up to the `END_FOR` keyword, while a progression of values is assigned to the `FOR` loop control variable. The control variable, initial value, and final value are expressions of the same integer type (e.g., `SINT`, `INT`, or `DINT`) and cannot be altered by any of the repeated statements. The `FOR` statement increments the control variable up or down





from an initial value to a final value in increments determined by the value of an expression; this value defaults to 1. The test for the termination condition is made at the beginning of each iteration, so that the statement sequence is not executed if the initial value exceeds the final value.

- **WHILE:** the `WHILE` statement causes the sequence of statements up to the `END_WHILE` keyword to be executed repeatedly until the associated Boolean expression is false. If the expression is initially false, then the group of statements is not executed at all.
- **REPEAT:** the `REPEAT` statement causes the sequence of statements up to the `UNTIL` keyword to be executed repeatedly (and at least once) until the associated Boolean condition is true.
- **EXIT:** the `EXIT` statement is used to terminate iterations before the termination condition is satisfied. When the `EXIT` statement is located within nested iterative constructs, `exit` is from the innermost loop in which the `EXIT` is located, that is, control passes to the next statement after the first loop terminator (`END_FOR`, `END_WHILE`, or `END_REPEAT`) following the `EXIT` statement.

**Note:** the `WHILE` and `REPEAT` statements cannot be used to achieve interprocess synchronization, for example as a "wait loop" with an externally determined termination condition. The SFC elements defined must be used for this purpose.

### Syntax

Note that square brackets include optional code, while braces include repeatable portions of code.

#### 1) FOR:

```
FOR control_var := init_val TO end_val [ BY increm_val ] DO
  stat_list
END_FOR ;
```

#### 2) WHILE:

```
WHILE expression DO
  stat_list
END_WHILE ;
```

#### 3) REPEAT:

```
REPEAT
  stat_list
UNTIL expression
END_REPEAT ;
```

### Examples

#### FOR statement:

```
j := 101 ;
FOR i := 1 TO 100 BY 2 DO
  IF arrvals[i] = 57 THEN
    j := i ;
    EXIT ;
  END_IF ;
END_FOR ;
```

#### WHILE statement:

```
j := 1 ;
WHILE j <=100 AND arrvals[i] <> 57 DO
  j := j + 2 ;
```



```

END_WHILE ;
REPEAT statement:
  j := -1 ;
  REPEAT
    j := j + 2 ;
  UNTIL j = 101 AND arrvals[i] = 57
END_REPEAT ;

```

## 11.6 SEQUENTIAL FUNCTION CHART (SFC)

This section defines Sequential Function Chart (SFC) elements to structure the internal organization of a PLC program organization unit (POU), written in one of the languages defined in this standard, for the purpose of performing sequential control functions. The definitions in this section are derived from IEC 848, with the necessary changes to convert the representations from a standard documentation to a set of execution control elements for a PLC program organization unit.

Since SFC elements require storage of state information, the only program organization units which can be structured using these elements are function blocks and programs.

If any part of a program organization unit is partitioned into SFC elements, the entire program organization unit is so partitioned. If no SFC partitioning is given for a program organization unit, the entire program organization unit is considered to be a single action which executes under the control of the invoking entity.

### SFC elements

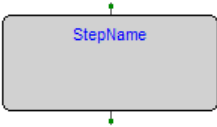
The SFC elements provide a means of partitioning a PLC program organization unit into a set of steps and transitions interconnected by directed links. Associated with each step is a set of actions, and with each transition is associated a transition condition.

### 11.6.1 STEPS

#### 11.6.1.1 DEFINITION

A step represents a situation where the behavior of a program organization unit (POU) with respect to its inputs and outputs follows a set of rules defined by the associated actions of the step. A step is either active or inactive. At any given moment, the state of the program organization unit is defined by the set of active steps and the values of its internal and output variables.

A step is represented graphically by a block containing a step name in the form of an identifier. The directed link(s) into the step can be represented graphically by a vertical line attached to the top of the step. The directed link(s) out of the step can be represented by a vertical line attached to the bottom of the step.

Representation	Description
	Step (graphical representation with direct links)

LogicLab does not set any constraint on the maximum number of steps per SFC.

### Step flag

The step flag (active or inactive state of a step) can be represented by the logic value of a Boolean variable `***_x`, where `***` is the step name. This Boolean variable has the value `TRUE` when the corresponding step is active, and `FALSE` when it is inactive. The scope of



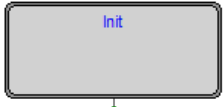
step names and step flags is local to the program organization unit where the steps appear.

Representation	Description
Step Name_x	Step flag = TRUE when Step Name_x is active = FALSE otherwise

### 11.6.1.2 INITIAL STEP

The initial state of the program organization unit is represented by the initial values of its internal and output variables, and by its set of initial steps, i.e., the steps which are initially active. Each SFC network, or its textual equivalent, has exactly one initial step. An initial step can be drawn graphically with double lines for the borders, as shown below. For system initialization, the default initial state is FALSE for ordinary steps and TRUE for initial steps.

LogicLab cannot compile an SFC network not containing exactly one initial step.

Representation	Description
	Initial step (graphical representation with direct links)

### 11.6.1.3 ACTIONS

An action can be:

- a collection of instructions in the IL language;
- a collection of networks in the FBD language;
- a collection of rungs in the LD language;
- a collection of statements in the ST language;
- a sequential function chart (SFC) organized as defined in this section.

Zero or more actions can be associated with each step. Actions are declared via one of the textual structuring elements listed in the following table.

Structuring element	Description
<pre>STEP StepName : (* Step body *) END_STEP</pre>	Step (textual form)
<pre>INITIAL_STEP StepName : (* Step body *) END_STEP</pre>	Initial step (textual form)

Such a structuring element exists in the *lsc* file for every step having at least one associated action.

### 11.6.1.4 ACTION QUALIFIERS

The time when an action associated to a step is executed depends on its action qualifier. LogicLab implements the following action qualifiers.




Qualifier	Description	Meaning
<i>N</i>	Non-stored (null qualifier).	The action is executed as long as the step remains active.
<i>P</i>	Pulse.	The action is executed only once per step activation, regardless of the number of cycles the step remains active.

If a step has zero associated actions, then it is considered as having a *WAIT* function, that is, waiting for a successor transition condition to become true.

**11.6.1.5 JUMPS**

Direct links flow only downwards. Therefore, if you want to return to a upper step from a lower one, you cannot draw a logical wire from the latter to the former. A special type of block exists, called Jump, which lets you implement such a transition.

A Jump block is logically equivalent to a step, as they have to always be separated by a transition. The only effect of a Jump is to activate the step flag of the preceding step and to activate the flag of the step it points to.

Representation	Description
	Jump (logical link to the destination step)

**11.6.2 TRANSITIONS**

**11.6.2.1 DEFINITION**

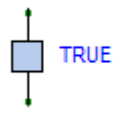
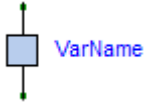
A transition represents the condition whereby control passes from one or more steps preceding the transition to one or more successor steps along the corresponding directed link. The transition is represented by a small grey square across the vertical directed link.

The direction of evolution following the directed links is from the bottom of the predecessor step(s) to the top of the successor step(s).

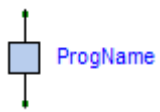
**11.6.2.2 TRANSITION CONDITION**

Each transition has an associated transition condition which is the result of the evaluation of a single Boolean expression. A transition condition which is always true is represented by the keyword `TRUE`, whereas a transition condition always false is symbolized by the keyword `FALSE`.

A transition condition can be associated with a transition by one of the following means:

Representation	Description
	By placing the appropriate Boolean constant { <code>TRUE</code> , <code>FALSE</code> } adjacent to the vertical directed link.
	By declaring a Boolean variable, whose value determines whether or not the transition is cleared.



Representation	Description
	<p>By writing a piece of code, in any of the languages supported by LogicLab, except for SFC. The result of the evaluation of such a code determines the transition condition.</p>

The scope of a transition name is local to the program organization unit (POU) where the transition is located.

### 11.6.3 RULES OF EVOLUTION

#### Introduction

The initial situation of a SFC network is characterized by the initial step which is in the active state upon initialization of the program or function block containing the network.

Evolutions of the active states of steps take place along the directed links when caused by the clearing of one or more transitions.

A transition is enabled when all the preceding steps, connected to the corresponding transition symbol by directed links, are active. The clearing of a transition occurs when the transition is enabled and when the associated transition condition is true.

The clearing of a transition causes the deactivation (or "resetting") of all the immediately preceding steps connected to the corresponding transition symbol by directed links, followed by the activation of all the immediately following steps.

The alternation Step/Transition and Transition/Step is always maintained in SFC element connections, that is:

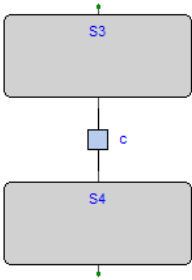
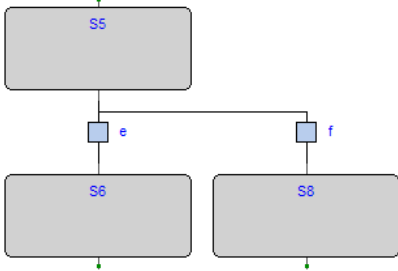
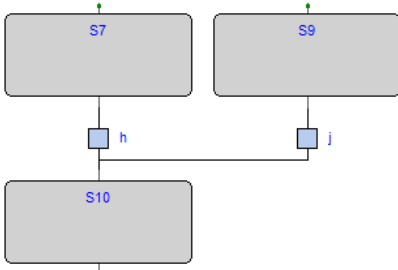
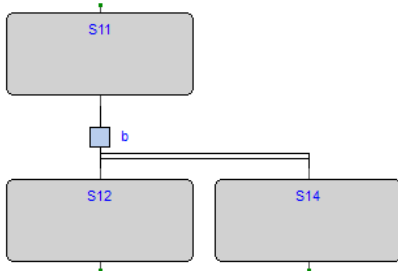
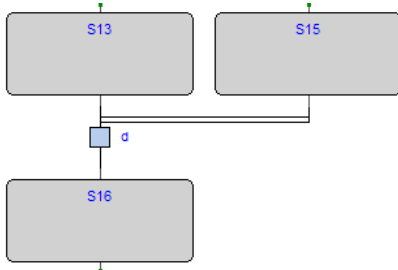
- two steps are never directly linked; they are always separated by a transition;
- two transitions are never directly linked; they are always separated by a step.

When the clearing of a transition leads to the activation of several steps at the same time, the sequences which these steps belong to are called simultaneous sequences. After their simultaneous activation, the evolution of each of these sequences becomes independent. In order to emphasize the special nature of such constructs, the divergence and convergence of simultaneous sequences is indicated by a double horizontal line.

The clearing time of a transition may theoretically be considered as short as one may wish, but it can never be zero. In practice, the clearing time will be imposed by the PLC implementation: several transitions which can be cleared simultaneously will be cleared simultaneously, within the timing constraints of the particular PLC implementation and the priority constraints defined in the sequence evolution table. For the same reason, the duration of a step activity can never be considered to be zero. Testing of the successor transition condition(s) of an active step shall not be performed until the effects of the step activation have propagated throughout the program organization unit where the step is declared.

#### Sequence evolution table

This table defines the syntax and semantics of the allowed combinations of steps and transitions.

Example	Rule
	<p>Normal transition</p> <p>An evolution from step <math>S_3</math> to step <math>S_4</math> takes place if and only if step <math>S_3</math> is in the active state and the transition condition <math>c</math> is <b>TRUE</b>.</p>
	<p>Divergent transition</p> <p>An evolution takes place from <math>S_5</math> to <math>S_6</math> if and only if <math>S_5</math> is active and the transition condition <math>e</math> is <b>TRUE</b>, or from <math>S_5</math> to <math>S_8</math> only if <math>S_5</math> is active and <math>f</math> is <b>TRUE</b> and <math>e</math> is <b>FALSE</b>.</p>
	<p>Convergent transition</p> <p>An evolution takes place from <math>S_7</math> to <math>S_{10}</math> only if <math>S_7</math> is active and the transition condition <math>h</math> is <b>TRUE</b>, or from <math>S_9</math> to <math>S_{10}</math> only if <math>S_9</math> is active and <math>j</math> is <b>TRUE</b>.</p>
	<p>Simultaneous divergent transition</p> <p>An evolution takes place from <math>S_{11}</math> to <math>S_{12}, S_{14}, \dots</math> only if <math>S_{11}</math> is active and the transition condition <math>b</math> associated to the common transition is <b>TRUE</b>. After the simultaneous activation of <math>S_{12}, S_{14}, \dots</math>, the evolution of each sequence proceeds independently.</p>
	<p>Simultaneous convergent transition</p> <p>An evolution takes place from <math>S_{13}, S_{15}, \dots</math> to <math>S_{16}</math> only if all steps above and connected to the double horizontal line are active and the transition condition <math>d</math> associated to the common transition is <b>TRUE</b>.</p>

Examples



Invalid scheme	Equivalent allowed scheme	Note
		<p>Expected behavior: an evolution takes place from S30 to S33 if a is FALSE and d is TRUE.</p> <p>The scheme in the leftmost column is invalid because conditions d and TRUE are directly linked.</p>
		<p>Expected behavior: an evolution takes place from S32 to S31 if c is FALSE and d is TRUE.</p> <p>The scheme in the leftmost column is invalid because direct links flow only downwards. Upward transitions can be performed via jump blocks.</p>

### 11.6.4 SFC CONTROL FLAGS

LogicLab provides some control flags for SFC program or function blocks.

To enable this feature, please refer to paragraph 4.6.2.

Those flags are:

- <POU name>\_HOLD\_SFC (type BOOL);
- <POU name>\_RESET\_SFC (type BOOL).

Where <POU name> means the name of the SFC POU (program or function block).

For example, if the SFC POU is named Main, the control flags will be named Main\_HOLD\_SFC and Main\_RESET\_SFC.

Another couple of actions is available for every SFC action, which also are contained in a SFC POU.

For example, if the above program Main contains a SFC action named Execute, the control flags of this action will be Main\_Execute\_HOLD\_SFC and Main\_Execute\_RESET\_SFC.

These flags functionalities are described in details on next paragraphs.

#### 11.6.4.1 HOLD FLAG

Following the main characteristics of the <POU name>\_HOLD\_SFC flag:

- default value is FALSE;



- when set to `TRUE`, the SFC block, which is referred to ( the one with the same name as `<POU name>`), it is kept in the current status (hold) and no code is executed;
- when the flag is set back to `FALSE`, the SFC block execution is recovered from exactly the same point in which was set to hold, trough `<POU name>_HOLD_SFC := TRUE`.

#### 11.6.4.2 RESET FLAG

Following the main characteristics of the `<POU name>_RESET_SFC` flag:

- default value is `FALSE`;
- when set to `TRUE`, the SFC block, which is referred to (the one with the same name as `<POU name>`), it is brought back to the initial state, that is the execution state of the init action.
- this is an auto-reset flag, which means that if it is set to `TRUE` his own state becomes `FALSE` after his reset action has been executed. It is therefore not necessary to bring the `<POU name>_RESET_SFC` value back to `FALSE`.

#### 11.6.4.3 FLAGS VISIBILITY

The `<POU name>_HOLD_SFC` and `<POU name>_RESET_SFC` flags are automatically generated from the code compiler and they belongs to the local variables of the POU which are referred to.

LogicLab does not show this flags in the variables list of the POU; they are hidden but in any case they can be used everywhere within the code.

#### 11.6.5 CHECK A SFC POU FROM OTHER PROGRAMS

To allow the managing of a SFC POU from other programs LogicLab provides the following functionalities:

- The compiler automatically generates the `<POU name>_RESET_SFC` and `<POU name>_HOLD_SFC` flags.
- If the SFC POU is a function block, the user has the possibility to declare, as `VAR_INPUT` and type `BOOL`, both flags having the name of the SFC POU control flags.
- If the SFC POU is a program, the user has the possibility to declare, as `VAR_GLOBAL` and type `BOOL`, both flags having the name of the SFC POU control flags.
- In both cases above, LogicLab compiler will use the variables declared among the `VAR_INPUT` or `VAR_GLOBAL` ones and not those automatically generated (therefore they will be not generated).

Using these techniques, user then can manage the working state of the SFC POU from other POU using the `INPUT` variables of the SFC POU.

#### Example

```

FUNCTION_BLOCK test
  VAR_INPUT
    ...
    test_RESET_SFC : BOOL; (* Control flag explicitly declared *)
  END_VAR
  ...
END_FUNCTION_BLOCK
PROGRAM Main
  VAR
    ...

```





```
    block : test; (* SFC block instance *)
    END_VAR
    ...
    (* Reset SFC block state *)
    block.test_RESET_SFC := TRUE;
    ...
END_PROGRAM
```

### 11.6.5.1 SFC MACRO LIBRARY

LogicLab makes available to user a library, called *SFCControl.pll*, to allow the manage of the SFC states trough commands instead of variable settings.

This library is composed by macros usable only in ST language.

### 11.6.5.2 USAGE EXAMPLE OF THE CONTROL FLAGS

Following are some example of control flags usage, assuming the SFC POU is named Main:

- Hold (freeze):

```
Main_HOLD_SFC := TRUE;
```

- Restart from hold state:

```
Main_HOLD_SFC := FALSE;
```

- Restart form initial state of a SFC block in hold state:

```
Main_RESET_SFC := TRUE;
```

```
Main_HOLD_SFC := FALSE;
```

- Reset to initial state and instant restart of SFC block:

```
Main_RESET_SFC := TRUE; (* automatic reset from compiler *).
```

## 11.7 LOGICLAB LANGUAGE EXTENSIONS

LogicLab features a few extensions to the IEC 61131-3 standard, in order to further enrich the language and to adapt to different coding styles.

### 11.7.1 MACROS

LogicLab implements macros in the same way a C programming language pre-processor does.

Macros can be defined using the following syntax:

```
MACRO <macro name>
PAR_MACRO
    <parameter list>
END_PAR
<macro body>
END_MACRO
```

Note that the parameter list may eventually be empty, thus distinguishing between object-like macros, which do not take parameters, and function-like macros, which do take parameters.

A concrete example of macro definition is the following, which takes two bytes and composes a 16-bit word:



```

MACRO MAKEWORD
  PAR_MACRO
  lobyte;
  hibyte;
  END_PAR
  { CODE:ST }
  lobyte + SHL( TO_UINT( hibyte ), 8 )
  END_MACRO

```

Whenever the macro name appears in the source code, it is replaced (along with the actual parameter list, in case of function-like macros) with the macro body. For example, given the definition of the macro `MAKEWORD` and the following Structured Text code fragment:

```
w := MAKEWORD( b1, b2 );
```

the macro pre-processor expands it to

```
w := b1 + SHL( TO_UINT( b2 ), 8 );
```

## 11.7.2 POINTERS

Pointers are a special kind of variables which act as a reference to another variable (the pointed variable). The value of a pointer is, in fact, the address of the pointed variable; in order to access the data stored at the address pointed to, pointers can be dereferenced.

Pointer declaration requires the same syntax used in variable declaration, where the type name is the type name of the pointed variable preceded by a `@` sign:

```

VAR
  <pointer name> : @<pointed variable type name>;
END_VAR

```

For example, the declaration of a pointer to a `REAL` variable shall be as follows:

```

VAR
  px : @REAL;
END_VAR

```

A pointer can be assigned with another pointer or with an address. A special operator, `ADR`, is available to retrieve the address of a variable.

```

px := py;          (* px and py are pointers to REAL (that is, variables of type @REAL) *)
px := ADR( x )    (* x is a variable of type REAL *)
px := ?x          (* ? is an alternative notation for ADR *)

```

The `@` operator is used to dereference a pointer, hence to access the pointed variable.

```

px := ADR( x );
@px := 3.141592;  (* the approximate value of pi is assigned to x *)
pn := ADR( n );
n := @pn + 1;    (* n is incremented by 1 *)

```

Beware that careless use of pointers is potentially dangerous: indeed, pointers can point to any arbitrary location, which can cause undesirable effects.

### 11.7.3 WAITING STATEMENT

LogicLab implements a *WAITING* statement that can be used in ST code as following example:

```
...  
WAITING      <condition> DO  
              <code to be executed waiting for condition becomes true>  
END_WAITING;  
...
```

Until the condition is not verified, the code will be executed (not as in a loop cycle but returning to caller in every execution).

The *WAITING* statement can be used only if the associated project option is enabled (see paragraph 4.6.2 for more details).

## 12. ERRORS REFERENCE

### 12.1 COMPILE TIME ERROR MESSAGES

ERROR CODE	SHORT DESCRIPTION	EXPLANATION
A4097	Object not found	The object indicated (variable or function block) has not been defined in the application.
A4098	Unsupported data type	The size (in bits) requested by the indicated data type isn't supported by the target system.
A4099	Auto vars space exhausted	The total allocation space requested by all local variables exceeds the space available on the target system.
A4100	Retentive vars space exhausted	The total allocation space requested by all local retentive variables exceeds the space available on the target system.
A4101	Bit vars space exhausted	The total allocation space requested by all local bit (boolean) variables exceeds the space available on the target system.
A4102	Invalid index in data block	The variable indicated is associated with an index that is not available in the relative data block.
A4103	Data block not found	The variable indicated is associated with a data block that doesn't exist (isn't defined) in the target system.
A4104	Code space exhausted	The total size of code used for POU (programs, functions and function blocks) exceed the space available on the target system.
A4105	Invalid bit offset	The variable indicated is associated with a bit index that is not available in the relative data block.
A4106	Image variable requested	Error code superseded.
A4107	Target function not found	The function indicated isn't available on the target system.
A4108	Base object not found	The indicated instance refers to a function block definition non defined.
A4109	Invalid base object type	The indicated variable is associated with a data type (including function block definition) that isn't defined.
A4110	Invalid data type	The data type used in the variable definition doesn't exist.
A4111	Invalid operand type	The operand type is not allowed for the current operator.
A4112	Function block shares global data and is used by more tasks	The indicated function block is called by more than one task but uses global variables with process image. For this reason the compiler isn't able to refer to the proper image variable for each instance of the function block.
A4113	Temporary variables allocation error	Internal compiler error.
A4114	Embedded functions do not support arrays as input variables	
A4115	Too many parameters input to embedded function	
A4116	Incremental build failed, perform a full build command	
A4117	Less then 10% of free data	
A4118	Less then 10% of free retain data	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
A4119	Less than 10% of free bit data	
A4120	Variable exceeds data block space	
A4121	Element not found	
A4122	Invalid bit mapped type	Bit mapped variables must be of type BOOL
A4123	Invalid access to private member	
A4124	Invalid datablock type for bit mapping	
A4126	Invalid label specification	
A4127	Not a function	Invalid function specification
A4128	Invalid bit mapping index	
A4129	Not a structured type	
A4130	Not a function block instance	
A4131	Incompatible external declaration	
A4132	Label not found	
A4133	Not a variable	
A4134	Index exceeds array size	Index value is out of the array range
A4135	Invalid index data type	
A4136	Missing index(es)	
A4137	Function block instance required	
A4138	Simple variable required	
A4139	Too many indexes	
A4140	Not a structure instance	
A4141	Not an array	
A4142	Invalid symbol specification	
A4143	Not a pointer	
A4144	Double pointer indirection not allowed	
A4145	To be implemented	
A4146	Bit datatype not allowed	
A4147	Unable to calculate variable offset	
A4148	Complex variables cannot have process image	
A4149	Cannot use directly represented variables with process image in function blocks (not implemented)	
A4150	Function block instance not allowed	
A4151	Structure not allowed	
A4152	16-bit variables must be aligned to a 16-bit boundary	
A4153	32-bit variables must be aligned to a 32-bit boundary	
A4154	Temporary string variable allocation error. Instruction shall be split.	

ERROR CODE	SHORT DESCRIPTION	EXPLANATION
A4155	Ext/aux auto vars space exhausted	
A4156	Ambiguous enum value, <enum># prefix required	
A4157	Invalid init element	
A4158	Invalid target function table entry	
A4159	Invalid bit access syntax	
A4160	Invalid bit string type	Bit access allowed only on bit string data types (BYTE, WORD, DWORD)
A4161	Invalid bit index	
A4162	Object is not a method	
A4163	Method not found	
A4164	Invalid usage of THIS/SUPER	
A4165	Parent function block not found	
A4166	Variable name already used into a parent	
A4167	Erroneus method override	Return value or input variables mismatch
A4168	Erroneus local method override	Override of a parent method belonging to a locally implemented interface
A4169	Not an interface instance	
A4170	Not a reference	
A4171	Error dereferencing interfaces	Interfaces can not be dereferenced from references/ pointers
A4172	Relocation table generation failure	
A4173	Bit mapped variables can't be arrays	
C0001	Parser not initialized	Internal compiler error.
C0002	Invalid token	Invalid word for the current language syntax
C0003	Invalid file specification	Internal compiler error.
C0004	Can't open file	The indicated file can't be opened due to a file system error or to a missing source file.
C0005	Parser table error	Internal compiler error.
C0006	Parser non specified	Internal compiler error.
C0007	Unexpected end of file	The indicated file is truncated or the syntax is incomplete.
C0009	Reserved keyword	The indicated word can't be used for declaration purposes because is a keyword of the language.
C0010	Invalid element	The indicated word isn't a valid one for the language syntax.
C0011	Aborted by user	
C0032	Too many parameters in macro call	
C0033	Invalid number of parameters in macro call	
C0034	Too many macro calls nested	
C4097	Invalid variable type	The data type indicated isn't allowed.



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
C4098	Invalid location prefix	The address string of the indicated variable isn't correct, '%' missing.
C4099	Invalid location specification	The address string of the indicated variable isn't correct, the data access type indication isn't 'I', 'Q' or 'M'.
C4100	Invalid location type	The address string of the indicated variable isn't correct, the data type indication isn't 'X', 'B', 'W', 'D', 'R' or 'L'.
C4101	Invalid location index specification	The address string of the indicated variable isn't correct, the index isn't correct.
C4102	Duplicate variable name	The name of the indicated variable has already been used for some other project object.
C4103	Only 0 admitted here	The compiler uses only arrays zero-index based
C4104	Invalid array dimension	The dimension of the array isn't indicated in the correct way (e.g.: contains invalid characters, negative numbers etc.).
C4105	Constant not initialized	Every constant need to have an initial value.
C4106	Invalid string size	
C4107	Initialization exceeding string size	
C4108	Invalid repetition in initialization	
C4109	Invalid data type for initialization	
C4110	Invalid binary file for initialization	
C4112	Duplicate type name	
C4353	Duplicate label	The indicated label has already been defined in the current POU (program, function or function block).
C4354	Constant not admitted	The operation indicated doesn't allow to use constants (typically store or assign operations).
C4355	Address of explicit constant not defined	
C4356	Maximum number of subscripts exceeded	
C4358	Invalid array base	
C4359	Invalid operand	
C4609	Invalid binary constant	A constant value with 2# prefix must contain only binary digits (0 or 1).
C4610	Invalid octal constant	A constant value with 8# prefix must contain only octal digits (between 0 and 7).
C4611	Invalid hexadecimal constant	A constant value with 16# prefix must contain only hexadecimal digits (between 0 and 9 and between A and F).
C4612	Invalid decimal constant	A decimal constant must contain only digits between 0 and 9, a leading sign + or -, a decimal separator '.' Or a exponent indicator 'e' or 'E'.
C4613	Invalid time constant	A constant value with t# prefix must contain a time indication in decimal notation and a time unit between 'ms', 's' or 'm'.
C4614	Invalid constant string	
C4618	Invalid constant wstring	
C4619	Time constant exceeds maximum value	
C4620	LTime constant exceeds maximum value	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
C4621	A non-most significant time unit exceeds its range	
C4622	A non-least significant time unit has a decimal part	
C4623	Invalid date constant	
C4624	Invalid Date and Time typed constant	
C4864	Duplicate function name	The indicated function name has already been used for another application object.
C4865	Invalid function type	The data type returned by the indicated function is not correct.
C5120	Duplicate program name	The indicated program name has already been used for another application object.
C5376	Duplicate function block name	The indicated function block name has already been used for another application object.
C5632	Invalid pragma	
C5633	Invalid pragma value	
C5889	Duplicate macro name	
C5890	Duplicate macro parameter name	
C6144	Invalid resource definition: two or more tasks have the same ID	
C16385	Invalid init value	
C16386	Empty init value	
C16387	Invalid structure init value	Invalid element name in structure init value
C16388	Unexpected token	
C16389	Syntax error	
C16390	Invalid function declaration	Function declaration must begin at line one
C16391	Invalid variable init value	Initial value must begin on the same line as the variable name
C16392	Invalid description	Description exceeded 1024 characters
C16393	Invalid POU name declaration	Declared POU name does not match actual POU name
C16394	Missing POU header	Missing POU header (e.g.: PROGRAM main)
F1025	Invalid network	The indicated FBD or LD network contains a connection error (the errors are normally indicated by red connections).
F1026	Unconnected pin	The indicated block (operator, function, contact or coil) has an unconnected pin.
F1027	Invalid connection (incomplete, more than a source etc.)	Internal compiler error.
F1028	More than one network per block	The network indicated contains more networks of blocks and variables not connected between them.
F1029	Ambiguous network evaluation	The compiler is not able to find an univocal way to establish the order of blocks execution.
F1030	Temporary variables allocation error	Internal compiler error.
F1031	Inconsistent network	The network indicated doesn't have input or output variables.





ERROR CODE	SHORT DESCRIPTION	EXPLANATION
F1032	Invalid object connected to power rail	
F1033	Invalid use of pin negation (ADR operator does not allow negated input)	
F1034	Invalid use of pin negation (SIZEOF operator does not allow negated input)	
F1035	Undefined function block	
F1036	Missing VAR_IN_OUT assignment	
G0001	Invalid operand number	The number of operands is not correct for the operand or the function indicated.
G0002	Variable not defined	The variable has not been defined in the local or global context.
G0003	Label not defined	The label indicated for the JMP operand isn't defined in the current POU (program, function or function block).
G0004	Function block not defined	The indicated instance refers to a function block not defined in the whole project.
G0005	Reference to object not defined	The indicated instance refers to an object not defined in the whole project.
G0006	Constant not admitted	The operation indicated doesn't allow to use constants (typically store or assign operations).
G0007	Code buffer overflow	The total size of code used for POU (programs, functions and function blocks) exceed the space available on the target system.
G0008	Invalid access to variable	The access made to the indicated variable is not allowed. An attempt to write a read-only variable or to read a write-only variable has been made.
G0009	Program not found	The indicated program doesn't exist in the current project.
G0010	Program already assigned to a task	The indicated program has been assigned to more than one task of the target system.
G0011	Can't allocate code buffer	There isn't enough memory on the PC to create the image of the code of the target system.
G0012	Function not defined	The indicated function doesn't exist in the current project.
G0013	Cyclic declaration of function blocks	The indicated function block call itself directly or by means of other functions.
G0014	Incompatible external declaration	The external variable declaration of the current function block doesn't match with the global variable definition it refers to (the one with the same name). Typically is the case of a type mismatch.
G0015	Accumulator extension	
G0016	External variable not found	The external variable doesn't refer to any of the global variables of the project (e.g.: there isn't a global variable with the same name).
G0017	Program is not assigned to a task	The indicated program hasn't been assigned to a task in the target system.
G0018	Task not found in resources	The indicated task isn't defined in the target system.



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0019	No task defined for the application	There aren't task definitions for the target system. The target definition file (*.TAR) is missing or incomplete. Contact the target system vendor.
G0020	Far data allowed only for load/store operations in PROGRAMs	Huge memory access isn't allowed for function blocks, only for programs (error code valid only for some target system with NEAR/FAR data access).
G0021	Invalid processor type	The processor indicated into the target definition file (*.TAR) isn't correct or isn't supported by the compiler.
G0022	Function block with process image variables can't be used in event tasks	
G0023	Process image variables can't be used in event tasks	
G0024	Accumulator undefined	
G0025	Invalid index	
G0026	Only constant index allowed	
G0027	Illegal reference to the address of a register	
G0028	Less then 10% of free code	
G0029	Index exceeds array size	
G0030	Access to array as scalar - assuming index 0	
G0031	Number of indexes not matching the var size	
G0032	Multidimensional variables not supported	
G0033	Invalid data type	
G0034	Invalid operand type	
G0035	Assembler error	
G0036	Aborted by user	
G0037	Element not defined	
G0038	Cyclic declaration of structures	
G0039	Cyclic declaration of typedefs	
G0040	Unresolved definition of typedef	
G0041	Exceeding dimensions in typedef	
G0042	Unable to allocate compiler internal data	
G0043	CODE GENERATOR INTERNAL ERROR	
G0044	Real data not supported	
G0045	Long real data not supported	
G0046	Long data not supported	
G0047	Operation not implemented	
G0048	Invalid operator	
G0049	Invalid operator value	
G0050	Unbalanced parentheses	
G0051	Data conversion	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0052	To be implemented	
G0053	Invalid index data type	
G0054	Negation without condition	
G0055	Operation not allowed on boolean	
G0056	Negation of a non-boolean operand	
G0057	Boolean operand required	
G0058	Floating point parameter not allowed	
G0059	Operand extension	
G0060	Division by zero	
G0061	Comparison between different types	
G0062	Function block must be instantiated	
G0063	String operand not allowed	
G0064	Operation not allowed on pointers	
G0065	Destination may be too small to store current result	
G0066	Cannot use a function block containing external variables with process image in more than one task	
G0067	Cannot load the address of an explicit constant	
G0068	Writing a real value into an integer variable	
G0069	Cannot use complex variables in functions. Not implemented	
G0070	Signed/unsigned mismatch	
G0071	Conversion data types mismatch, possible loss of data	
G0072	Implicit type conversion of boolean to integer	
G0073	Implicit type conversion of boolean to real	
G0074	Implicit type conversion of integer to boolean	
G0075	Implicit type conversion of integer to real	
G0076	Implicit type conversion of real to boolean	
G0077	Implicit type conversion of real to integer	
G0078	Arithmetic operations require numerical operands	
G0079	Bitwise logical operations require bitstring/integer operands	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0080	Comparison operations require elementary (i.e., not user-defined) operands	
G0081	Cannot take the address of a bit variable	
G0082	Writing a signed value into an unsigned variable	
G0083	Writing an unsigned value into a signed variable	
G0084	Implicit conversion from single to double precision	
G0085	Implicit conversion from double to single precision	
G0086	Function parameter extension	
G0087	Casting to the same type has no effects	
G0088	Function parameters wrong number	
G0089	Embedded target function not found	
G0090	Recursive type declaration	
G0091	Wrong initial value. Signed/unsigned mismatch	
G0092	Wrong initial value. Conversion data types mismatch, possible loss of data	
G0093	String will be truncated	
G0094	Init value type mismatch	
G0095	Improper init value	
G0096	Init value object not found	
G0097	Invalid assignment to pointer	
G0098	Unsupported data type	
G0099	Variable bit access not supported	
G0100	Symbolic initialization of constants not supported	
G0101	Type mismatch in assignment	
G0102	Array size mismatch in assignment	
G0103	Copy of array or structures not supported	
G0104	Data size mismatch in assignment	
G0105	Copy of data having a large size (see threshold in project options)	
G0106	Object oriented features not supported	
G0107	Recursive usage of function	
G0108	Recursive usage of method	
G0109	Recursive usage of function block	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0110	Parent function block not found (with EXTENDS)	
G0111	Recursive inheritance (with EXTENDS)	
G0112	Object oriented programming not supported by target system	
G0113	Undefined interface (with IMPLEMENTS)	
G0114	Incomplete interface implementation (with IMPLEMENTS)	
G0115	Method prototype differs from interface definition	
G0116	Redundant interface implementation	
G0117	Function block does not implements interface	
G0118	Copy between different interfaces	
G0119	Parent interface not found	
G0120	Recursive interface hierarchy (EXTENDS)	
G0121	Method redefinition in interface hierarchy (EXTENDS)	
G0122	Invalid operands for query interface operator ?=	
G0123	Invalid assignment to reference	
G0124	Can not load reference/address of an interface	
G0125	Invalid operation on reference	
G0126	Improper assignment to a reference, different type	
G0127	Usage of deprecated pointer initialization, use NULL instead	
G0128	Comparison between pointer and non-pointer	
G0129	Comparison between reference and non-reference	
G0130	Operation between pointer and non-pointer	
G0131	Check for division by zero unsupported for LREAL type	
G0132	Mismatch in ENUM data types	
G0133	Operation between ENUM and generic constant	
G0134	Operation requires explicit type cast	
G0135	Operation required an implicit type cast	
G0136	Type cast is not allowed	
G0137	Initialization of constants with addresses is not allowed	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0138	Illegal conversion to pointer	
G0139	Array dimension constant not found	
G0140	Invalid constant for array size	
G0141	Invalid pointer arithmetic operation	
G0142	VAR_IN_OUT can't be a reference	
G0143	VAR_IN_OUT can be assigned to other VAR_IN_OUT only	
G0144	Only variables can be assigned to VAR_IN_OUT	
G0145	Invalid MOVE operation	
G0146	Found invalid instruction in patch code, could not set breakpoint/trigger	
G0147	Variable bit access with variable index not supported	
G0148	Invalid array size indication	
G0149	Invalid operand on function call	
G0150	Argument types mismatch on function call	
G0151	Operand types mismatch on function invocation	
G0152	Time parameter not allowed	
G0153	Converting a time into a number	
G0154	Converting a time into a string	
G0155	Converting a time into a bool	
G0156	Converting a number into a time	
G0157	Converting a string into a time	
G0158	Implicit conversion of Time to LTime	
G0159	Cannot convert an LTime into a Time implicitly	
G0160	Invalid operation with a time typed operand	
G0161	Operation not allowed on Time operand	
G0162	Destination type not supported for Time type	
G0163	Destination type not supported for LTime type	
G0164	Implicit conversion of DATE to LDATE	
G0165	Destination type not supported for DATE type	
G0166	Destination type not supported for LDATE type	
G0167	Cannot convert an LDATE into a DATE implicitly	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0168	Converting a date into a number	
G0169	Converting a date into a string	
G0170	Converting a date into a bool	
G0171	Converting a number into a date type	
G0172	Operation not allowed on date operand	
G0173	Operation between a date type operand and a non date type operand is not allowed	
G0174	Operation between different date type operands (Date and LDate) is not allowed	
G0175	Operation not allowed on TIME or LTIME	
G0176	Operation not allowed on DATE or LDATE	
G0177	Cannot convert a DATE_AND_TIME into a DATE implicitly	
G0178	Cannot convert a DATE_AND_TIME into an LDATE implicitly	
G0179	Cannot convert an LDATE_AND_TIME into a DATE implicitly	
G0180	Cannot convert an LDATE_AND_TIME into an LDATE implicitly	
G0181	Implicit conversion of DATE_AND_TIME to LDATE_AND_TIME	
G0182	Cannot convert an LDATE_AND_TIME into a DATE_AND_TIME implicitly	
G0183	Operation between a date and time type operand and a non date and time type is not allowed	
G0184	Operation between different date and time type operands (DT and LDT) is not allowed	
G0185	Operation not allowed on date and time type operand	
G0186	Operation not allowed on DATE_AND_TIME or LDATE_AND_TIME	
G0187	Converting a date and time type into a string	
G0188	Converting a DATE into a DATE_AND_TIME	
G0189	Converting a LDATE into a DATE_AND_TIME	
G0190	Converting a DATE into a LDATE_AND_TIME	
G0191	Converting a LDATE into a LDATE_AND_TIME	
G0192	Destination type not supported for DATE_AND_TIME type	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
G0193	Destination type not supported for LDATE_AND_TIME type	
G0194	Date typed parameter not allowed	
G0195	Date and time typed parameter not allowed	
G0196	Converting a floating point into a time	
G0197	Converting a bool into a time type	
G0198	Converting a bool into a date and time type	
G0199	Converting a bool into a date and time type	
G0200	Converting a string into a date and time type	
G0201	Converting a number into a date and time type	
G0202	Converting a date type into a time type	
G0203	Converting a date and time type into a time type	
G0204	Converting a floating point into a date type	
G0205	Converting a string into a date type	
G0206	Converting a bool into a date type	
G0207	Converting a time type into a date type	
G0208	Converting a time type into a date and time type	
G0209	Converting a date type into a floating point	
G0210	Converting a date and time type into a floating point	
G0211	Converting a date and time type into a number	
G0212	Converting a date and time type into a bool	
G0213	Converting a String into WString	
G0214	Converting a WString into String	
G0215	Converting a string into a bool	
G8193	Type definition of unknown data type	
G8194	Type definition has exceeding array dimensions	
G8195	Cyclic definition of data type	
G8196	Double pointers are not supported	
G8197	No enumerative elements	
G8199	Invalid or undefined initialization constant	





<b>ERROR CODE</b>	<b>SHORT DESCRIPTION</b>	<b>EXPLANATION</b>
G8200	Global variable and ENUM field with the same name	
G10241	Too many initializers for variable	
G10242	Too less initializers for variable	
G10243	Constant without init values	
L1153	Unconnected pin	
L1154	Jump to non existing label	
L1155	Invalid operand	
L1156	Undefined contact	
L1157	Undefined variable	
L1158	Undefined constant	
L1159	Undefined coil	
L1160	Undefined jump destination	
L1161	Undefined expression	
L1162	Assignment not admitted in expressions	
L1163	Comments not admitted in expressions	
L1164	Undefined function block	
L1165	VAR_IN_OUT must be assigned in function block invocation	
P2062	Support for processor isn't available	
P2063	Less than 10% of free code	
P2064	Less than 10% of free data	
P2065	Less than 10% of free retain data	
P2066	Less than 10% of free bit data	
P2067	Task not found in resources	
P2068	No task defined for the application	
P2069	Project is in the old PPJ format. It will be saved in the actual PPJX format	
P2070	Can't open auxiliary source file	
P2071	Can't read file	
P2072	Application name is longer than 10 characters: only the first 10 characters will be downloaded into the target	
P2073	Downloadable source code file is not password-protected	
P2074	Downloadable PLC application binary file not created	
P2075	Less than 10% of free ext/aux data	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
P2076	Project private copy of this library was missing and has been replaced with a new copy of the library (from the original path)	
P2077	Cannot load library! Project private copy of this library was missing and the original path to the library is invalid: library has been dropped	
P2079	Debug symbols package (for following download to the target device) not created	
P2080	Source code package (for following download to the target device) not created	
P2081	Invalid task definition	
P2083	Invalid or incoherent task period	
P2084	Broken library link	
P2085	Missing external aux source	
S1281	Generic ST error	
S1282	Too many expressions nested	
S1283	No iteration to exit from	
S1284	Missing END_IF	
S1285	Invalid ST statement	
S1286	Invalid assignment	
S1287	Missing “;”	
S1288	Invalid expression	
S1289	Invalid expression or missing DO	
S1290	Missing END_WHILE	
S1291	Missing END_FOR	
S1292	Missing END_REPEAT	
S1293	Invalid expression or missing THEN	
S1294	Invalid expression or missing TO	
S1295	Invalid expression or missing BY	
S1296	Invalid statement or missing UNTIL	
S1297	Invalid assignment, := expected	
S1298	Invalid address expression	
S1299	Invalid size expression	
S1300	Function return value ignored	
S1301	Invalid parameter passing	
S1302	Function parameter not defined	
S1303	Useless expression	
S1304	Unbalanced parentheses	
S1305	Unknown function	



ERROR CODE	SHORT DESCRIPTION	EXPLANATION
S1306	Invalid function parameter(s) specification	
S1307	Function parameter doesn't exist	
S1308	Multiple assignment not allowed (in accordance with IEC 61131-3)	
S1309	ST preprocessor buffer overflow	
S1310	Function block invocation of a non-function block instance	
S1311	Missing END_WAITING	
S1312	Syntax error	
S1313	Invalid range in CASE definition	
S1314	Value overlap in CASE definition	
S1315	Exceeding number of parameters	
S1316	Wrong number of function parameters	
S1317	Duplicated function parameter	
S1318	Improper use of THIS/SUPER	
S1319	Improper usage of query interface operator ?=	
S1320	Invalid reference to expression	
S1321	Missing IL block end marker ({IL})	
S1322	Function in/out variable doesn't exist	
S1323	VAR_IN_OUT must be assigned in function block invocation	
S1537	Generic SFC error	
S1538	Initial step missing	
S1539	Output connection missing	
S1540	The output pin must be connected to a transition	
S1541	Every output pin of a transition must be connected to a step/jump block	
S1542	Transition expected	
S1543	Step or jump expected	
S1544	Could not find the associate program code	
S1545	Could not find the condition code	
S1546	Unknown-type transition	
S1547	Invalid jump destination	
S1548	Duplicates action. Same SFC action cannot be used in more than one step	
S1549	Unconnected block in SFC schema	

ERROR CODE	SHORT DESCRIPTION	EXPLANATION
T8193	Communication timeout	The communication with the target system failed because there is no answer from the system itself. More common causes of this problem are wrong cable connection, invalid target address in communication settings, invalid settings of communication parameters (such as baud rate), target system failure.
T8194	Incompatible target version	Error code not used.
T8195	Invalid code file	The target system image file (with IMG extension) is invalid or corrupted. Try to upload and create new version of the image file using the "Communication Upload image file" menu option.
T8196	Invalid data block index	The image file (with IMG extension) contains a data block that has an index greater than the largest index supported by the target system. Try to upload and create new version of the image file using the "Communication Upload image file" menu option. If the problem persists, contact the target system vendor.
T8197	Invalid target information address	Internal compiler error.
T8198	Flash erase failure	The target system was not able to complete the flash erasure procedure. Contact the target system vendor for details.
T8199	Code write failure	The target system was not able to complete the flash programming procedure. Contact the target system vendor for details.
T8200	Communication device unavailable	The compiler tried to communicate with the target system but the communication channel is not available. If the problem persists and there are other applications that communicate with the target system, deactivate the communication on the other applications and try again.
T8201	Invalid function index	Internal compiler error.
T8202	Invalid database information address	The address of the parameter's database memory area of the target system isn't correct or valid. Try to upload and create new version of the image file using the "Communication Upload image file" menu option.
T8203	Invalid target information	
T8204	Rebuild required	
T8205	Invalid task	
T8206	Application-level communication protocol error: PLC run-time was not able to understand the received command	
T8207	Not implemented	
T8209	No room for source file on the target	
T8210	Error while uploading source code from target device	
T8211	No room for debug symbols on the target	
T8212	Memory read error	
T8213	Memory write error	
T8214	Not enough space available on the target device for the PLC application binary	



<b>ERROR CODE</b>	<b>SHORT DESCRIPTION</b>	<b>EXPLANATION</b>
T8215	Generic communication failure	