

CODESYS V3.5

First Start

User manual

Version 1.1

24.03.2026

CODESYS V3.5 First Start v1.1 EN 2026

© All rights reserved

Subject to technical changes and misprints

Contents

1. Introduction 5

1.1 About document 5

1.2 About CODESYS V3.5 5

1.3 About CODESYS versions 5

2. Software installation 6

2.1 CODESYS installation 6

2.2 Installation of targets and other packages 10

2.3 Installation of libraries 13

3. Quick start 15

3.1 Project creation 15

3.2 Connection between PC and akYtec PLC 21

3.2.1 Connection over Ethernet 21

3.2.2 Connection over USB 24

3.3 Connection between akYtec PLC and CODESYS 25

3.4 Connection between virtual PLC and CODESYS 29

3.5 Project download, start and management 32

3.6 FAQ 37

3.6.1 No boot application after device reboot 37

3.6.2 How can I remove the application from the PLC? 38

3.6.3 Is it possible to transfer an application from one PLC into another? 38

3.6.4 The correct way to transfer the project to another PC with CODESYS installed 39

3.6.5 How can I view the PLC message log? 40

3.6.6 How do I know if there are errors in the project? 40

3.6.7 With what interval is the program code executed? 41

3.6.8 How to enable grid in visualization editor? 42

3.6.9 Is the size of memory occupied by application limited? 42

4. CODESYS interface 44

4.1 CODESYS interface overview 44

4.2 Main menu 44

4.2.1 File menu 45

4.2.2 Edit menu 45

4.2.3 View menu 46

4.2.4 Project menu 46

4.2.5 Build menu 47

4.2.6 Online menu 48

4.2.7 Debug menu 49

4.2.8 Tools menu 50
4.2.9 Window menu 51
4.2.10 Help menu 51
4.3 Project tree 52
4.3.1 Controller and component nodes 52
4.3.2 Application node 55
4.3.3 POUs tab 56
4.3.4 Project template composition 57
5. Programming basics 59
5.1 IEC 61131-3 programming languages 59
5.2 Variables 62
5.3 Data types 63
5.3.1 BOOL 64
5.3.2 Integer types 64
5.3.3 Floating-point types 65
5.3.4 Strings 66
5.3.5 Date and time 67
5.3.6 Type conversion 68
5.3.7 Arrays 69
5.3.8 User data types 72
5.3.9 Structures 73
5.3.10 Unions 75
5.3.11 Enumerations 76
5.3.12 Aliases 77
5.3.13 Special data types 78
5.4 Variable declaration areas 79
5.4.1 Types of variable declaration areas 79
5.4.2 Local variables (VAR) 79
5.4.3 Input variables (VAR_INPUT) 79
5.4.4 Output variables (VAR_OUTPUT) 79
5.4.5 Input-output variables (VAR_IN_OUT) 79
5.4.6 Global variables (VAR_GLOBAL) 80
5.4.7 Other declaration areas 81
5.5 Declaration area modifiers 81
5.5.1 CONSTANT 81
5.5.2 RETAIN and PERSISTENT 82
5.6 Operators 84

5.6.1 Arithmetic operators 85
5.6.2 Logical operators 85
5.6.3 Comparison operators 86
5.6.4 Bit shift operators 87
5.6.5 Mathematical operators 88
5.6.6 Trigonometric operators 88
5.6.7 Selection operators 89
5.6.8 Conversion operators 90
5.6.9 Assignment operator 91
5.6.10 Size operators 91
5.6.11 Other operators 92
5.6.12 Execution order of operators 92
5.7 Control statements 93
5.7.1 IF conditional statement 93
5.7.2 CASE multiple choice statement 94
5.7.3 FOR loop statement 95
5.7.4 Other control statements 98
5.8 Program objects 99
5.8.1 Functions 99
5.8.2 Functional blocks 101
5.8.3 Programs 103
5.8.4 Comparison of functions, functional blocks, and programs 103
5.8.5 Nested program objects (methods, etc.) 103
5.9 Tasks 104
5.10 Libraries 106
5.10.1 Library installation 106
5.10.2 Add library to project 106
5.10.3 Using library objects in a project. Namespaces. 107
5.10.4 Standard library 110
6. Process control project example 114
6.1 Task definition 114
6.2 Mx210 module configuration 114
6.3 Mx210 module polling configuration using templates 116
6.3.1 New project creation and connection to controller 116
6.3.2 Mx210 template pack installation 119
6.3.3 Modbus component configuration 119
6.3.4 Communication test 123

6.4 Program development 124
6.4.1 Creating structure 124
6.4.2 Assigning variables to Mx210 templates 127
6.4.3 Creating IsValueOutOfLimits function 129
6.4.4 Program code - First version 130
6.4.5 Program code - Second version 131
6.4.6 Creating StatusChange FB 133
6.5 Creating visualization 136
6.5.1 Creating frame 136
6.5.2 Basic elements - Adding and configuring 142
6.5.3 visuMainScreen configuration 146
6.5.4 Alarm configuration 149
6.5.5 visuAlarmTable screen configuration 152
6.6 Connecting controller to akYtec Cloud service 154
7. SPC and PLC comparison 162

1. Introduction

1.1 About document

This document contains the information necessary to get started with CODESYS V3.5. It is intended for beginners without significant experience with programmable PLCs. The document presents only basic information with an emphasis on simplicity and clarity of presentation. The introduction to the CODESYS V3.5 environment is based on akYtec PLCs, but the main part of the document is universal and can be used to learn to work with any PLC programmable in this environment. It is also considered that the reader may not have a PLC at all, so the execution of an application on a virtual PLC installed together with CODESYS is discussed separately.

The document corresponds to the CODESYS V3.5 SP17 Patch 3 version.

Currently (2025) akYtec manufactures the following lines of PLCs programmable in CODESYS V3.5.17.30:

- PLC210-XX
- SPC210

1.2 About CODESYS V3.5

CODESYS V3.5 is an application development environment for programmable PLCs developed by German company CODESYS Group. PLCs from more than 500 manufacturers including the modern line of akYtec PLCs can be programmed in this environment. CODESYS provides IEC 61131-3 programming language editors, visualization editor, configurators for industrial communication protocols, debugging tools and other components. The software is distributed free of charge. Installation package can be downloaded here:

CODESYS 3.5.17.30 IDE 64-Bit

CODESYS 3.5.17.30 IDE 32-Bit

1.3 About CODESYS versions

CODESYS V3.5 is the successor of the CoDeSys V2.3 environment, support for which has been terminated in 2019.

The transition from the V2.3 platform to the V3.0 platform began in 2005. The V3.5 version was released in 2011, the development and support of which continues until now. A major update to the environment, called service pack (SP), is released once a year. Between service pack releases, several patches with bug fixes are issued. Therefore, the full version designation looks as follows:

V3.5 SP17 Patch 3

where

- ***V3.5*** – platform version
- ***SP17*** – service pack number
- ***Patch 3*** – patch number

The version of the environment is written often in a short form: ***3.5.17.30***. The short form is used not only for the environment version, but also for its components: plug-ins, libraries, etc.

A specific PLC cannot be programmed in any version of CODESYS, only the version specified by the PLC manufacturer must be used. The firmware version of the PLC can be viewed in the Web Configurator. The firmware version of an SPC210 is also shown on the device display if the CODESYS user application has not yet been downloaded to the device.

2. Software installation

2.1 CODESYS installation

To install CODESYS, run the downloaded ***.exe*** file.

NOTE

- CODESYS requires an Internet connection for installation, as additional software is downloaded during installation to ensure the full performance of the environment.
- Disable your antivirus software before installing CODESYS.
- Supported operating systems: Windows 8 / 10 / 11.
- Windows 7 is no longer supported by V3.5 SP17 Patch 3.
- To install CODESYS under Linux, read this article.

In the open window (Fig. 2.1.1), click the ***Install*** button to install ***CODESYS Installer***, a utility required to install the target file package (sect. 2.2).

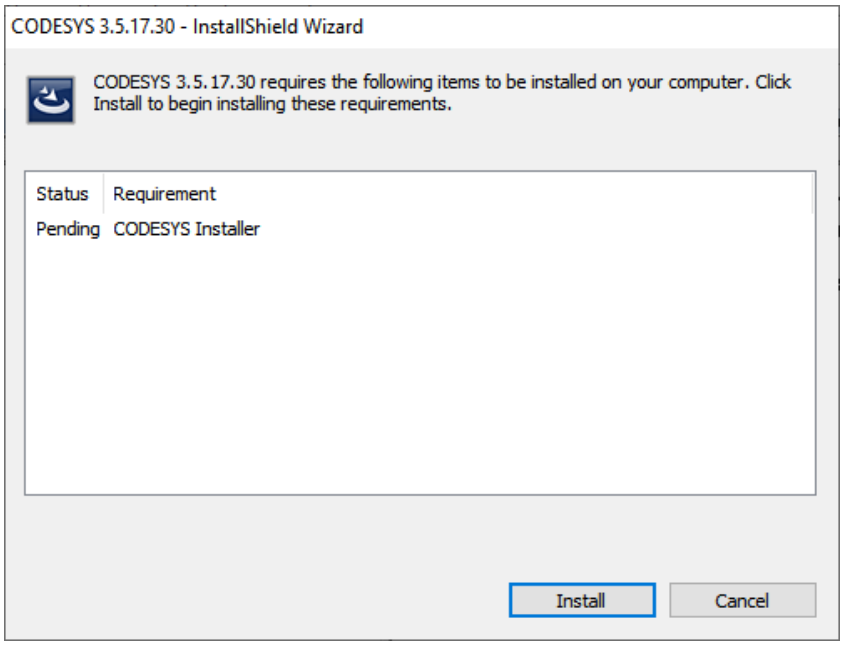


Fig. 2.1.1 CODESYS Installer installation

In the next window (Fig 2.1.2), click the ***Next*** button.

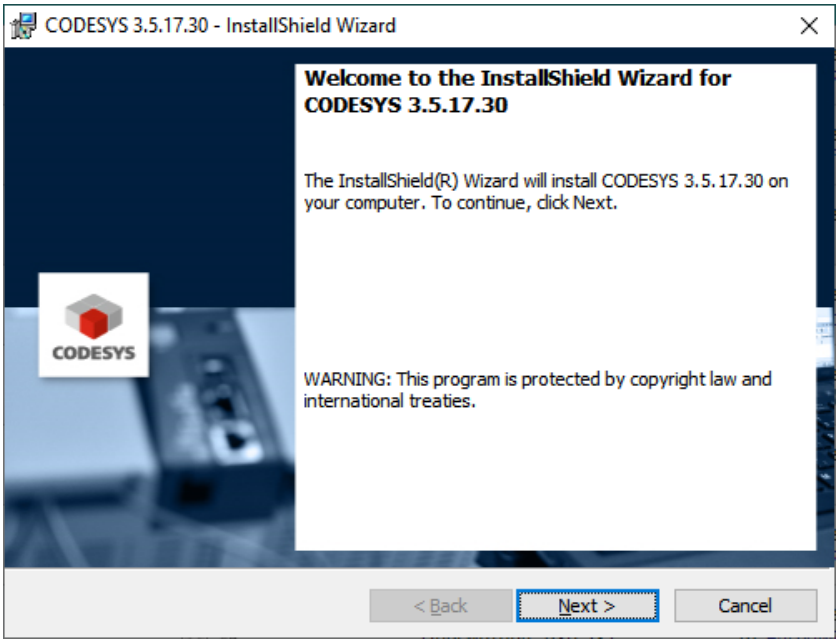


Fig. 2.1.2 Welcome window

In the next window (Fig 2.1.3), read the text of the license agreement, select ***I accept the terms in the license agreement*** and click ***Next***.

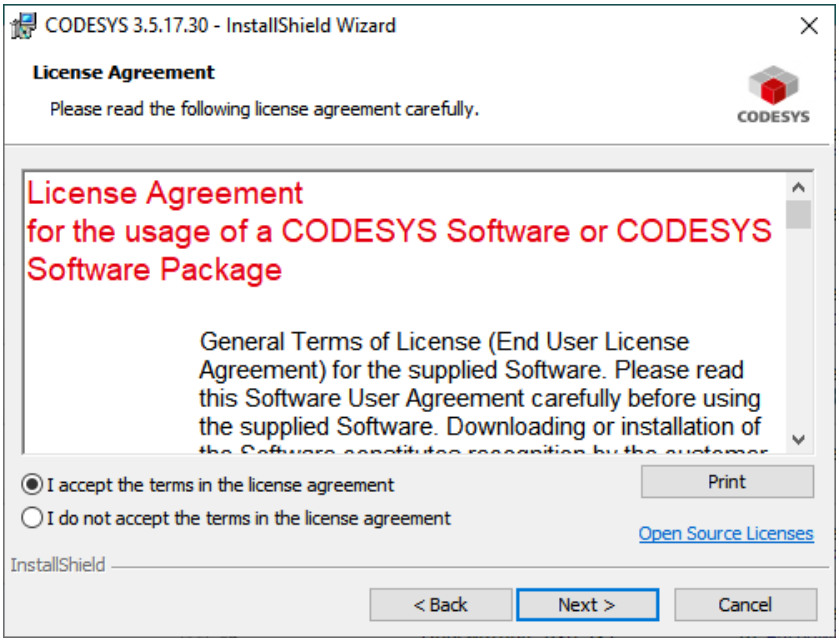


Fig. 2.1.3 License agreement window

In the next window (Fig 2.1.4), read the information about the major changes in the new version, select ***I have read the information***, and click ***Next***.

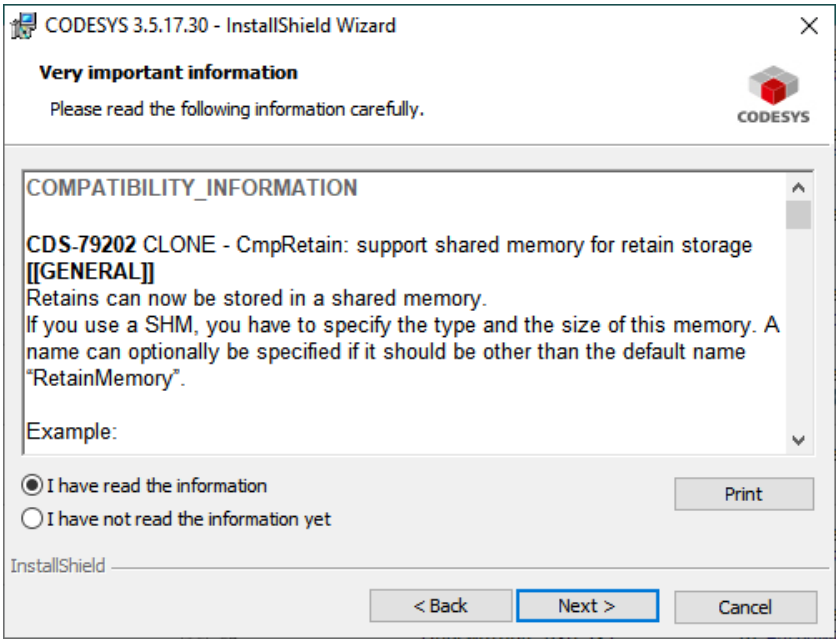


Fig. 2.1.4 Information about the major changes in the new version

In the next window (Fig 2.1.5), select the directory where CODESYS will be installed, and click ***Next***.

NOTE

If you want to use different CODESYS versions on your PC, each version must be installed in a separate directory.

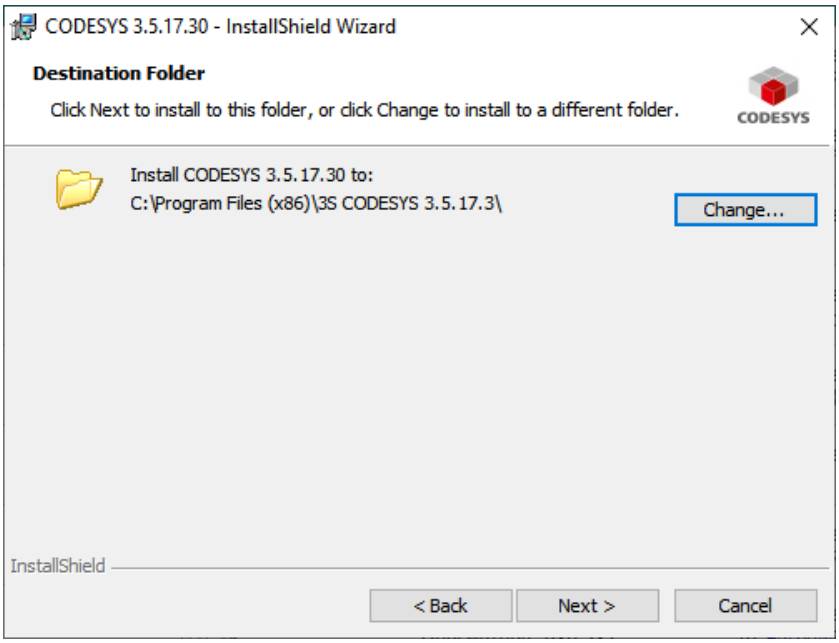


Fig. 2.1.5 Installation directory selection

In the next window (Fig 2.1.6), select a setup type and click ***Next***.

It is recommended to select ***Complete*** type to install all available plug-ins.

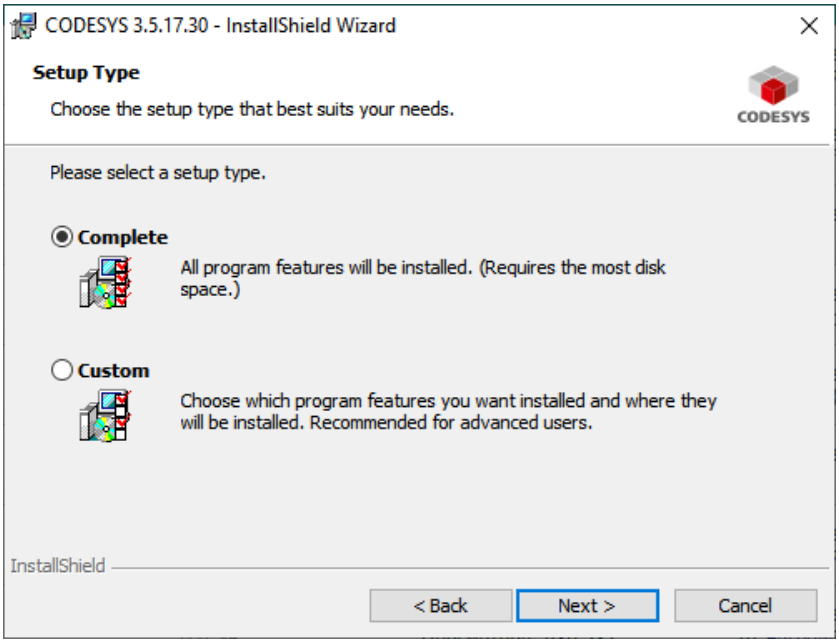


Fig. 2.1.6 Installation type selection

In the next window (Fig 2.1.7), click ***Install*** to start the installation.

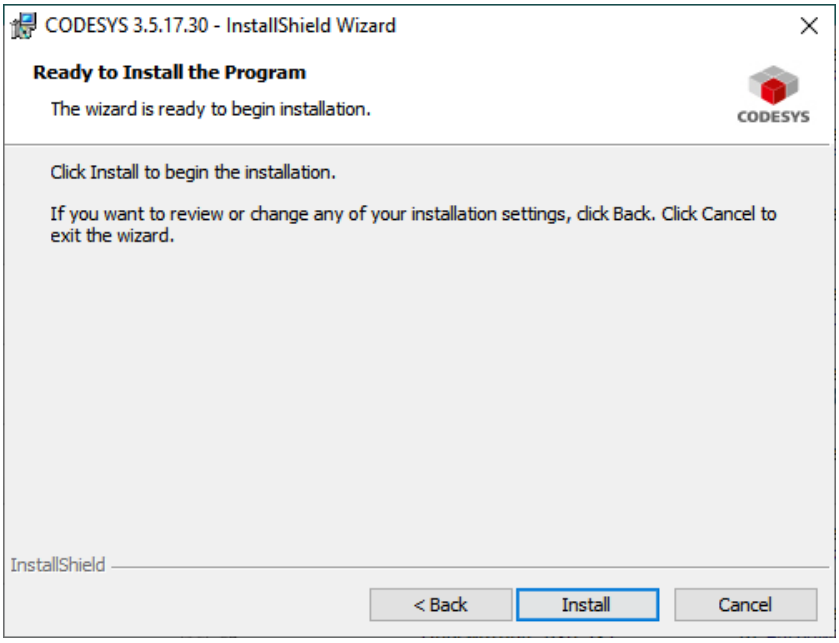


Fig. 2.1.7 Installation start

The installation process takes from 10 minutes to one hour depending on the characteristics of your PC. When the installation is successfully completed, the following window opens:

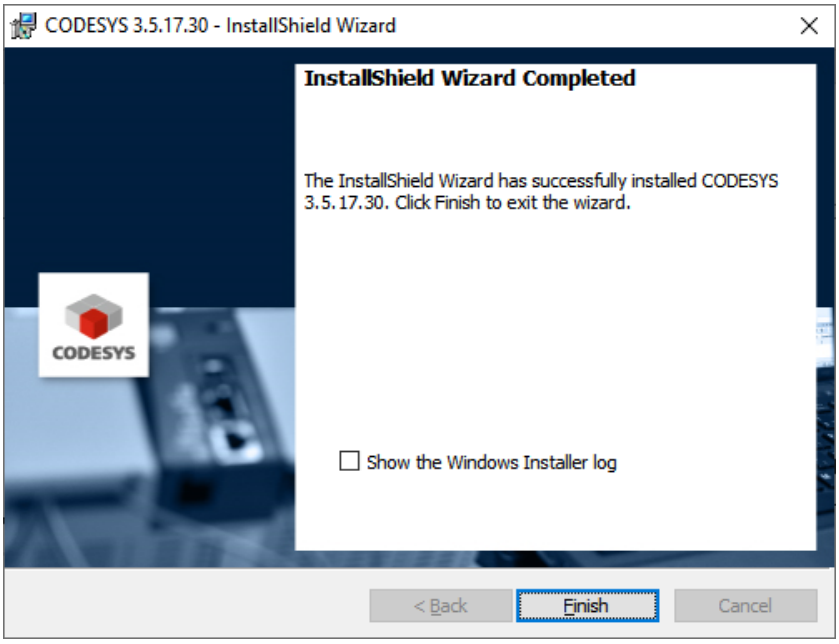


Fig. 2.1.8 Installation completion

Click ***Finish*** to close the window.

2.2 Installation of targets and other packages

Many manufacturers use CODESYS to program their PLCs. Since different PLCs have different hardware and software resources, CODESYS must "know" about them to select a suitable compiler, show the supported functions and hide the unsupported, etc. A ***target file*** that contains all information about hardware and software resources of the device is an integral part of the CODESYS project (e.g., without a target file it is impossible to establish connection to the PLC and download a user application).

Target files for all akYtec PLCs are distributed as a single package file (****.package***) that can be downloaded here. Alternatively, you can download the required version of the target package for your PLC via the Web Configurator.

The firmware version of your PLC determines both the version of CODESYS used to program it and the version of the target package.

The versioning of the target package corresponds to the CODESYS versioning. E.g., the target package version 3.5.17.34 is currently used for PLCs programmed in CODESYS 3.5.17.30. The last digit (4) is the ordinal number of the package, i.e., the next package version will be 3.5.17.35.

In the examples we will use the akYtec target files, so it is necessary to install the akYtec target package.

Starting with CODESYS V3.5 SP17, the packages are installed using the ***CODESYS Installer*** utility. The utility can be launched via Windows Start menu (***CODESYS*** > ***CODESYS Installer***) or from the CODESYS interface menu (***Tools*** > ***CODESYS Installer***).

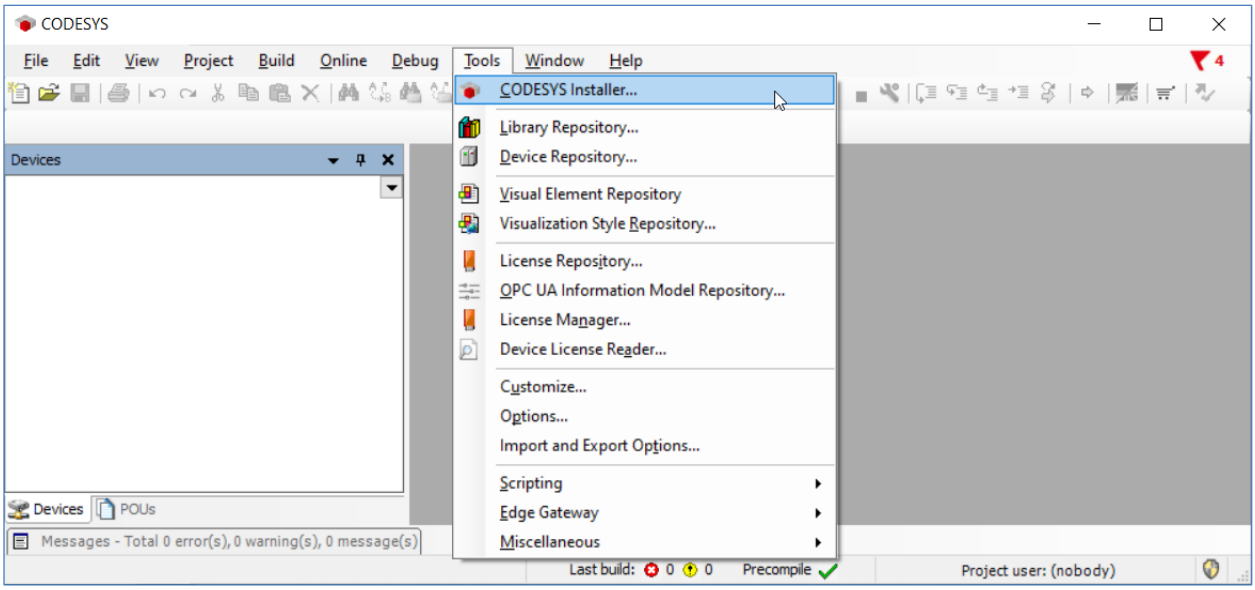


Fig. 2.2.1 Calling CODESYS Installer via CODESYS menu

If the ***CODESYS Installer*** utility is launched from the ***Start*** menu, the list of environments installed on this PC will be displayed (Fig. 2.2.2). An environment is a specific version of CODESYS with a specific set of plug-ins of specific versions. You can create a new environment directly in CODESYS Installer using the ***Add Installation*** button. E.g., if you already have a CODESYS V3.5 SP17 Patch 3 environment installed, you can create another environment with the same CODESYS version and update its plugins to the latest available versions to test their functionality. In this case, you will still have the previously installed "stable" environment with the versions of plug-ins included in the "clean" CODESYS V3.5 SP17 Patch 3 installation package, and these two environments will not affect each other.

Select the environment in which you are going to install the target package and click the ***Change*** button.

If you run CODESYS Installer from CODESYS interface (Fig. 2.2.1), the dialog of this environment will open (Fig. 2.2.3).

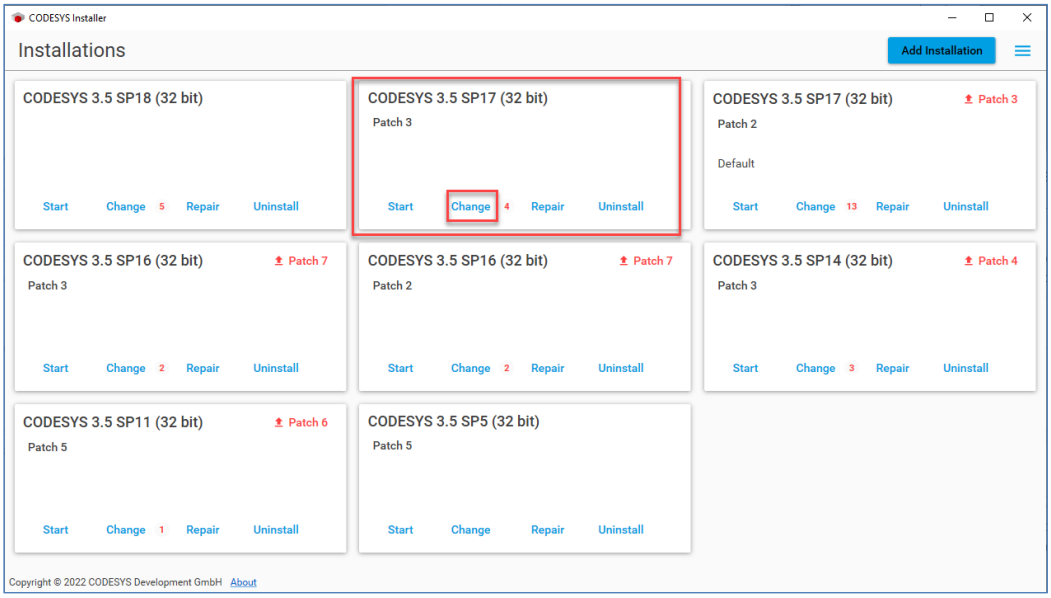


Fig. 2.2.2 List of CODESYS environments installed (user specific)

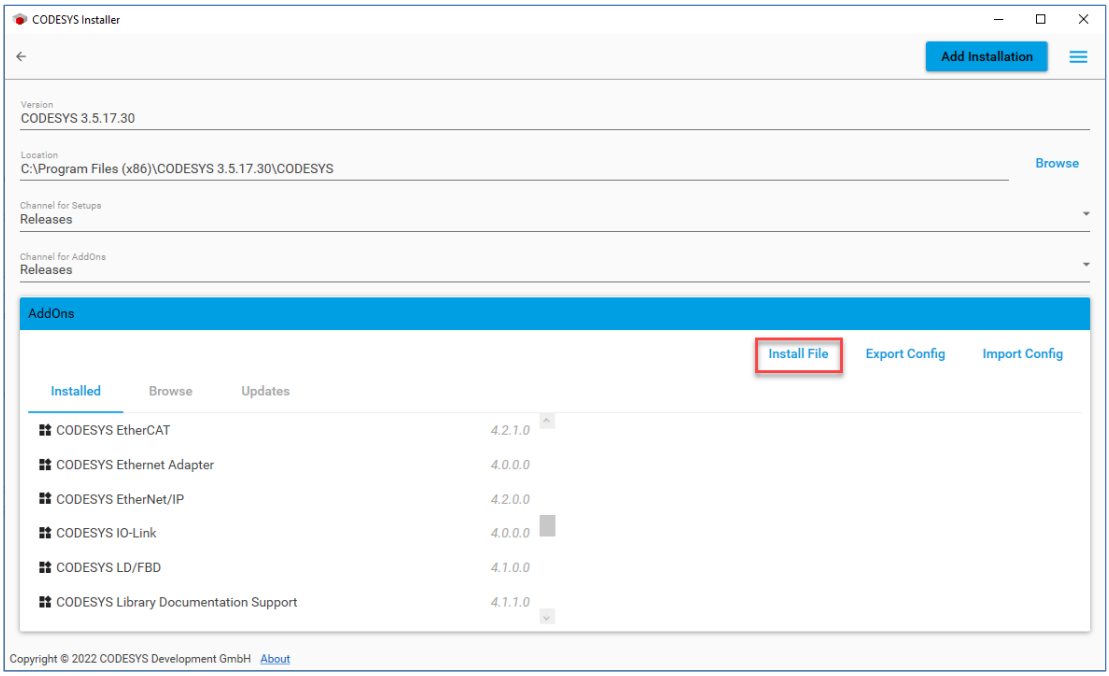


Fig. 2.2.3 Environment view

When the environment dialog window is open (Fig. 2.2.3), close all running instances of the CODESYS, click ***Install File***, select the target package (Fig. 2.2.4) and click ***Open***.

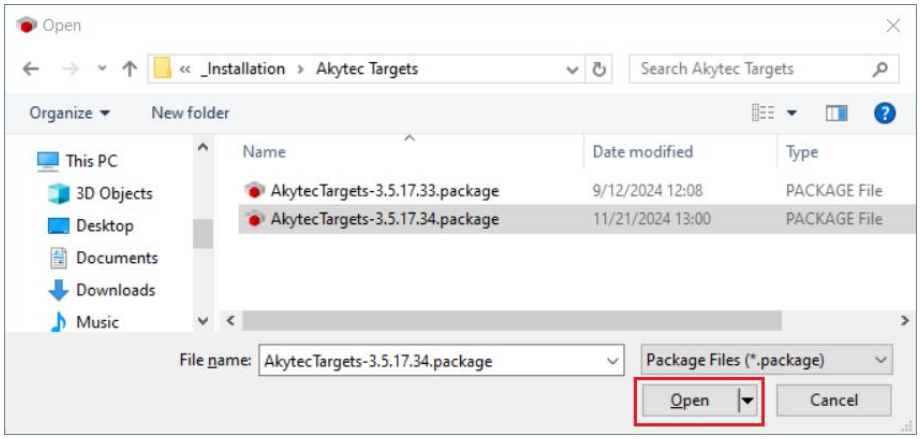


Fig. 2.2.4 Target package selection

In the new window (Fig. 2.2.5), click ***OK*** to confirm the installation of the package.

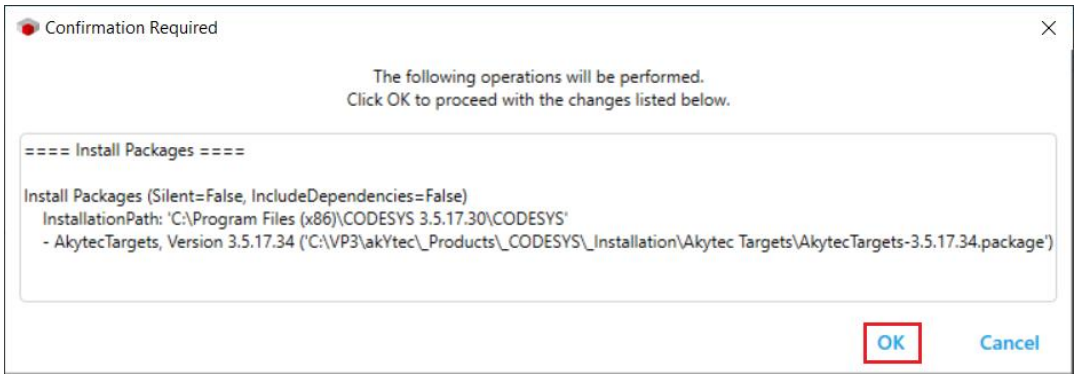


Fig. 2.2.5 Confirm package installation

In the new window (Fig. 2.2.6), check the checkbox "***I want to continue despite the missing signature(s)***" to confirm the installation of the package without digital signature and click ***Continue***.

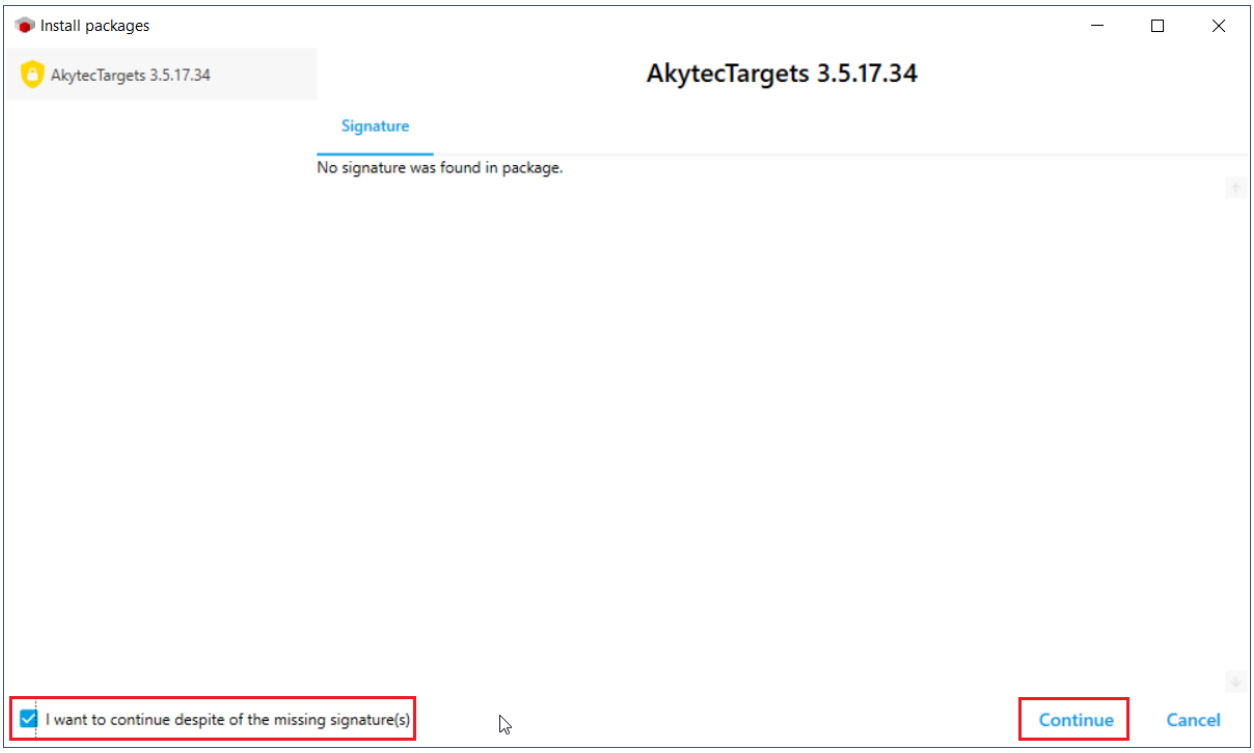


Fig. 2.2.6 Confirm package installation without digital signature

When the installation is completed, click the **Ok** button in the last window.

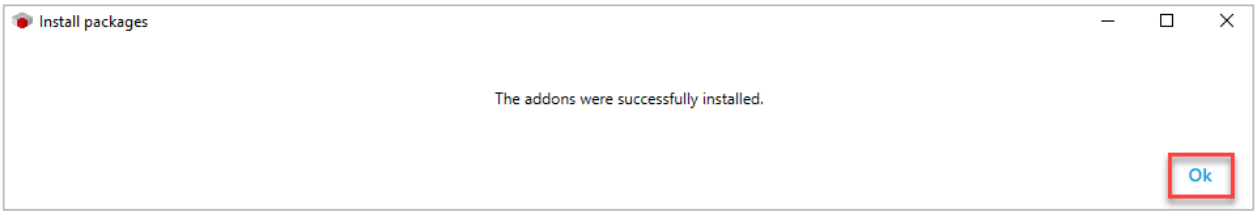


Fig. 2.2.7 Installation completion window

Other packages, such as polling template packages for the akYtec devices, can be installed in the same way.

2.3 Installation of libraries

To extend CODESYS capabilities, PLC manufacturers and users can develop their own objects (function blocks, visualization dialogs, templates etc.) and share them as libraries.

Libraries developed by akYtec and some free libraries from CODESYS users can be downloaded here.

To install a library in CODESYS, select ***Tools*** > ***Library Repository***, click ***Install*** and specify the path to the library file (Fig. 2.3.1).

Libraries can be distributed in two formats:

- ***.compiled-library*** (with protected source code)
- ***.library*** (with unprotected source code)

NOTE

If an unsuitable format is specified in the library file selection window, you will not be able to install the library. Therefore, it is recommended to switch the filter to ***All files***.

In the following examples we will only use the libraries included in the CODESYS installation package. No new libraries need to be installed.

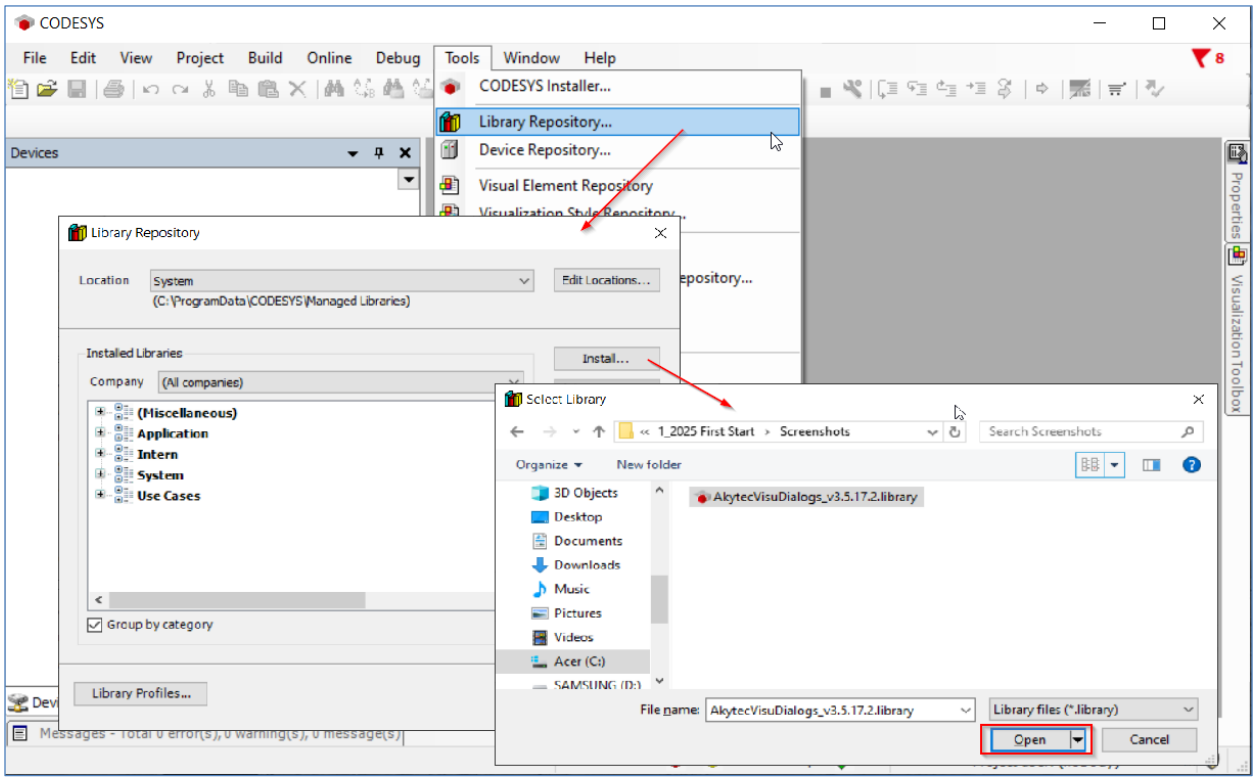


Fig. 2.3.1 Library Installation in Library Repository

3. Quick start

In this section we will learn to create a simple project and download it to the PLC. The aim of the section is to get a user application running on the PLC as soon as possible, so most aspects of project creation are not explained in detail.

3.1 Project creation

To create a new project, use the menu items ***File*** > ***New project***. To open an existing project, select ***Open project*** or ***Recent projects*** (Fig. 3.1.1).

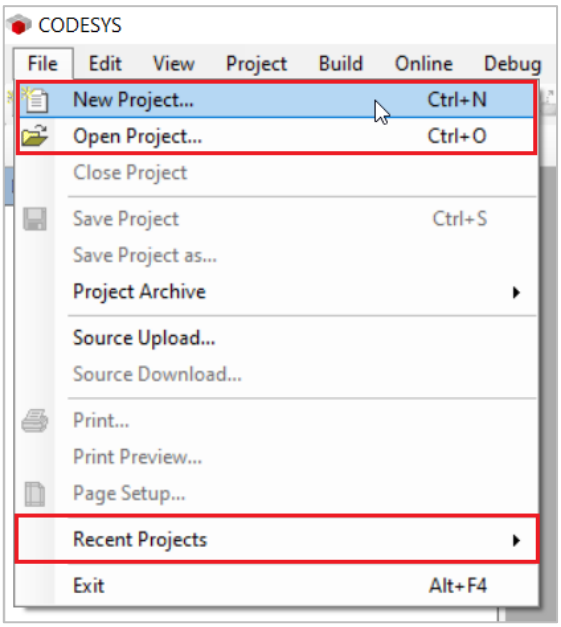


Fig. 3.1.1 Create a new project or open an existing one

After that, the project wizard window will open. (Fig. 3.1.2). If you have already installed the akYtec target package, the window will show the project templates. If not yet, it is time to install it (sect. 2.2).

Target files can be treated as project templates because they contain not only the necessary information about the device resources, but also a number of pre-configured components: interfaces, visualizations etc. It is much easier to create a project based on templates than to do it from scratch. We will discuss the SPC210 panel as an example, so we will take its template. If you are working with another akYtec PLC, select the appropriate template. If you do not have a PLC at all, you can take any template. In sect. 3.4 we will explain how to work with a virtual PLC.

In the lower part of the window, you can determine the name of the project file and the directory in which it will be saved.

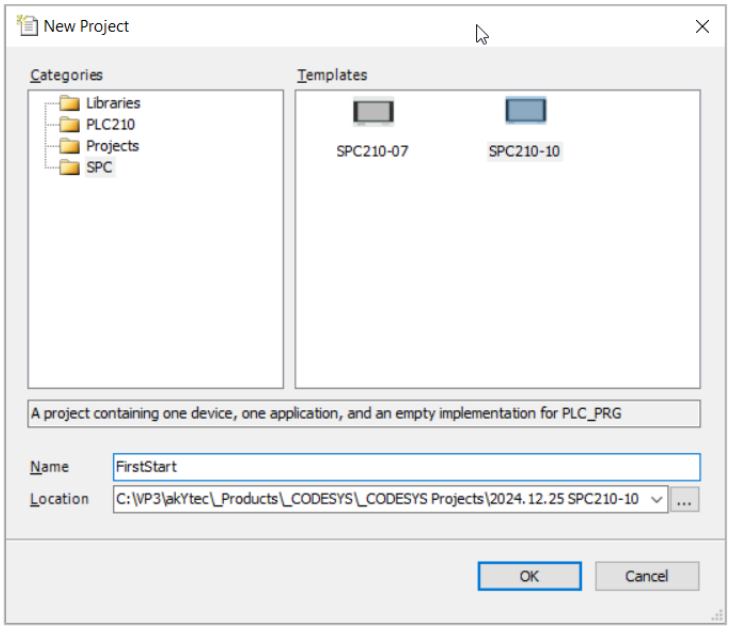


Fig. 3.1.2 Project template selection

The CODESYS interface looks as follows:

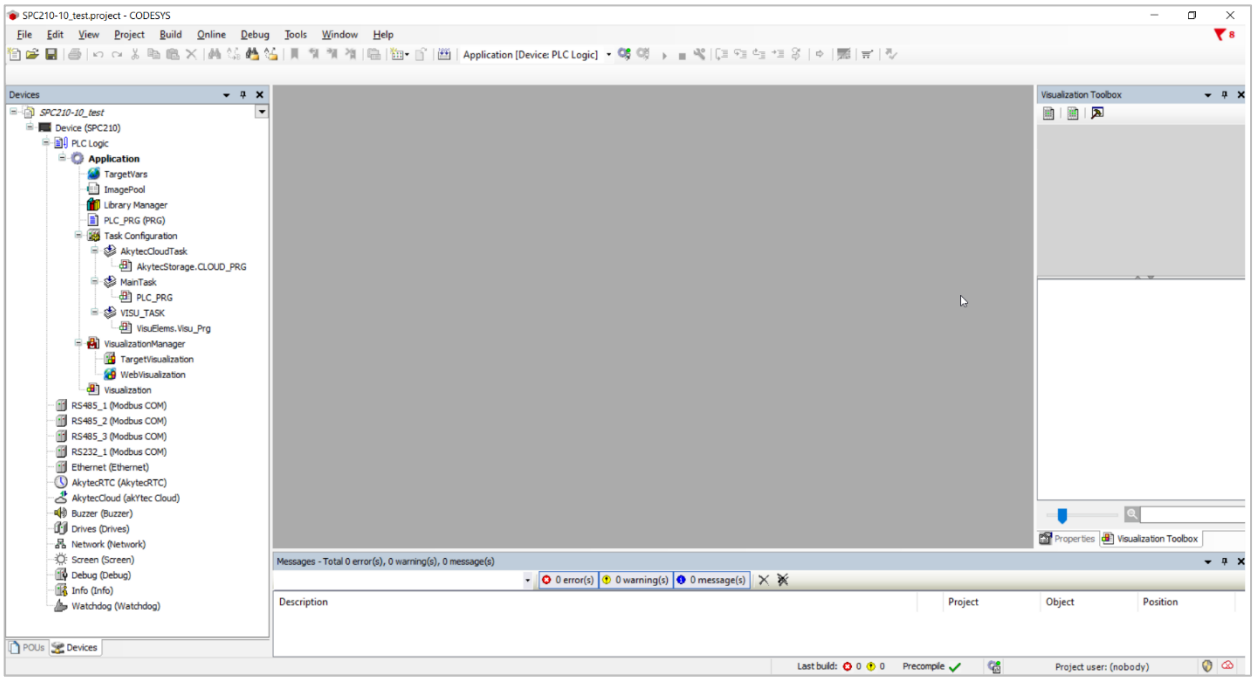


Fig. 3.1.3 CODESYS user interface

The panel ***Devices*** (left) contains a tree with project components: programs, function blocks, visualizations, exchange protocol nodes, etc. The currently open editor (the programming language editor or the visualization editor) is displayed in the center panel. The panel ***ToolBox*** (to the right) is the toolbar of the open editor. Various panels can be displayed at the window bottom: the compilation message panel, the monitoring panel, etc.

To open the main program in an editor, double-click the ***PLC_PRG*** item in the node ***Application*** in the project tree. It will open in the ST editor since the ST language is selected for the main program in the project template.

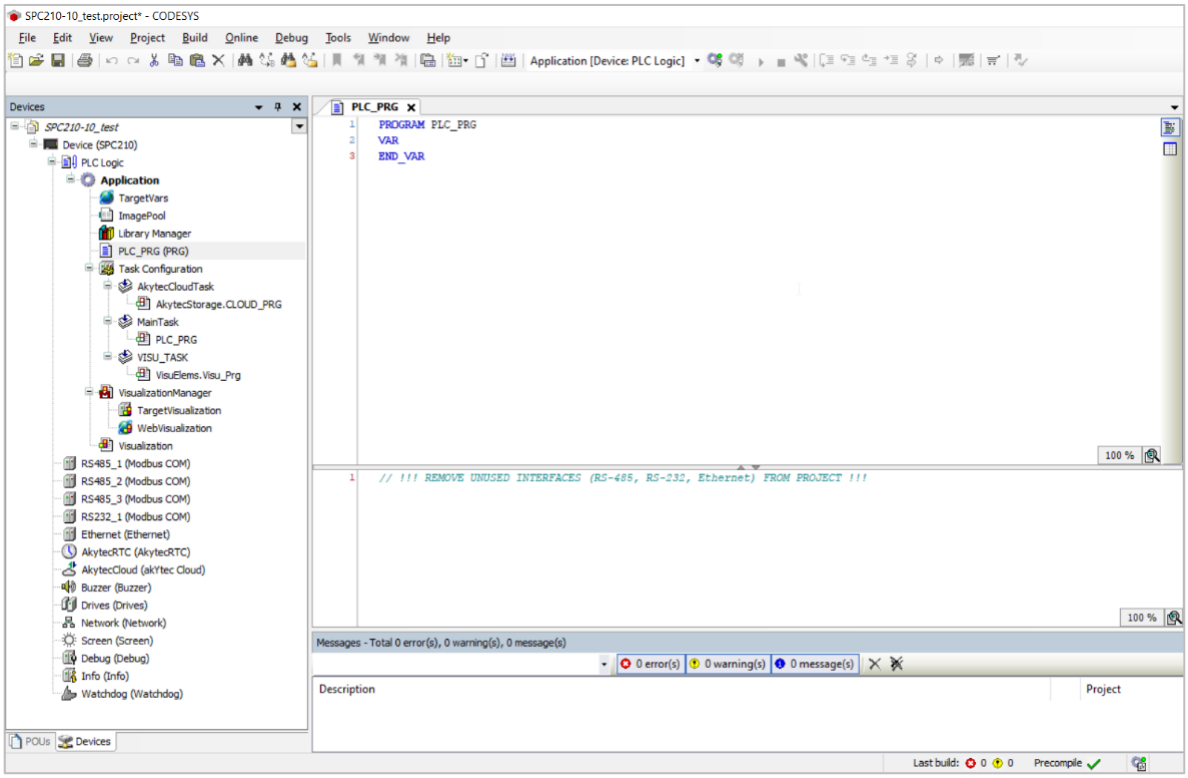


Fig. 3.1.4 ST Editor

The ST editor panel (as well as other programming editors in CODESYS) consists of two parts – in the upper one (declaration part) the variables are declared and the lower one (Instruction part) contains the program code and uses the resources of the declaration part.

Variables are declared within specific blocks (e.g. ***INPUT***, ***OUTPUT***, ***GLOBAL***, etc.). By default, the ***PLC_PRG*** program contains only one empty block between the keywords ***VAR*** and ***END_VAR*** – the block of local (internal) variables.

We declare two variables:

- ***iValue*** – variable of ***INT*** type represents the value of a certain parameter (e.g. a signal from the temperature sensor connected to the input module, or a slider position)
- ***xAlarm*** – variable of ***BOOL*** type represents an alarm signal that indicates that the value of ***iValue*** has exceeded a permissible limit (e.g., ***50***).

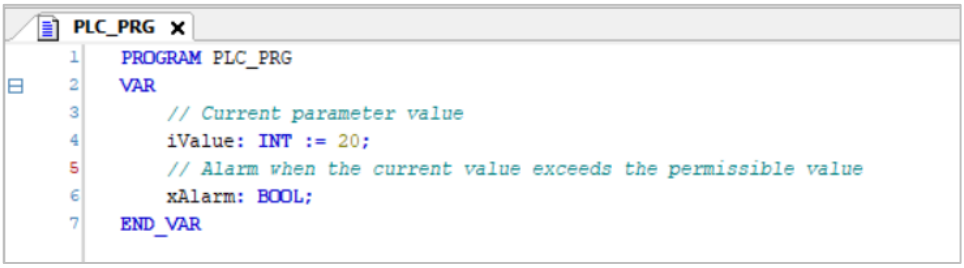


Fig. 3.1.5 Variable declaration

When declaring a variable, you must specify its name and data type. If necessary, you can specify the initial value – in the example above, ***iValue*** has an initial value of ***20***. The variable name and its type are separated by a colon '**:**', and the declaration line ends with a semicolon '**;**'.

Add a comment explaining the meaning of the variable. One-line comment begins with double forward slash '*//*'.

Now we will write the program code – in the example it consists of only one line.

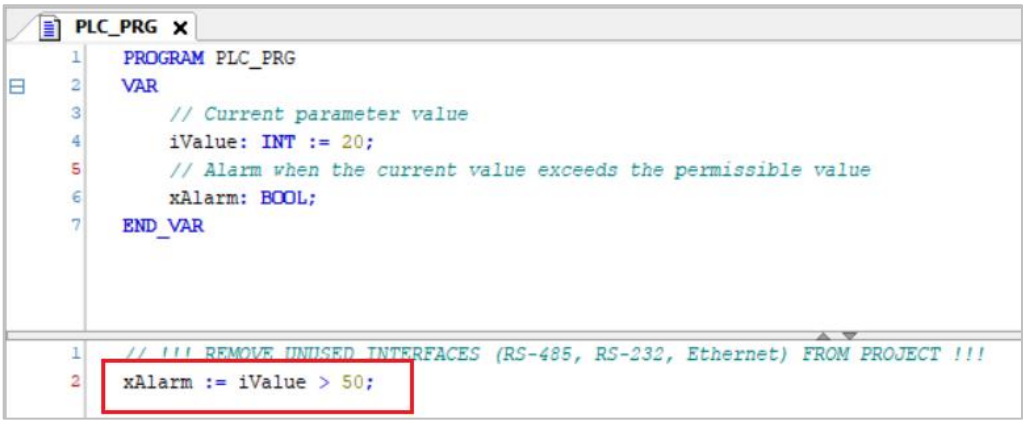


Fig. 3.1.6 Program code

In this line, the value of the variable ***iValue*** is compared with the constant value ***50***. The result of the comparison is a logical value: if ***iValue*** > ***50***, the expression result is ***TRUE***, and if it is not (***iValue*** ≤ ***50***), the result is ***FALSE***. The result of the comparison is written to the variable ***xAlarm*** using the assignment operator '**:=**'. Space between these two characters is not allowed.

In the first line of the ***PLC_PRG*** program in the project template there is a comment with a recommendation to delete unused interfaces from the project, otherwise they will waste the PLC's resources. In our example, the interfaces are not used at all, so we will select all of them in the project tree (multiple selection with ***<Shift>*** key) and select the item ***Delete*** from the context menu (right mouse button on the selection) (Fig. 3.1.7).

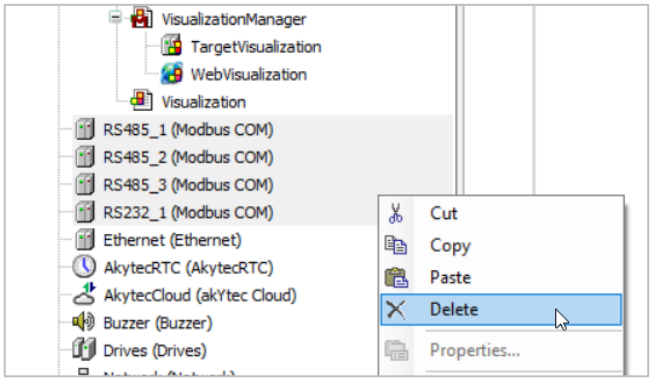


Fig. 3.1.7 Removing unused interfaces from the project tree

Now we will create the visualization screen. Double-click the node ***Visualization*** to open the visualization editor.

For convenient working in the visualization editor, it is recommended to enable the grid: open ***Tools*** > ***Options*** > ***Visualization*** > ***Grid*** and check the checkbox ***Visible***.

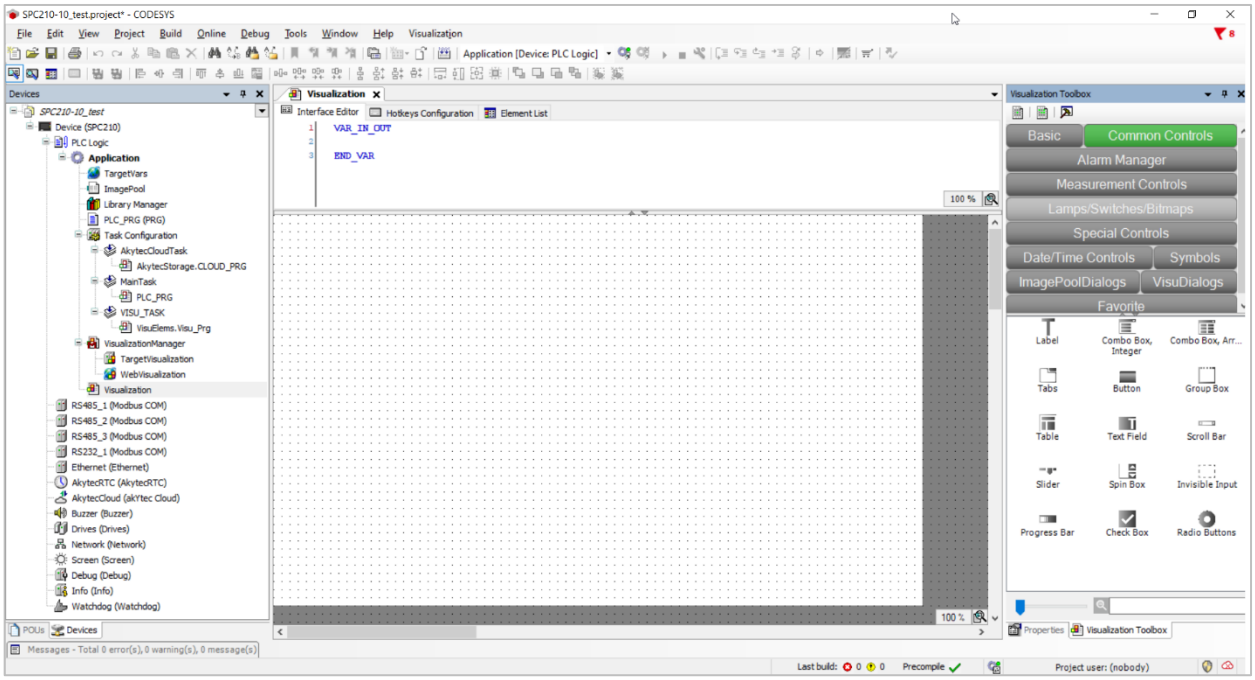


Fig. 3.1.8 Visualization editor with Visualization Toolbox panel

The visualization elements in the ***Visualization Toolbox*** panel (Fig. 3.1.8) are grouped in tabs. Select the ***Slider*** element in the ***Common controls*** tab and drag-and-drop it on the screen (or select it and click in the screen). Add the ***Lamp*** element from the ***Lamps/Switches/Bitmaps*** tab in the same way.

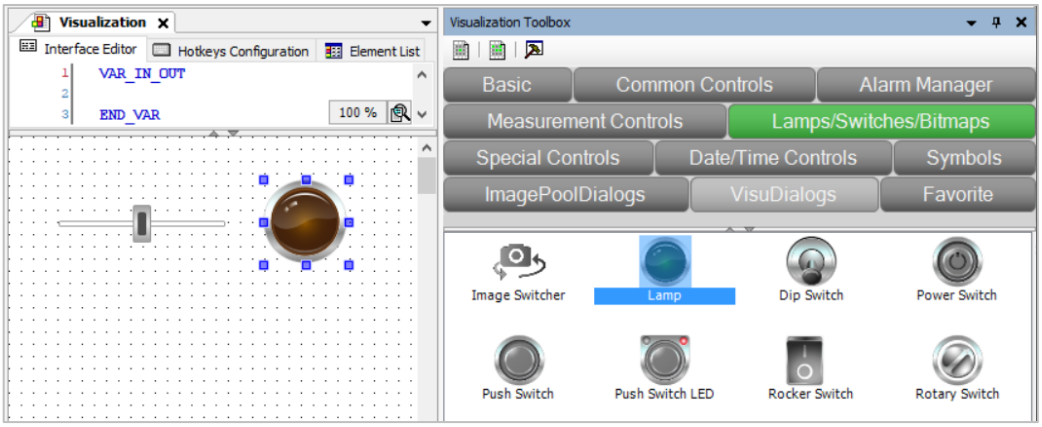


Fig. 3.1.9 Placing Slider and Lamp elements on the screen

To position an element in the screen, drag and drop it to the desired position. To resize an element, move one of its reference points with the left mouse button pressed (highlighted in blue in Fig. 3.1.9).

When an element is selected, the properties panel opens to the right of the screen.

Now we will customize the ***Slider*** element: set its size, define the scaling and assign the variable ***iValue*** to it. To assign a variable, double-click the left mouse button on the empty line next to the parameter ***Variable***, then click the "**...**" button that appears to the right of it and select the desired variable from the list of project variables.

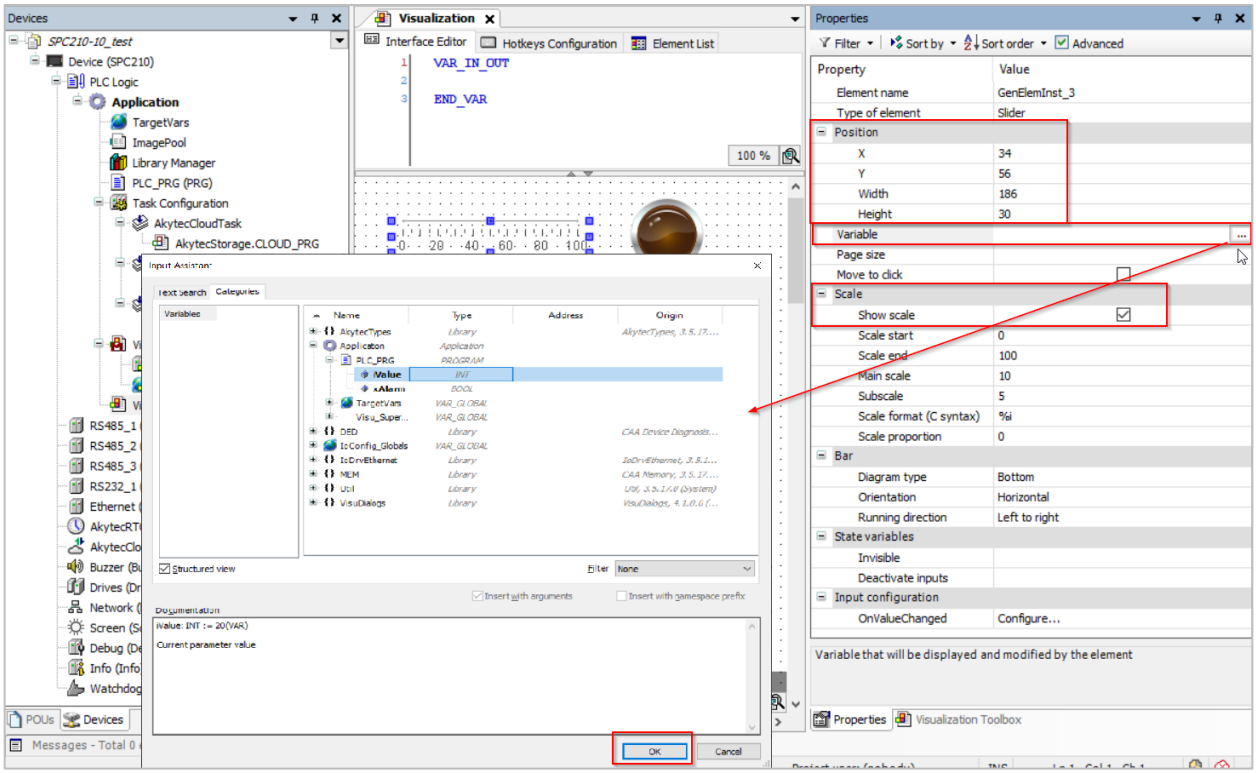


Fig. 3.1.10 Slider element settings

In the same way, set the position and dimensions of the ***Lamp*** element, set its color to red and assign the ***xAlarm*** variable to it.

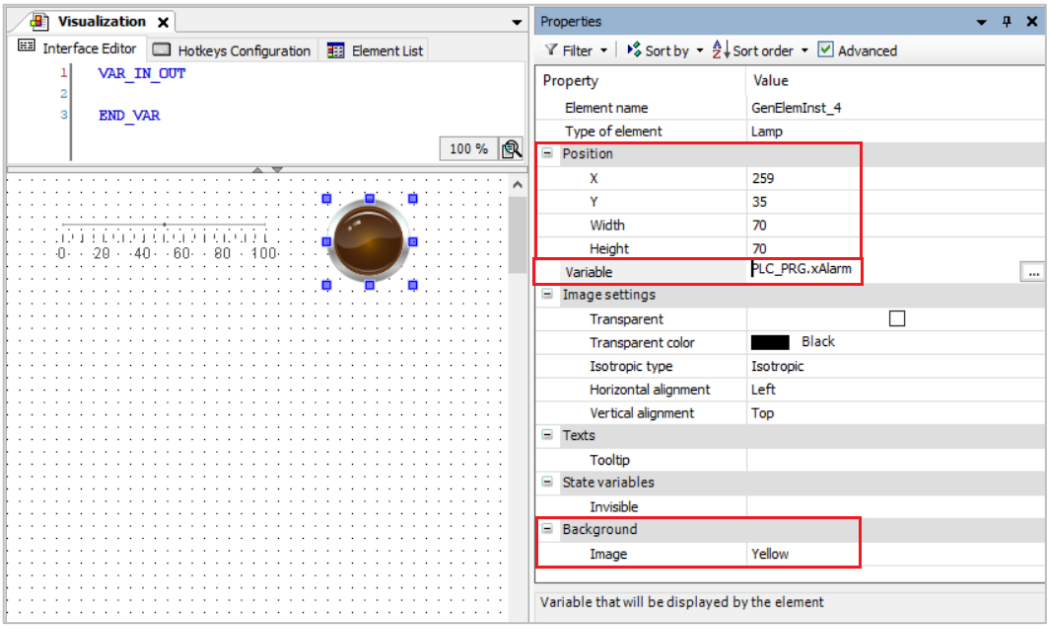


Fig. 3.1.11 Lamp element settings

Our simple project is ready. Now we can download it to the PLC and make sure that it functions properly. If you already have an akYtec PLC, go to sect. 3.2. If you don't have a PLC yet, go to sect. 3.3 that describes how to run the project on a PC using a virtual PLC ***CODESYS Control Win V3***.

3.2 Connection between PC and akYtec PLC

Connection between PC and akYtec PLC can be established over Ethernet (sect. 3.2.1) or USB interface (sect. 3.2.2).

3.2.1 Connection over Ethernet

There are two basic variants of mutual positioning of PLC and PC in the network:

- The PLC and the PC are in the **same local area network** (direct connection between PLC and PC is a special case of this variant).
- The PLC and the PC are in **different local area networks**, connected using appropriate network devices (routers).

We will consider an example of the first variant with Ethernet ports of the PLC and PC connected directly using RJ45-RJ45 cable.

> **NOTE**
>
> There must be no IP address conflicts in a local network, i.e., different devices must not have the same address.

First, the network parameters of the PLC must be defined. For the default network settings of the akYtec-PLC's Ethernet interfaces, see Tab. 3.2.1.

Table 3.2.1 Default Ethernet settings

PLC	Interface	DHCP Client	IP address	Subnet mask	Gateway
PLC210	Ethernet 1–3	off	192.168.0.10	255.255.0.0	192.168.0.1
	Ethernet 4	on	-	-	-
SPC210	Ethernet	off	192.168.0.10	255.255.0.0	192.168.0.1

We consider the connection between the PC and the PLC with the default network settings.

The interface with a static IP address should be used for connection.

The DHCP client mode is used to obtain network settings automatically when the PLC is connected to a local network with a DHCP server. Since our PLC is directly connected to the PC, this option will not work in most cases.

To change the PLC's network settings, use the Web Configurator. If you are working with an SPC210 touchscreen controller, you can also use the display configurator.

To start the Web Configurator, connect the PLC to the PC over Ethernet interface and enter the IP-address of the interface in the address bar of your web browser. On the open login page (Fig. 3.2.1), enter the username "**root**" and the password. The default password is "***akytec***".

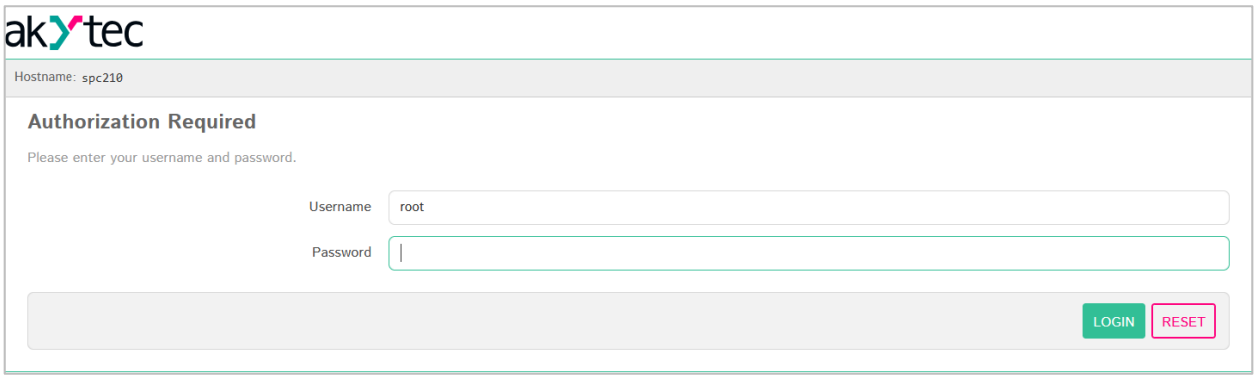


Fig. 3.2.1 Login page in Web Configurator

The first time the PLC is connected to the Web Configurator, the ***Setup Wizard*** will be launched to configure the PLC's network settings. If necessary, the ***Setup Wizard*** can be restarted at any time on the ***System*** > ***Setup Wizard*** tab. Network settings can also be set on the ***Network*** > ***Interfaces*** tab. Here, select the required interface and click the ***Edit*** button.

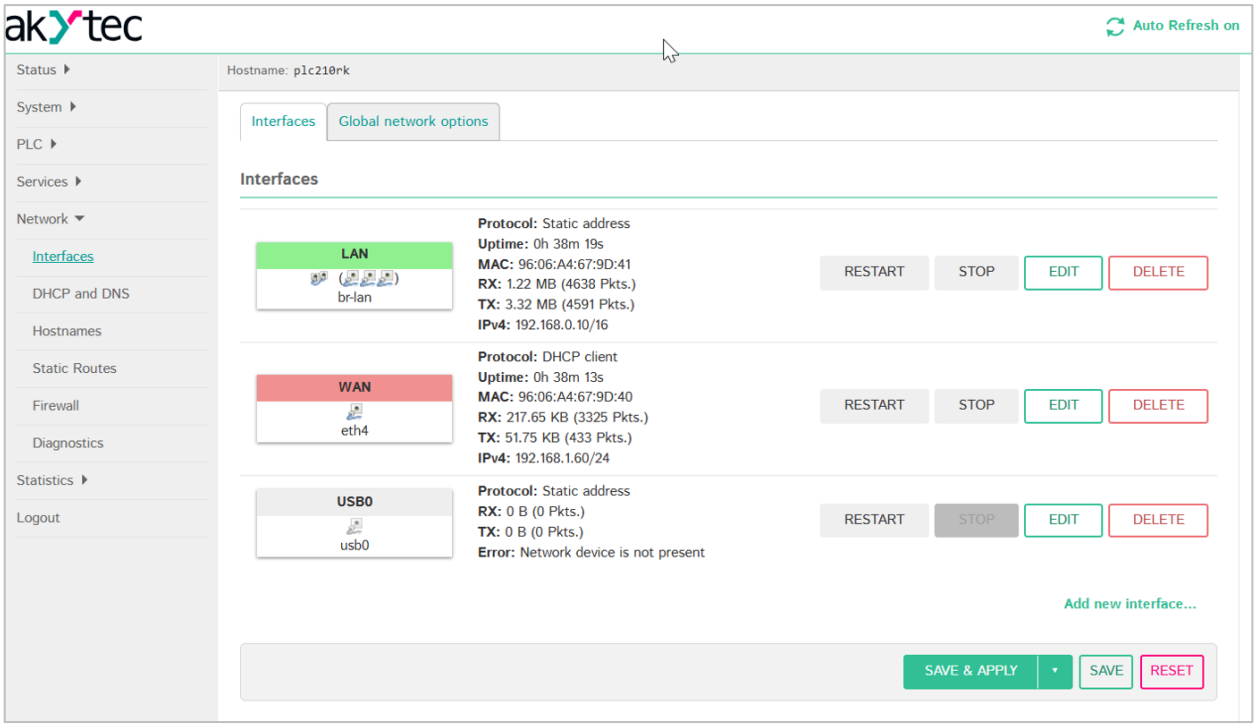


Fig. 3.2.2 List of PLC interfaces

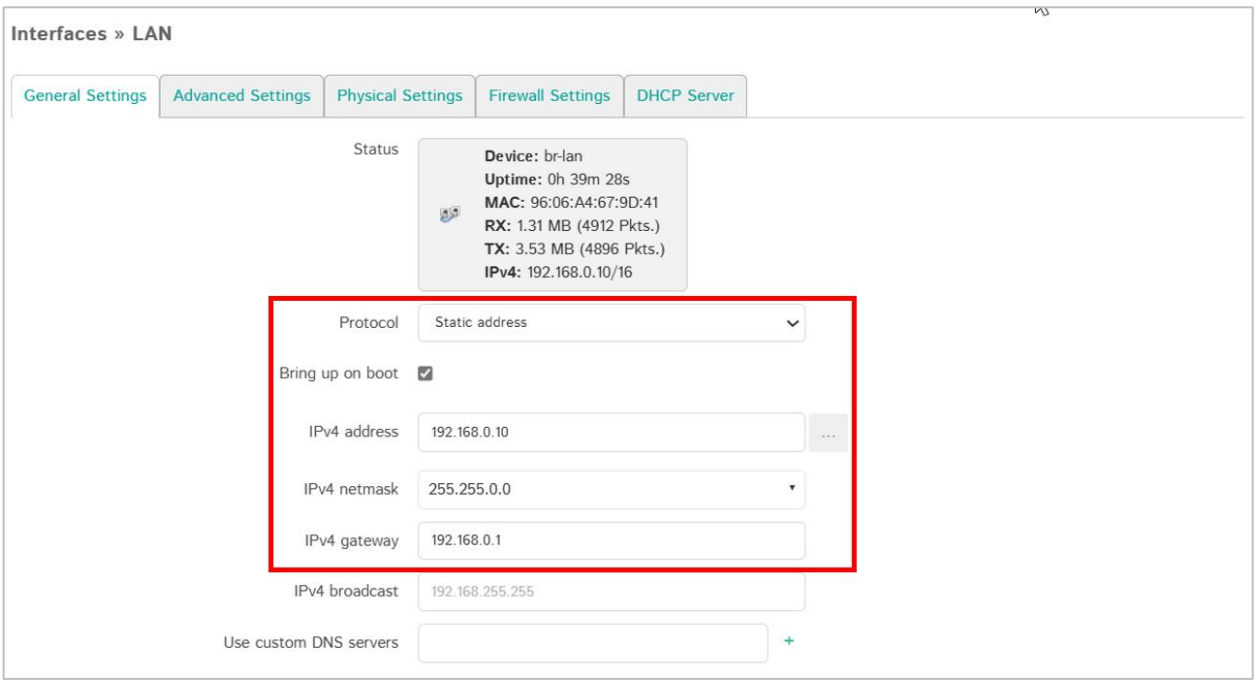
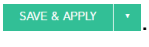


Fig. 3.2.3 Ethernet settings

Click ***Save and Apply*** button to apply the new network settings 

. We will not change the default network settings.

NOTE

After changing network settings, you must reconnect Web Configurator for the new settings to be applied.

To configure the PC network settings, open ***Start*** > ***Settings*** > ***Network & Internet*** and select ***Change adapter options*** (Fig. 3.2.4).

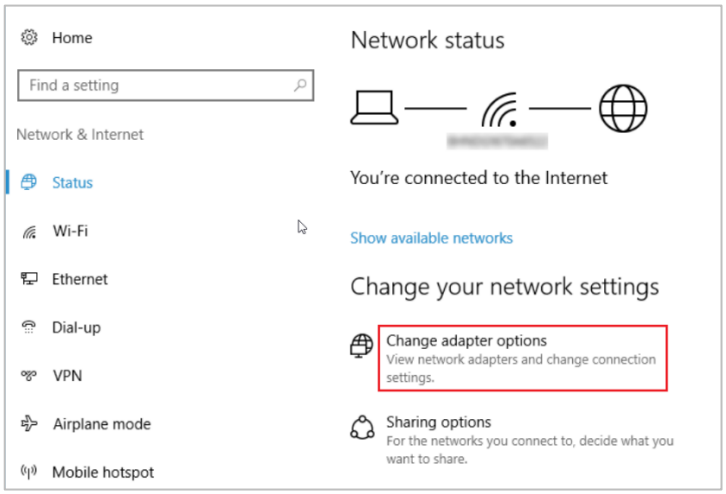


Fig. 3.2.4 Settings > Network & Internet

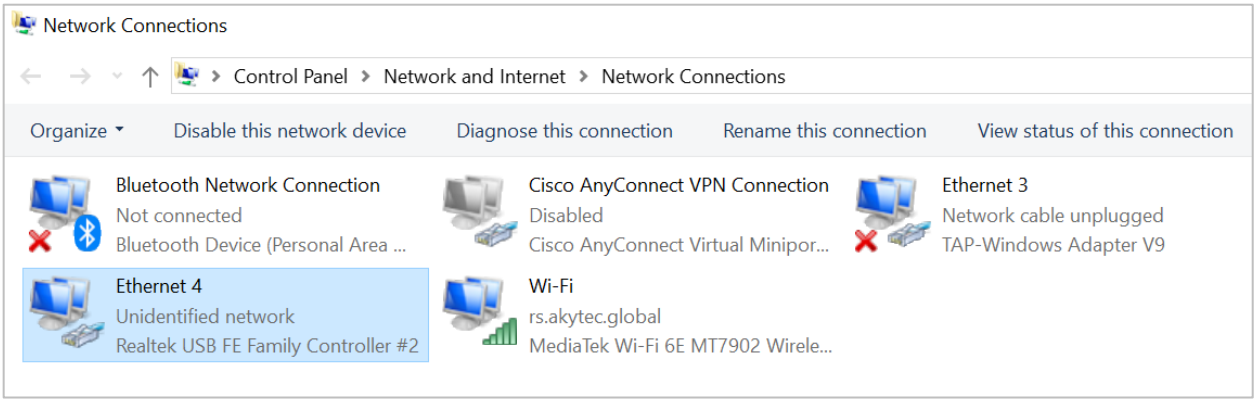


Fig 3.2.5 Change adapter options

Connect the PLC Ethernet interface with the Ethernet interface on the PC over an Ethernet cable. One of the network adapters become active.

Open ***Properties*** in the context menu of the adapter, select ***Internet Protocol version 4 (TCP/IPv4)*** and click the ***Properties*** button.

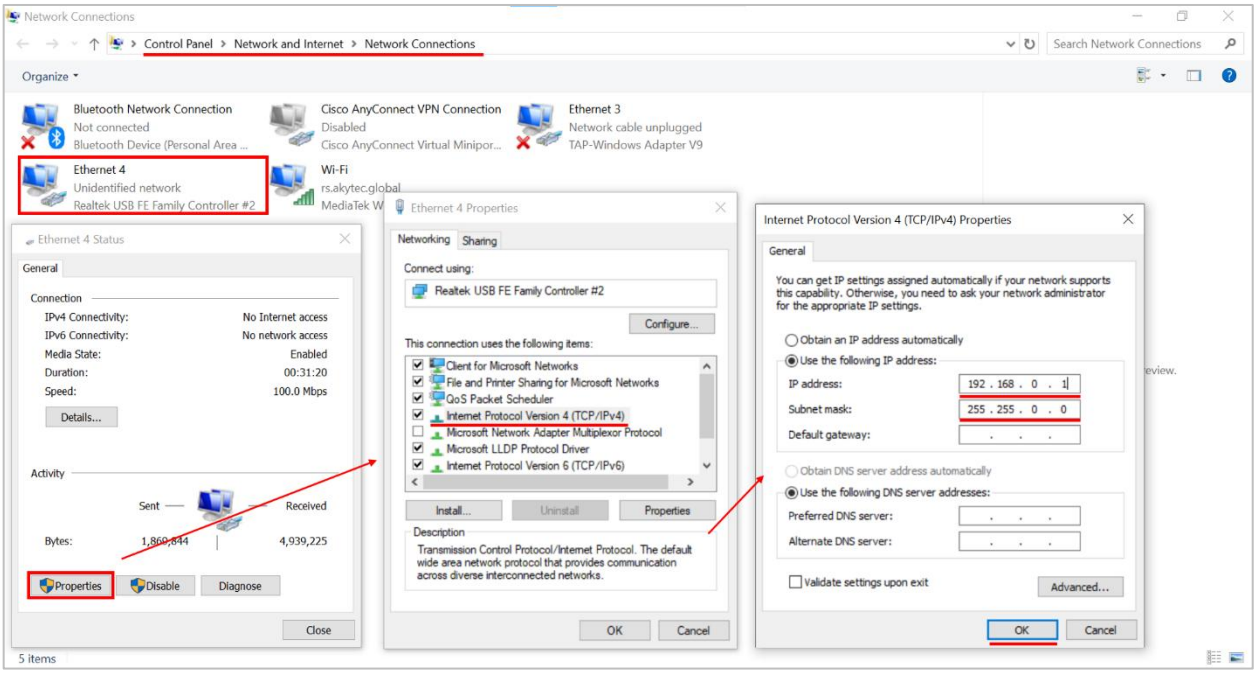


Fig. 3.2.6 PC Ethernet settings

Set the IP address to ***192.168.0.1*** and subnet mask to ***255.255.0.0***. The other fields remain empty. After clicking ***OK***, it will take a few seconds for the adapter to apply the new settings.

To check the PC-PLC connection, open the command line (+ ***R*** and enter ***cmd*** in the input field). Enter the command "***ping 192.168.0.10 -t***" in the command line. If the connection is established, the result is as follows:

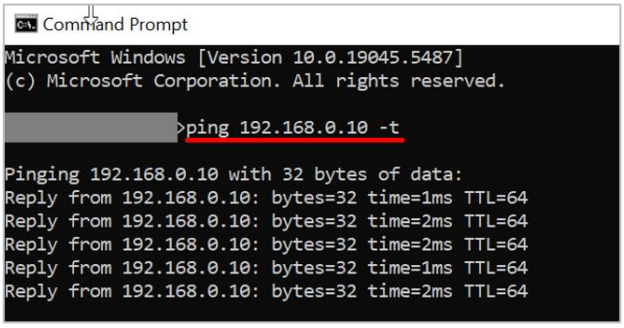


Fig. 3.2.7 Result of successful ping test

Close the window. Now you can connect the PLC to CODESYS (sect. 3.3).

3.2.2 Connection over USB

Use USB A-to-B cable for SPC210 and USB-to-MicroUSB cable for PLC210, to connect them to a PC with running CODESYS. The default network settings for the USB interface of the device are as follows:

Table 3.2.2 USB default settings

PLC	DHCP Server	IP address	Subnet mask	Gateway
PLC210	on	172.16.0.1	255.255.255.248	-
SPC210	on	10.0.6.10	255.255.255.248	-

The USB interface of the PLC210 or SPC210 works in DHCP server mode (Fig. 3.2.8), therefore the PC network adapter receives the required settings automatically.

To check the PC-PLC communication, open the command line (+ ***R*** and enter ***cmd*** in the input field). Enter the command ***ping 172.16.0.1 -t*** for PLC210, and ***ping 10.0.6.10 -t*** for SPC210 in the command line.

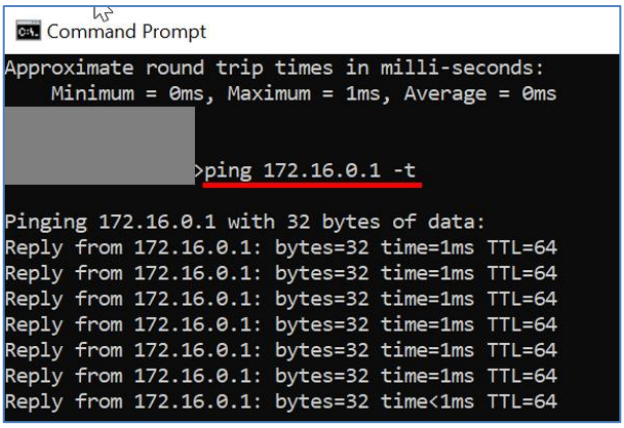


Fig. 3.2.9 Result of successful ping test

Close the window. Now you can connect the PLC to CODESYS (sect. 3.3).

If it is necessary to change the USB interface network settings, use the Web Configurator. If you are working with an SPC210, you can also use the on-screen configurator.

To adjust the settings of the network adapter on the PC, open ***Properties*** in the context menu of the adapter, select ***Internet Protocol version 4 (TCP/IPv4)*** and click the ***Properties*** button.

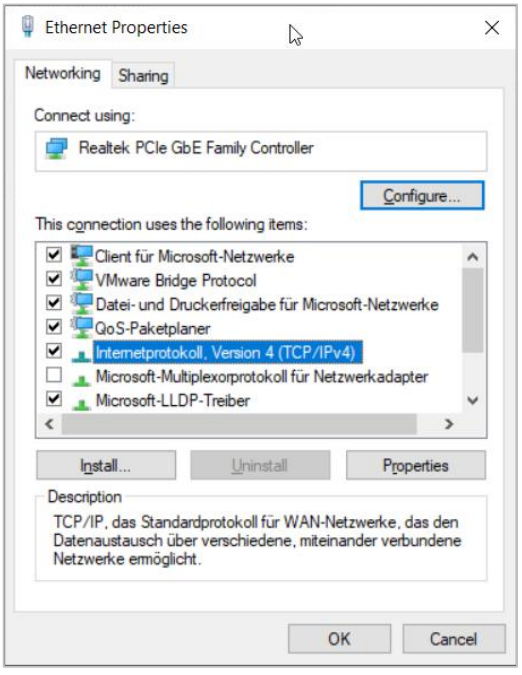


Fig. 3.2.8 Network adapter settings

Make changes in the open input dialog and click ***OK*** to confirm.

3.3 Connection between akYtec PLC and CODESYS

If the network parameters of the PLC and the PC are configured, communication between the PLC and CODESYS can be established.

First, check the ***Device*** node in the project tree (defined by the target file) matches the model of the PLC to be connected. If an inappropriate PLC type has been selected at project creation, it can be changed using the command ***Update Device*** in the context menu of the ***Device*** node.

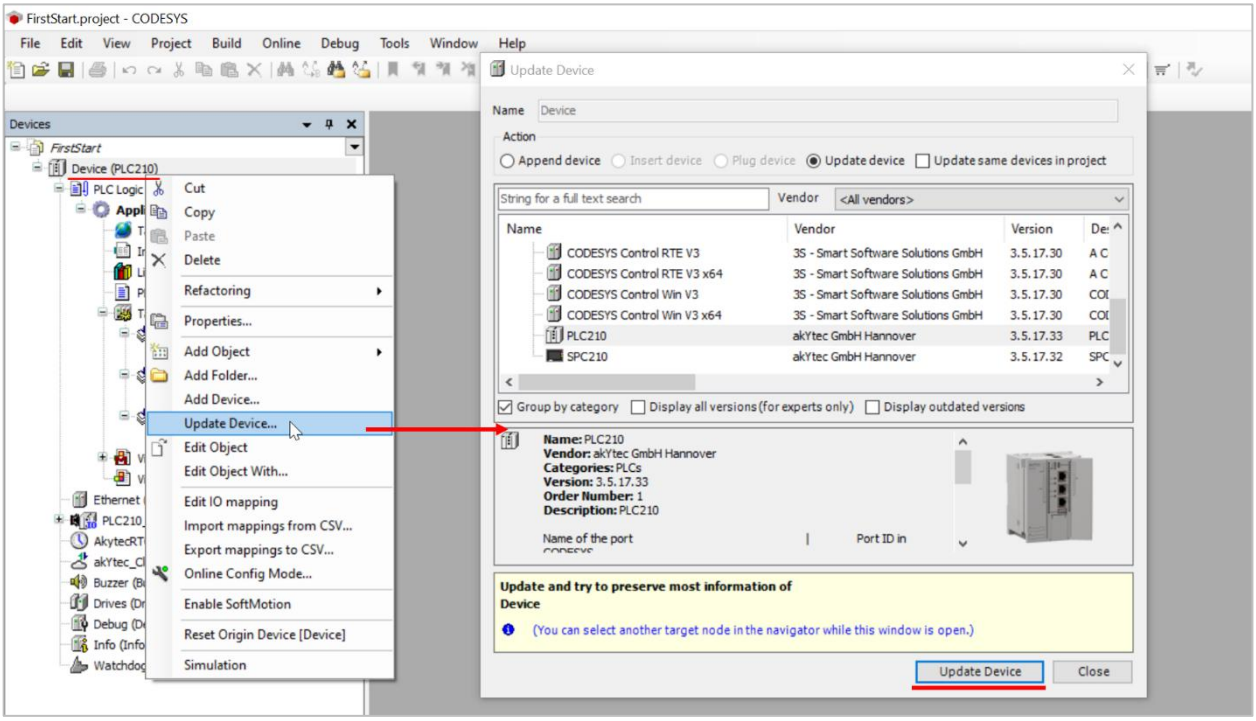


Fig. 3.3.1 Change target device

Select the required target file from the list. Its version, as well as the CODESYS version, must match the PLC firmware version.

By default, only the newest versions of the installed target files are displayed. To view all available versions, check the checkbox ***Display all versions***.

Click the ***Update Device*** button and then the ***Close*** button. The name of the selected device will be displayed in the ***Device*** node in the project tree.

Next, you need to configure the gateway. This configuration should only be done once and is usually required if the current version of CODESYS is the first one installed on this PC.

Double-click the **Device** node to open the ***Device*** tab in the workspace. Select ***Communication Settings*** > ***Gateway*** > ***Add New Gateway***.

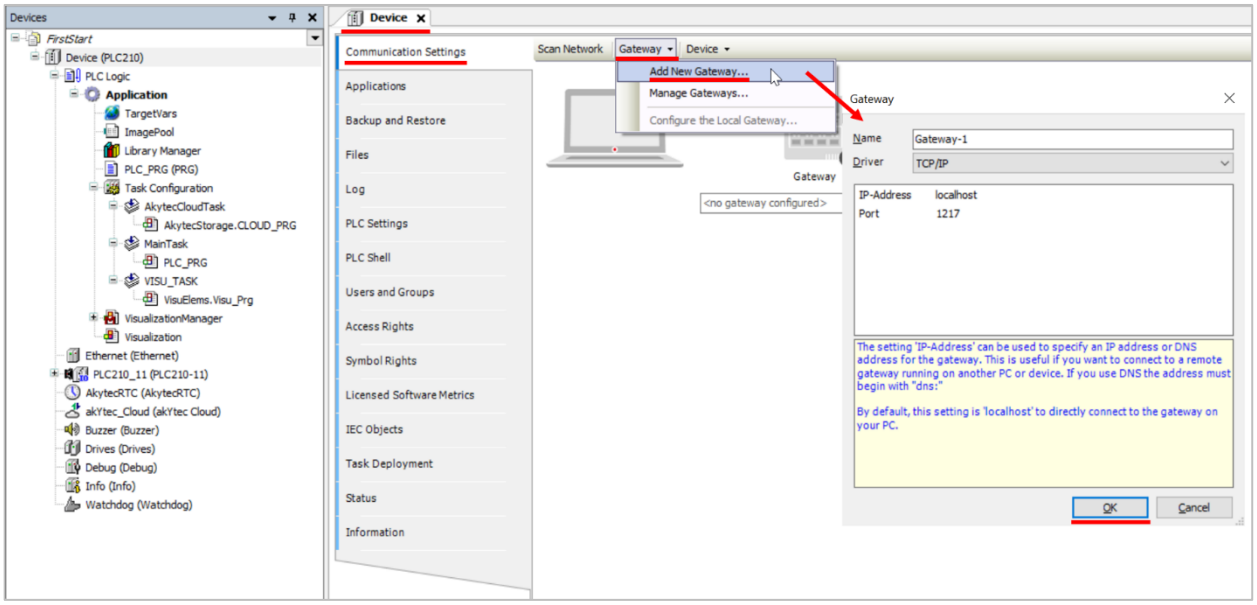


Fig. 3.3.2 Create new gateway

It is recommended to use the default settings (***name*** = ***Gateway-1***, ***IP address*** = ***localhost***). Close the ***Gateway*** window and click the ***Scan Network*** button. Select the required PLC in the open device list and click ***OK*** to establish communication.

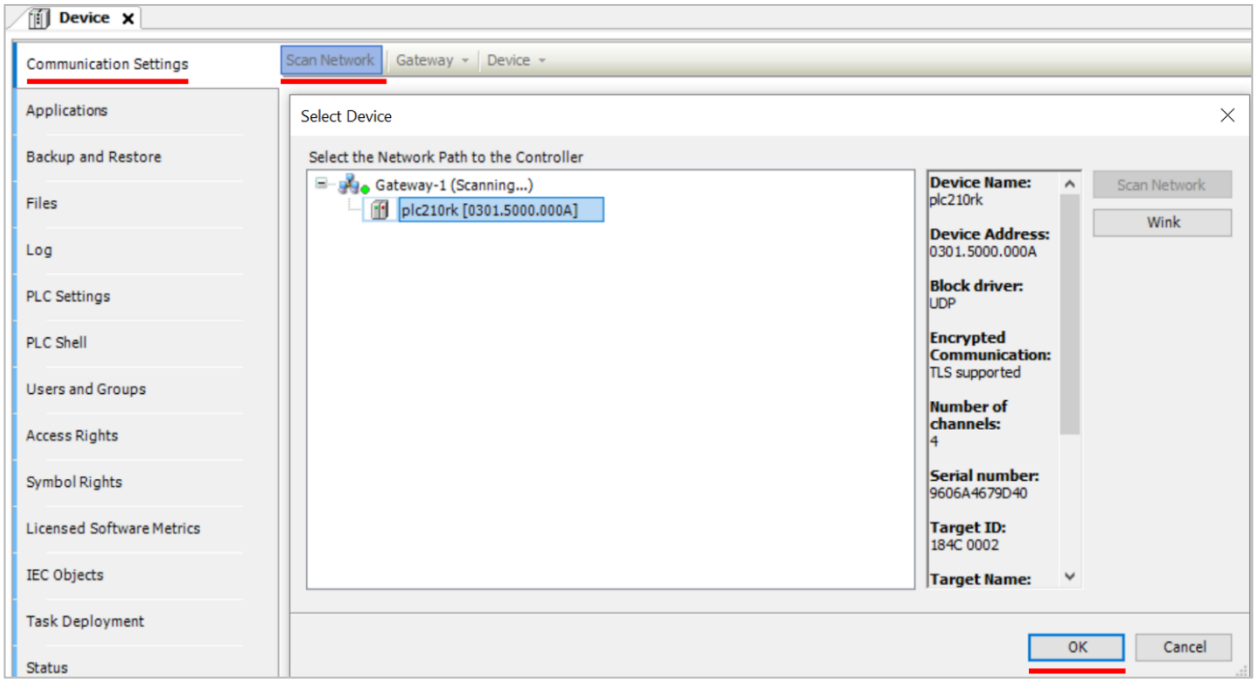


Fig. 3.3.4 "Select Device" window

In case of successful connection, the gateway and PLC indicators will light green.

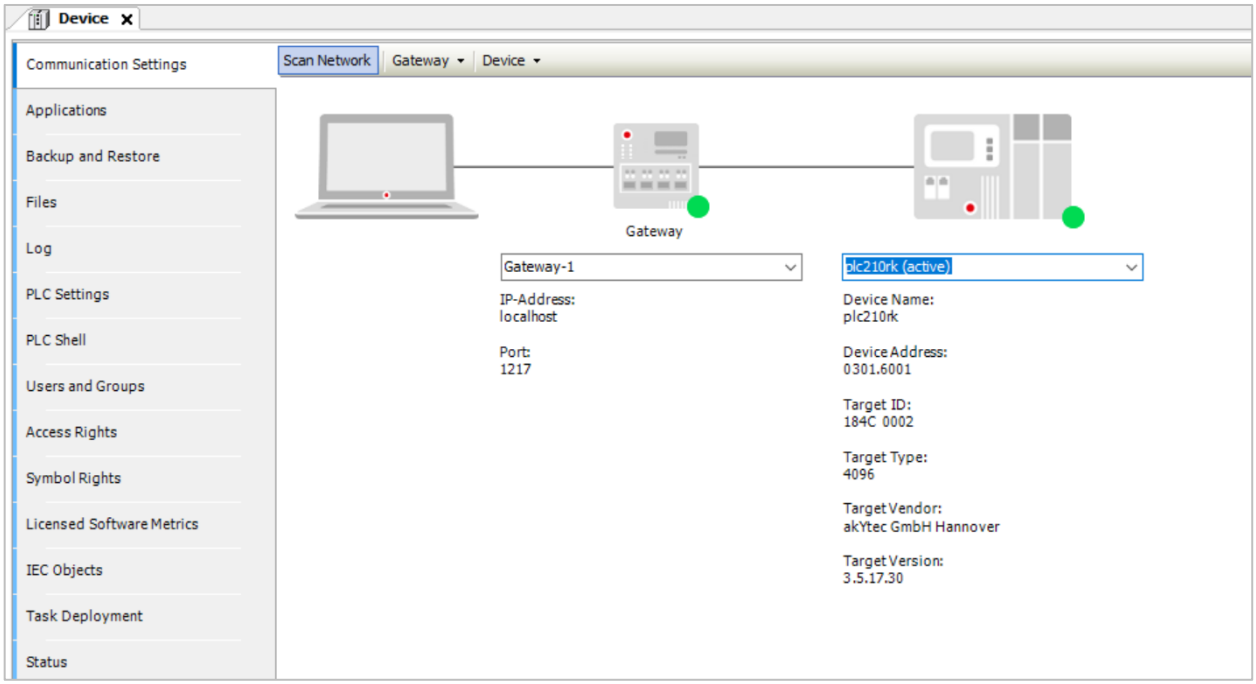


Fig. 3.3.5 Connection successfully established

In some cases (e.g. if the antivirus is running on the PC, which blocks sending UDP broadcast packets or receiving packets on ports 1740-1743) the PLC may not be detected during network scanning. In this case, enter the PLC's IP address manually and press ***Enter***.

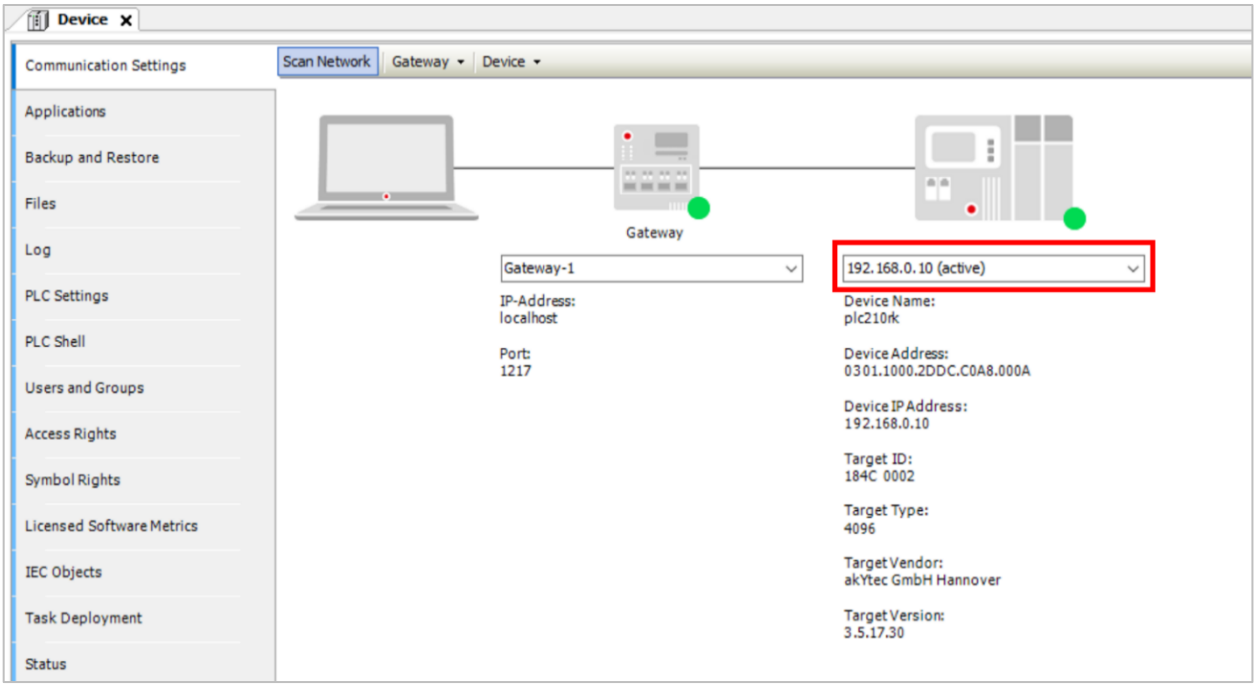


Fig. 3.3.6 Enter device IP address to establish connection without network scanning

NOTE

Connection with an SPC210 is only possible if the on-screen configurator is not active.

After successfully established connection, you can download the project to the device (sect. 3.5). If you do not have a real PLC, you can use a virtual PLC (sect. 3.4).

3.4 Connection between virtual PLC and CODESYS

The virtual PLC is a full-fledged control system running on PC. The CODESYS installation package includes a virtual PLC ***CODESYS Control Win V3*** with a continuous operation time limit of 2 hours. After expiration of this time, it can be restarted.

The virtual PLC enables us to test all basic functions of the project, including web-visualization, communication with other devices and file management, without connecting a real PLC. Only PLC-specific functions cannot be tested, such as operation of I/Os and external libraries (e.g. ***CmpSysExec*** library, which is used to call PLC OS utilities).

To start the virtual PLC, open the Windows System Tray, click on the virtual PLC icon and select the command ***Start PLC***. If you start the virtual PLC for the first time, an information window will open. Click ***OK*** to continue.

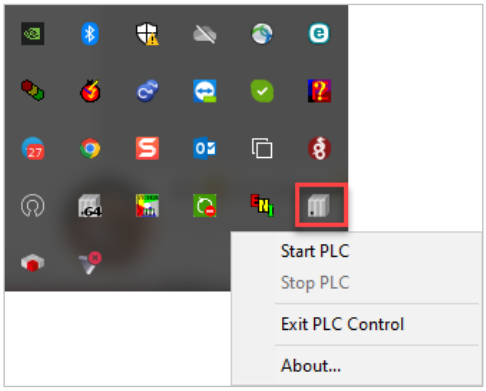


Fig. 3.4.1 Start virtual PLC

Next, you need to replace the PLC in the project with the virtual PLC (Fig. 3.4.2). Use the command ***Update Device*** from the ***Device*** node context menu and select the target file ***CODESYS Control Win V3***. Its version must correspond to the version of CODESYS (e.g., for CODESYS V3.5 SP17 Patch 3, use target file version 3.5.17.30).

By default, only the newest versions of the installed target files are displayed. To view all available versions, check the checkbox ***Display all versions***.

NOTE

If a 64-bit version of CODESYS is installed, select the target file ***CODESYS Control Win V3 x64***.

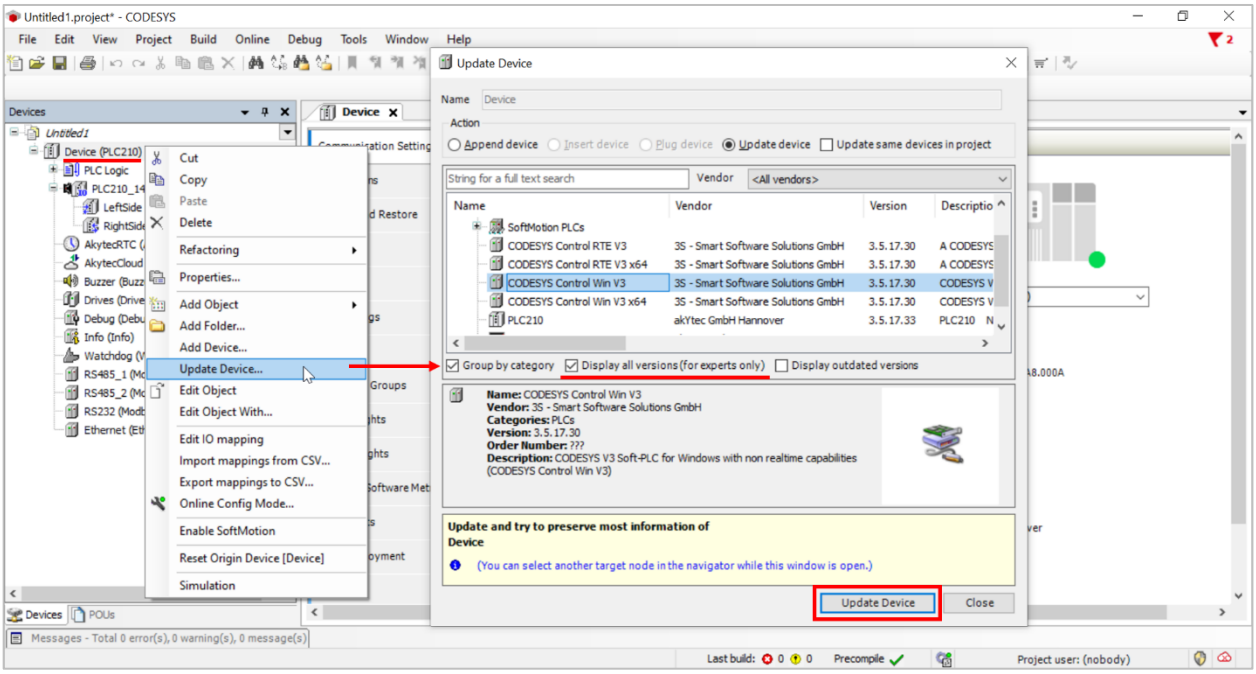


Fig. 3.4.2 Replacing the project PLC with a virtual PLC

> **NOTE**
>
> If the project was originally created for an akYtec PLC, open the node ***Task Configuration*** and remove the task ***AkytecCloudTask*** from the project tree, otherwise the project cannot be downloaded to the virtual PLC.

Next, you need to configure the gateway. This configuration should only be done once and is usually required if the current version of CODESYS is the first one installed on this PC.

Double-click the ***Device*** node to open the ***Device*** tab in the workspace. Select ***Communication Settings*** > ***Gateway*** > ***Add New Gateway***.

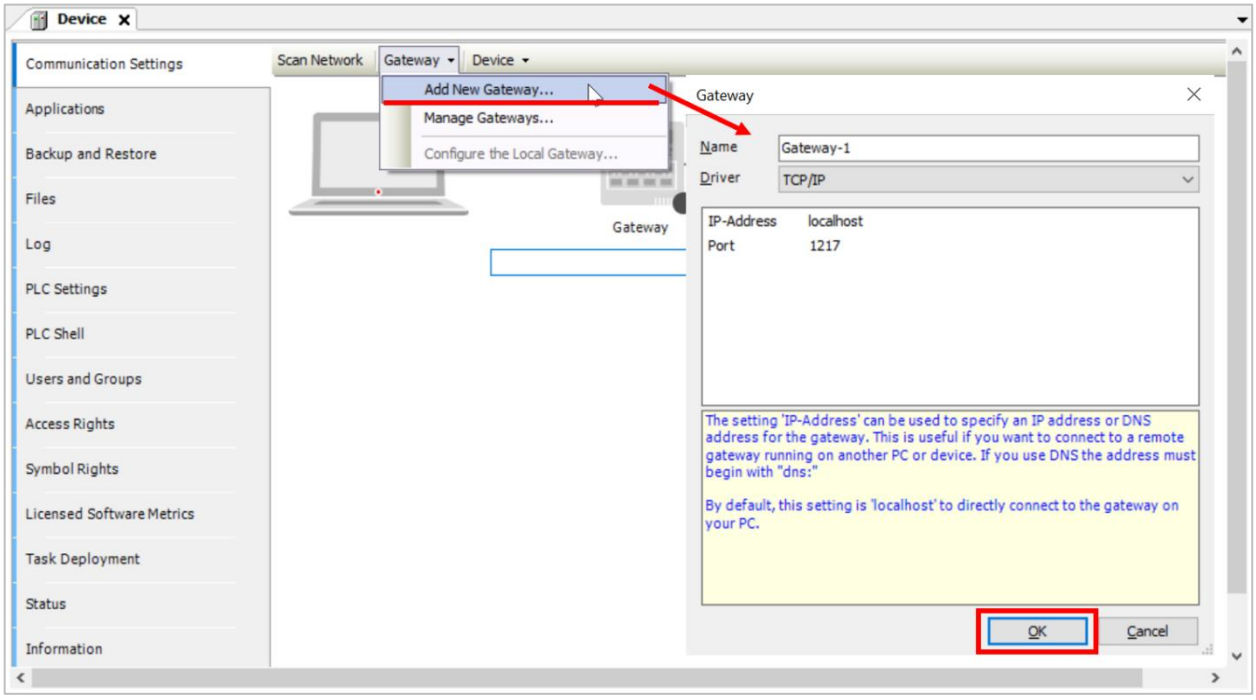


Fig. 3.4.3 Create new gateway

It is recommended to use the default settings (***name*** = ***Gateway-1***, ***IP address*** = ***localhost***). Close the ***Gateway*** window and click the ***Scan Network*** button. Select the required PLC in the open device list and click ***OK*** to establish connection.

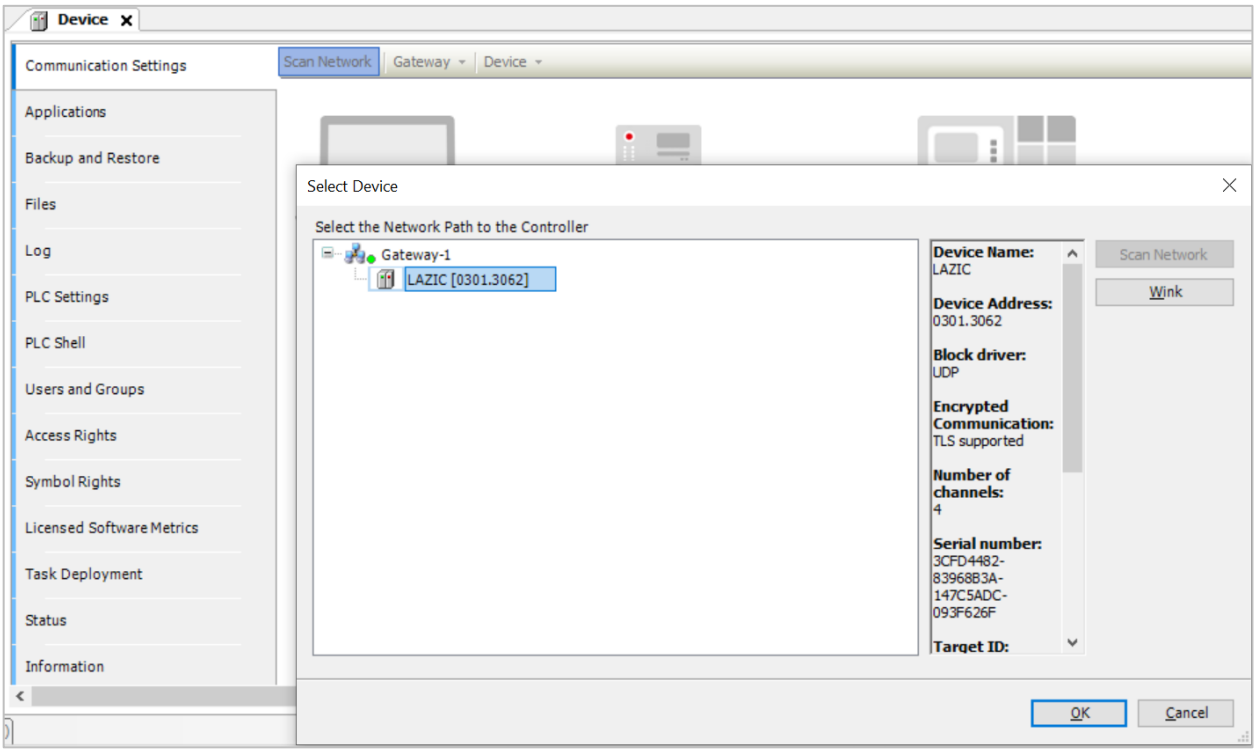


Fig. 3.4.4 “Select Device” window

If you are using CODESYS V3.5 SP17 or higher, after clicking the ***OK*** button, a window opens with a prompt to create a PLC user with a login and a password. This login and password must be entered each time you connect to the virtual PLC.

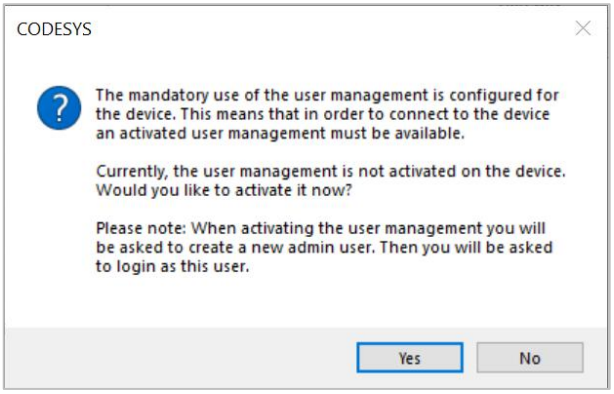


Fig. 3.4.5 Prompt to create a user at first connection to virtual PLC

If you want to avoid this, you can disable the mandatory user management:

- open the folder ***C:\ProgramData\CODESYS\CODESYSControlWinV3\<PLC ID>***,
- open the file ***CODESYSControl.cfg***,
- go to the component ***[CmpUserMgr]***
- in the line ***;SECURITY.UserMgmtEnforce=NO*** remove the semicolon (first character in the line) to uncomment the parameter.

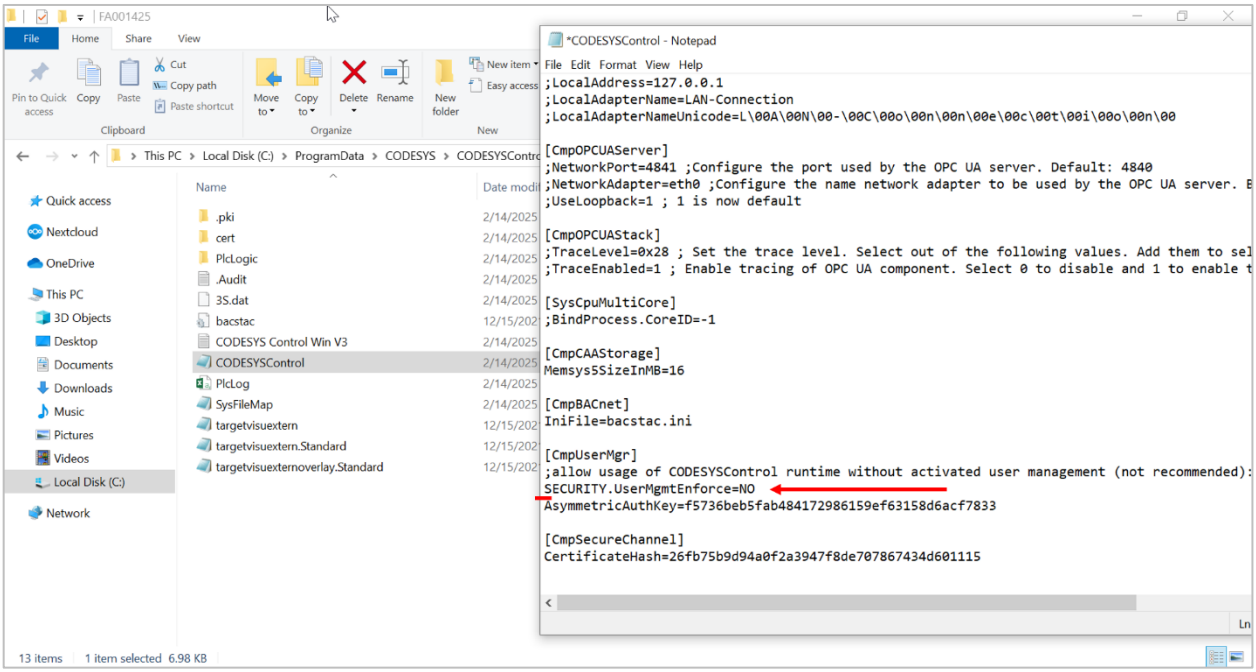


Fig. 3.4.6 CODESYSControl.cfg

Save the changes in the file and restart the virtual PLC: click on its icon in the System Tray, select the command ***Stop PLC*** and then the command ***Start PLC*** again. When reconnecting the PLC in CODESYS, this time there will be no prompt to create a user.

In case of successful connection, the gateway and PLC indicators will light green.

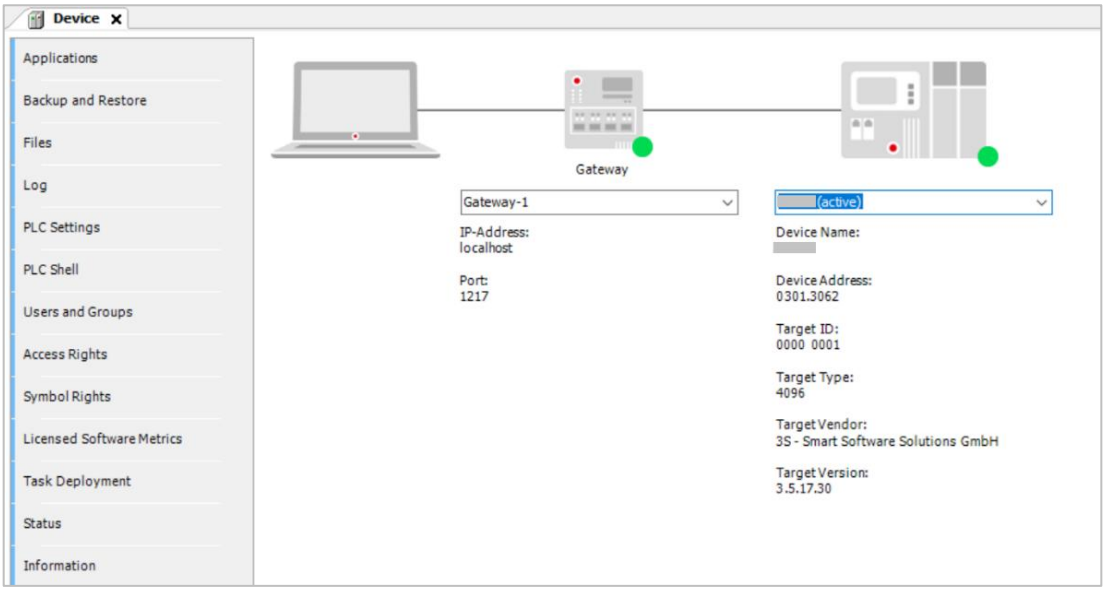


Fig. 3.4.7 Communication successfully established

After successful connection, you can download the project to the device (sect. 3.5).

3.5 Project download, start and management

To download the project to the device, use the command ***Online*** > ***Login*** .

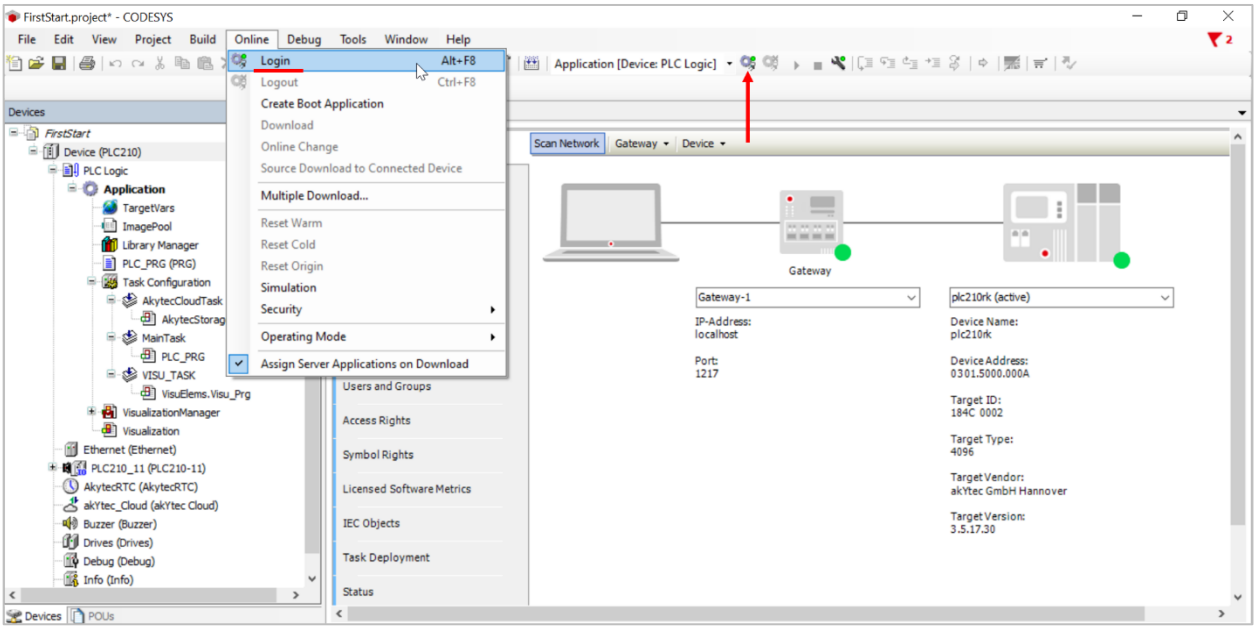


Fig. 3.5.1 Login to PLC

If there is no project in the PLC yet, you will be prompted to create a boot project and download it into the device. Click ***Yes*** to continue.

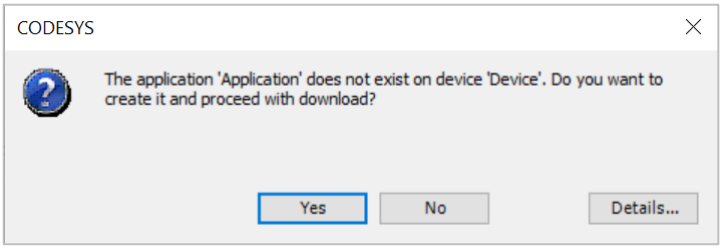


Fig. 3.5.2 Prompt to create a boot application

As a result, the project file will be converted to a binary file, which will be downloaded to the PLC, but not executed. This can be recognized by ***[stop]*** status of the ***Application*** node in the project tree, as well as by gray icons near tasks and components.

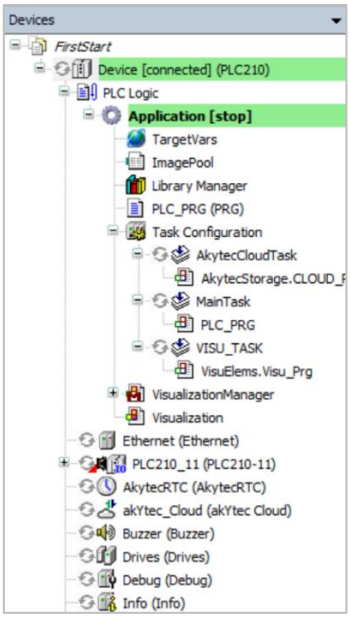


Fig. 3.5.3 Application status: downloaded, stopped

To start the application, use the menu command ***Debug*** > ***Start*** (or the icon in the toolbar). If you run the project on a virtual PLC, the target visualization will open. If the visualization is opened in full-screen mode, use ***Win*** or ***Alt+Tab*** keys to exit the mode. The window can be closed then.

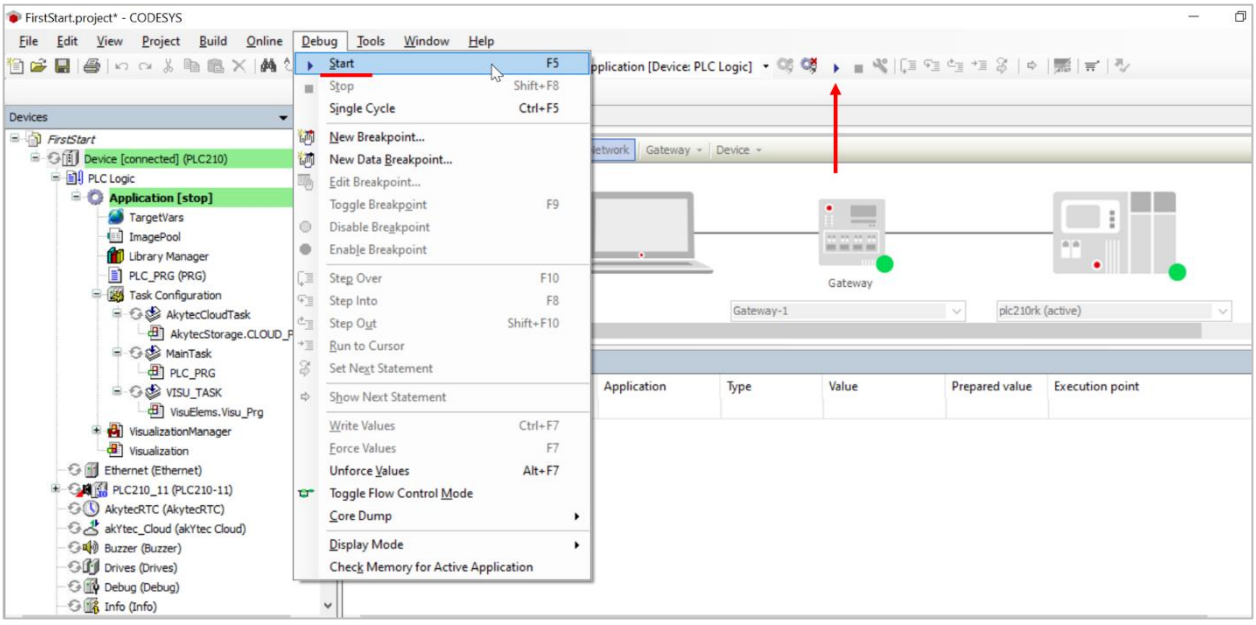


Fig. 3.5.4 Start application

Now the application is running: the status in the ***Application*** node has changed to ***[run]***, and the icons near tasks and components have turned green .

NOTE

Important! As a result of the previous operations, the application is downloaded to the device RAM, but it is not yet saved in its flash memory. If you reboot the PLC now, the application will be removed. To avoid this, make it bootable using the command ***Online*** > ***Create Boot Application***. Then it will be saved in the flash memory of the PLC and will be automatically launched after its restart.

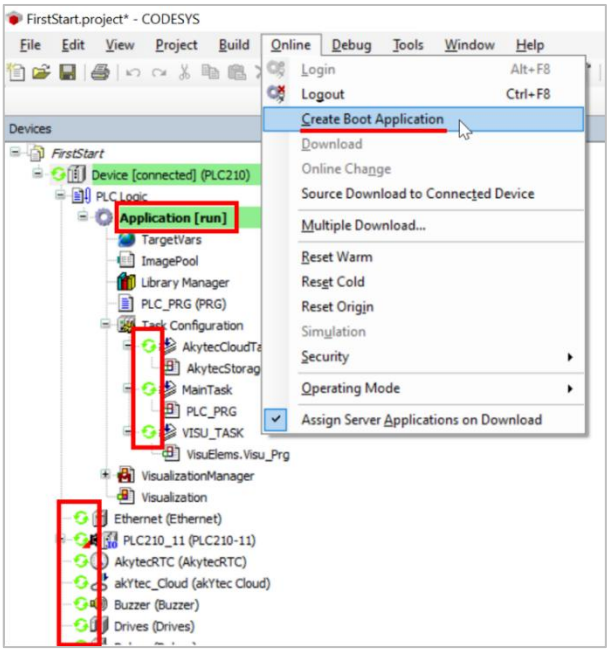


Fig. 3.5.5 Indication of the running application in the project tree

Above we have outlined the difference between the terms “**project**” and “**application**”. The “project” is the ***.project** file you work with in the CODESYS.

When you click the menu item ***Online > Login***, the boot application file is created on the PLC.

When the project is started, you can check the visualization program in web-visualization. Enter the IP address of the PLC and the port of the web-visualization server (8080 by default) in the browser address line.

- If your PC is connected to the PLC with factory settings over Ethernet interface (Tab. 3.2.1), enter ***192.168.0.10:8080***
- If your PC is connected to the PLC with factory settings over USB interface (Tab. 3.2.2), enter ***172.16.0.1:8080*** for PLC210 or ***10.0.6.10:8080*** for SPC210.
- If you use a virtual controller, enter ***localhost:8080***.

After that the web-browser will display the visualization page we created in Sect. 3.1, and if you use SPK210, the same visualization will be displayed on the device screen immediately after the application start.

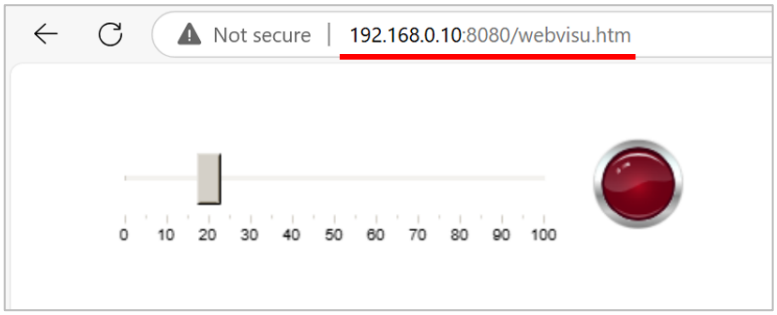


Fig. 3.5.6 Web-visualization in browser

Change the slider position. If its value exceeds 50, the alarm indicator will turn on. If the value becomes less than or equal to 50, the indicator will turn off. Thus, the program works exactly as it was intended.

At this moment CODESYS is still connected to the PLC. If we open the PLC_PRG program, we can see that next to the variable names, their current values are displayed. It is often useful to change these values „on

the fly" and see how the program reacts. Of course, it can be done through web-visualization - but in a real project there may be hundreds or even thousands of variables, and a significant part of them will not be used in the visualization.

The easiest way to do this:

1. in the variable table, select a variable and double-click in the field ***Prepared value***
2. enter the value you want to assign to the variable and press ***Enter***
3. use the command ***Debug*** > ***Write Values*** (or press Ctrl+F7)

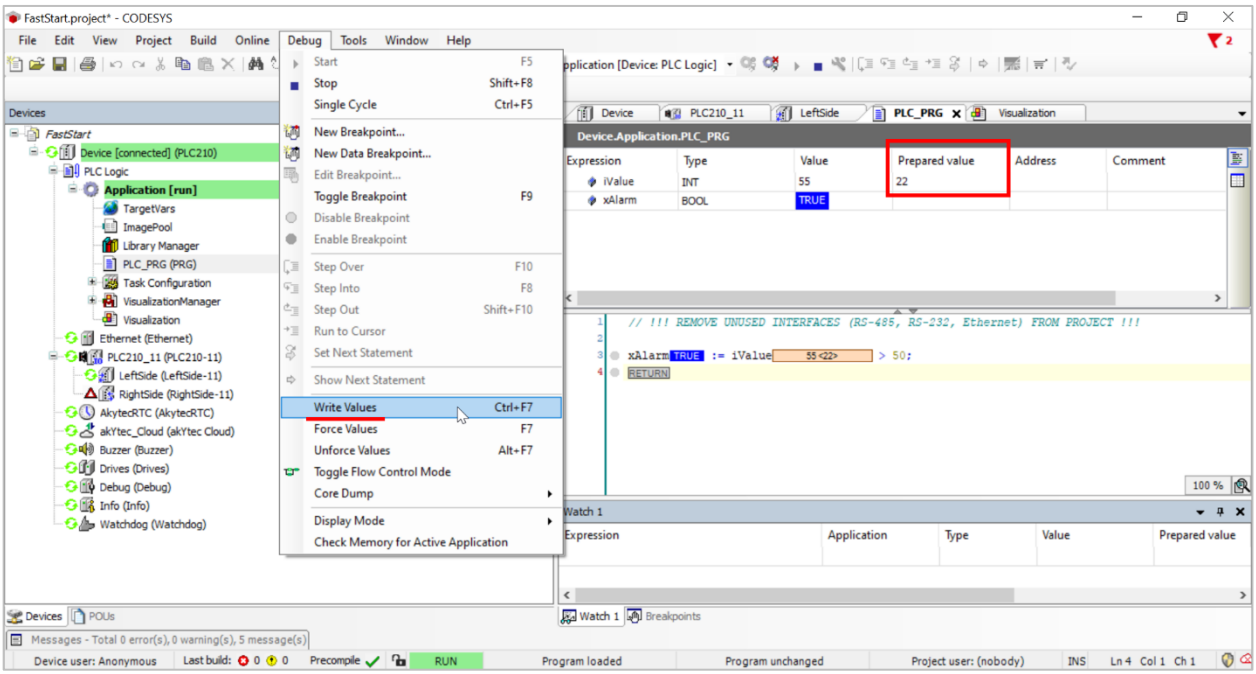


Fig. 3.5.7 Changing the variable value

After writing, you can change the value of the variable by moving the slider in the visualization.

But if you use the command ***Force Values*** instead of ***Write Values***, it will be impossible to move the slider, because the prepared value will be written to the variable every program cycle.

To disconnect the PLC from CODESYS for project editing, use the command ***Online*** > ***Logout*** .

If you make any changes to the project (e.g., change the initial value of the variable ***iValue*** from ***20*** to ***30***) and use the command ***Login***, the following window will open:

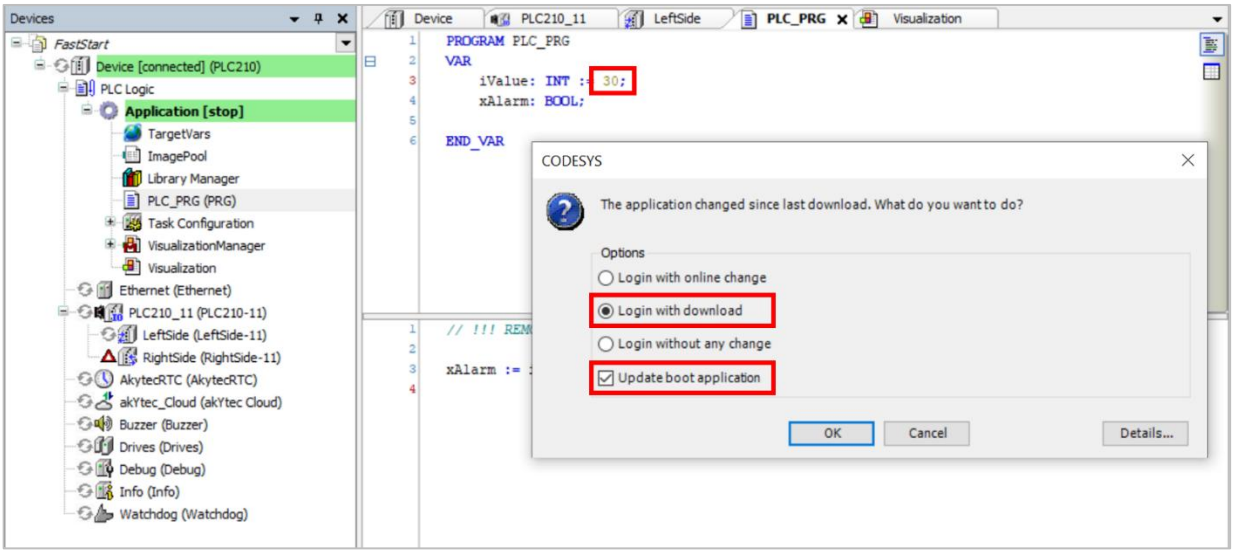


Fig. 3.5.8 Application loading window after making changes to the application

This window prompts you to select one of the options for connecting to the PLC.

NOTE

Important! It is not recommended to select the option ***Login with online change*** (active by default). This option implies a "hot update" of the application code, during which only the changed code fragments will be downloaded. In practice, this procedure may work incorrectly in many cases, as a result, you may encounter a range of hard-to-detect errors (e.g., unintended changes of variable values due to incorrect reallocation of their addresses in memory). Therefore, it is safer to use the option ***Login with download***, because in this case the complete application will be downloaded to the PLC.

You can also check the checkbox ***Update boot application***, in this case, the command ***Create boot application*** will be automatically executed after the download of the application, and you do not have to execute it manually.

So, in this section we have:

- created a simple project with a program in ST language and visualization
- established connection between the PLC and CODESYS
- downloaded this project to the PLC
- worked with web-visualization
- learned how to change the values of variables when connecting online to the PLC
- learned about different options for downloading the application

If you already have some experience with PLCs, you probably already have some questions. In the following section we will try to answer the most predictable of them.

3.6 FAQ

3.6.1 No boot application after device reboot

If after rebooting the PLC there is no boot application in the device. If you see the message from Fig. 3.5.2, this means you have not made the application bootable. To fix this, see Fig. 3.5.5 and the information above.

3.6.2 How can I remove the application from the PLC?

In Sect. 3.5 we have learned how to download the application to the PLC. If it is necessary to remove it, use the command ***Online*** > ***Reset Origin***. The CODESYS application will be deleted while all other settings, including network settings, will retain their current values.

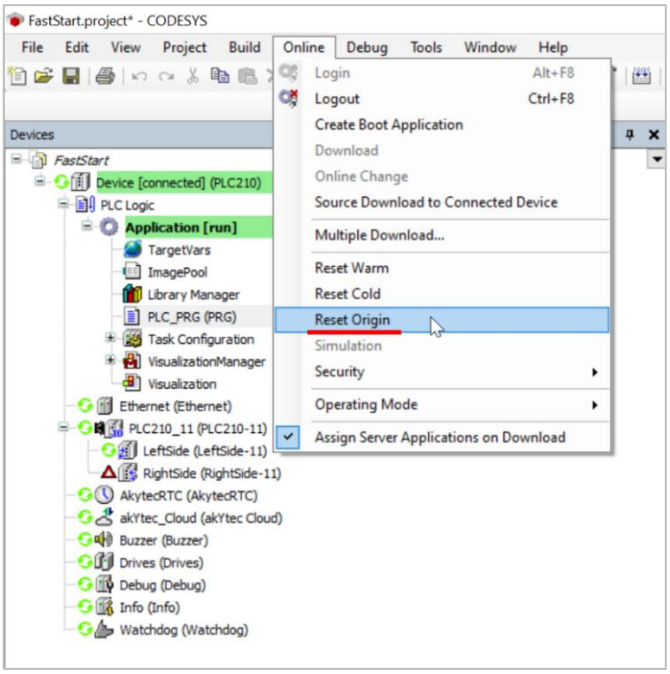


Fig. 3.6.1 Removing boot application from PLC

3.6.3 Is it possible to transfer an application from one PLC into another?

Generally, there are two different interpretations for this question:

- The operation engineer is interested in the possibility of preparing a backup, so that in case of a PLC failure he can get a spare device, download the same project into it and restore the operation of the automation system.
- The software engineer is interested in the issue of intellectual property protection, so that the project developed by him cannot be changed or replicated to other PLC without his permission.

The main aspects of this issue:

- When executing the command ***Create boot application***, a binary file with the boot application is saved in the PLC. A copy of this file does not give another person access to the source file, the project file (*.***project***) cannot be restored from it.
- Generally, the boot application file can be transferred to another PLC of the same model and with the same firmware version, but this feature may be password protected or completely disabled. Besides, it is possible to read out the device serial number from the program code, compare it with the specified one and, e.g., prohibit the execution of any actions in the program in case of serial numbers mismatch. This makes the transfer of the boot application to another PLC meaningless, since the serial number is assigned to the PLC at the production stage and cannot be changed by user.
- CODESYS has functionality to back up the application and then deploy its copy. The akYtec PLCs also enable, using the Web Configurator as well as the SPC on-screen configurator, to create backups that include both the user application and all device settings (including network settings). Both tools provide the ability to create password protected backups.

- Source files (***.project***) can be downloaded into the PLC using the command ***Online*** > ***Source Download to Connected Device*** (Fig. 3.6.2). To upload this file to CODESYS, use the command ***File*** > ***Source Upload…*** (Fig. 3.6.3).

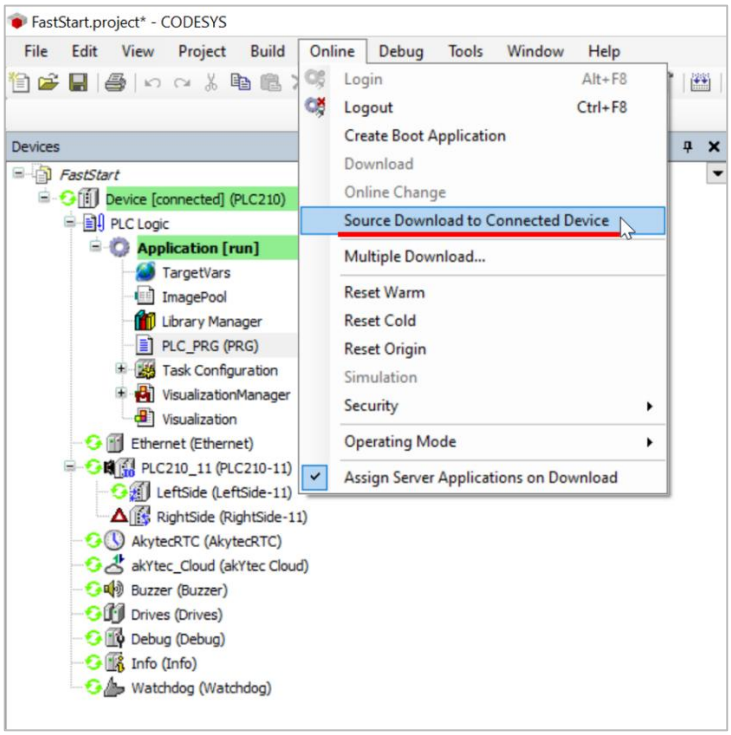


Fig. 3.6.2 Source Download to Connected Device

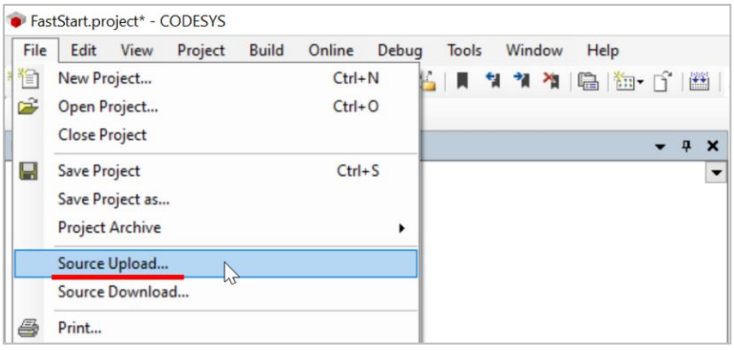


Fig. 3.6.3 Source Upload

3.6.4 The correct way to transfer the project to another PC with CODESYS installed

Before transferring project files between different PCs, make sure that the version of CODESYS in which the project was created is installed on all these PCs.

The project may contain additional components, libraries, etc. To avoid transferring all these files together with the project, you should use project archives (***.projectarchive***). To create a project archive, use the command ***File*** > ***Project Archive*** > ***Save/Send Archive***. The archive file is much larger than the project file (usually ≥ 60 MB), but it contains all the components and libraries used in the project.

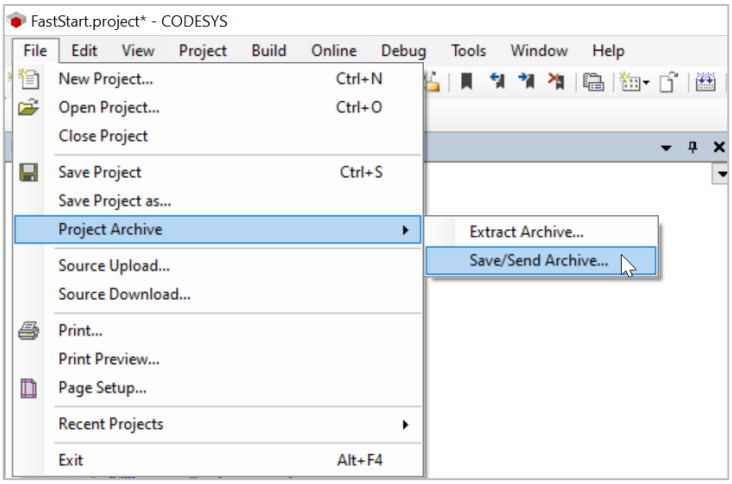


Fig. 3.6.4 Project archive creation

3.6.5 How can I view the PLC message log?

To view the message log, double-click the ***Device*** node to open the ***Device*** tab in the workspace and select the ***Log*** section. It can also be viewed in the Web Configurator in the section ***PLC*** > ***Log Files***.

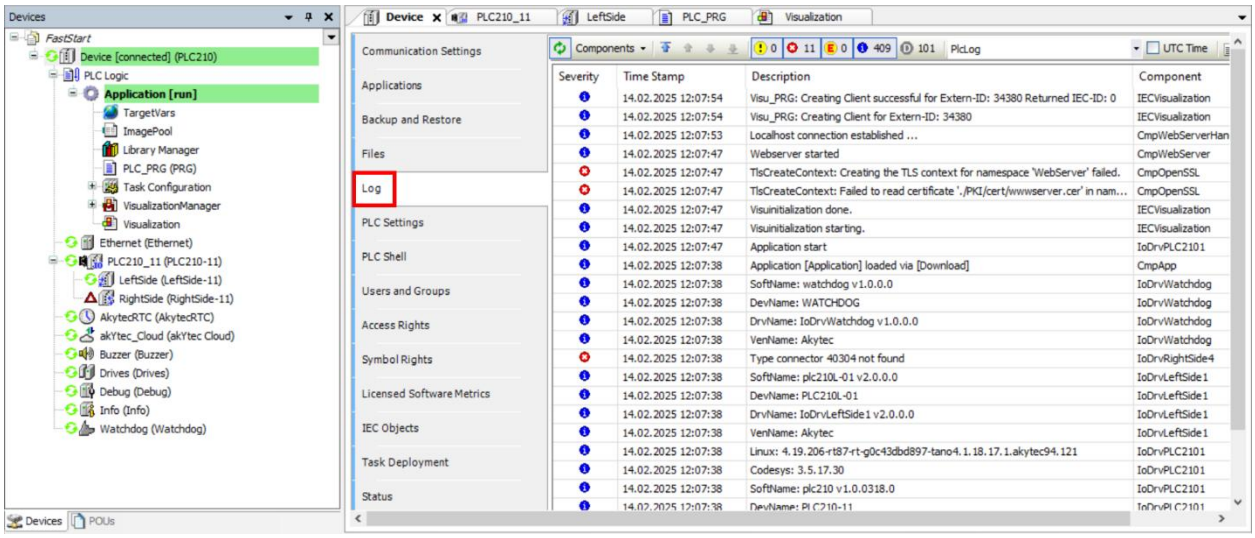


Fig. 3.6.5 PLC message log in CODESYS

3.6.6 How do I know if there are errors in the project?

If there is an error in the project, in most cases it will be detected by compiler during automatic check.

Suppose you forgot to put a semicolon at the end of an expression in the program code. The object in which the error is detected (in our case, it is the ***PLC_PRG*** program) will be underlined in the project tree with a red wavy line, and the compilation message bar displayed at the bottom of the screen will show an error message indicating the object in which the error occurred and the expected position (in our case, it is the line number).

The error can be detected at different stages (precompilation, compilation, etc.). To view messages for a particular stage, you should select it in the drop-down list.

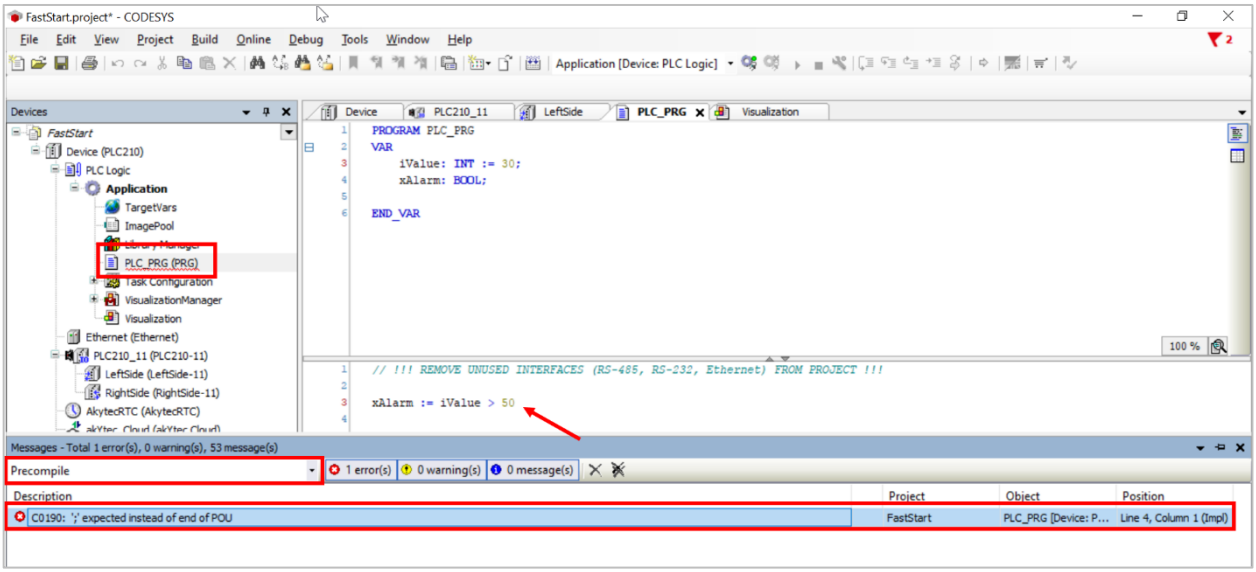


Fig. 3.6.6 Error message in compilation message bar

If you accidentally close the message panel, you can get it back using the command ***View*** > ***Messages***.

In some cases, an error cannot be detected during automatic checks. Therefore, it is recommended to execute the command ***Build*** > ***Generate Code*** regularly. This command checks the project for errors, and if there are no any, creates an application file that will be downloaded to the PLC if connected.

The check results are saved in internal CODESYS files. Sometimes these files may be corrupted or incorrectly overwritten (e.g., during regular automatic checks). This can lead to records about errors that do not actually exist in the program. Therefore, it is recommended to execute the command ***Build*** > ***Clear All*** before ***Generate Code*** to remove information about previous program checks.

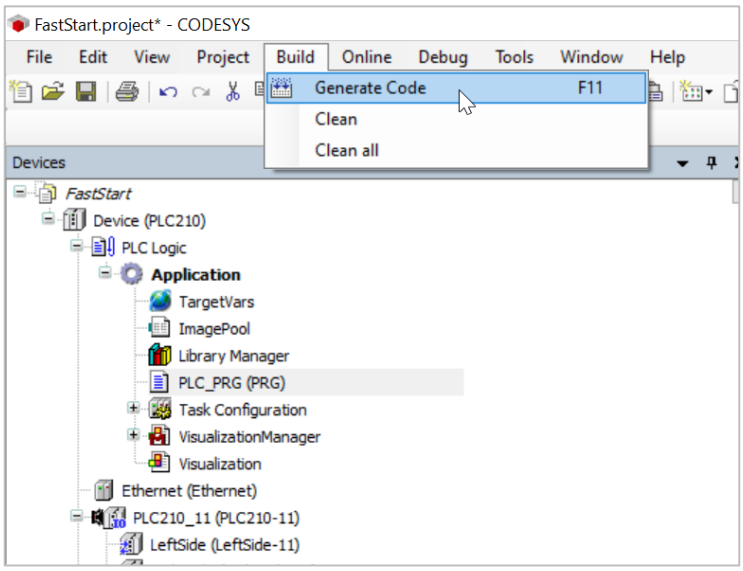


Fig. 3.6.7 Build menu

3.6.7 With what interval is the program code executed?

In sect. 3.1 we added the code to the program that compares the value of the ***iValue*** variable with the set value 50. How often is this check performed?

The interval at which the program is called is determined by the task to which the program is attached. The interval is usually specified in milliseconds. We have created a project based on a template in which the

PLC_PRG program is assigned to the task ***MainTask*** in the node ***Task Configuration*** (Fig. 3.6.8). Double-click ***MainTask*** to open it in the workspace. The interval of ***MainTask*** is 10 ms, it is also the interval of ***PLC_PRG*** program execution.

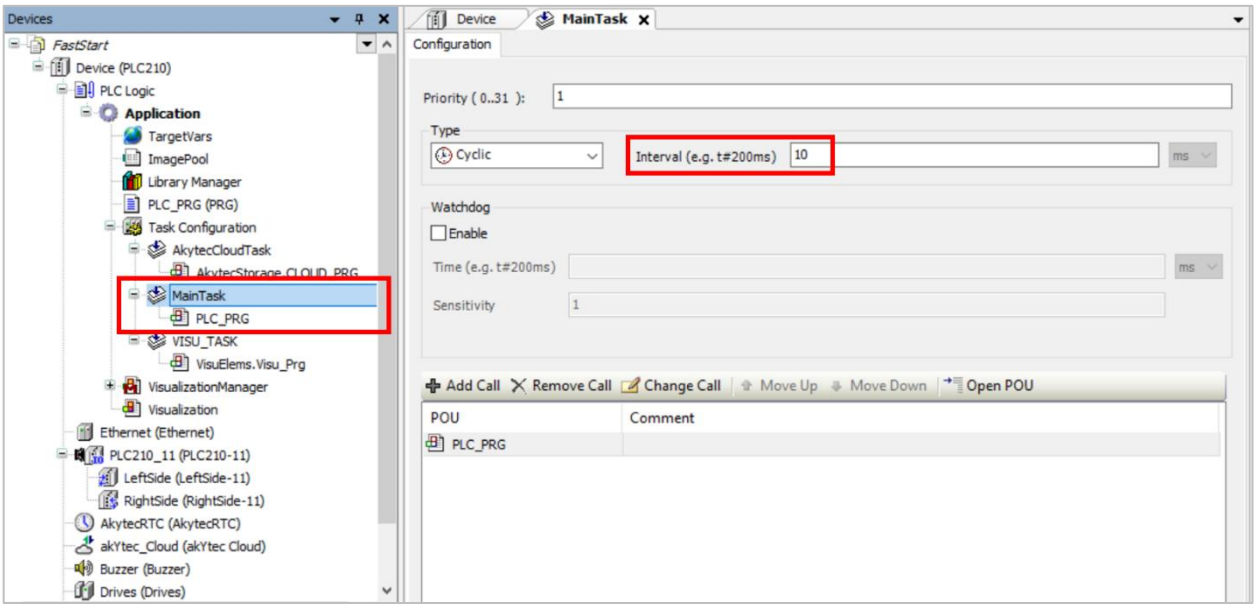


Fig. 3.6.8 MainTask parameters

3.6.8 How to enable grid in visualization editor?

To enable grid in the visualization editor, open ***Tools*** > ***Options*** > ***Visualization*** > ***Grid*** and check the checkbox ***Visible***.

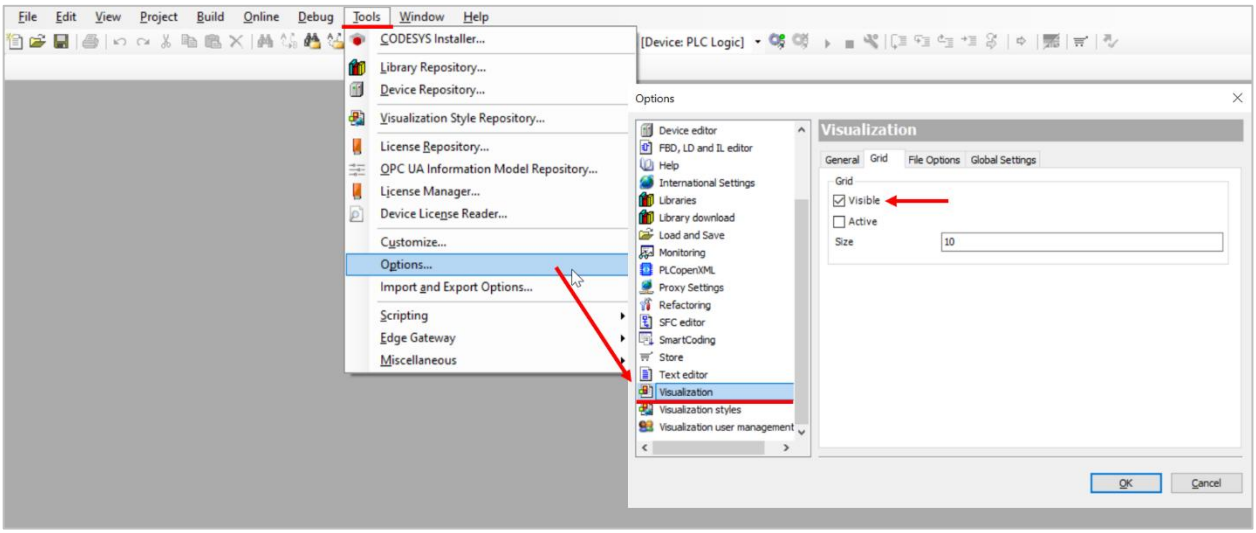


Fig. 3.6.9 Make grid in visualization editor visible

3.6.9 Is the size of memory occupied by application limited?

There are no explicit limitations on the size of the application, there is only a limitation on the size of free device memory. For PLC210 the size of memory available to the user is about 70 MB, for SPC210 – 2 GB.

You can view the current size of free memory with a connected PLC in the ***Drives*** node. Double-click the node and set the value in the channel ***Enable Drives*** to ***TRUE***. The size of free memory in bytes will be displayed in the channel ***FS free***.

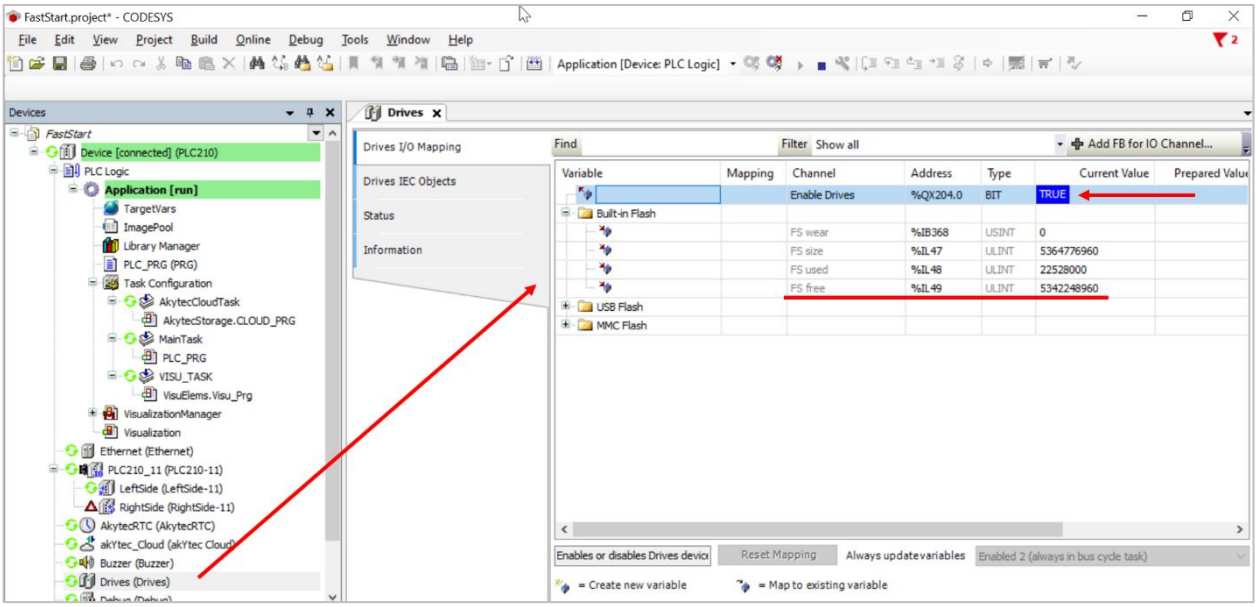


Fig. 3.6.10 Size of free PLC memory in Drives node

4. CODESYS interface

In this section, we will take a closer look at the CODESYS environment interface and learn about the most frequently used menu items and commands.

4.1 CODESYS interface overview

The main areas of the CODESYS interface are shown in Figure 4.1.1.

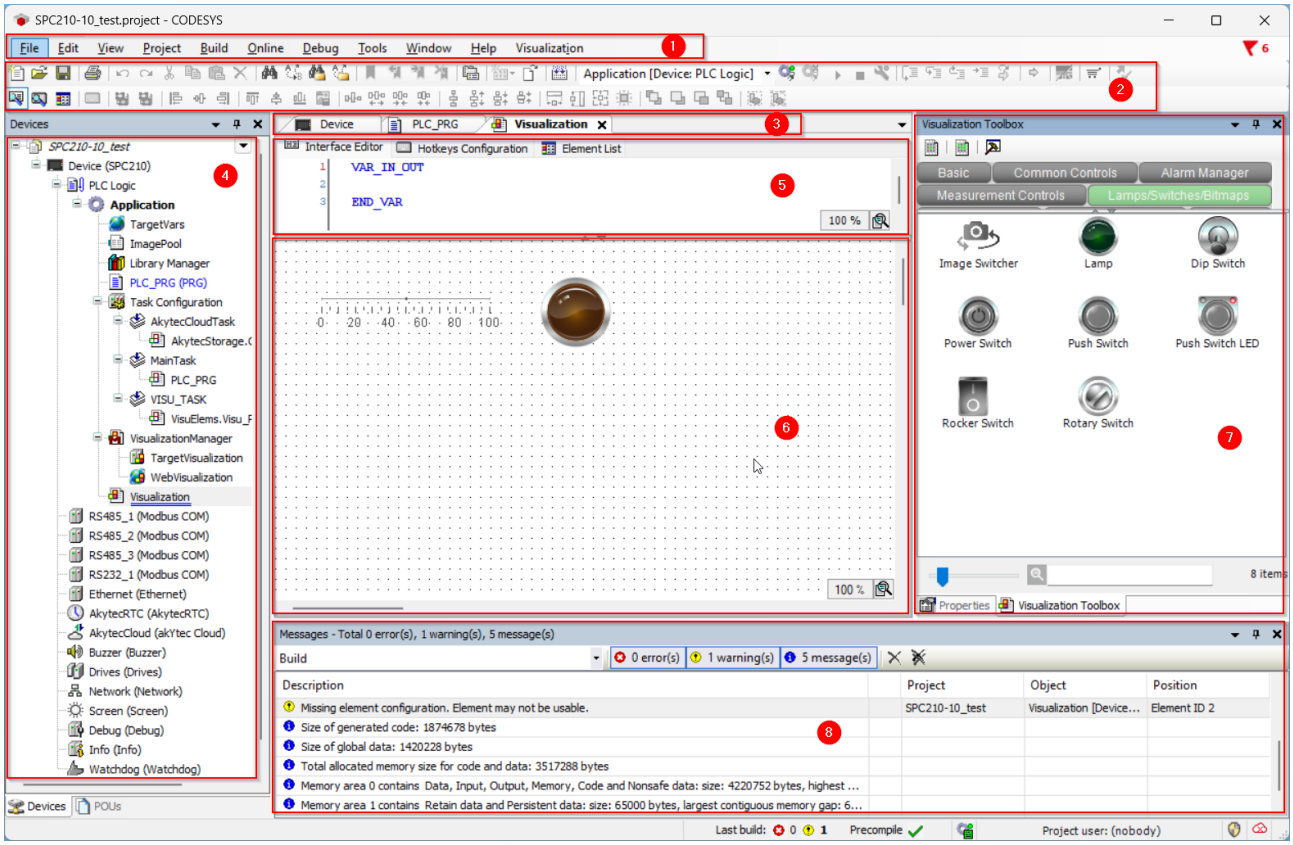


Fig. 4.1.1 Main areas of the CODESYS interface

1. Main menu with drop-down lists of actions / options
2. Toolbar contains shortcuts for the most frequently used actions / options
3. Tabs for open CODESYS components (editors, configurators, etc.). To open a new tab, double-click with the left mouse button on a node in the project tree.
4. Project tree contains a tree structure of project objects
5. Variable declaration area of programming language and visualization editors
6. Panel of the active editor
7. Panel with elements that can be used in the active editor panel (e.g., CFC language element panel, visualization element panel, etc.)
8. Message panel. Other panels may open in the same area, such as the variable monitoring panel, the breakpoint panel, etc.

Tabs and panels can be moved around the screen by holding down the left mouse button on the tab/panel title. The set of commands in the menu and toolbar shortcuts can be changed in the ***Tools*** > ***Settings*** menu.

4.2 Main menu

4.2.1 File menu

The ***File*** menu contains basic actions for working with a project: opening, closing, saving, print etc.

The ***Project Archive*** action saves the project in ***.projectarchive*** format. Unlike a project file (***.project***), an archive file includes components, libraries, device descriptions, etc. used in the project. This is very convenient when transferring a project from one PC to another. In this case, you will not need to separately install the libraries and components required for a specific project on another PC.

The ***Source Upload*** and ***Source Download*** commands are used to upload and download the project file (***.project***) to the controller. In case you need to make changes to the project, but the source code has been lost, you can download it from the controller.

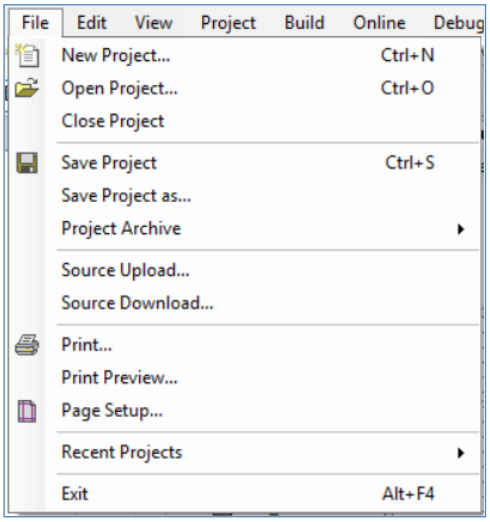


Fig. 4.2.1 File menu

4.2.2 Edit menu

The ***Edit*** menu contains basic actions for working with project content: undo/redo, copy/paste, find, replace, delete etc.

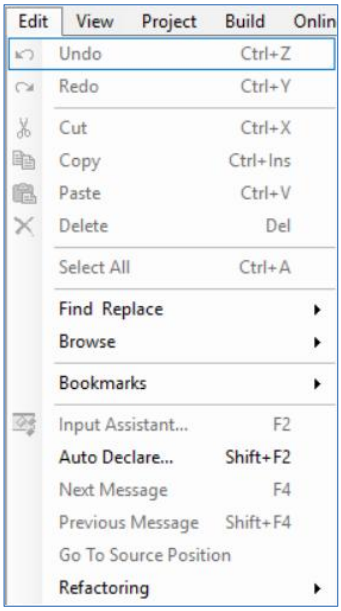


Fig. 4.2.2 Edit menu

4.2.3 View menu

The ***View*** menu controls the visual presentation of the interface: you can show/hide different panels, such as ***Devices*** (project tree), ***Messages*** (compilation and other messages), ***Toolboxes*** and create watchlists for variables. If you accidentally close any panels you need, you can reopen them using this menu.

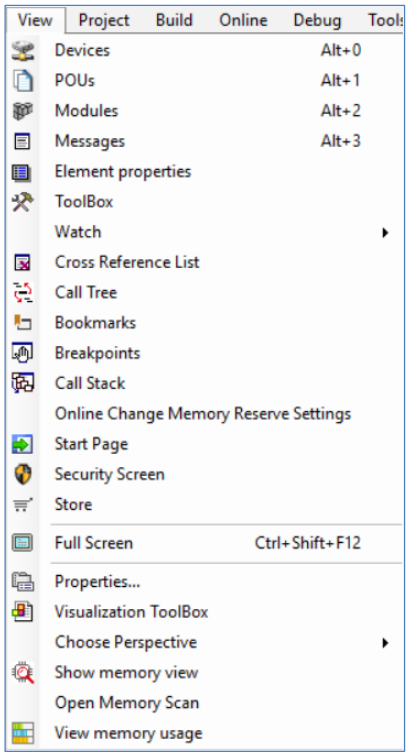


Fig. 4.2.3 View menu

4.2.4 Project menu

The ***Project*** menu contains commands for configuring the project and exporting/importing its contents in various formats.

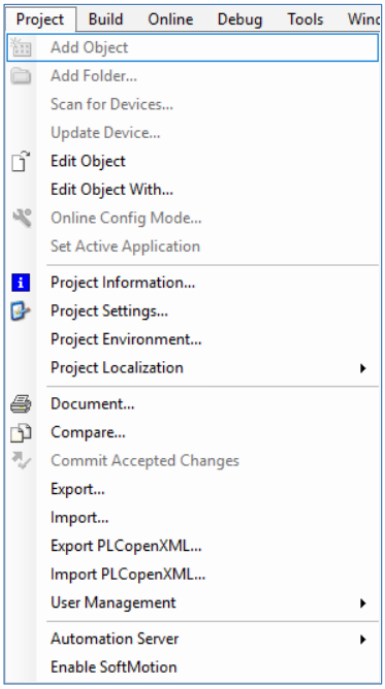


Fig. 4.2.4 Project menu

The most frequently used commands are:

- ***Project settings*** – opens the settings window for this project. The most frequently used settings are:

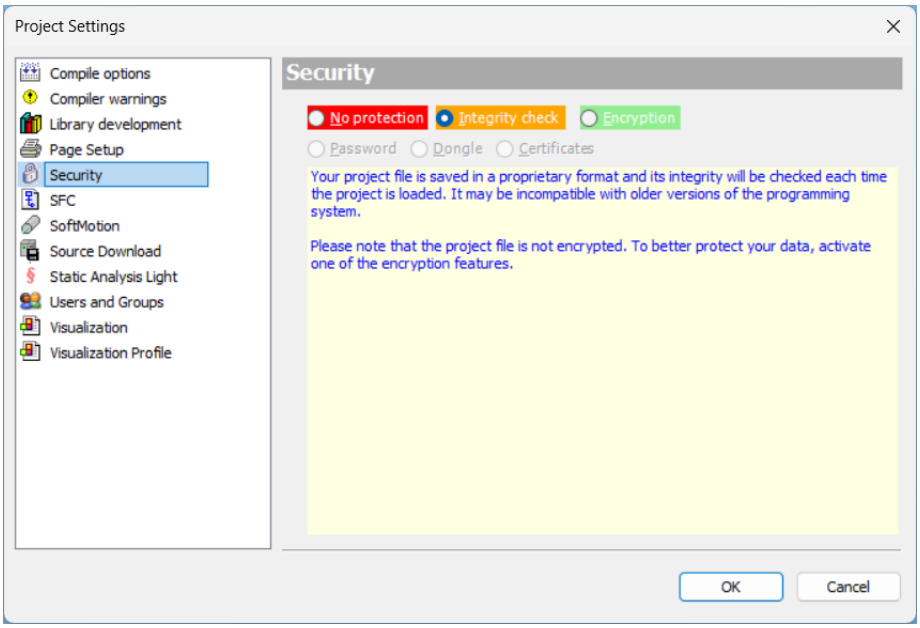


Fig. 4.2.5 Project settings > Security

  - ***Security*** – enables you to restrict access to the project using a password/certificate/digital key to protect intellectual property. The actions in this tab apply to the project file as a whole.
  - ***Compile options*** and ***Visualization profile*** – the settings in these tabs must correspond to the version of the controller target file. These settings must be changed when migrating a project created in an older version of CODESYS to a newer version.

- ***Export... / Import...*** – these commands are used to export/import a project or any of its fragments (e.g., function blocks, visualization screens, etc.) in a CODESYS-specific XML-like format. This is useful when you need to transfer a fragment of your project to another user without transferring the project itself.
- ***PLCopenXML Export*** / ***PLCopen XML Import*** – these commands are used to export/import project software objects in ***PLCopen XML*** format. This is a universal format for transferring software objects between different development environments for controllers, specified in the IEC 61131-10 standard.
- ***User Management*** – with this command you can create a number of users with different rights for a specific project (e.g., one user will have read-only access to the project, another will have no access to specific objects, etc.). You can configure access rights for each object in the project tree. Each user is assigned a login and a password.

  Contrary to the ***Security*** settings mentioned above, the settings in this tab apply to specific project elements and not to the project file as a whole.

4.2.5 Build menu

Generate Code command of the ***Build*** menu checks the project for errors and, if none are found, creates an application file that will be loaded into the controller when connected to it.

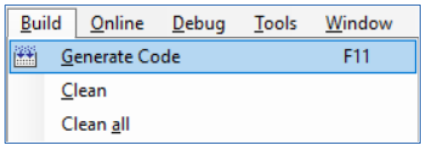


Fig. 4.2.6 Build menu

Information about checks is stored in internal CODESYS files. In some cases, these files may be damaged or overwritten incorrectly (e.g., during constant regular checks). This can lead to error messages that do not actually exist in the project. Therefore, it is recommended to run the ***Clear All*** command before running the ***Generate Code*** command to delete information about previous project checks.

Clear command removes the compilation information only for the active application, contrary to the ***Clear all*** command, that removes the compilation information for all applications in the project.

If errors are detected during code generation, information about them will be displayed in the compilation message panel (Sect 3.6.6).

4.2.6 Online menu

The Online menu contains commands for connecting to the controller, downloading the application and project source code, and resetting the controller.

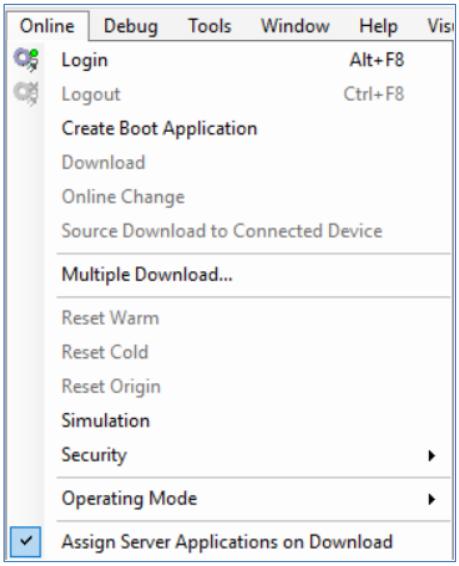


Fig. 4.2.7 Online menu

The most frequently used commands are:

Login – the command to connect to the controller. During the login process, a new application can be downloaded to the controller (Sect. 3.5).

Create Boot Application – if the command is executed while the controller is connected, the loaded application is saved to the controller's flash memory. If this command is not executed after the application is loaded, the application will not be saved after the controller reboot (Sect. 3.6.1). If the command is executed when there is no connection to the controller, the boot application is saved on the PC; it can then be loaded into the controller from a USB/SD drive or via the configurator.

Source Download to Connected Device – similar to the ***Source Download*** command in the ***File*** menu but enables you to download the project file directly to the connected controller. The ***Source Download*** command opens a list of controllers found on the network for downloading the project to the selected one.

Reset – the command to reinitialize all project variables except non-volatile ones. After it is executed, the variables take on the initial values specified when they were declared. If no initial values are specified, default values are used (e.g., ***0*** for numeric types).

Cold reset – the command to reinitialize project variables, including non-volatile (***RETAIN***) variables.

Factory reset – deletes the current application and its source code (if loaded) from the controller. All other settings (network settings, etc.) will not be reset to factory defaults – they will retain their current values.

Emulation – the command switches the project to emulation mode. use the Login command to connect to the emulator. Emulation mode has serious limitations. In particular, it cannot be used to check:

- web visualization
- file operations
- communication with other devices

Therefore, it is not recommended to use emulation mode at all and instead check the functioning of the project on a virtual controller (Sect. 3.4).

4.2.7 Debug menu

The Debug menu contains commands for analyzing errors in the project code. Most commands are only available when connected to the controller.

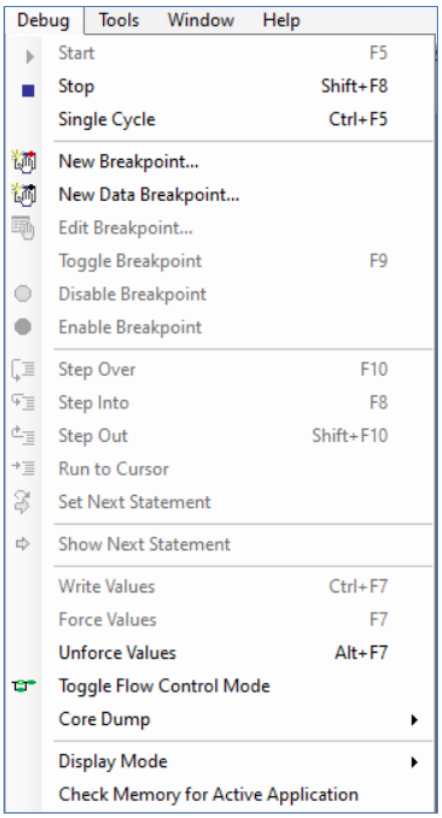


Fig. 4.2.8 Debug menu

The most frequently used commands are:

- ***Start*** – the command to launch the application, usually executed by the user immediately after downloading a new application.
- ***Breakpoint commands*** – these commands enable you to set breakpoints in project code editors. When these points are reached, the application will automatically pause. This enables you, e.g., to track whether certain code branches are being executed, to determine the values of project variables at the moment certain conditions are met, etc.
- ***Steps-related commands*** – if the application is stopped at a breakpoint, these commands can be used to execute it in fragments, going through each line of code separately.
- ***Write values*** – the command is used to assign user-defined values to project variables (see Fig. 3.5.7 and the explanation before and after it). If the value of a variable changes cyclically within the project (e.g., the variable is linked to an input of the controller or an input module connected to it, or it

is assigned the value of another variable), the ***Write Values*** command will not work – use instead the ***Force Values*** command.

- ***Force Values*** – the command is used to forcibly write the user-specified value to the variable at the beginning of each cycle (e.g., the variable is linked to the analog input of the controller, to which no sensor is connected, so the input has a value of ***0.0***). To simulate the sensor value, use the ***Force Values*** command. If you use the ***Write Values*** command, the value written by the user will immediately be reset to 0.0, since the sensor is polled cyclically.
- ***Unforce Values*** – the command disables the locking of the values of the selected variables.
- ***Display mode*** – switches the display mode for integer variables (available modes: BIN/DEC/HEX).

Along with the ***Debug*** menu commands, it is helpful to use additional panels that can be opened in the ***View*** menu (Sect. 4.2.3):

- ***Watch*** – used to create a list of variables to monitor their values with the ability to change them using the ***Write Values*** and ***Force Values*** commands. Variables can be declared in different project objects.
- ***Watch*** –> ***Watch all Forces*** – displays all project variables that are in a forced state.
- ***Breakpoints*** – displays all breakpoints in the project.

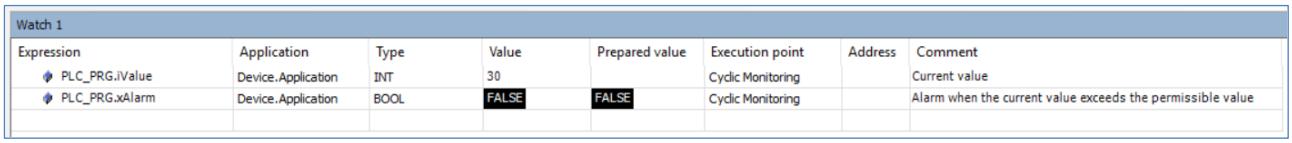


Fig. 4.2.9 Watch panel

4.2.8 Tools menu

The menu contains commands for installing additional components (packages, libraries, etc.) in CODESYS, as well as for configuring the development environment.

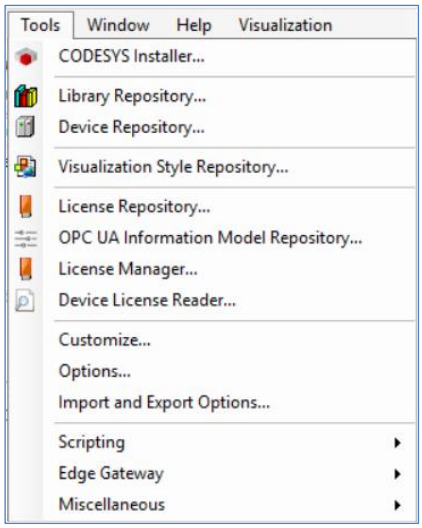


Fig. 4.2.10 Tools menu

The most frequently used commands are:

- ***CODESYS Installer*** – used to install packages (in particular, target file packages), CODESYS updates, and create environments (Sect. 2.2).
- ***Library Repository*** – used to install additional libraries (Sect. 2.3).

- ***Customize*** – used to customize the appearance of the development environment (set of commands in the menu, set of shortcuts on the toolbar, hotkey combinations).
- ***Options*** – used to configure CODESYS. In this menu you can:
  - change the CODESYS interface language (***International Settings*** tab)
  - enable or disable the grid in the visualization editor (***Visualization*** tab)
  - configure the color and font of the text in the ST language editor (***Text editor*** tab)

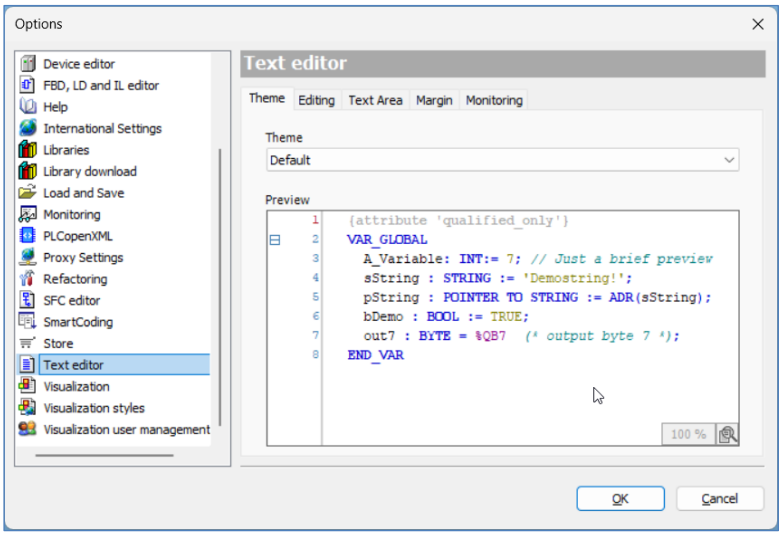


Fig. 4.2.11 Options menu

4.2.9 Window menu

The menu contains commands used to arrange the window layout (switching between and closing tabs and panels opened in the environment).

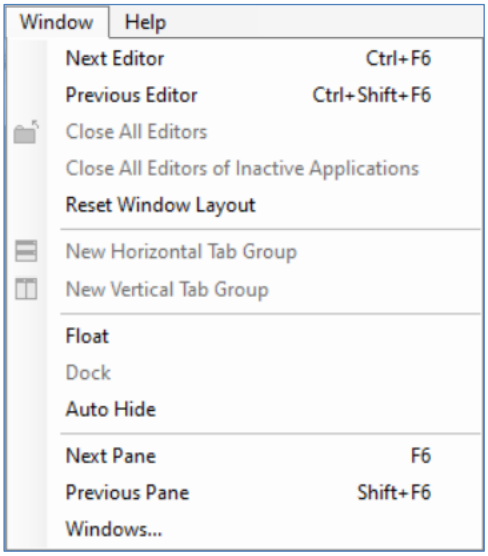


Fig. 4.2.12 Window menu

4.2.10 Help menu

The menu contains commands for displaying information about the version of the environment being used (About) and for accessing CODESYS online help. If there is no Internet connection, the offline version of the help is open.

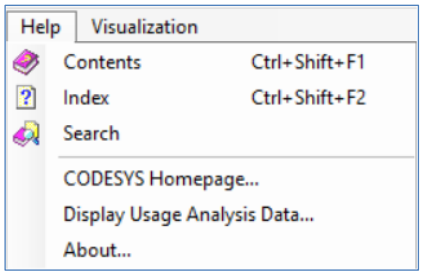


Fig. 4.2.13 Help menu

4.3 Project tree

4.3.1 Controller and component nodes

The project tree is a hierarchical structure of objects used in the project. The top-level object is the controller (***Device***). There can be several controllers in the project. CODESYS provides convenient tools for configuring communication among them, which are not covered in the “First Start”.

Initially, the project includes one controller. If you need to add other controllers to the project, right-click on the project name or on the free area under the tree and select ***Add Device*** from the context menu.

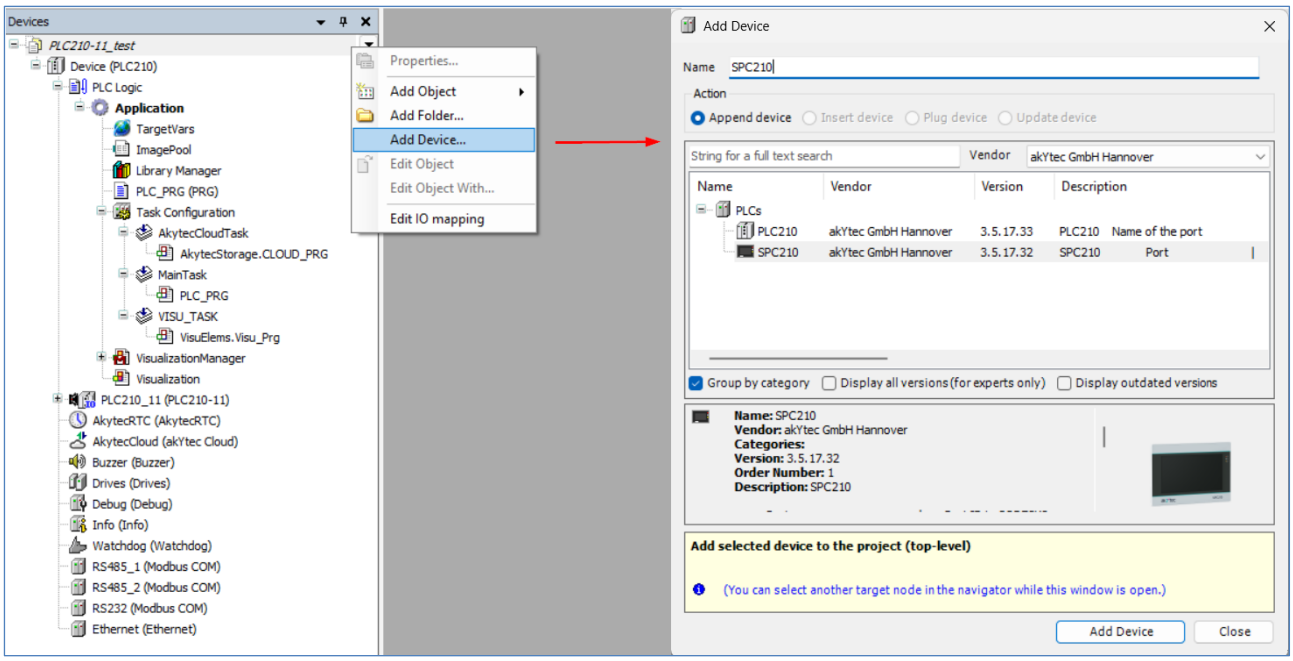


Fig. 4.3.1 Add controller to project

The controller node includes the ***Application*** node and additional components (e.g., communication driver nodes, system nodes from the target file, etc.). To add a component, select ***Add Device*** from the context menu of the controller node (named ***Device*** by default).

In the new window (Fig. 4.3.2) you can see the protocol components supported in CODESYS: Ethernet/IP, Modbus, Profinet, etc. Not all of these protocols can be used in akYtec PLCs because a license is required for their use. akYtec controllers are licensed for Modbus Serial, Modbus TCP, and OPC UA (server mode). The ability to activate licenses for other protocols is not currently supported. However, a number of additional protocols have been implemented as libraries (e.g., SNMP, etc.).

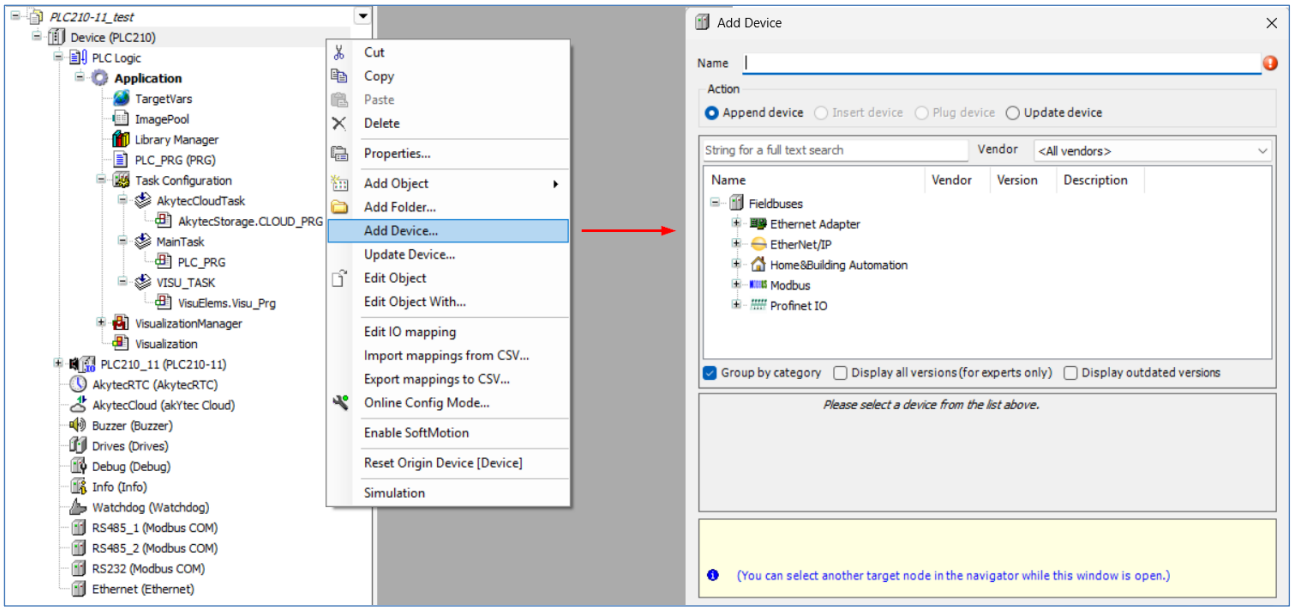


Fig. 4.3.2 Add component to device

Some components may include child components. To add them, use the ***Add Device*** command from the component context menu. Figure 4.3.3 shows how to add a ***Modbus Master*** or ***Modbus Serial Device*** component to the ***Modbus COM*** component, which determines the operating mode of the selected COM port of the controller (Master or Slave).

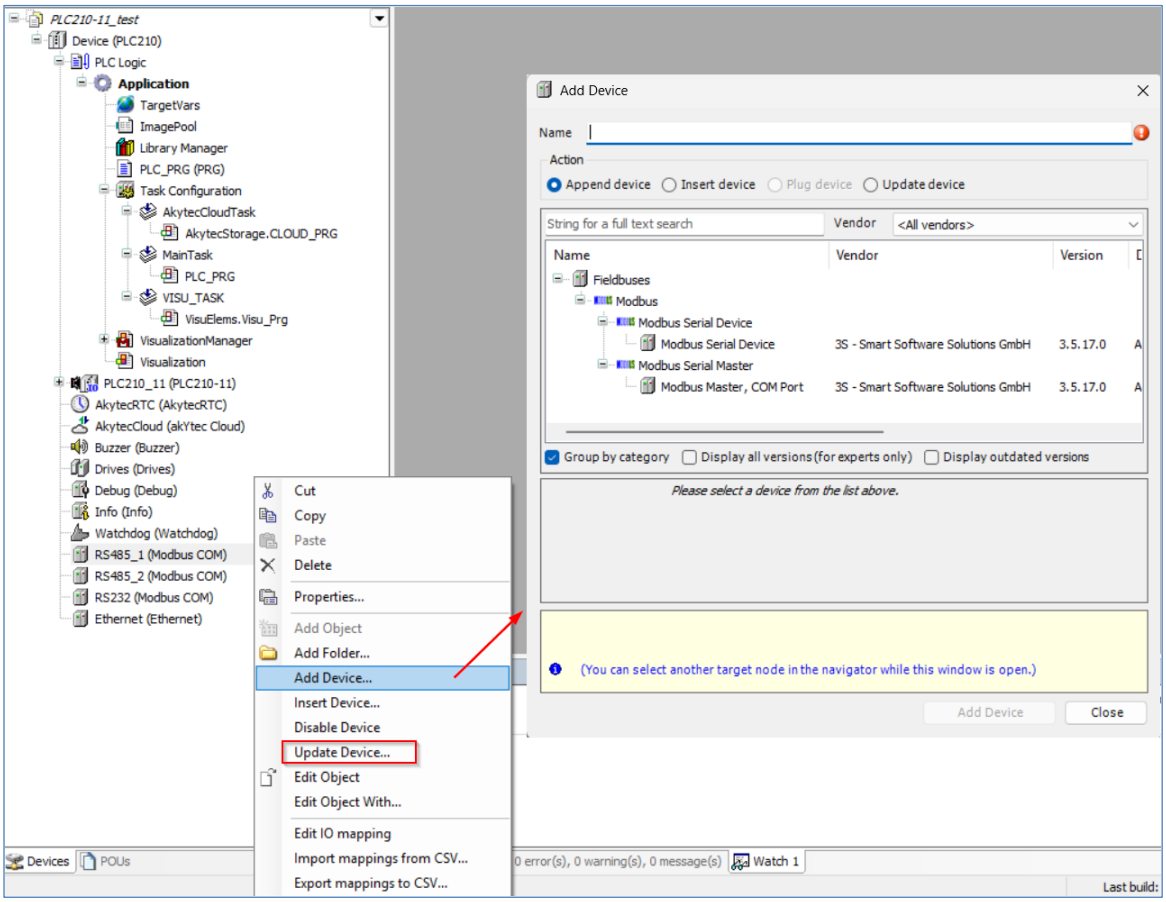


Fig. 4.3.3 Add child component to component

You can use the ***Update Device*** command to change the version of a component (e.g., to update the version of the controller target file) or to replace one component with another.

The target files of akYtec controllers include a number of system nodes, each of which simplifies working with specific PLC functionality. These nodes cannot be deleted from the project tree. Below is an overview of them:

- ***AkytecRTC*** – reading/writing the controller's system time
- ***AkytecCloud*** – settings for connecting to the AkytecCloud cloud service. Connecting to the cloud service is described in Sect. 6.10.
- ***Buzzer*** – control of the controller's piezo emitter (buzzer).
- ***Drives*** – enables you to read information about the controller memory and connected drives, as well as determine which drive is mounted and unmount it.
- ***Network*** – read/write the controller's network settings (for SPC controllers only).
- ***Screen*** – enables you to control the brightness of the controller screen backlight and configure the screen sleep mode settings (for SPC controllers only).
- ***Debug*** – enables the reading of debug parameter values (processor load, etc.).
- ***Info*** – enables the reading of information about the controller and project (firmware version, serial number, project name, etc.).
- ***Watchdog*** – enables you to obtain information from the controller's watchdog timer (information about the last exception, reboot counters, etc.) and reboot the controller from the program code.

A detailed description of the target file system nodes is provided in the document “CODESYS V3.5 Target File”. PLC210 and PLC200 controllers also have special nodes for working with inputs and outputs.

All system nodes have an ***I/O Mapping*** tab, where project variables are linked to component channels. In the project template, variables are already linked to all channels of all nodes (Sect. 4.3.4).

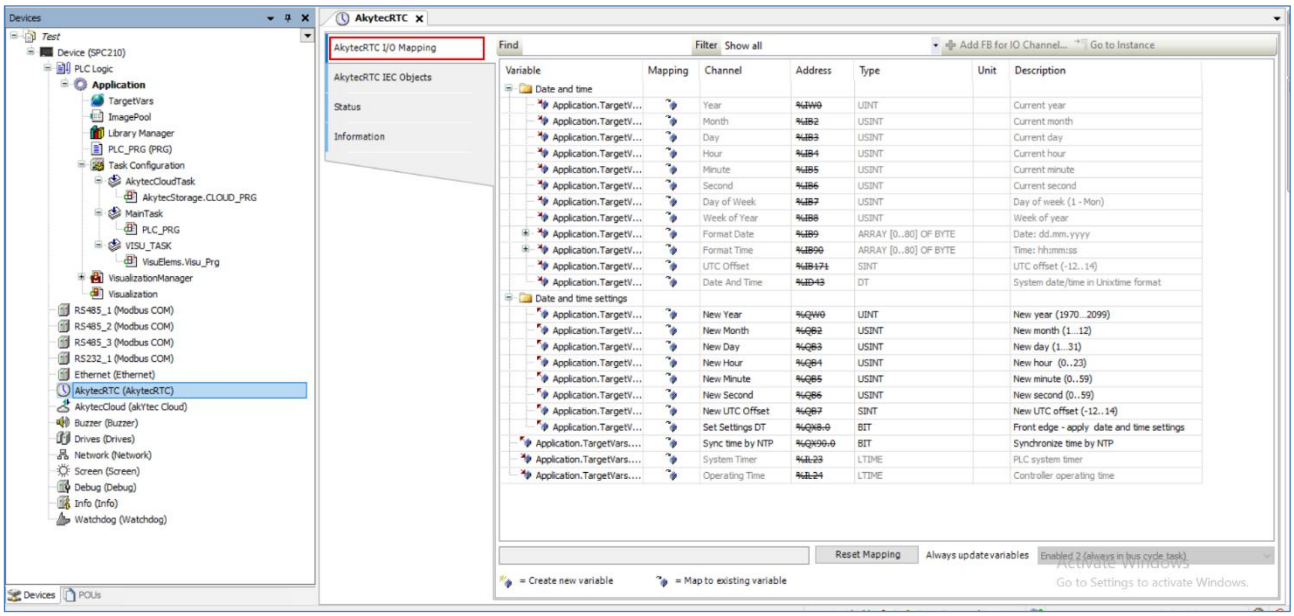


Fig. 4.3.4 Variables linked to component channels

Some nodes also have a ***Parameters*** tab. Here, the user can set the parameter values that cannot be changed while the application is running. To change them, a new application must be loaded into the controller.

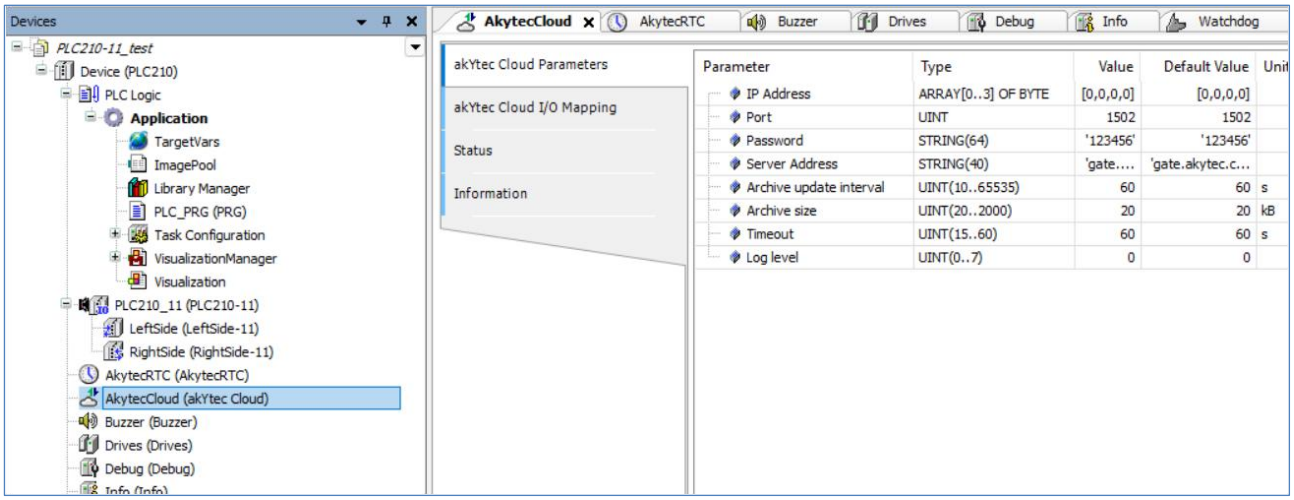


Fig. 4.3.5 Component parameters

4.3.2 Application node

An akYtec controller in the project can only contain one application which is named ***Application*** and cannot be renamed. The ***Application*** node includes objects such as programs, functional blocks, visualization screens, etc.

To add an object to the project, select the ***Add Object*** command from the context menu of the ***Application*** node.

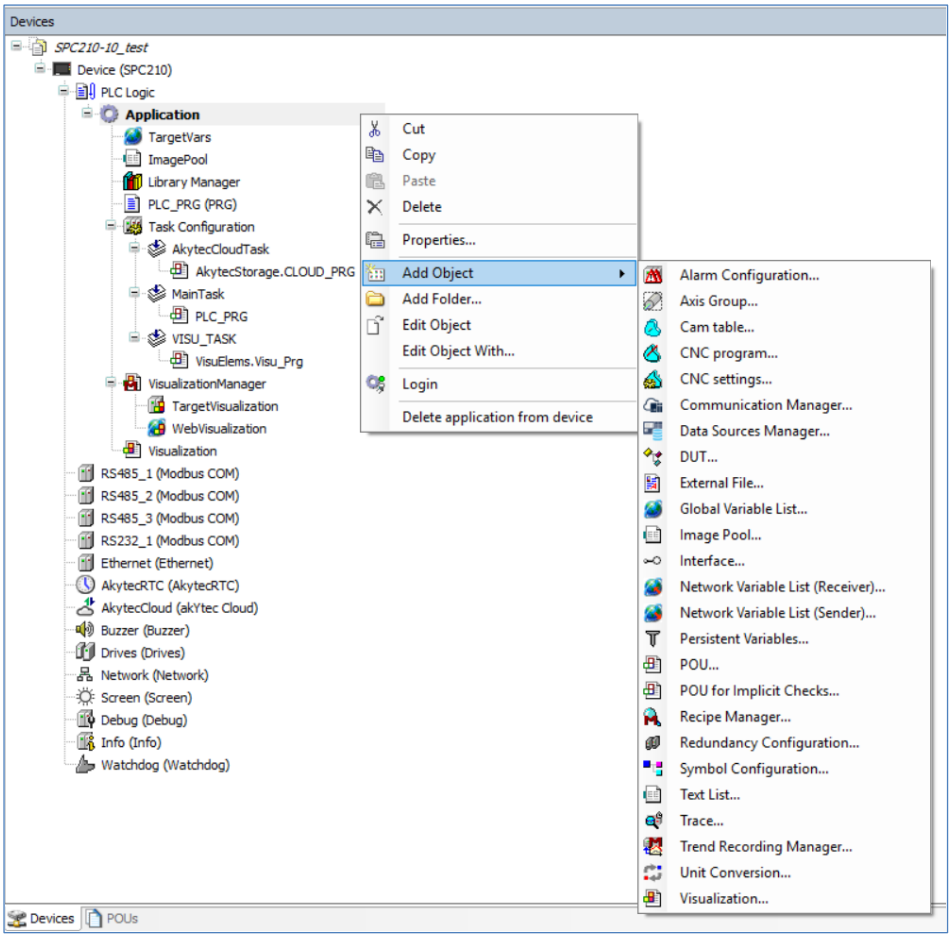


Fig. 4.3.6 Add object to application

Objects can be grouped into several categories:

Table 4.3.1 Categories of objects

Category	Objects
Program objects	DUT, Persistent Variables, POU, POU for Implicit Checks, Interface, Global Variable List, Task Configuration, Library Manager
Visualization objects	Visualization, Visualization Manager, Unit Conversion, Alarm Configuration, Trend Recording Manager, Recipe Manager, Image Pool, Text List
Communication objects	Communication Manager, Data Sources Manager, Symbol Configuration, Network Variable List
Other objects	Trace, External File
Objects not supported by akYtec controllers	Axis Group, Cam table, CNC Program, CNC Settings, Redundancy configuration

Some objects may include child objects (sub-objects) (e.g., in Fig. 4.3.6, the child objects ***Target Visualization*** and ***WebVisualization*** have been added to the ***Visualization Manager*** object).

Some objects have already been briefly discussed in Sect. 3.1, and we will discuss a few more in Sect. 6 while creating a demo project.

4.3.3 POUs tab

As mentioned in Sect. 4.3.1, a project may include several controllers, each with its own application. In this case, it may be necessary for certain objects (e.g., functional blocks) to be accessible for all controllers (applications) in the project. Such globally accessible objects are located in the ***POUs*** tab.

To open POUs, click on the ***POUs*** tab at the bottom of the project tree (Fig. 4.3.7) or use the ***POUs*** command from the ***View*** menu.

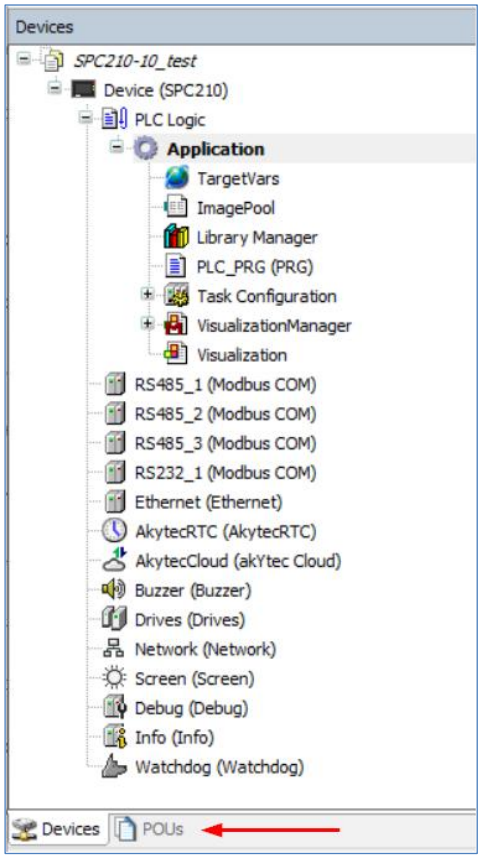


Fig. 4.3.7 Switch to POUs tab

This tab already contains several objects by default:

- ***GlobalTextList*** – a global list of texts containing static texts for project visualization.
- ***Library Manager*** – similar to the ***Library Manager*** in the ***Application***, but the libraries added to it are available to all project objects.
- ***Project Information*** and ***Project Settings*** – links to the windows that can also be opened using the commands of the same name in the ***Project*** menu.

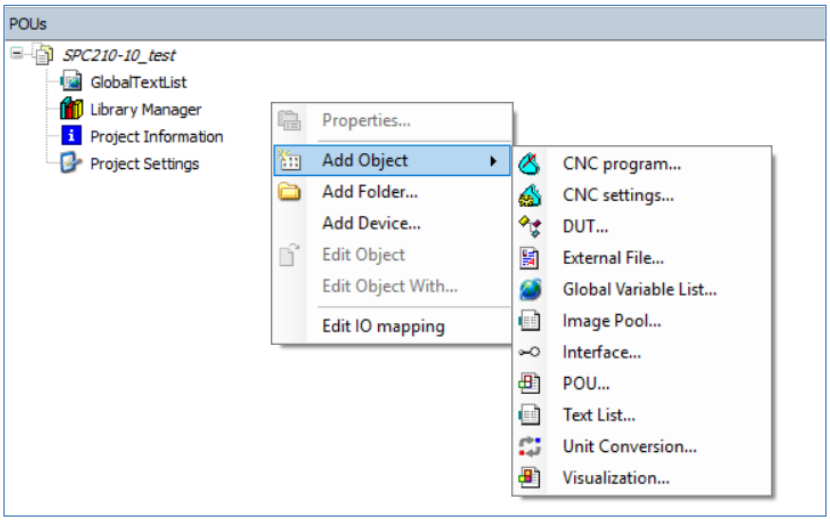


Fig. 4.3.8 Add object to POU node

4.3.4 Project template composition

In Sect. 3.1, we created a project based on a template. Project templates for akYtec controllers are installed together with the akYtec target file package.

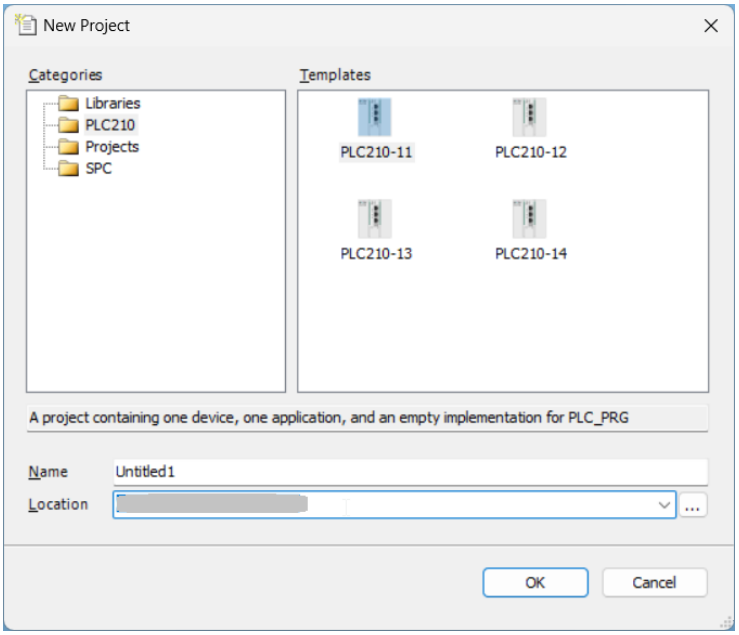


Fig. 4.3.9 Project template selection

Compared to the “Standard project” (which you can select in the Templates pane), the template additionally includes:

- ***TargetVars*** – list of global variables. The list contains instances of structures whose variables are linked to the system nodes of the target file. This provides access to a wide range of “system” variables at the template level (e.g., to read the controller's system time, its serial number, control its inputs/outputs, etc.). Further details on structures and global variables are explained in Sect. 5.3.9 and 5.4.6 respectively.
- ***ImagePool*** – graphic files added to this object can be used when creating visualizations.
- ***Visualization Manager*** – contains settings for a specific controller model (e.g., the correct screen resolution is specified for SPC panel controllers).
- ***Visualization*** – screen visualization for the controller
- Communication components for RS485/RS232 and Ethernet interfaces. For RS485/RS232, the COM port identifiers of the controller are specified. If there are interfaces that are not used in the project, it is recommended to remove their components from the project tree.

The Library Manager of the template contains frequently used libraries:

- ***akYtecTypes*** – library of parameter structures for system nodes of akYtec target files
- ***Standard64*** – library of functions for working with WSTRING strings, 64-bit pulse counters, etc.
- ***Util*** – library of frequently used functional blocks (linear scaling, hysteresis, PID controller, etc.)
- ***CAA Memory*** – library of functions for working with memory (copying memory blocks, filling memory blocks with a specified value, etc.).

Template files are stored in the CODESYS installation directory in the folder “…/CODESYS/Templates”.

You can edit the templates to suit your needs or create your own templates.

5. Programming basics

In this section, we'll introduce you to the basics of PLC programming in IEC 61131-3 languages, focusing on the ST language.

Application architecture in CODESYS:

- The application contains a ***Task Configuration*** object with one or more cyclically executed tasks with defined intervals for program calls.
- Each task is associated with one or more ***programs*** – large “building blocks” of the application that implement the overall control system algorithm or a significant part of it.
- Programs are build using ***variables***, ***operators*** (regular and conditional), and program objects – ***functions*** and ***function blocks***.
- ***Functions*** and ***function blocks*** use ***variables***, ***operators*** (regular and conditional), and function calls. Function blocks can call other function blocks.
- ***Operators*** are symbols that perform standard operations on data (***variables*** and values).
- ***Variables*** are labeled containers for storing data of a certain data type (e.g., integer or string) and are declared in certain areas (e.g., input, output, or global variable declaration areas). Declaration areas can be marked with special modifiers to make variables non-volatile.

In the following sections, we will discuss the elements of this architecture in detail.

5.1 IEC 61131-3 programming languages

The IEC 61131 standard defines requirements for programmable logic controllers. The third section of the standard (usually referred to as IEC 61131-3) is devoted to programming languages for controllers. The first edition of this section was released in 1993. The third edition from 2013 introduced aspects of object-oriented programming (OOP) such as classes, methods, interfaces, and namespaces to improve code reusability and structuring. This was further expanded in the fourth edition from 2025.

The IEC 61131-3 standard defines five programming languages for controllers:

- Ladder diagram (***LD***) – graphic language for relay contact diagrams. Historically, it is the first PLC programming language.
- Function block diagram (***FBD***) – graphic language for function blocks.
- Sequential function chart (***SFC***) – graphical state diagram language. Unlike other languages in the standard, it is a design language rather than a programming language. SFC describes the structure of a program as a finite state machine consisting of steps, transitions between steps, and actions performed at these steps. Actions must be coded in one of the other languages in the standard.
- Instruction list (***IL***) – low-level text language similar to assembly language. In the third edition of the standard, it is recognized as obsolete (“deprecated”). In recent versions of CODESYS, it is disabled by default; you can enable its support in the menu ***Tools*** > ***Options*** > ***FDB, LD, and IL Editor*** > ***IL***.
- Structured text (***ST***) – high-level text-based programming language similar to Pascal language

CODESYS supports all five languages of IEC 61131-3 and, in addition to them, the graphical language Continuous Function Chart (***CFC***) of function blocks (not to be confused with ***SFC***).

The differences between ***FBD*** and ***CFC***:

- FBD stands for “rigid” placement of blocks on the canvas – blocks are added one by one, and each added block is automatically linked to one of the previously added ones. The order in which blocks are executed is determined by the order in which they are placed.

- ***CFC*** supports free placement of blocks on the canvas – the user can first add multiple blocks to the canvas and then draw connections between them, Reverse connections, where the output of one block is connected to the input of one of the previous blocks, are also enabled. The execution order is determined either automatically, depending on their relative positions (execution proceeds from left to right and from top to bottom), or by the user for each block manually.

Fig. 5.1.1…5.1.6 show examples of code in various languages supported by CODESYS.

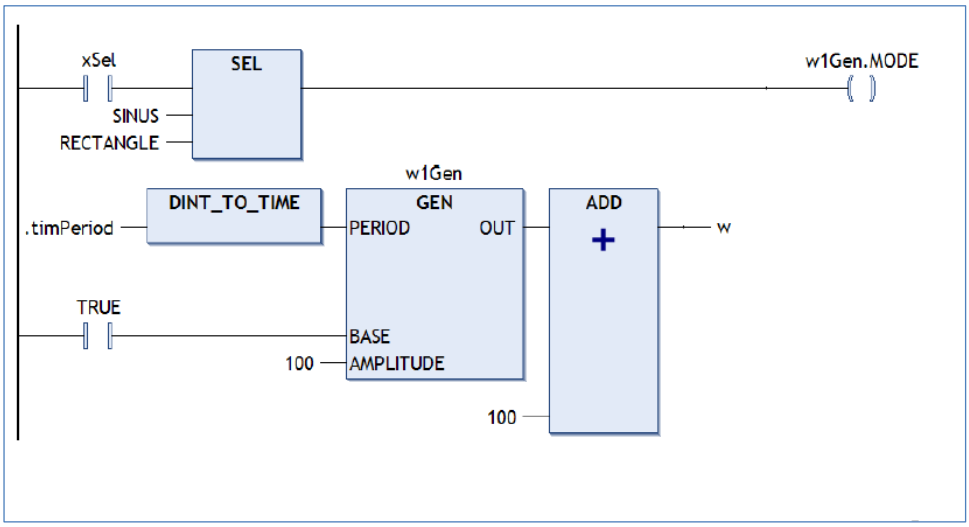


Fig. 5.1.1 LD language program code

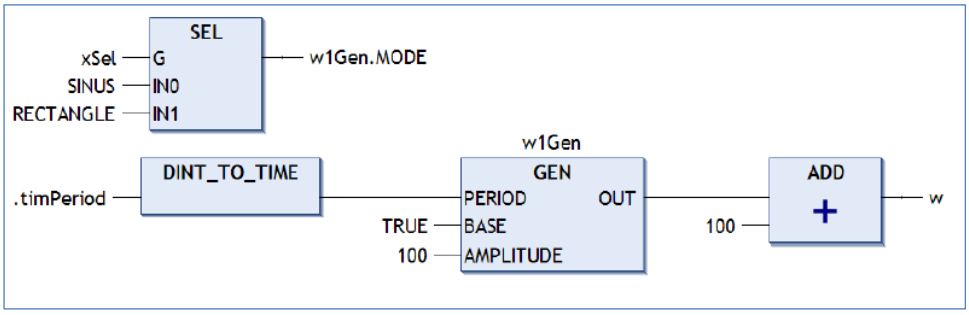


Fig. 5.1.2 FBD language program code

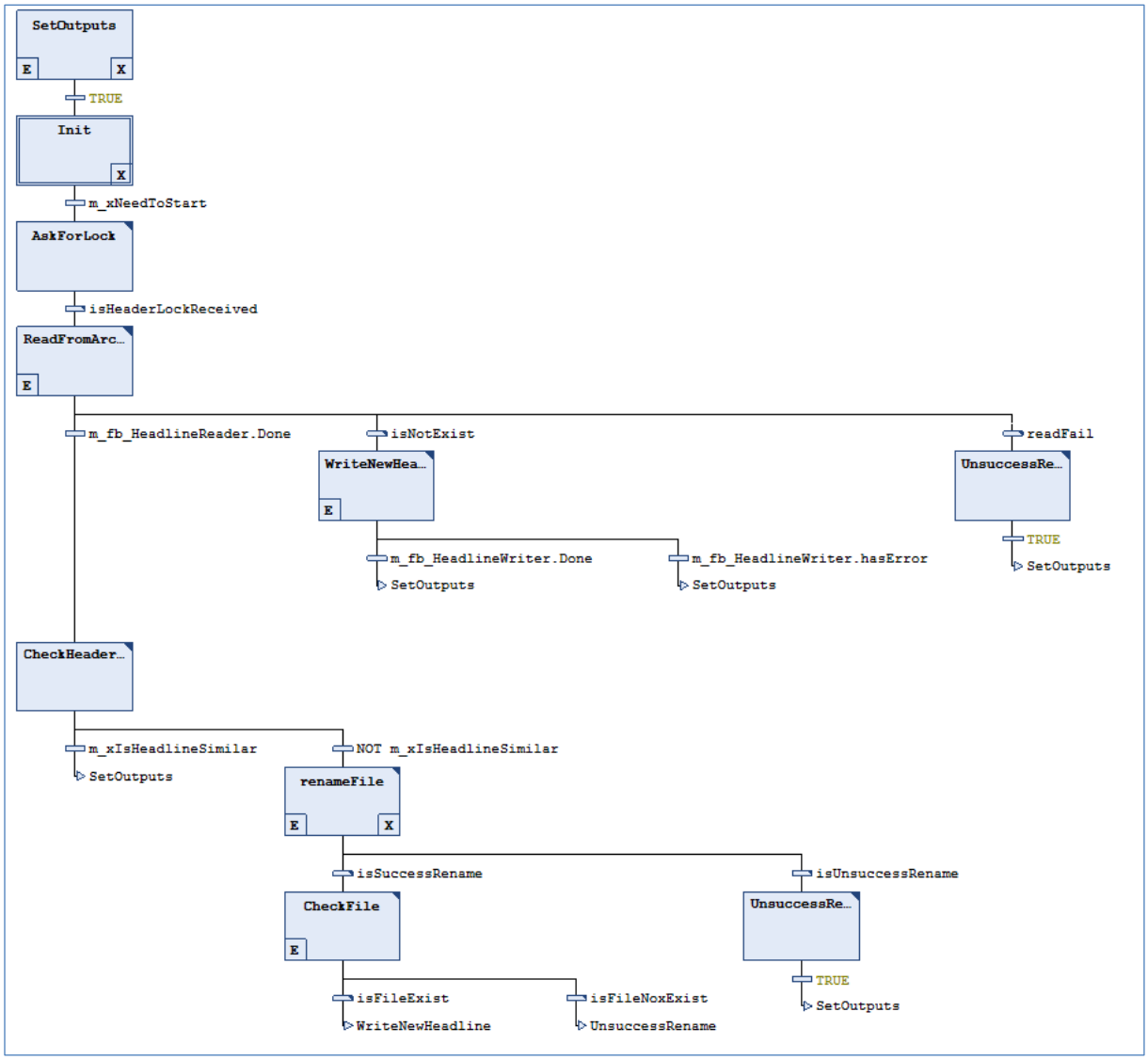


Fig. 5.1.3 SFC language program code

```
LD              xSel
SEL             SINUS
                RECTANGLE
ST              w1Gen.MODE
LD              .timPeriod
DINT_TO_TIME
ST              w1Gen.PERIOD
CAL             w1Gen(
          BASE:= TRUE,
     AMPLITUDE:= 100)
LD              w1Gen.OUT
ADD             100
ST              w
```

Fig. 5.1.4 IL language program code

```
_uiBufferSize := UDINT_TO_UINT (szBuffer);

Mem.MemMove
(
    pSource         := pBuffer,
    pDestination    := ADR (_abyBuffer),
    uiNumberOfBytes := _uiBufferSize
);

FOR i := 0 TO UINT_TO_INT (_uiBufferSize) - 1 DO
    _abyBuffer [i] := Util.BCD_TO_BYTE (_abyBuffer [i]);
    _usiExponent := INT_TO_USINT (UINT_TO_INT (_uiBufferSize) -1 - i);
    IF _usiExponent > 0 THEN
        _udiValue := _udiValue + LREAL_TO_UDINT(_abyBuffer [i] * EXPT (10, UINT_TO_INT (_uiBufferSize) - i));
    ELSE
        _udiValue := _udiValue +_abyBuffer [i];
    END_IF
END_FOR

IF uiAccuracy > 0 THEN
    _rValue := UDINT_TO_REAL (_udiValue) / uiAccuracy;
ELSE
    _rValue := 0.0;
END_IF

GetRealByHEXBytes := _rValue;
```

Fig. 5.1.5 ST language program code

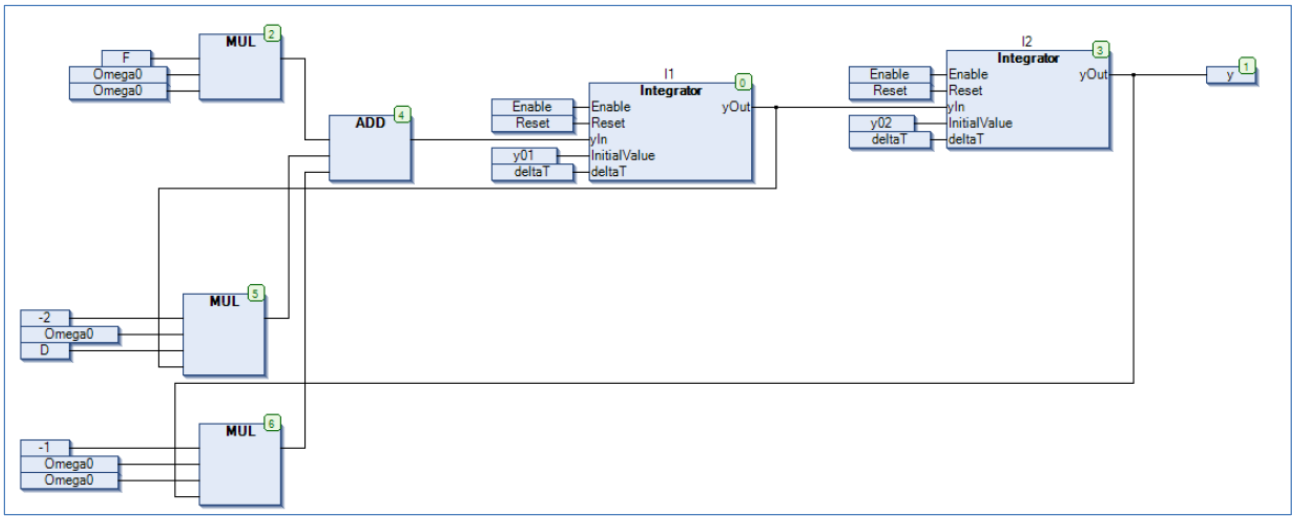


Fig. 5.1.6 CFC language program code

CODESYS provides the possibility to use different programming languages within a single project: one program can be written in ***ST*** language, another in ***CFC***, etc.

5.2 Variables

A program is a set of operations performed on data stored in the controller's memory, e.g.:

- temperature control and alarm on exceeding emergency limits
- cyclical rotation of pumps based on operating time
- lighting control depending on the time of day and motion sensor activity

etc.

To simplify memory management, data is represented as variables – named objects with specified characteristics (e.g., data type). A variable must be declared. In CODESYS, variables are declared in the

upper area of the programming language editor. We already did this when creating our first project (Sect. 3.1).

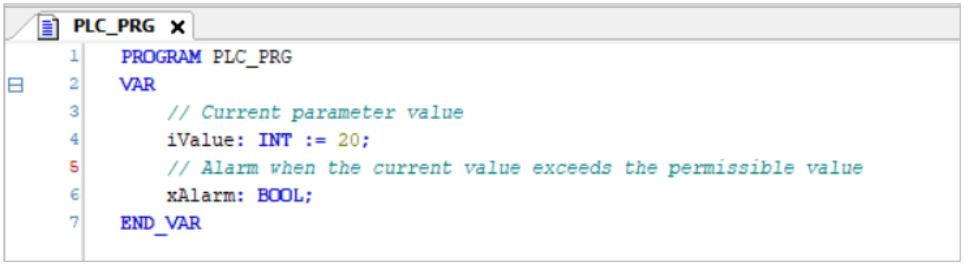


Fig. 5.2.1 Variable declaration

First, the variable name is specified, then its type, and optionally its initial value. The variable name and type are separated by a colon. The type and initial value are separated by an assignment operator '`:=`'. The declaration is terminated by a semicolon.

As in any part of the program, explanatory comments can be inserted in the declaration area. CODESYS supports single-line comments beginning with the double slash '`//`' and multi-line comments enclosed between symbols '`(*`' and '`*)`'.

Example 5.2.1

```
PROGRAM PLC_PRG
VAR
      // single-line comment
(*
   multi-line comment
   multi-line comment
*)
iVar: INT;
END_VAR
```

The names of variables (and all other CODESYS objects) are subject to the following rules:

- The name must not contain spaces or special characters (e.g., '`!`', '`@`', etc.). The exception is the underscore symbol '`_`'.
- The name must not begin with a number.
- The name must not contain several underscore symbols '`_`' in a row: the name '`i__Test`' is not allowed, but the name '`_i_Test`' is allowed.
- The case of object name is not considered: '`iTest`' and '`iTEST`' will be interpreted as the same name.
- There are no restrictions on the name length.
- The name must not coincide with one of the reserved keywords (e.g., `VAR`, `INT`, etc.).
- It is recommended to use ***Hungarian notation*** (CODESYS variant) and ***lowerCamelCase*** style for variable names.

By default, CODESYS only supports Latin characters in variable names. You can enable support for other alphabets (Greek, Cyrillic, etc.) in the ***Project*** menu: ***Project Settings*** > ***Compile options*** > check the option ***Allow unicode characters for identifiers***.

5.3 Data types

The data type determines the size of the variable, the range of its possible values, and the set of permissible operations.

5.3.1 BOOL

- acceptable values: **FALSE** (0) or **TRUE** (1)
- size: 1 byte (special implementation of this type in CODESYS)
- used to describe binary events and commands (e.g., signal from a discrete sensor, absence/presence of an alarm, motor start/stop, etc.)
- support of logical operations (**AND**, **OR**, **XOR**, **NOT** etc.)

Example 5.3.1

```
PROGRAM PLC_PRG
VAR
	xVar1: BOOL;
	xVar2: BOOL;
	xVar3: BOOL;
END_VAR

// code area

xVar1 := TRUE;

// xVar3 is TRUE, if xVar1 and xVar2 are TRUE
// if xVar1 or xVar2 are FALSE, than xVar3 also is FALSE

xVar3 := xVar1 AND xVar2;
```

5.3.2 Integer types

- can be signed or unsigned
- size and range are defined for the specific type (Tab. 5.3.1)
- support arithmetic and comparison operations
- support bitwise access

Table 5.3.1 Characteristics of integer data types

Type	Size (Byte)	Range
USINT / BYTE	1	0…255
SINT	1	-128…127
UINT / WORD	2	0…65535
INT	2	-32768…32767
UDINT / DWORD	4	$0…2^{32}-1$
DINT	4	$-2^{31}…2^{31}-1$
ULINT / LWORD	8	$0…2^{64}-1$
LINT	8	$-2^{63}…2^{63}-1$

The **USINT** / **BYTE** types and other similar pairs in the CODESYS implementation are identical. The double designation is retained for compatibility with IEC 61131-3, in which only arithmetic operations are defined for the **USINT** / **UINT** / **UDINT** / **ULINT** types, and only bitwise access is defined for the **BYTE** / **WORD** / **DWORD** / **LWORD** types. In CODESYS, arithmetic operations and bitwise access are possible for any of these types.

When performing arithmetic operations, an overflow effect is possible. E.g., if a variable of type `USINT` has a value of '`255`' (the upper limit of its range) and we add '`1`' to it, the variable will receive a value of '`0`' (i.e., its value will become equal to the lower limit of its range).

Division by '`0`' will cause an exception in the controller application and the application will be stopped.

Example 5.3.2

```
PROGRAM PLC_PRG
VAR
	iVar1: INT;
	siVar1: SINT;
	siVar2: SINT;
	wVar: WORD;
	dwVar: DWORD;
END_VAR

// code area

iVar1 := 123 + 2 * siVar1;
siVar1 := 9;

// siVar2 will take a value of '-120' due to the overflow effect
siVar2 := 127 + siVar1;

// Values can be specified in hexadecimal notation (HEX).
wVar := 16#00FF;

// bitwise access to integer variables
// a variable cannot be used as a bit number
dwVar.0 := TRUE;
dwVar.2 := TRUE;
```

5.3.3 Floating-point types

- implementation complies with IEEE 754-2019
- size, range, and precision (number of significant digits after the decimal point) are defined for the specific type
- support arithmetic and comparison operations

Table 5.3.2 Characteristics of floating-point data types

Type	Size (Byte)	Range	Precision
REAL	4	± (≈ 10^{-38}…≈ 10^{38})	7…8 digits after decimal point
LREAL	8	± (≈ -10^{-308}…≈ 10^{308})	15…17 digits after decimal point

A floating-point number always represents an approximation of the "true" value. Arithmetic operations with such numbers may result in rounding errors (Example 5.3.3), so it is incorrect to check them for strict equality.

Example 5.3.3

```
PROGRAM PLC_PRG
VAR
	rVar1: REAL;
	rVar2: REAL;
	rVar3: REAL;

	iVar1: INT := 10;
END_VAR
```

```
// code area

rVar1 := 0.6;

// due to rounding error, rVar1 is set to '0.700000048'
rVar1 := rVar1 + 0.1;

// rVar2 is set to '3.0' because iVar1 and '3' are integers
// The result of dividing an integer by an integer is an integer.
rVar2 := iVar1 / 3;

(* To obtain the "correct" result, one of the operands must be of the floating-point
type. Therefore, you need to do the following: *)
rVar3 := iVar1 / 3.0;

// or like this (see information about type conversion operators in section 5.3.6):
rVar3 := TO_REAL (iVar1) / 3;
```

5.3.4 Strings

- Two types of strings are supported: **STRING** (ASCII-like encoding) and **WSTRING** (UTF-16 encoding) (Tab. 5.3.3).
- **STRING** type values are enclosed in single quotation marks (**'test'**), **WSTRING** type values are enclosed in double quotation marks (**"test"**).
- The maximum number of characters in a string is specified when it is declared, and memory is allocated for a string of the maximum length.
- The number of characters in a string is not limited, but standard libraries (***Standard*** and ***Standard64***) can only proceed with strings with a maximum length of 255 characters.
- Strings are null-terminated (as in C). Memory for **NULL** is allocated automatically, the user does not need to think about it.
- Strings support escape sequences (line breaks, etc.). See examples below.
- Strings support indexed access like arrays, with indexing starting at zero.
- Strings are processed (concatenation, length determination, etc.) using functions from ***Standard*** library for **STRING** and ***Standard64*** for **WSTRING**.
- If it is necessary to display text in a non-Latin encoding in the visualization, use only the **WSTRING** type.

Table 5.3.3 Characteristics of string data types

Characteristic	STRING	WSTRING
Encoding	8-bit, ASCII-based	UCS-2 (subtype of UTF-16)
Syntax	'test'	"test"
Character size (byte)	1	2
Variable size	Maximum string length + 1	(Maximum string length + 1) *2
Library	Standard	Standard64

Example 5.3.4

```
PROGRAM PLC_PRG
VAR
	// string with a maximum length of 20 characters
	// variable size is 21 bytes (20 characters + 1 byte for NULL)
```

```
        sVar1: STRING(20) := 'test';

        // 80 characters are allocated for the string by default
        // so the variable size is 81 bytes
        sVar2: STRING;

        // variable size is 42 bytes (2 bytes for NULL)
        // escape sequence characters are enclosed in $ characters
        // below is an example of line wrapping after 'hello'
        wsVar1: WSTRING(20) := "hello$n$rworld";

        // 80 characters are allocated for the string by default
        // so the variable size is 162 bytes
        wsVar2: WSTRING;

END_VAR

// code area

sVar2 := 'temperature';
wsVar2 := "pressure";

// example of index access to a string
// sVar1 takes the value 'rest', because 16#72 is the ASCII code for the character 'r'.
sVar1[0] := 16#72;
```

5.3.5 Date and time

Date and time types:

- used to specify timestamps and time intervals
- size and range are defined for the specific type
- for **DT** / **LDT** / **DATE** / **LDATE** types, the time '**00:00:00 01.01.1970**' (as in Unix time) is used as the reference point
- specific syntax for specifying values (Example 5.3.5)
- support certain arithmetic operations (Sect. 5.6.1) and comparison operations (Sect. 5.6.3)

Table 5.3.3 Characteristics of date and time data types

Type	Description	Size (Byte)	Range	Precision
TIME	Time interval	4	T#0d0h0m0s0ms…T#49d17h2m47s295ms	ms
LTIME		8	LTIME#0d0h0m0s0ms0us0ns… LTIME#213503d23h34m33s709ms551us615ns	ns
DT	Date and time	4	DT#1970-1-1-0:0:0…DT#2106-02-07-06:28:15	s
LDT		8	LDT#1970-1-1-0:0:0… LDT#2554-7-21-23:59:59.99999999	ns
DATE	Date	4	D#1970-01-01…D#2106-02-07	s
LDATE		8	LD#1970-01-01…LD#2554-7-21	ns
TOD	Time of date	4	TOD#0:0:0…TOD#23:59:59.999	ms
LTOD		8	LTOD#0:0:0…LTOD#23:59:59.999999999	ns

Example 5.3.5

```
PROGRAM PLC_PRG
VAR
        dtVar: DT;
        tVar: TIME;
```

```
	// number of hours as an integer
	uiHourCount: UINT := 8;
END_VAR

// code area

dtVar := DT#2022-7-22-11:55:11;

// tVar takes the value 'T#8h'
tVar2 := T#1H * uiHourCount;

// Possible arithmetic operations:
// TIME + TIME = TIME
// TIME - TIME = TIME
// TIME * integer = TIME

// DT + TIME = DT
// DT - TIME = DT
// DT - DT = TIME

// TOD + TIME = TOD
// TOD - TIME = TOD

// the same applies for LTIME / LDT / LTOD
```

5.3.6 Type conversion

All types discussed in the previous sections can be converted into each other. For example, if you plan to display an integer value with corresponding units of measurement (with the ability to change these units) in the visualization, you will first need to convert this integer into a string.

Conversion operators look as follows:

TO_<conversion result type>

A complete list of available conversion operators is presented here.

Example 5.3.6

```
PROGRAM PLC_PRG
VAR
	xVar: BOOL;
	iVar1: INT;
	iVar2: INT;
	uiVar1: UINT;
	uiVar2: UINT;
	uiVar3: UINT;
	usiVar: USINT;
	rVar: REAL;
	sVar: STRING;
	dtVar: DT;
	udiVar: UDINT;
END_VAR

// code area

xVar := TRUE;

// iVar1 takes '1'
iVar1 := TO_INT(xVar);

// sVar takes 'TRUE'
sVar := TO_STRING(xVar);
```

```
iVar2 := -123;

// uiVar takes '65413'
// Since the UINT type is unsigned:
// 65535 (upper limit of UINT) + 1 - 123 = 65413
uiVar1 := TO_UINT(iVar2);

// sVar takes '-123'
sVar := TO_STRING(iVar2);

uiVar2 := 513;

// The upper limit of USINT is '255', therefore, usiVar takes '1' due to overflow.
usiVar := TO_USINT(uiVar2);

rVar := 11.22;
// uiVar takes '11'
uiVar3 := TO_UINT(rVar);

dtVar := DT#2022-7-22-11:55:11;
(* The variable udiVar is the number of seconds that have elapsed between 1970-1-1-0:0:0
and 2022-7-22-11:55:11. *)
udiVar := TO_UDINT(dtVar);
```

NOTE

When converting **STRING** to **WSTRING** and **WSTRING** to **STRING**, encodings are not converted. Therefore, if the string contained non-Latin characters (e.g., Cyrillic), after conversion to another string type, it would contain mojibake.

5.3.7 Arrays

In the previous sections, we only discussed elementary data types. A variable of such a type stores only one value. However, even in medium-sized projects, the number of signals to be processed is rarely less than a few hundred. In addition, groups of values often need to be processed in the same way (e.g., temperatures of identical control system nodes).

Arrays are used for convenient work with groups of objects of the same type. A usual task: an alarm signal must be generated if the temperature exceeds the permissible limit. Suppose our system has 100 points at which we monitor the temperature. Without using arrays, the code might look like this:

Example 5.3.7

```
PROGRAM PLC_PRG
VAR
	rTemp_1: REAL;
	rTemp_2: REAL;

	// declaration of another 97 variables...

	rTemp_100: REAL;
	rTempHighAlarmLimit_1: REAL := 11.22;
	rTempHighAlarmLimit_2: REAL := 22.33;

	// declaration of another 97 variables...

	rTempHighAlarmLimit_100: REAL := 33.44;
```

```
	xAlarm_1: BOOL;
	xAlarm_2: BOOL;

	// 97 more lines, differing only in their indexes...

	xAlarm_100: BOOL;
END_VAR

// code area

xAlarm_1 := rTemp_1 > rTempHighAlarmLimit_1;
xAlarm_2 := rTemp_2 > rTempHighAlarmLimit_2;

// 97 more lines, differing only in their indexes...

xAlarm_100 := rTemp_100 > rTempHighAlarmLimit_100;
```

There are many lines of code (a total of more than 300 lines of code), but all these lines differ only slightly, which makes it very easy to make a mistake when copying and renaming.

Using arrays, you can do the following:

Example 5.3.8

```
PROGRAM PLC_PRG
VAR
	arTemp: ARRAY [1..100] OF REAL;
	arTempHighAlarmLimit: ARRAY [1..100] OF REAL := [11.22, 22.33, 97(32.23), 33.44];
	axAlarm: ARRAY [1..100] OF BOOL;
	i: INT;
END_VAR

// code area
(*
FOR - loop operator. It is discussed in Sect. 5.7.3.
[i] - accessing the i-th element of the array
The code below performs a pairwise comparison (checking for greater than) of array
elemnts ‘arTemp’ and ‘arTempHighAlarmLimit’.
The comparison results are stored in the ‘axAlarm array’.
*)
FOR i := 1 TO 100 DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

7 lines of code versus 300 – that's a very significant difference.

We will discuss the **FOR**-loop operator in detail in Sect. 5.7.3.

When declaring an array, its lower and upper bounds are specified, which determine the number of elements in the array. Arrays in IEC 61131-3 are static, and the number of their elements must be known at the variable declaration stage. It is not possible to declare a dynamic array.

Arrays are often numbered starting from 0, but this is not mandatory – the bounds of an array can be any integers (including negative ones), and the only obvious restriction is that the upper bound of the array must be greater than the lower one.

When declaring an array, you can specify the initial values of its elements. In the example above, this is done for acceptable temperature limits. You can initialize several consecutively located array elements with the same value. In the example above, the element with index 1 takes the initial value ‘**11.22**’, the element with index 2 takes the initial value ‘**22.33**’, and the following 97 elements (i.e., with indexes 3...99) take the value ‘**32.23**’.

To access an array element, you must specify its index in square brackets.

Example 5.3.9

```
// write the value to the 5th element of the array
arTemp[5] := 55.44;
```

The index can be either a number or a variable. In the case of variable indexes, the execution system does not check whether the current value of the variable belongs to the array bounds range, i.e., such code will be executed without any warnings from CODESYS:

Example 5.3.10

```
i := 1000;
// writing a value to a non-existent array element
arTemp[i] := 55.44;
```

In fact, a certain block of memory in our application will be overwritten. If this memory is used by the execution system, we may get an exception with the **AccessViolation** code, which is a sign of a memory segmentation error. Another possible scenario is that our application's memory will be overwritten, and as a result, one or more other variables will receive incorrect values, which will disrupt the program's logic. Therefore, it is necessary to carefully control the index values used, when accessing an array.

Array elements are stored in the application memory sequentially and contiguously.

In Example 5.3.8, we considered a one-dimensional array. Besides these, there are also multidimensional arrays, but they are not used very often in real projects, so we will not consider them in this “First Start”. For the same reason, we will not consider variable-length arrays. The latter are used in functions and functional blocks that must be able to process arrays of different sizes.

Back to Example 5.3.8:

Example 5.3.11

```
PROGRAM PLC_PRG
VAR
	arTemp: ARRAY [1..100] OF REAL;
	arTempHighAlarmLimit: ARRAY [1..100] OF REAL := [11.22, 22.33, 97(32.23), 33.44];
	axAlarm: ARRAY [1..100] OF BOOL;
	i: INT;
END_VAR

// code area

FOR i := 1 TO 100 DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

In this example, the same number describing the upper limit of the array ‘**100**’ is used four times. If we need to change the number of elements in our array, we must remember to do this in four different places. To avoid this, we can declare a constant (Sect. 5.5.1):

Example 5.3.12

```
PROGRAM PLC_PRG
VAR
	arTemp:                    ARRAY [1..c_iMaxTempPointCount] OF REAL;
	arTempHighAlarmLimit:      ARRAY [1..c_iMaxTempPointCount] OF REAL := [11.22,
		22.33, 97(32.23), 33.44];
	axAlarm:                   ARRAY [1..c_iMaxTempPointCount] OF BOOL;
	i: INT;
END_VAR
```

```
VAR_CONSTANT
	c_iMaxTempPointCount: INT := 100;
END_VAR

// code area

FOR i := 1 TO c_iMaxTempPointCount DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

Now, if the number of temperature measuring points changes, we only need to edit one constant.

We could also declare the lower bound of the array as a constant, but this approach is not commonly used – arrays are almost always numbered from '**0**' or '**1**'.

Instead of a constant, you can use a special operator to define the bounds of the array. Its second argument represents the number of measuring points in the array. Since this operator returns the value of type **DINT**, we need to change the type of the loop counter variable.

Example 5.3.13

```
PROGRAM PLC_PRG
VAR
	// array declaration, as in the previous example
	//...
	i: DINT;
END_VAR

// code area

FOR i := LOWER_BOUND(arTemp, 1) TO UPPR_BOUND(arTemp, 1) DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

Strings are also arrays and support index access. You can work with a variable of type **STRING** as with an array of type **ARRAY [...] OF BYTE**, and with a variable of type **WSTRING** as with an array of type **ARRAY [...] OF WORD**.

Summary of experiments with arrays:

- Arrays are useful for representing groups of objects of the same type.
- Arrays support index access.
- Accessing a non-existent array element is not controlled by CODESYS and may lead to errors. Therefore, the developer must ensure that the array index value falls within its bounds.
- Array elements are stored in the application memory sequentially and contiguously.
- When declaring arrays, it is preferable to use the constant as the upper bound.
- The **LOWER_BOUND** and **UPPER_BOUND** operators can be used to define the array length.
- Strings, like arrays, support index access.

5.3.8 User data types

All of the types discussed above are "built-in", i.g., declaring variables of these types does not require any preliminary actions. However, in some cases, it is useful to create a user type designed for a specific task.

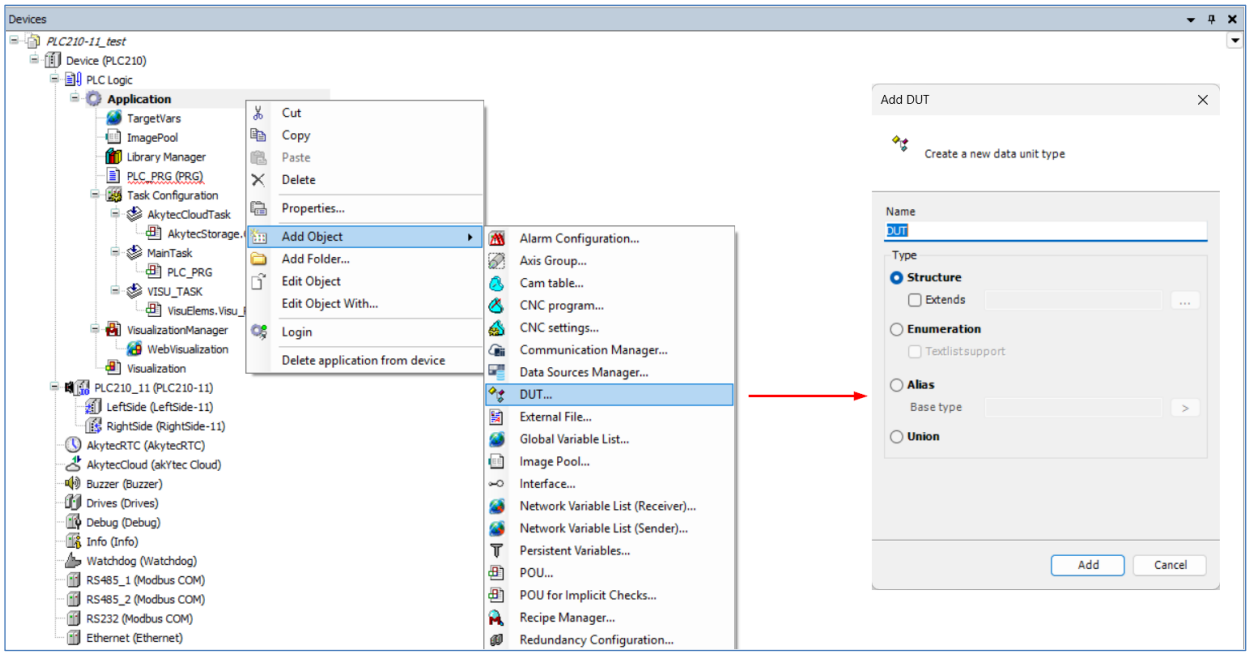


Fig. 5.3.1 Add new user data type

To create an object of such type, use the ***Add Object*** command from the ***Application*** node context menu and select the ***DUT*** object (Data Unit Type). In the new window, select the type to edit. There are 4 user types: structures (Sect. 5.3.9), unions (Sect. 5.3.10), enumerations (Sect. 5.3.11) and aliases (Sect. 5.3.12).

After creating a user type, you can use it when declaring variables.

5.3.9 Structures

Structures enable grouping variables of different types within a single object. This distinguishes them from arrays (Sect. 5.3.7), which are used to group variables of the same type.

We will create a structure that describes a single temperature measuring point.

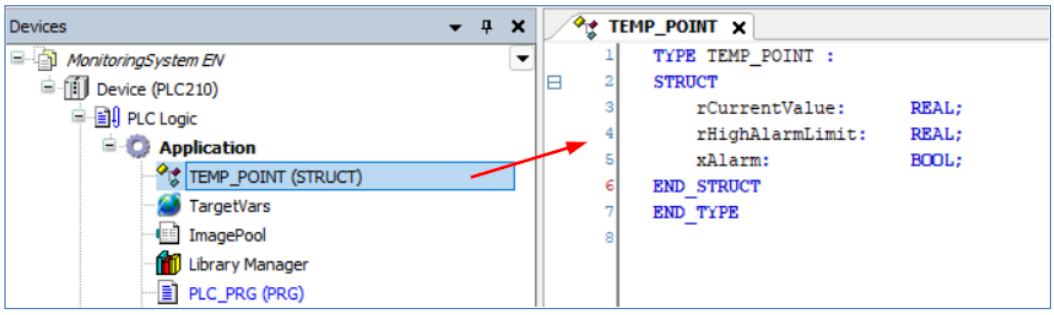


Fig. 5.3.2 Creating a structure

Example 5.3.14

```
PROGRAM PLC_PRG
VAR
	stTempPoint: TEMP_POINT := (rHighAlarmLimit := 22.33);
END_VAR

// code area

stTempPoint.xAlarm := stTempPoint.rCurrentValue > stTempPoint.rHighAlarmLimit;
```

The example above shows the syntax for initializing structures and accessing their variables in the program code via the number of the measuring point.

We recall Example 5.3.12…

Example 5.3.15

```
PROGRAM PLC_PRG
VAR
	arTemp:                    ARRAY [1..c_iMaxTempPointCount] OF REAL;
	arTempHighAlarmLimit:      ARRAY [1..c_iMaxTempPointCount] OF REAL := [11.22,
22.33, 97(32.23), 33.44];
	axAlarm:                   ARRAY [1..c_iMaxTempPointCount] OF BOOL;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iMaxTempPointCount: INT := 100;
END_VAR

// code area

FOR i := 1 TO c_iMaxTempPointCount DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

… and rewrite it using the structure:

Example 5.3.16

```
PROGRAM PLC_PRG
VAR
	astTempPoint: ARRAY [1.. c_iMaxTempPointCount] OF TEMP_POINT;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iMaxTempPointCount: INT := 100;
END_VAR

// code area

FOR i := 1 TO c_iMaxTempPointCount DO
	astTempPoint[i].xAlarm := astTempPoint[i].rCurrentValue >
		astTempPoint[i].rHighAlarmLimit;
END_FOR
```

The code became more compact – instead of three arrays, we used a single array of structures.

In the last example, we did not specify the initial bounds – the initialization syntax for an array of structures is quite complex (if you are interested, see the relevant article in the CODESYS online help). In such cases, initial values are often assigned directly in the program code:

Example 5.3.17

```
PROGRAM PLC_PRG
VAR
	astTempPoint: ARRAY [1.. c_iMaxTempPointCount] OF TEMP_POINT;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iMaxTempPointCount: INT := 100;
END_VAR

// code area
```

```
IF NOT(xInit) THEN
	astTempPoint[1].rHighAlarmLimit := 11.22;
	astTempPoint[2].rHighAlarmLimit := 22.33;
	// ...
	astTempPoint[100].rHighAlarmLimit := 33.44;
	xInit := TRUE;
END_IF
```

This code is executed once on startup of the application: when the application starts, the variable **xInit** has the value **FALSE**, which causes the **IF** statement (Sect. 5.7.1) to be executed. Inside the **IF** statement, the necessary elements of the structure array are initialized, after which the variable **xInit** is assigned the value **TRUE**, as a result, this block of code will no longer be executed.

The elements of structures are arranged in the memory sequentially, but not necessarily contiguously. Because of this, the size of the structure may not be equal to the sum of the sizes of its elements. Let's go back to Fig. 5.3.2. A variable of type **REAL** occupies 4 bytes of memory, and a variable of type **BOOL** occupies 1 byte (Sect. 5.3.1, 5.3.3). Thus, the total size of the elements of this structure is 9 bytes, but the actual size of this structure is 12 bytes due to memory alignment – the compiler optimizes the allocation of variables in memory, leaving “gaps” between them to speed up access to them by the processor.

In some cases, it is necessary for structure elements to be placed close to each other in memory, without gaps. In CODESYS, a special attribute called ***pack_mode*** is used for this purpose. More information about this attribute you can read here.

5.3.10 Unions

Unions are similar to structures with one difference: all their elements are located in the same memory area. Each element of a union represents a different value of the same parameter. Suppose we read a value of **REAL** type from a sensor. The error code is encoded in the most significant byte of the value. The error could be a sensor break, an ADC error, etc. To determine the presence and nature of the error, we need to compare the value of the most significant byte with the known error codes. For this purpose, we will create a union:

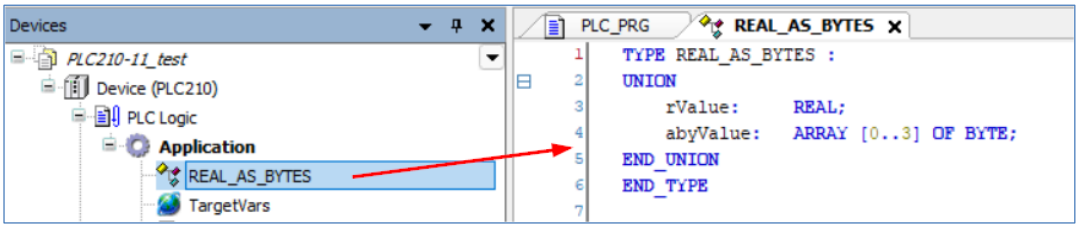


Fig. 5.3.3 Creating Union

Declaration of the variable of the type **REAL_AS_BYTES**:

Example 5.3.18

```
PROGRAM PLC_PRG
VAR
	uTemp: REAL_AS_BYTES;
	xSensorFailed: BOOL;
END_VAR

// code area

	// For debugging purposes, we will write some value to the REAL variable of the
structure.
```

```
uTemp.rValue := 11.22;
      // Checking the most significant byte of the value
      // 240 is error code “sensor break”
      // In our case, there is no error, and xSensorFailed takes FALSE.
xSensorFailed := (uTemp.abyValue[3] = 240);
```

As in the case of structures, access to union elements is performed via the number of the measuring point. Since union elements are located in the same memory area, to check the most significant byte of the **abyValue** array, we must check the most significant byte of the **rValue** variable of **REAL** type.

5.3.11 Enumerations

Enumerations are used to assign symbolic names to a set of integer values. This improves the readability of the program code. We recall Example 5.3.18:

Example 5.3.19

```
PROGRAM PLC_PRG
VAR
      uTemp: REAL_AS_BYTES;
      xSensorFailed: BOOL;
END_VAR

// code area

      // For debugging purposes, we will write some value to the
      // REAL variable of the structure.
uTemp.rValue := 11.22;
      // Checking the most significant byte of the value
      // 240 is error code “sensor break”
xSensorFailed := (uTemp.abyValue[3] = 240);
```

The code uses the number 240, which refers to a specific sensor error code. The meaning of the number is explained in the comment above. However, in a real project, sensor errors may occur in many parts of the program, and we would have to duplicate this comment in each of them. If the error code changes in the new sensor firmware, we will have to change it in all parts of the code where this code appears, as well as edit all comments. It is quite possible that we will forget to make a correction somewhere, and the program will not work correctly. In addition, comments can be lost when transferring code to another project, etc.

Therefore, it is more convenient to use enumerations. We will create an enumeration with sensor error codes.

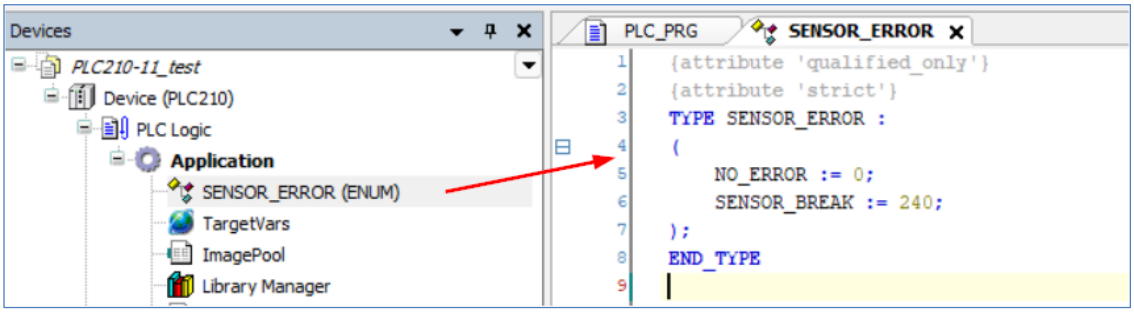


Fig. 5.3.4 Creating Enumeration

We have declared the normal sensor status code and one error code in the enumeration; you can add as many elements to the enumeration as you need.

Now you can change the program code as follows:

Example 5.3.20

```
PROGRAM PLC_PRG
VAR
	uTemp: REAL_AS_BYTES;
	xSensorFailed: BOOL;
END_VAR

// code area

	// For debugging purposes, we will write some value to the
	// REAL variable of the structure.
uTemp.rValue := 11.22;

	// Checking the most significant byte of the value
xSensorFailed := (uTemp.abyValue[3] = SENSOR_ERROR.SENSOR_BREAK);
```

Usually, when creating an enumeration, a variable of the corresponding type is declared. This is convenient in cases where the variable contains an error code, an equipment operating mode, a technological operation currently being performed, etc. Consider the following example:

Example 5.3.21

```
PROGRAM PLC_PRG
VAR
	rValue: REAL;
	eSensorError: SENSOR_ERROR;
END_VAR
VAR CONSTANT
	rValueLowLimit: REAL := 4.0;
	rValueHighLimit: REAL := 20.0;
END_VAR

// code area

IF rValue < rValueLowLimit OR rValue > rValueHighLimit THEN
	eSensorError := SENSOR_ERROR.SENSOR_BREAK;
END_IF
```

By default, an enumeration occupies 2 bytes of memory. If necessary (for example, if your error codes exceed 65535 and therefore cannot be represented by a 2-byte integer variable), you can specify the enumeration type when creating it.

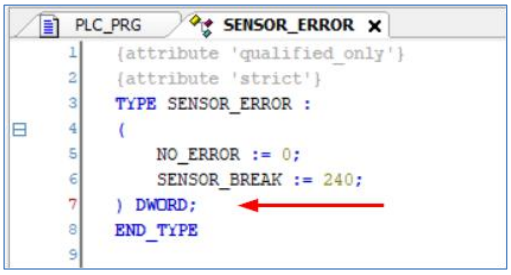


Fig. 5.3.5 Specifying the data type of enumeration

5.3.12 Aliases

Aliases enable you to assign an additional name to one of the existing data types. We will consider cases where this feature may be useful. Suppose that our program actively works with strings. It was determined in the design phase that the length of strings cannot exceed 100 characters. Accordingly, our program declares a set of string variables with their permissible length specified:

Example 5.3.22

```
PROGRAM PLC_PRG
VAR
	sUserId: STRING(100);
	sVendorId: STRING(100);
	sCategoryId: STRING(100);
	sProductId: STRING(100);
	// ...more variables of type STRING(100)...
END_VAR
```

Suppose that the requirements changed during the design process, and now strings must contain up to 120 characters. You would have to edit the declaration of each string variable, changing its maximum length. Alternatively, you could create an alias:

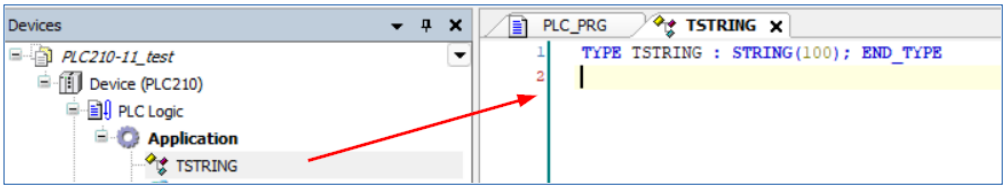


Fig. 5.3.6 Creating alias

Now we can use this alias when declaring our string variables:

Example 5.3.23

```
PROGRAM PLC_PRG
VAR
	sUserId: TSTRING;
	sVendorId: TSTRING;
	sCategoryId: TSTRING;
	sProductId: TSTRING;
END_VAR
```

Changes in requirements are no longer a concern, as we only need to make an update in one place in the project – the alias.

Another option would be to use a constant as the string length.

5.3.13 Special data types

There are also several special data types that are sometimes used by library developers, but rarely by application developers. Brief overview:

- BIT – similar to the **BOOL** type but occupies only 1 bit of memory (**BOOL** occupies 1 byte). Variables of this type can only be declared in structures, unions, and function blocks.
- POINTER – a data type used to work with variable addresses. Operations with pointers enable the creation of compact and efficient code (for example, for copying large amounts of data), but in the case of programming errors, the controller may malfunction (including application shutdown). Therefore, beginner developers are advised to avoid using pointers.
- REFERENCE – a safer but less functional alternative to pointers. Typically used in combination with object-oriented programming techniques.
- UXINT, XINT, XWORD – platform-dependent data types, interpreted as **UDINT** / **DINT** / **DWORD** for 32-bit applications and as **ULINT** / **LINT** / **LWORD** for 64-bit applications. Used by developers to create hardware-independent libraries.

- ANY – a data type that can only be used when declaring input variables (`VAR_INPUT`) (Sect. 5.4.3). Any type of variable can be assigned to an `ANY` input. This makes it possible to create universal functions and function blocks that can process different data types.
- __VECTOR – a type of array that can be processed faster than regular arrays by some processors using special instructions.

5.4 Variable declaration areas

5.4.1 Types of variable declaration areas

In the previous sections, we declared variables in the PLC_PRG program between the keywords `VAR` and `END_VAR`.

Example 5.4.1

```
PROGRAM PLC_PRG
VAR
	iVar1: INT;
	siVar1: SINT;
	siVar2: SINT;
	wVar: WORD;
	dwVar: DWORD;
END_VAR
```

These keywords define the area of variable declarations. The area imposes restrictions on access to variables from this (where the variable is declared) or other program objects. Each area begins with a unique keyword and ends with the keyword `END_VAR`. The following sections describe the main variable declaration areas. Specific examples will be discussed in Sect. 5.8.1, 5.8.2.

5.4.2 Local variables (VAR)

Local (internal) variables are only available for reading and writing within the program object in which they are declared. Technically, they can be read from other program objects, but this is not recommended because it violates the logic of data transfer within the application. If you need variables to be readable from other objects, make them output variables (`VAR_OUTPUT`).

5.4.3 Input variables (VAR_INPUT)

Input variables are read-only within the program object in which they are declared and are available for reading and writing by other program objects that call this object. Input variables enable a program object to receive data from other program objects.

5.4.4 Output variables (VAR_OUTPUT)

Output variables are available for reading and writing within the program object in which they are declared and are available for reading by other program objects that call this object. Output variables enable a program object to transfer data to other program objects.

5.4.5 Input-output variables (VAR_IN_OUT)

Input-output variables are available for reading and writing both within the program object in which they are declared and from other program objects that call this object. In most cases, using this area offers no significant advantage, and instead complicates understanding the data transfer logic within the program. It is not recommended to use this area in program objects, but in some cases, it is acceptable to use it (e.g., in visualization objects) (Sect. 6.6.1).

5.4.6 Global variables (VAR_GLOBAL)

Global variables are available for reading and writing within any program object of the application. Contrary to other areas, global variables are declared in a separate project tree object:

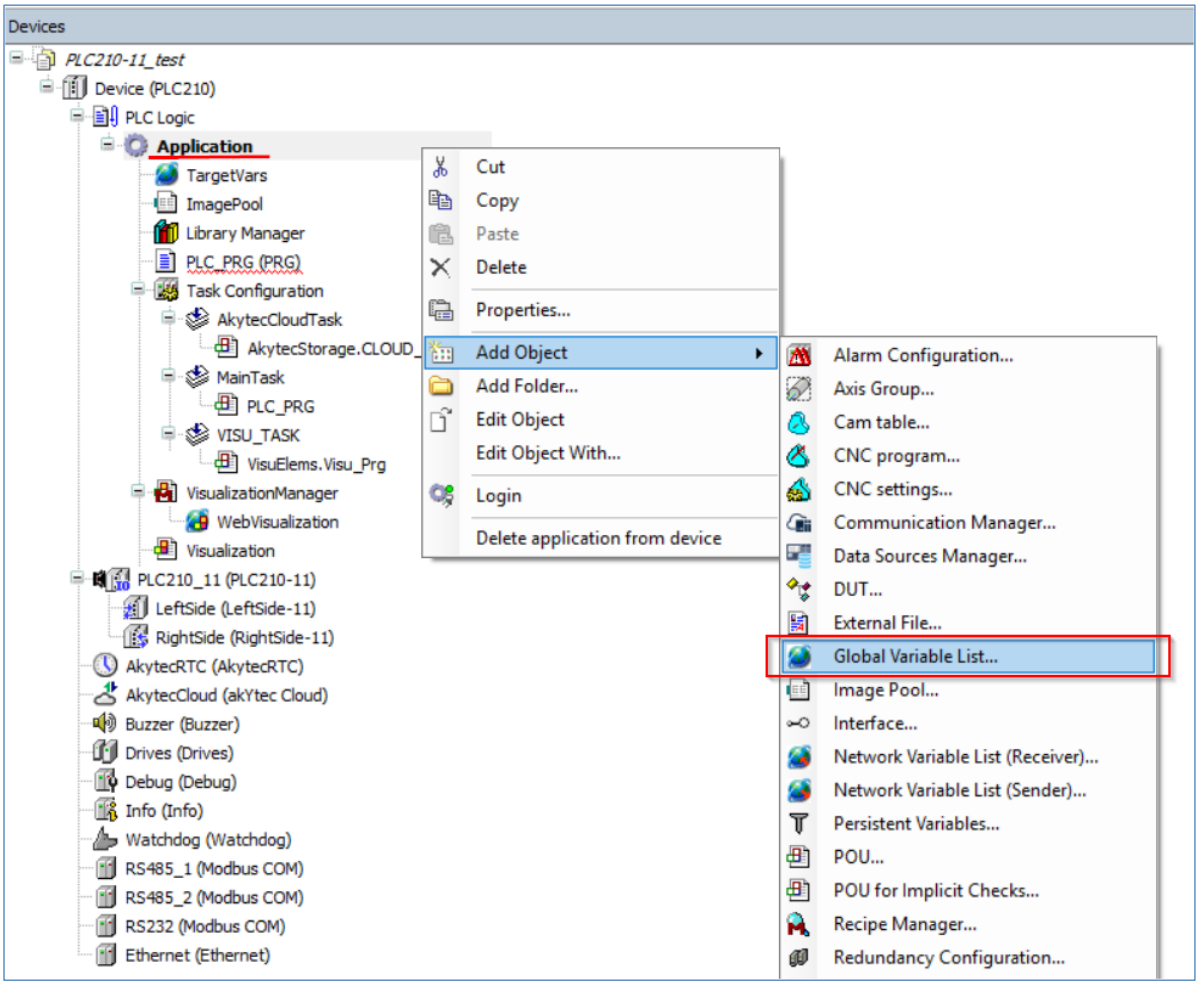


Fig. 5.4.1 Creating Global Variable List

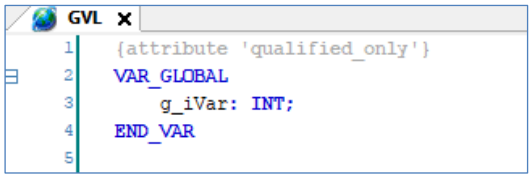


Fig. 5.4.2 Global variable declaration

To access a global variable, the name of its list must be specified:

Example 5.4.2

```
PROGRAM PLC_PRG
VAR
END_VAR

// code area

GVL.g_iVar := 11;
```

Global variables should be used consciously. The device template (Sect. 3.1) includes a list of global variables ***TargetVars***, whose variables are linked to the channels of the target system nodes (***AkytecRTC***, ***Buzzer***, etc.). The variables in this list can be considered system variables. E.g., system time may be used in different parts of the application, so it is necessary to make it global.

However, ill-considered use of global variables complicates the logic of data transfer within the application and can lead to errors (especially when attempting to write to a single variable from multiple program objects). Therefore, only use this area when it is absolutely necessary.

5.4.7 Other declaration areas

CODESYS offers several additional variable declaration areas, such as `VAR_STAT`, `VAR_EXTERNAL`, and `VAR_CONFIG`. Some of them are only present to maintain compatibility with IEC 61131-3; some areas are only used in specific cases that are not relevant for the “First Start”.

In most projects, three areas are sufficient: local, input, and output variables.

5.5 Declaration area modifiers

Variable declaration area modifiers are used to give areas specific properties. There are two types of modifiers in CODESYS: for constants and non-volatile variables.

5.5.1 CONSTANT

Some values in the application must always remain constant. For example, if the control system uses eight pumps, it is unlikely that this number will change during normal system operation. If the number of pumps changes during system modernization, the project will most likely still need to be revised.

Instead of using specific numbers in the code, it is better to declare constants. First, this will simplify code maintenance, i.e., if the number changes, you will have to search for all instances in the program where it was used and edit them. In the case of a constant, it will be enough to change the constant declaration line. Second, constants are write-protected, i.e., even if the programmer makes an error in the code, the constant cannot be overwritten from the program code.

To declare constants, you have to create a variable declaration area with the `CONSTANT` modifier added.

We recall Example 5.3.12:

Example 5.5.1

```
PROGRAM PLC_PRG
VAR
	arTemp:                    ARRAY [1..c_iMaxTempPointCount] OF REAL;
	arTempHighAlarmLimit:      ARRAY [1..c_iMaxTempPointCount] OF REAL := [11.22,
22.33, 97(32.23), 33.44];
	axAlarm:                   ARRAY [1..c_iMaxTempPointCount] OF BOOL;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iMaxTempPointCount: INT := 100;
END_VAR

// code area

FOR i := 1 TO c_iMaxTempPointCount DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

Constants can be declared local or global.

Input constants (`VAR_INPUT CONSTANT`) can be used in CFC. They are no different from regular constants in CFC in terms of processing, but their values can be changed in the program editor using a special menu:

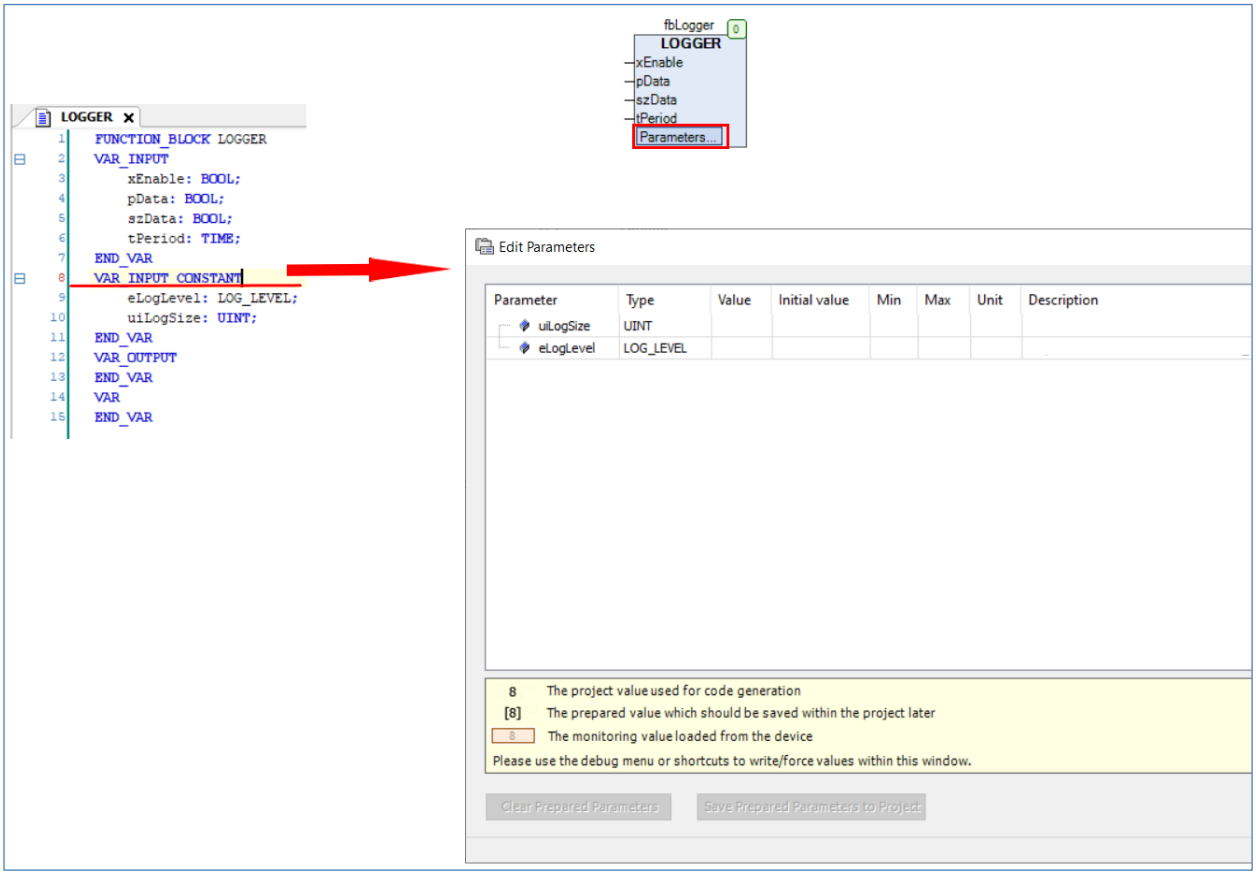


Fig. 5.5.1 Changing input constants in CFC editor

Processing `VAR_INPUT CONSTANT` in other programming languages is no different from processing regular input variables.

Furthermore, the `CONSTANT` modifier can be added to the `VAR_IN_OUT` area. Since we do not recommend using this area, we will not focus on this issue. However, you can refer to the relevant article in the CODESYS online help.

5.5.2 RETAIN and PERSISTENT

After the controller is restarted, the variables of all areas discussed in Sect. 5.4 assume the default values specified in their declarations. If they are not specified, the system default values are used (e.g., ***0*** for numeric types, an empty string for strings, etc.).

It is unacceptable in some situations, e.g., technological process settings may be changed by the operator during operation and must be retained after the controller is restarted. In this case, such variables must be declared in the non-volatile variable area.

There are two non-volatile modifiers: `RETAIN` and `PERSISTENT`. They differ in their behaviors when a new project is loaded into the controller:

- `RETAIN` variables are re-initialized when loading a new project.
- `PERSISTENT` variables retain their values when a new project is loaded, but only those whose names and types have not changed in the new version of the project.

`PERSISTENT` variables are useful in cases where it is known in advance that the project will be modified (e.g., after the extended test operation). During this time, operators can change setpoints, recipe

parameters, etc. If there are dozens (or even hundreds) of such changes, it will be difficult to rewrite all these values after loading each new project version. When using **PERSISTENT** variables, this can be avoided (unless the names or types of **PERSISTENT** variables have been changed).

In other cases, **RETAIN** variables can be used.

The declaration of **RETAIN** and **PERSISTENT** variables differs significantly.

To declare **RETAIN** variables, a variable area must be created with the **RETAIN** modifier added. Both local and global **RETAIN** variables are used. Declaring **RETAIN** variables in functions does not make sense, and in function blocks it is undesirable due to excessive memory consumption (Sect. 5.8.1, 5.8.2).

Example 5.5.2

```
PROGRAM PLC_PRG
VAR
	iNonRetainVar: INT := 10;
END_VAR
VAR_RETAIN
	iRetainVar: INT := 100;
END_VAR
```

After loading the application, the **iRetainVar** variable will take the value 100. Further in the operation, this value may change. E.g., an operator in the visualization will set the variable value to 200. After restarting the controller, the value of the **RETAIN** variable will be retained and set to 200.

PERSISTENT variables are declared in a special object of the same name, which must be added to the project tree:

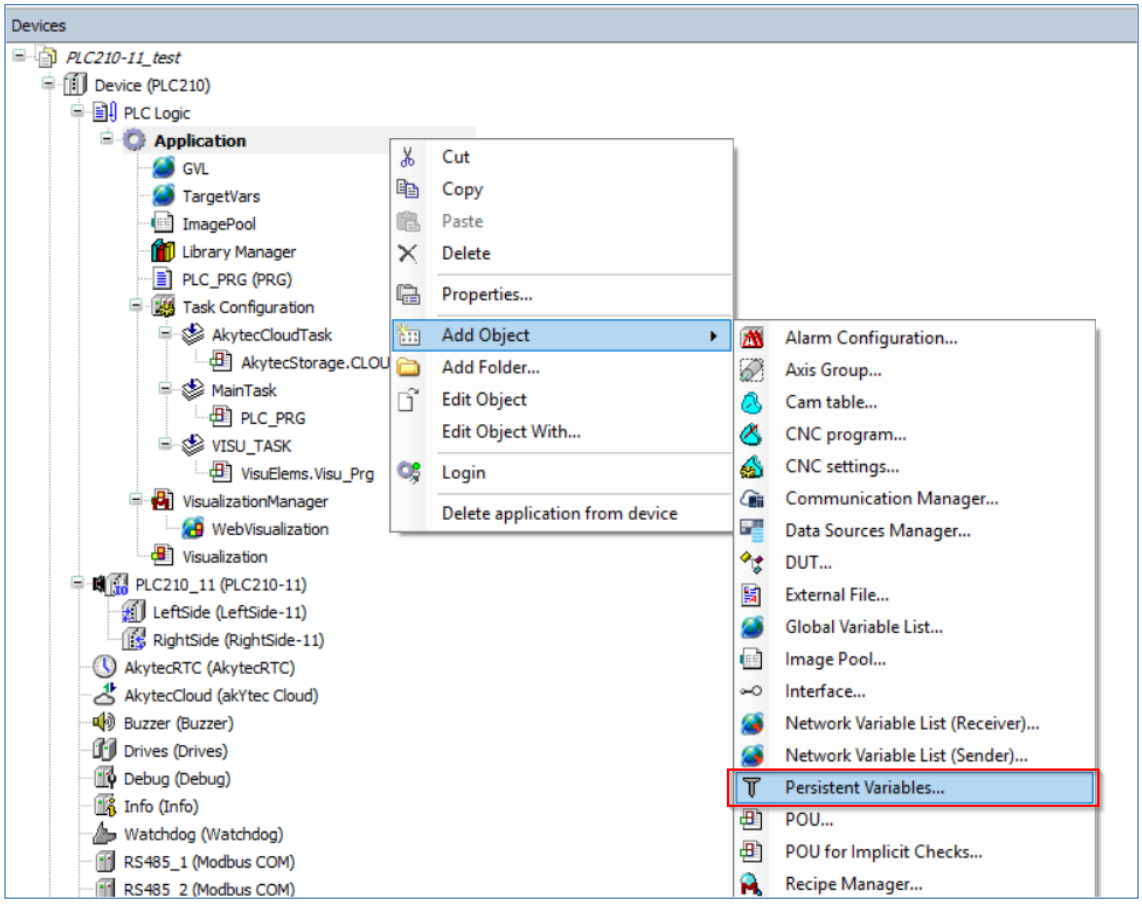


Fig. 5.5.2 Creating a list of PERSISTENT variables

Variables declared in this list are global **PERSISTENT** variables. As with regular global variables, the list name must be specified in order to access them from within the program.

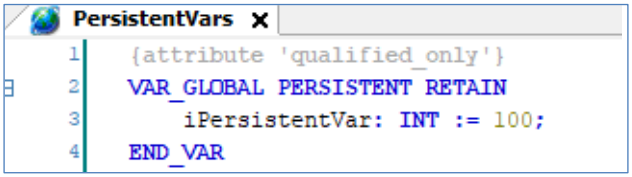


Fig. 5.5.3 Declaration of a PERSISTENT variable

Example 5.5.3

```
      // in the programme code:
      // PersistentVars – name of the list of PERSISTENT variables
PersistentVars.iPersistentVar := 200;
```

Physically, **RETAIN** and **PERSISTENT** variables are stored in the same memory area. A special MRAM microchip is used for their storage. Non-volatile variables are saved upon a change in any of the variables, but no more than once per second. In this case, the entire MRAM area is rewritten. The number of MRAM rewriting cycles is practically infinite. MRAM includes two redundant memory areas. At startup, the controller checks the checksums of these areas. If the calculated checksum for one area does not match the stored checksum (e.g., this can happen if the power fails at the moment of saving), the variable values are restored from the other area.

The non-volatile memory capacity of akYtec controllers currently produced is 64 kB.

In Sect. 4.2.6, we discussed the different options for the ***Reset*** command. Now we will show how the different reset options affect the values of non-volatile variables:

Table 5.5.1 Behavior of variables after different reset types

Reset	Standard	RETAIN	PERSISTENT
Reset Warm (Application Reset)	reset to initial/default	retain	retain
Reset Cold	reset to initial/default	reset to initial/default	retain
Application Download	reset to initial/default	reset to initial/default	retain *

* only variables whose names and types have not changed compared to the previous version of the project

5.6 Operators

A controller program is a set of actions performed on variables. These actions are called operators, and the variables and values on which these actions are performed are called operands (or arguments). A sequence of related operators is called an expression.

Example 5.6.1

```
xAlarm := xSensorFailed AND xAlarmDelayTimeout;
```

AND and '**:=**' are operators.

xSensorFailed and **xAlarmDelayTimeout** are the operands of the **AND** operator.

xAlarm and the result of the **AND** operator are the operands of the '**:=**' operator.

The entire line of code is an expression.

In the following sections, we will discuss the basic CODESYS operators. Most of them are defined in IEC 61131-3.

5.6.1 Arithmetic operators

Arithmetic operators are used to perform basic arithmetic operations: addition, subtraction, multiplication, and division. These operators can be applied to variables and values of integer type (Sect. 5.3.2), floating-point type (Sect. 5.3.3), and, with some restrictions, of date and time type (Sect. 5.3.5). The type of return value is determined by the types of the operands.

Example 5.6.2

```
PROGRAM PLC_PRG
VAR
	iVar1: INT := 2;
	iVar2: INT := 3;
	iVar3: INT;
	iVar4: INT;
	iVar5: INT;
	rVar: REAL;
END_VAR

// code area

iVar1 := iVar2 + 5;
iVar3 := 23 - iVar2;
iVar4 := iVar3 * 7;
rVar := TO_REAL(iVar4) / 3;
	// Any combination of operators can be used in the expression.
iVar4 := iVar1 + iVar2 + 3 - iVar3 * 2;
	// Modulo-operator: the remainder from division
iVar5 := iVar2 MOD iVar1;
```

In the division example, one of the operands is converted to the **REAL** so that the result of the division is a floating-point value. Without type conversion, the result of the division would be an integer, as when dividing two integers.

The **MOD** operator returns the remainder of the division of two integers.

5.6.2 Logical operators

Logical operators perform logical operations (**AND**, **OR**, **NOT**) on variables and values of type **BOOL**, integers and individual bits of integer variables, i.e., these operators are bitwise. The return type is determined by the type of the operands.

Example 5.6.3

```
PROGRAM PLC_PRG
VAR
	xVar1: BOOL;
	xVar2: BOOL;
	xVar3: BOOL;
	xResult1, xResult2, xResult3, xResult4, xResult5: BOOL;
	byBitMask: BYTE;
	byVar1: BYTE := 1;
	byVar2: BYTE := 2;
	byResult1, byResult2: BYTE; END_VAR
END_VAR

// code area

	// AND
	// xResult1 takes TRUE if both operands are TRUE.
	// xResult1 takes FALSE if at least one of the operands is FALSE.
xResult1 := xVar1 AND xVar2 AND byBitMask.0;
```

```
	// OR
	// xResult2 takes TRUE if at least one of the operands is TRUE.
	// xResult2 takes FALSE only if all operands are FALS.
xResult2 := xVar1 OR xVar2 OR byBitMask.0;

	// XOR
	// xResult3 takes TRUE if only one of the operands is TRUE.
	// In all other cases, xResult3 takes FALSE.
xResult3 := xVar1 XOR xVar2;

	// multiple XORs in a single expression
	// Assume that all three variables are TRUE.
	// The expression becomes 'TRUE XOR TRUE XOR TRUE'.
	// The result of 'TRUE XOR TRUE' is FALSE.
	// Substituting the result into the expression above, we get 'FALSE XOR TRUE'.
	// The result of the expression is TRUE
	// because only one of the operands is TRUE.
xResult4 := xVar1 XOR xVar2 XOR xVar3;

	// NOT
	// If xVar1 is TRUE, xResult5 takes FALSE
	// If xVar1 is FALSE, xResult5 takes TRUE
xResult5 := NOT(xVar1);

	// bitwise OR
	// When substituting values, we get '1 OR 2'.
	// byResult1 takes 3 because logical operators are applied to integer bitwise:
	// 2#00000001 OR 2#00000010 -> 2#00000011 (3 in the decimal system)
byResult1 := byVar1 OR byVar2;

	// bitwise NOT
	// byResult2 takes 254, since the values of all bits are inverted.
	// byVar1 is initially 1 (2#00000001)
	// After inversion, the value takes 2#11111110 (254 in the decimal system)
byResult2 := NOT(byVar1);
```

The **AND** and **OR** operators always evaluate both of their operands, i.e. even if the first operand of the **AND** operator is **FALSE** and the result of the operator is **FALSE** regardless of the value of the second operand, this value will still be evaluated.

In CODESYS V3.5, the ST language implements special **AND_THEN** and **OR_ELSE** operators that perform reduced evaluation of their operands, i.e., the second operand is evaluated only if it matters.

5.6.3 Comparison operators

Comparison operators are used to compare variables and values of integer (Sect. 5.3.2), floating-point (Sect. 5.3.3), and date and time types (Sect. 5.3.5). The return type is BOOL.

Example 5.6.4

```
PROGRAM PLC_PRG
VAR
	iVar1: INT := 11;
	iVar2: INT := 22;
	rVar: REAL := 33.44;
	tVar: TIME := T#10m20s;
	xResult1: BOOL;
	xResult2: BOOL;
	xResult3: BOOL;
	xResult4: BOOL;
	xResult5: BOOL;
	xResult6: BOOL;
```

```
      xResult7: BOOL;
END_VAR

// code area

      // Check whether the first operand is greater than the second
xResult1 := iVar2 > iVar1;

      // Check whether the first operand is less than the second
xResult2 := iVar2 < iVar1;

      // Check whether the first operand is greater than or equal to the second
xResult3 := rVar >= 0.5;

      // Check whether the first operand is less than or equal to the second
xResult4 := tVar <= T#15m;

      // Check whether the value belongs to the range [10, 20]
xResult5 := iVar1 >= 10 AND iVar1 <= 20;

      // Equality check
xResult6 := iVar1 = 11;

      // Inequality check
xResult7 := iVar1 <> 11;
```

5.6.4 Bit shift operators

Bit shift operators have the following form: `<bit shift operator(argument_1, argument_2)>`.

The first operand of bit shift operators is a variable or integer value, and the second is the number of bits to be shifted. The return type is determined by the type of the operand.

Example 5.6.5

```
PROGRAM PLC_PRG
VAR
      bySource: BYTE := 2#00110001;
      byResult1: BYTE;
      byResult2: BYTE;
      byResult3: BYTE;
      byResult4: BYTE;
END_VAR

// code area

      // left shift by 3 bits with zero padding
      // byResult1 takes the value 2#10001000
byResult1 := SHL(bySource, 3);

      // right shift by 3 bits with zero padding
      // byResult2 takes the value 2#00000110
byResult2 := SHR(bySource, 3);

      // cyclic left shift by 3 bits
      // byResult3 takes the value 2#10001001
byResult3 := ROL(bySource, 3);

      // cyclic right shift by 3 bits
      // byResult4 takes the value 2#0100110
byResult4 := ROR(bySource, 3);
```

5.6.5 Mathematical operators

The arguments of mathematical operators are variables or values of integer (Sect. 5.3.2) or floating-point type (Sect. 5.3.3). The return type is floating-point, except for the ABS operator, whose type is determined by the type of the argument.

Example 5.6.6

```
PROGRAM PLC_PRG
VAR
	bySource: BYTE := 2#00110001;
	byResult1: BYTE;
	byResult2: BYTE;
	byResult3: BYTE;
	byResult4: BYTE;
END_VAR

// code area

	// absolute value
	// iResult1 takes 11
iResult := ABS(iVar1);

	// square root
rResult1 := SQRT(iVar2);

	// exponent (e^b, where where e is Euler's number, approximately 2.71828)
rResult2 := EXP(iVar2);

	// power function (a^b, wher a = argument_1 is base, b = argument_2 is exponent)
rResult3 := EXPT(iVar2, 3);

	// natural logarithm
rResult4 := LN(iVar2);

	// logarithm to base 10
rResult5 := LOG(iVar2);
```

Root extraction and logarithm operations only make sense for positive numbers. If the argument is a negative number, `NaN` will be returned as a result. `NaN` is a special state of a floating-point number that can be obtained as a result of incorrect mathematical operations. To check for `NaN`, you can use functions from the ***FloatingPointUtils*** library.

5.6.6 Trigonometric operators

The arguments of trigonometric operators are variables or values of integer or floating-point type that define the angle in radians. For inverse trigonometric operators (arc sine, arc cosine, and arc tangent), the arguments are dimensionless. The return type is floating-point, both REAL and LREAL can be used.

Example 5.6.7

```
PROGRAM PLC_PRG
VAR
// 0.785398 radians correspond to 45°
	rAngleInRadian: REAL := 0.785398;
	rValue: REAL := 0.707106;
	rResult1: REAL;
	rResult2: REAL;
	rResult3: REAL;
	rResult4: REAL;
	rResult5: REAL;
	rResult6: REAL;
```

```
END_VAR

// code area

       // sine
       // rResult1 takes ≈ 0.707106
rResult1 := SIN(rAngleInRadian);

       // cosine
       // rResult2 takes ≈ 0.707106 (the sine and cosine values of 45° are the same)
rResult2 := COS(rAngleInRadian);

       // tangent
       // rResult3 takes ≈ 1
rResult3 := TAN(rAngleInRadian);

       // arcsine
       // rResult4 takes ≈ 0.785398
rResult4 := ASIN(rValue);

       // arccosine
       // rResult5 takes ≈ 0.785398
rResult5 := ACOS(rValue);

       // arctangent
       // rResult6 takes ≈ 0.61548
rResult6 := ATAN(rValue);
```

5.6.7 Selection operators

The arguments of selection operators are variables or values of any type. The return type is determined by the type of the arguments.

Example 5.6.8

```
PROGRAM PLC_PRG
VAR
       xSel: BOOL;
       sSel: STRING;
       iVar1: INT := 10;
       iVar2: INT := 20;
       iVar3: INT := 30;
       iMax: INT;
       iMin: INT;
       rVar: REAL := 15.25;
       rLimit: REAL;
       usiIndex: USINT := 2;
       udiMux1: UDINT;
       udiMux2: UDINT;
END_VAR

// code area

       // SEL - binary selection operator
       // First argument is of BOOL type
       // If it has the value FALSE,
       // the operator returns the value of the second argument.
       // If it has the value TRUE, the operator returns the value of the third argument
sSel := SEL(xSel, 'Norm', 'Failure');

       // MAX - maximum value from an arbitrary number of arguments
iMax := MAX(iVar1, iVar2, iVar3);
```

```
      // MIN - minimum value from an arbitrary number of arguments
iMin := MIN(iVar1, iVar2, iVar3);

      // LIMIT - limit operator
      // First argument - minimum value
      // Second argument - controlled value
      // Third argument - maximum value
      // If the controlled value is less than the minimum,
      // the operator returns the minimum
      // If the controlled value is greater than the maximum,
      // the operator returns the maximum
      // If the controlled value is greater than or equal to the minimum
      // or less than or equal to the maximum, the operator returns the controlled value
      // rLimit takes 12.0, since 15.25 > 12.0
rLimit := LIMIT(10.0, rVar, 12.0);

      // MUX - multiple choice operator
      // The first argument is an integer index of the value (numbering starts from 0)
      // The remaining arguments (any number) are available values
      // The index from the first argument determines which value is returned
      // udiMux1 takes 30
udiMux1 := MUX(usiIndex, 10, 20, 30);

      // since the value with index 2 is missing,
      // the last available value will be returned
      // udiMux2 takes 20
udiMux2 := MUX(usiIndex, 10, 20);

      // The use of MIN, MAX, and LIMIT operators
      // does not make sense for string variables.
```

5.6.8 Conversion operators

The conversion operators were discussed in section 5.3.6.

There are two other specific conversion operators: **TRUNC** and **TRUNC_INT**.

They are equivalent to the **TO_DINT** and **TO_INT** operators respectively, with **REAL** arguments. They differ in one respect: while standard conversion operators round the value to the nearest integer, the **TRUNC** and **TRUNC_INT** operators truncate its fractional part.

Example 5.6.9

```
PROGRAM PLC_PRG
VAR
      diResult1: DINT;
      diResult2: DINT;
      diResult3: DINT;
      diResult4: DINT;
END_VAR

// code area

      // diResult1 takes 1
diResult1 := TO_DINT(1.4);

      // diResult2 takes 2
diResult2 := TO_DINT(1.6);

      // diResult3 takes 1
diResult3 := TRUNC(1.4);

      // diResult4 takes 1
```

```
diResult4 := TRUNC(1.6);
```

5.6.9 Assignment operator

We have already used the assignment operator many times. We have already used the assignment operator many times. The only other thing to add is that group assignment is allowed within a single expression.

Example 5.6.10

```
PROGRAM PLC_PRG
VAR
	iVar1: INT;
	iVar2: INT;
	iVar3: INT
END_VAR

// code area

	// All variables take 10
iVar1 := iVar2 := iVar3 := 10;
```

5.6.10 Size operators

The **SIZEOF** operator returns the size of an argument in bytes. The argument can be a variable of any type. The return type depends on the size of the variable:

- if the variable size is less than 256 bytes, then the return type is **USINT**
- if the variable size is in the range 256…65535 bytes, the return type is **UINT**
- if the variable size is in the range 65536...4294967295 bytes, the return type is **UDINT**
- if the variable size is greater than 4294967295 bytes, the return type is **ULINT**

The **XSIZEOF** operator has been added to CODESYS V3.5 SP16. This operator works similarly to **SIZEOF**, but the return type is always **UDINT** for 32-bit platforms and **ULINT** for 64-bit platforms. Starting with this version of CODESYS, it is recommended to use only the **XSIZEOF** operator.

Example 5.6.11

```
PROGRAM PLC_PRG
VAR
	udiSize1: UDINT;
	udiSize2: UDINT;
	udiSize3: UDINT;
	udiSize4: UDINT;
	iVar: INT;
	rVar: REAL;
	wsVar: WSTRING;
	awVar: ARRAY [0..11] OF WORD;
END_VAR

// code area

	// udiSize1 takes 2
udiSize1 := XSIZEOF(iVar);

	// udiSize2 takes 4
udiSize2 := XSIZEOF(rVar);
```

```
	// udiSize3 takes 162
udiSize3 := XSIZEOF(wsVar);

	// udiSize4 takes 24
udiSize4 := XSIZEOF(awVar);
```

5.6.11 Other operators

In sections 5.6.1...5.6.10, we discussed the main operators used in most user projects. The following remain unaddressed:

- The operators used to work with pointers are not covered in this document.
- The **INDEXOF** and **INI** operators, retained for compatibility with projects created in CoDeSys V2.3. These operators should not be used in projects created in CODESYS V3.5.
- The system operators. The names of such operators begin with the symbol '__' (e.g., **__VARINFO**). These operators are rarely used and mainly only by library developers, so they are not discussed in this document either.

5.6.12 Execution order of operators

Expressions usually consist of several operators. In some cases, the order in which operators are executed plays a significant role.

Example 5.6.12

```
PROGRAM PLC_PRG
VAR
	uiVar: UINT := 200;
	xAlarm: BOOL;
END_VAR

// code area

	// xAlarm takes FALSE
xAlarm := NOT uiVar < 123;
```

The following behavior can be expected: the variable **uiVar** has a value of 200, which is greater than 123, so the comparison operator will return **FALSE**. Next, the **NOT** operator should be executed, and the variable **xAlarm** takes the value **TRUE**.

But in reality, the variable **xAlarm** will get the value **FALSE**, because the order of execution of the operators is different. First, the **NOT** operator will be applied to the variable **uiVar**, resulting in the value of 65335 (because the **NOT** operator is bitwise; Sect. 5.6.2). After that, the comparison will be executed. Since 65335 is greater than 123, the result of the comparison will be **FALSE**.

The table of operator priorities (binding rules) is provided here.

Operators with the same priority are processed in order from left to right.

Parentheses are used to control priority. Statements enclosed in parentheses are executed first. To ensure that the comparison statement is executed first, parentheses need to be added to our example.

Example 5.6.13

```
	// xAlarm takes TRUE
xAlarm := NOT(uiVar < 123);
```

5.7 Control statements

To implement even the simplest algorithm, you will need to use statements that control the execution of the program. In this section, we will discuss the control statements of the ST language.

All of these statements can be used within each other, i.e., within IF, you can use CASE, FOR, and IF itself, etc.

5.7.1 IF conditional statement

The **IF** statement is the most frequently used control statement. It is used to check one or more conditions, and if at least one of the conditions is true, it executes the expressions specified for that condition. After the expressions are executed, the statement is exited, i.e., the remaining conditions are no longer checked, and control passes to the line after **END_IF**.

We will consider how the statement works using the example of an alarm being triggered when the temperature exceeds the limits.

Example 5.7.1

```
PROGRAM PLC_PRG
VAR
	rRoomTemp: REAL;
	xHighTempAlarm: BOOL;
	xLowTempAlarm: BOOL;
END_VAR
VAR CONSTANT
	c_rHighTempAlarm: REAL := 32.5;
	c_rLowTempAlarm: REAL := 13.0;
END_VAR

// code area

IF rRoomTemp > c_rHighTempAlarm THEN
	xHighTempAlarm := TRUE;
ELSIF rRoomTemp < c_rLowTempAlarm THEN
	xLowTempAlarm := TRUE;
ELSE
	xHighTempAlarm := FALSE;
	xHighTempAlarm := FALSE;
END_IF
```

- If the condition in the **IF** statement is **TRUE** (the **rRoomTemp** variable is greater than the **c_rHighTempAlarm** constant), then the **xHighTempAlarm** variable takes **TRUE** and the statement is exited (the next condition is not evaluated).
- If the condition is **FALSE**, the next condition in the nested **ELSIF** statement is evaluated.
- If the condition in **ELSIF** is **TRUE** (the **rRoomTemp** variable is less than the **c_rLowTempAlarm** constant), the **xLowTempAlarm** variable takes **TRUE** and the statement is exited (the next condition is not evaluated).
- If none of the conditions in **IF** and **ELSIF** are **TRUE** (the temperature is within the limits), the expressions in the nested **ELSE** statement are executed, i.e., the variables **xHighTempAlarm** and **xLowTempAlarm** take **FALSE**.

The use of nested **ELSIF** and **ELSE** statements is optional, but the number of **ELSIF** statements within an **IF** statement is not limited.

Now we will consider a specific case in which the use of the **IF** statement is unnecessary:

Example 5.7.2

```
PROGRAM PLC_PRG
VAR
	xMotorControlButton: BOOL;
	xStartMotor: BOOL;
END_VAR

// code area

IF xMotorControlButton THEN
	xStartMotor := TRUE;
ELSE
	xStartMotor := FALSE;
END_IF

	// A shorter version of the same code without using the IF statement:
xStartMotor := xMotorControlButton;
```

5.7.2 CASE multiple choice statement

The **CASE** statement is used to compare the value of a given integer variable (selector) with a set of constants or integer values (labels), and if there is a match, execute the expressions specified for that label. After the expressions are executed, the statement is exited.

Example 5.7.3

```
PROGRAM PLC_PRG
VAR
	iSelector: INT := 2;
	xOutput1: BOOL;
	xOutput2: BOOL;
	xOutput3: BOOL;
	xOutput4: BOOL;
	xOutput5: BOOL;
END_VAR

// code area

CASE iSelector OF
	0:
		xOutput1 := TRUE;
	1, 5:
		xOutput2 := TRUE;
		xOutput3 := TRUE;
	2..4:
		xOutput4 := TRUE;
ELSE
	xOutput5 := TRUE;
END_CASE
```

As shown in the example, a label can include several values separated by commas or a range.

The values of one label must not coincide with the values of others.

The nested **ELSE** operator is optional; the expressions placed in it are executed if the selector value does not match any of the labels.

Example 5.7.4

```
	// An alternative version of the same code using the IF statement:

IF iSelector = 0 THEN
```

```
        xOutput1 := TRUE;
ELSIF iSelector = 1 OR iSelector = 5 THEN
        xOutput2 := TRUE;
        xOutput3 := TRUE;
ELSIF iSelector >=2 AND iSelector <=4 THEN
        xOutput4 := TRUE;
ELSE
                xOutput5 := TRUE;
        END_IF
```

The code variant with the **CASE** operator looks very concise, as there is no need to explicitly write comparisons.

Enumeration elements are often used as CASE operator labels:

Example 5.7.5

```
        // Enumeration code

{attribute 'qualified_only'}
{attribute 'strict'}
TYPE WORK_MODE:
(
                // Local control mode
        LOCAL := 0,
                // Remote control mode
        REMOTE := 1,
                // Automatic control mode
        AUTO := 2
);
END_TYPE

PROGRAM PLC_PRG
VAR
        eWorkMode: WORK_MODE;
END_VAR

// code area

CASE eWorkMode OF
        WORK_MODE.LOCAL:
        // place for local control mode expressions

        WORK_MODE.REMOTE:
        // place for remote control mode expressions

        WORK_MODE.AUTO:
        // place for automatic control mode expressions
END_CASE
```

5.7.3 FOR loop statement

The **FOR** operator is used to organize a loop with a fixed number of iterations. It is commonly used for operations on data arrays. We recall Example 5.3.12:

Example 5.7.6

```
PROGRAM PLC_PRG
VAR
        arTemp:                     ARRAY [1..c_iMaxTempPointCount] OF REAL;
        arTempHighAlarmLimit:       ARRAY [1..c_iMaxTempPointCount] OF REAL;
```

```
	axAlarm:                    ARRAY [1..c_iMaxTempPointCount] OF BOOL;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iMaxTempPointCount: INT := 100;
END_VAR

// code area

FOR i := 1 TO c_iMaxTempPointCount DO
	axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

The variable **i** is called the loop iterator or loop counter. This variable must be of a signed integer type (usually **INT** or **DINT**). After each execution of the loop body, the counter value changes by one step (by default, the step is equal to 1, but the user can specify the step using the nested **BY** operator). After that, the program jumps to the next iteration, and the entire loop is executed again.

Example 5.7.7

```
// i sequentially takes the values 1, 3, 5, 7, etc.
	FOR i := 1 TO c_iMaxTempPointCount BY 2 DO
		axAlarm[i] := arTemp[i] > arTempHighAlarmLimit[i];
END_FOR
```

After the last iteration is completed, the counter changes its value last time, i.e., the variable **i** takes the value 101.

The following example shows that strings can be treated as arrays and demonstrates a loop with a decrementing counter and early loop termination after finding the desired character in the string using the **EXIT** statement.

Example 5.7.8

```
PROGRAM PLC_PRG
VAR
	sPouName: STRING := 'Device.Application.PLC_PRG';
	i: INT;
END_VAR
VAR_CONSTANT
	c_sCharSeparator: STRING(1) := '.';
END_VAR

// code area

FOR i := LEN(sPouName) - 1 TO 0 BY -1 DO
	IF sPouName[i] = c_sCharSeparator[0] THEN
		EXIT;
	END_IF
END_FOR
```

- The **sPouName** string contains the path to the program object in the project. Suppose we want to extract the name of the program object '**PLC_PRG**' from this path for subsequent logging. We need to determine the number of the first character in this string.
- To achieve this, we can walk through the string as an array from the last character to the first one using the '**BY -1**' step. String array indexing starts from zero, so we know the final index. To determine the starting index, we use the **LEN** function from ***Standard*** library, which returns the number of characters in the string. Since we are treating the string as an array, we need to reduce

the length of the string by 1, because the first character of the string corresponds to the array element with number 0.

- Within the loop, we check whether the current character being processed is a period. If it is, we exit the loop using the **EXIT** statement to skip the remaining unnecessary iterations.
- As a result, **i** takes 18. This is the number of the character we need (the last period in the string). Using this number, we can extract the part of the string that follows it and represents the name of the program object '**PLC_PRG**'.

Another available nested statement for the **FOR** loop is **CONTINUE**. It is used to interrupt the current loop iteration and jump to the next one. In the example below, it is used to avoid division by 0:

Example 5.7.9

```
PROGRAM PLC_PRG
VAR
	// Assume that the arrays are filled with values in another part of the code.
	arDividend: ARRAY [1..c_iCountOfNumbers] OF REAL;
	aiDivisor: ARRAY [1..c_iCountOfNumbers] OF INT;
	arResult: ARRAY [1..c_iCountOfNumbers] OF REAL;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iCountOfNumbers: INT := 5;
END_VAR

// code area

FOR i := 1 TO c_iCountOfNumbers DO
	IF aiDivisor[i] = 0 THEN
		CONTINUE;
	ELSE
		arResult[i] := arDividend[i] / aiDivisor[i];
	END_IF
END_FOR
```

This code fragment could be rewritten without the **CONTINUE** statement:

Example 5.7.10

```
PROGRAM PLC_PRG
VAR
// Assume that the arrays are filled with values in another part of the code.
	arDividend: ARRAY [1..c_iCountOfNumbers] OF REAL;
	aiDivisor: ARRAY [1..c_iCountOfNumbers] OF INT;
	arResult: ARRAY [1..c_iCountOfNumbers] OF REAL;
	i: INT;
END_VAR
VAR_CONSTANT
	c_iCountOfNumbers: INT := 5;
END_VAR

// code area

FOR i := 1 TO c_iCountOfNumbers DO
	IF aiDivisor[i] <> 0 THEN
		arResult[i] := arDividend[i] / aiDivisor[i];
	END_IF
END_FOR
```

In general, **CONTINUE** is not used very often.

Since the **FOR** statement is often used with arrays, we recommend reviewing Sect. 5.3.7.

Please note that it is not advisable to manipulate the value of the loop counter within the loop itself. In most cases, this does not provide any benefit and only results in code that is difficult to read and prone to errors.

5.7.4 Other control statements

There are several other statements that are rarely used in practice, so we will only briefly review them.

WHILE and **REPEAT** – the statements are used to create a loop with an unknown number of iterations. The loop is repeated until the specified condition is met. For **WHILE**, the condition is checked before the expressions are executed; for **REPEAT**, it is checked after. In practice, in most cases, the number of iterations is known in advance (e.g., the size of arrays is strictly defined in the declaration). Furthermore, controller programs are usually tied to cyclical tasks, i.e., they themselves are executed in an infinite loop. Finally, the use of **WHILE** and **REPEAT** statements by beginners often leads to an infinite loop in the program, resulting in an exception and the application stopping execution.

JMP – unconditional jump statement, similar to the **GOTO** statement in other programming languages. Using this statement usually results in “spaghetti” code that is difficult to read and debug, with a high probability of errors.

RETURN – ehe statement is used to exit a software object. Usually, you can easily avoid using this statement by slightly modifying the program code. For example, we have a large piece of code that should only be executed if the operator has entered the correct password in the visualization. Using the **RETURN** statement, you can write the following code:

Example 5.7.11

```
// code area

IF sVisuPassword <> sExpectedPassword THEN
       RETURN;
ELSE
       // ... expressions to be executed when a correct password is entered ...
END_IF
       // end of program object
```

This code fragment can be easily rewritten without the RETURN statement by simply changing the condition being checked:

Example 5.7.12

```
// code area

IF sVisuPassword = sExpectedPassword THEN
       // ... expressions to be executed when a correct password is entered ...
END_IF
       // end of program object
```

5.8 Program objects

In the previous examples, all code was placed in the `PLC_PRG` program. This program is automatically created as part of a new CODESYS project. In principle, you can limit yourself to this program by adding all of your application code to it, but this is not a good approach. As a result, you will end up with a single program consisting of several hundred (or possibly even thousands) lines of code, a significant portion of which will likely be created using copy-and-paste operations. This program will be difficult to debug and maintain, and it will most likely be prone to errors.

Therefore, a more reasonable option is to divide the program into isolated program objects that can be reused and edited independently of each other. Such objects in CODESYS are called ***POU*** (Program Organization Unit).

To create a program object, select the ***Add Object*** command from the ***Application*** node context menu and add a ***POU*** object (Fig. 5.8.1).

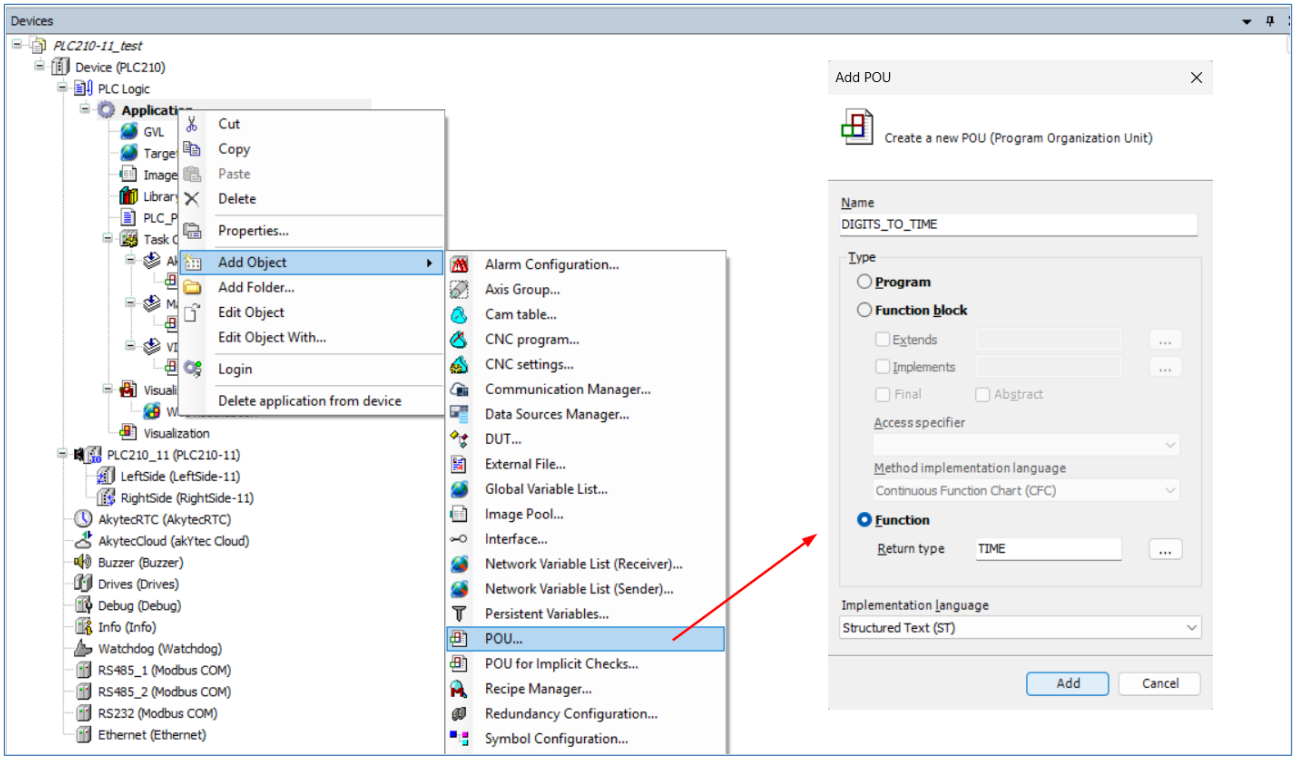


Fig. 5.8.1 Add POU to project

There are three types of program objects: functions (Sect. 5.8.1), functional blocks (Sect. 5.8.2), and programs (Sect. 5.8.3).

5.8.1 Functions

A function is a program object that does not have memory to store the values of its variables between calls. Functions are used for arithmetic calculations, signal scaling, converting data from one format to another, various checks (for example, whether a value belongs to an acceptable range), etc. Since a function has no memory, it cannot be used to organize a counter, timer, or other elements designed to store the temporary values of their variables

In most cases, functional blocks are not declared inside functions because these blocks usually require memory to store values between calls.

For the same reason, non-volatile variables cannot be declared inside functions.

When creating a function, you specify the type of value it returns, i.e., a variable is implicitly created in it, whose name coincides with the name of the function, and it is the type of this variable that is specified.

Below is an example of a declaration of a simple function used to assemble a variable of type **TIME** from individual time fields.

Example 5.8.1

```
FUNCTION DIGITS_TO_TIME : TIME
VAR_INPUT
	usiHour: USINT := 0;
	usiMinute: USINT := 0;
	usiSecond: USINT := 0;
END_VAR

// code area

DIGITS_TO_TIME := usiHour * T#1H + usiMinute * T#1M + usiSecond * T#1S;
```

Calling this function in a program:

Example 5.8.2

```
PROGRAM PLC_PRG
VAR
	tVar1: TIME;
	tVar2: TIME;
END_VAR

// code area

	// tVar1 takes the value T#20H5M3S
tVar1 := DIGITS_TO_TIME(20, 5, 3);

	// tVar1 takes the value T#12H7M1S
tVar2 := DIGITS_TO_TIME(usiHour := 12, usiMinute := 7, usiSecond:= 1);
```

When calling a function, you do not need to specify the names of its input variables. This is an “informal call.” In this case, the order of passing values must match the order of the function's input variables.

If the names of the function's input variables are specified when calling the function, it is a “forma call”. The order of listing inputs in a formal function call can be arbitrary.

If the input variables of a function have initial values, they can be omitted when calling the function if no values need to be passed to them. For example, our function can be called as follows:

Example 5.8.3

```
	// tVar1 takes the value T#7M
tVar2 := DIGITS_TO_TIME(usiMinute := 7);
```

The function may have additional outputs declared in the **VAR_OUTPUT** area.

We will modify our function so that it returns an error signal and resets its main output if the entered time values do not correspond to the permissible values for the time of day:

Example 5.8.4

```
FUNCTION DIGITS_TO_TIME : TIME
VAR_INPUT
	usiHour: USINT := 0;
```

```
	usiMinute: USINT := 0;
	usiSecond: USINT := 0;
END_VAR
VAR_OUTPUT
	xError: BOOL;
END_VAR

// code area

DIGITS_TO_TIME := usiHour * T#1H + usiMinute * T#1M + usiSecond * T#1S;
xError := usiHour > 23 OR usiMinute > 59 OR usiSecond > 59;
IF xError THEN
	DIGITS_TO_TIME := T#0s;
END_IF
```

An example of calling this function in a program with reference to its additional output (only formal calls are allowed for such outputs):

Example 5.8.5

```
VAR
	tVar1: TIME;
	xWrongParams: BOOL;
END_VAR

// code area

tVar1 := DIGITS_TO_TIME(20, 30, 30, xError => xWrongParams);
```

The values of function's outputs depend only on the values of its inputs. Calling a function in one place in the program does not affect other calls to it. A function can be called many times in a project with different arguments. Each function call with the same arguments will result in the same output values. This is because function variables do not retain their values between calls.

5.8.2 Functional blocks

A function block (FB) is a software object whose variable values are retained between calls. All elements that require memory (counters, timers, archive blocks, exchange blocks, etc.) are function blocks.

Below is an example of an FB declaration for collecting statistics on value changes: while the **xEnable** input is **TRUE**, the block stores the minimum and maximum values received by the **iValue** input and transmits them to the **iMin** and **iMax** outputs, respectively. If the **xReset** input takes the value **TRUE**, these outputs are initialized with the value **iValue** – the statistics are reset. The **xActive** output indicates that the block is active: while the **xEnable** input has the value **TRUE**, this output also has the value **TRUE**.

Example 5.8.6

```
FUNCTION_BLOCK STAT
VAR_INPUT
	xEnable: BOOL;
	iValue: INT;
	xReset: BOOL;
END_VAR
VAR_OUTPUT
	xActive: BOOL;
	iMin: INT;
	iMax: INT;
END_VAR
VAR
END_VAR
```

```
// code area

       // Upon start, the block keeps statistics relative to the first value received.
IF xEnable THEN
       IF iValue < iMin THEN
              iMin := iValue;
       END_IF
       IF iValue > iMax THEN
              iMax := iValue;
       END_IF
END_IF
xActive := xEnable;
```

FBs are generally not used in functions because blocks need memory to store their values, and functions do not have memory. FBs are mainly used in other FBs and programs. To use an FB in program, you need to create an instance of it (there can be many instances). Instances are variables whose type is the FB. When an FB instance is declared, its data structure (Sect. 5.3.9) is also declared.

Example of declaration and call of the FB created above in the program:

Example 5.8.7

```
PROGRAM PLC_PRG
VAR
       fbStat: STAT;
       xStatEnable: BOOL;
       xResetStatistics: BOOL;
       iUserValue: INT;
       iStatMin: INT;
       iStatMax: INT
END_VAR

// code area

fbStat
(
       xEnable := xStatEnable,
       iValue := iUserValue,
       xReset := xResetStatistics,
       iMin => iStatMin,
       iMax => iStatMax
);
```

The '**=>**' operator is used to copy values from the FB output variables.

FBs only support formal calls, with input and output names specified, but some inputs and outputs can be omitted: in our example, we do not refer to the **xActive** output.

To obtain statistics on several variables, it is necessary to declare the corresponding number of FB instances, or an array of them.

Example 5.8.8

```
PROGRAM PLC_PRG
VAR
       fbStat1: STAT;
       fbStat2: STAT;
       fbStat3: STAT;
       afbStat: ARRAY [1..3] OF STAT;
END_VAR
```

All instances of the blocks operate independently of each other, using their own memory.

Technically, non-volatile variables can be declared within an FB; however, if at least one such variable is declared, all variables in the block will automatically be stored in non-volatile memory. Therefore, non-volatile variables are not usually declared within blocks in order to avoid wasting this memory. For the same reason, FB instances are rarely declared as non-volatile, although in some specific cases this is a quite reasonable approach (for example, if the block occupies relatively little memory).

When creating an FB, several options are available (for example, `EXTENDS` and `IMPLEMENTS`). These options can be applied when using an object-oriented approach to application development. This approach is not discussed in this document.

5.8.3 Programs

Program is the main container where all logic, including calls to functions and FBs, takes place. Programs are assigned to tasks, have persistent memory, and are considered the top-level unit of code execution. The differences between programs and FBs are as follows:

The program can be assigned to a task.

An instance of the program cannot be declared.

The program allows the declaration of non-volatile variables, and this does not lead to additional consumption of non-volatile memory, as is the case with FB.

5.8.4 Comparison of functions, functional blocks, and programs

Table 5.8.1 Comparison of functions, functional blocks, and programs

	FC	FB	PROG
Storing variable values between calls	-	+	+
Using through instances	-	+	-
Non-volatile variables	-	~	+
Call in functions	+	~	~
Informal call	+	-	-
Namespaces (Sect. 5.10.3)	-	+	+

The symbol “~” indicates cases in which the use of the functionality is syntactically correct but, in most cases, contradicts logic.

5.8.5 Nested program objects (methods, etc.)

If you right-click on any POU and use the Add Object command, you can add nested POUs to it.

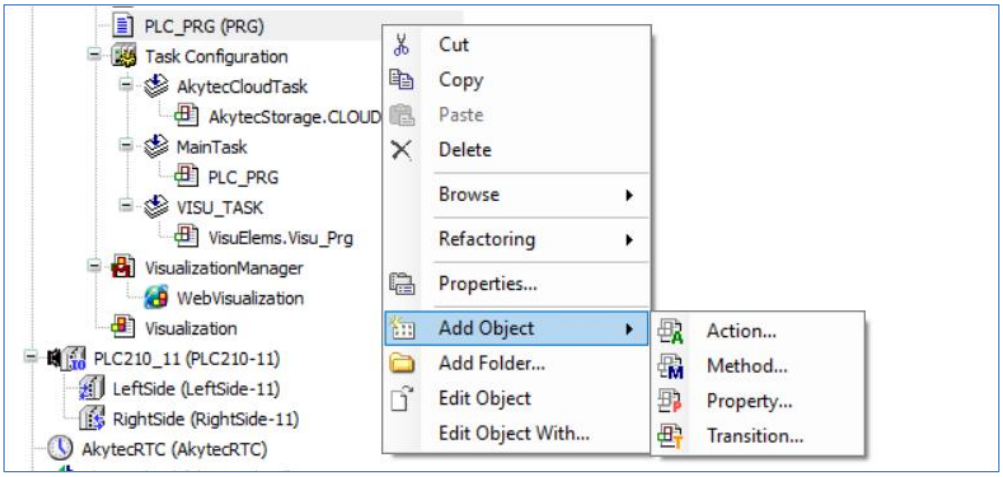


Fig. 5.8.2 Add nested POU

Brief overview of the nested program objects:

- ***Action*** – allows a POU fragment to be separated into an isolated object with its own name. In other programming languages, similar objects are called procedures (an action is a procedure without parameters). In SFC language it is always preferable to use a method rather than an action, as its functionality overlaps with that of an action. Actions are typically only used in projects that use the SFC language.
- ***Method*** – a function embedded in FB with access to its variables. Unlike an action, a method can have its own input, output, and local variables (as well as constants). Just as POUs are used to increase the modularity of projects, methods can be used to increase the modularity of individual POUs. E.g., a functional block for logging may contain several hundred lines of code, some of which are very similar to each other. Such fragments can be organized as methods. Methods are a convenient tool for structuring code; they can be used both within and outside of the object-oriented approach.
- ***Property*** – a variable with automatically created read (Get) and write (Set) methods. Properties are one of the elements of object-oriented programming and are rarely used outside of it.
- ***Transition*** – analogous to the action used in SFC language to describe the conditions for transitions between steps. Transitions are typically only used in projects that use the SFC language.

5.9 Tasks

Tasks are objects that define the conditions for calling programs. They enable the execution of project code. The list of project tasks is displayed in the ***Task Configuration*** tree component.

There are three tasks in the project template for the akYtec controller:

- ***MainTask*** – user logic task
- ***AkytecCloudTask*** – task for connecting to the akYtec Cloud service
- ***VISU_TASK*** – visualization task

Some tasks are automatically created when the relevant components are added to the project. Users also have the option to create tasks themselves, but we strongly advise inexperienced users not to create tasks manually or edit the settings of existing tasks. This is because the processing principle in CODESYS is quite complex (it uses a preemptive multitasking mechanism) and poorly documented.

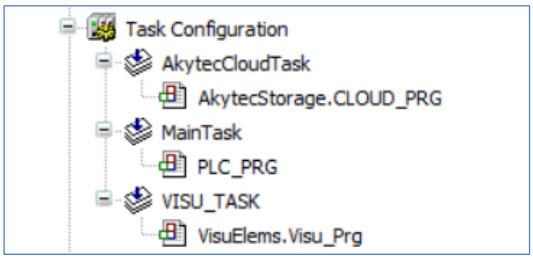


Fig. 5.9.1 Task Configuration structure

The task has the following settings (Fig. 5.9.2):

- ***Priority*** determines the importance of a task. It is relevant in cases where, during program execution, there is a moment when several tasks must be called simultaneously. In this case, the task with the highest priority will be executed first; the execution of tasks with lower priority will be postponed. In CODESYS, 0 corresponds to the highest priority, and 31 – to the lowest. Tasks with priorities 0...15 have real-time priority and can displace some operating system services.
- ***Type*** of task determines how it is called. Basically, only cyclic tasks are used. Other types of tasks are not recommended for use.

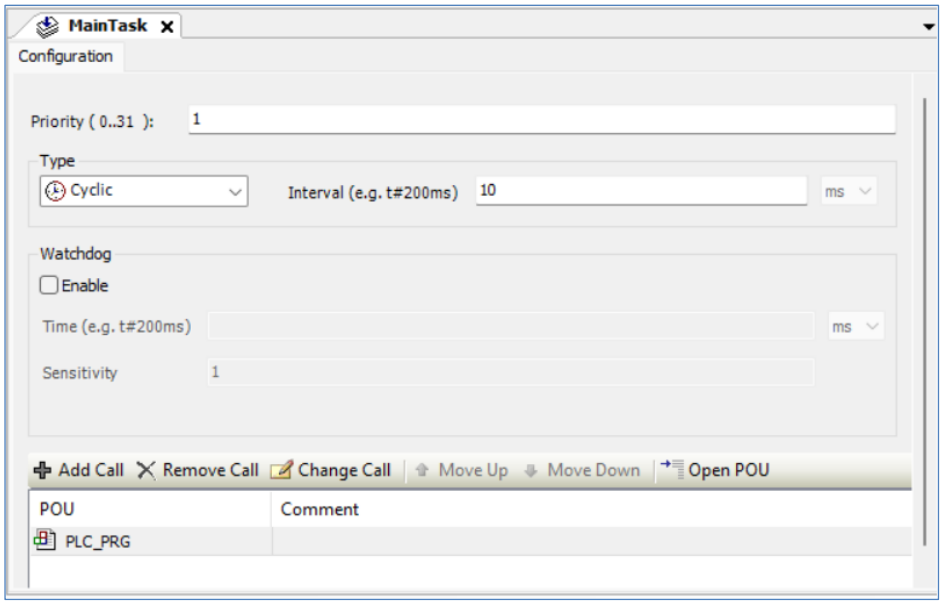


Fig. 5.9.2 Task Configuration settings

- ***Interva***l determines the period of time at which the cyclic task is called.
- ***Watchdog*** is a timer which interrupts task execution if the actual execution time exceeds the time specified for the watchdog parameter ***Time***. The ***Sensitivity*** parameter determines the number of consecutive task calls during which this condition must be met. The task will also be interrupted if, within a single task call, the actual execution time exceeds the time specified in the ***Time*** parameter by the number of times specified in the ***Sensitivity*** parameter. Watchdogs are rarely used, since during the debugging phase, all code fragments that take a long time to execute can be detected and optimized using online task monitoring.

If you open the task configuration when the controller is connected, the ***Monitoring*** tab will display information about the actual task execution time. Note that all times are displayed in ***microseconds***, although the task call interval is set in ***milliseconds***.

When the application is started, additional operations are performed, which cause the first few task cycles to run significantly slower than subsequent ones. As a result, the ***Max. Cycle Time*** column displays fairly large values. Therefore, after starting the application, it is advisable to reset the statistics using the ***Reset*** command in order to see reliable results. The most informative parameter is the ***Average Cycle Time***. In a correct project, it should always be less than the set ***Interval*** time. If the ***Average Cycle Time*** is greater than the ***Interval*** time, this is a sign that the project needs to be optimized.

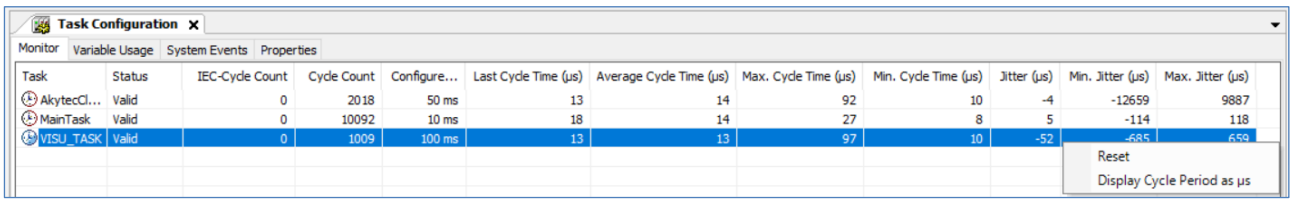


Fig. 5.9.3 Online task monitoring

Finally, we would like to remind you once again that we strongly advise inexperienced users not to create tasks manually or edit the settings of existing tasks.

5.10 Libraries

To extend the capabilities of CODESYS, controller manufacturers and users can develop their own project elements (functional blocks, visualization dialogs, etc.) and distribute them as libraries.

To use the library, it must first be installed in the CODESYS environment library repository and then added to the Library Manager of a specific project.

5.10.1 Library installation

To install the library in CODESYS, select ***Library Repository*** from the ***Tools*** menu, click ***Install***, and specify the path to the desired library file.

Libraries can be distributed in two formats:

- ***.compiled-library*** – access to source code is prohibited
- ***.library*** – source code is available

Please note that if the library file in the selection window has an unsuitable format, you will not be able to install the library. In this case, you need to switch the filter to ***All Files***.

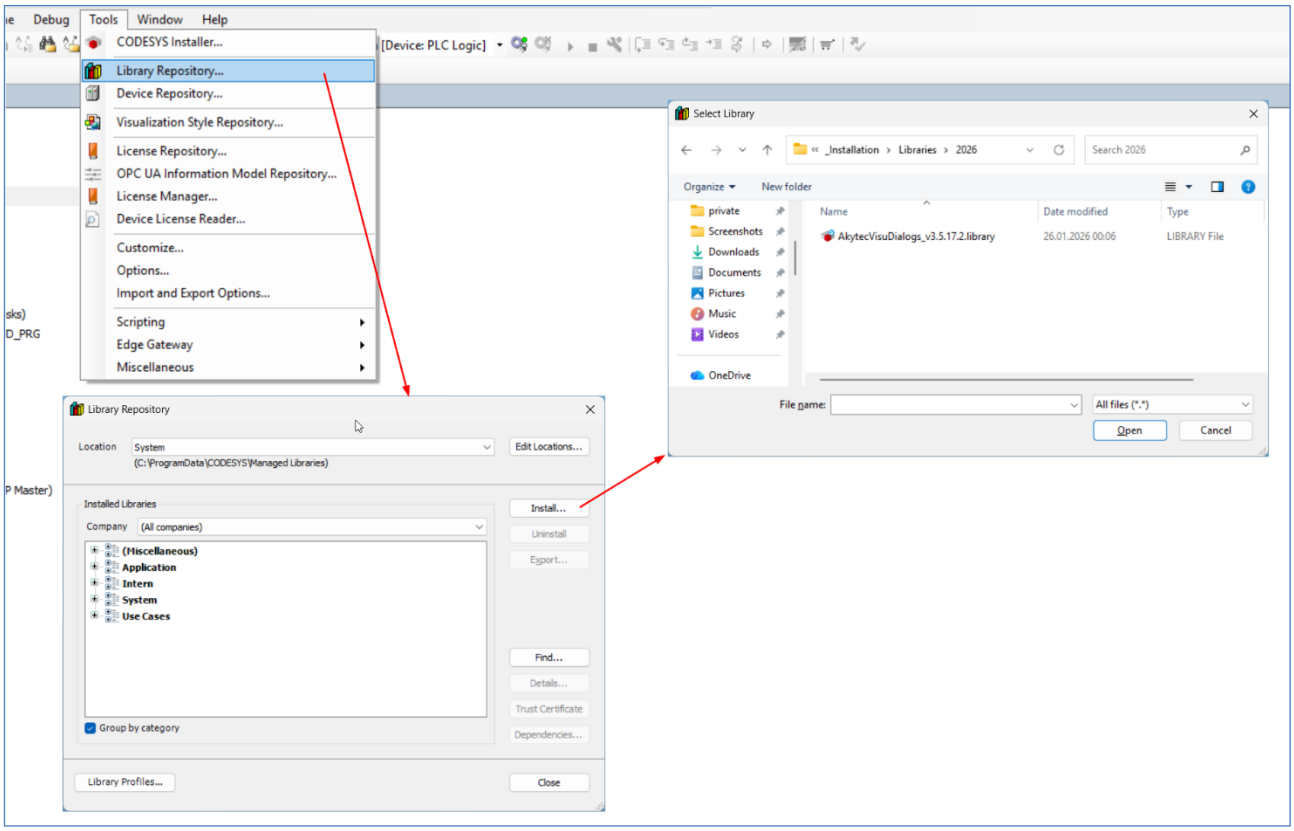


Fig. 5.10.1 Installing a library in the library repository

5.10.2 Add library to project

To add a library to a CODESYS project, go to the ***Library Manager*** component and click the ***Add Library*** button in the toolbar. A window with a list of libraries opens. It is recommended to switch it to the list view using the button in the upper right corner of the window. In the tree view, it can be quite difficult to find the library you need.

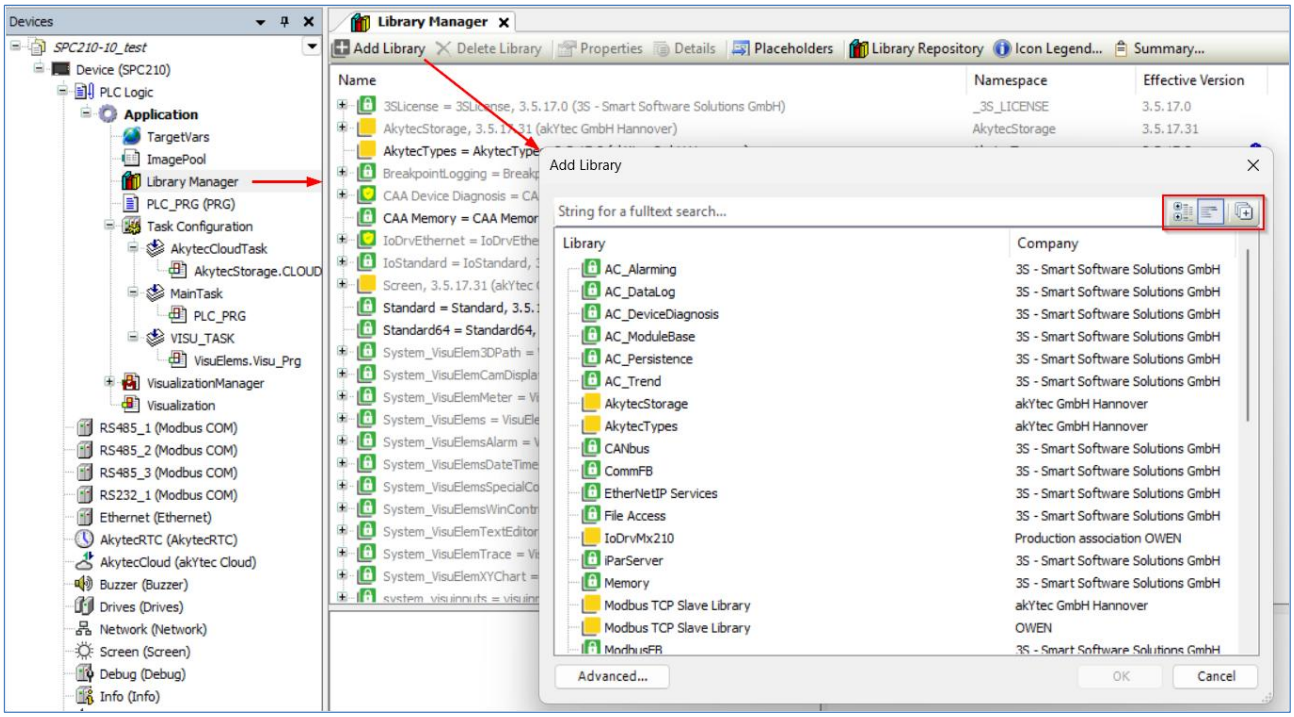


Fig. 5.10.2 Add a library to the project library manager

5.10.3 Using library objects in a project. Namespaces.

When you select a library in the library manager, a list of objects in that library is displayed in the lower left pane of the window. When you select a specific object, reference information about that object is displayed in the lower right pane.

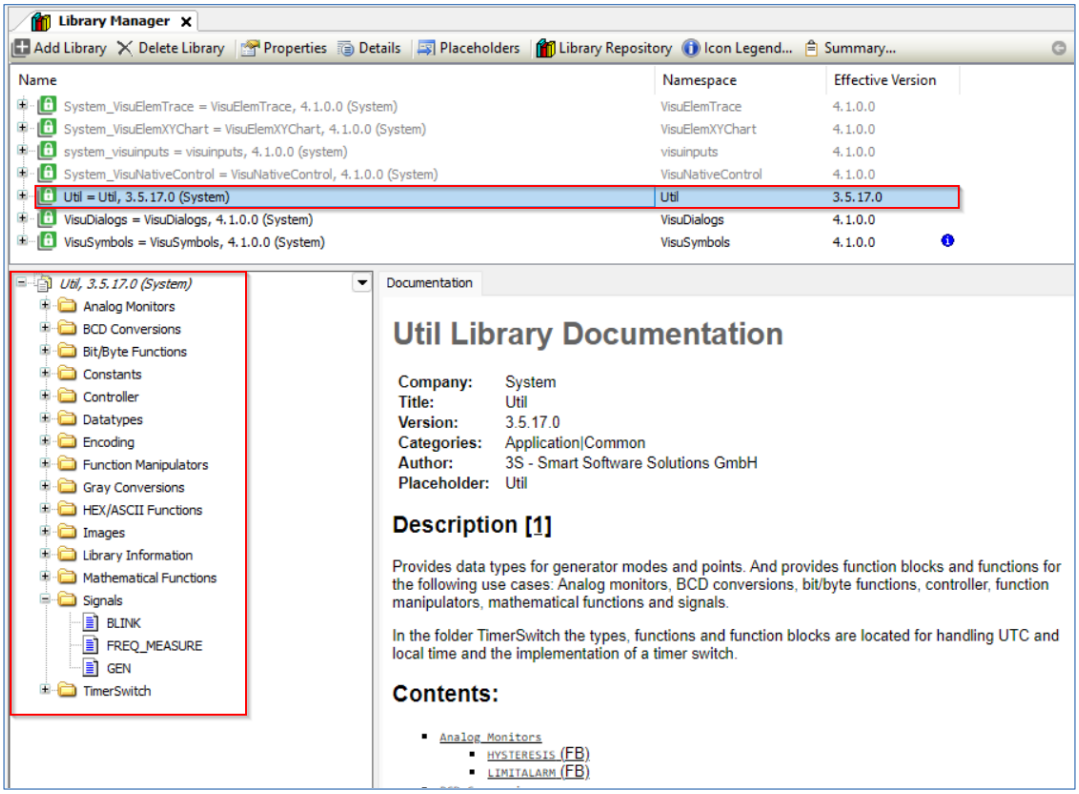


Fig. 5.10.3 Library objects and built-in help in the library manager

To use a library object in a project, its namespace must be specified. In the library manager, namespaces are displayed in the ***Namespace*** column. E.g., you need to declare an instance of the ***GEN*** function block from the ***Util*** library, which has the ***Util*** namespace (Fig. 5.10.3). This can be done as follows:

Example 5.10.1

```
PROGRAM PLC_PRG
VAR
	fbGen: Util.GEN;
END_VAR
```

We have already seen the method of accessing objects “through a point” (e.g., when we discussed structures, unions, and enumerations). Indeed, all of these objects (as well as function blocks, programs, lists of global variables, and some other objects in the project tree) have their own namespaces, which cover their nested objects.

Namespaces make it easy to determine the ownership of an object and to use different objects with the same names. For example, many libraries have an ***ERROR*** enumeration with error codes, and each library has its own codes. Using namespaces avoids confusion, it will always be clear which library a particular instance of the enumeration belongs to.

Example 5.10.2

```
	// The example can only be compiled when the CAA File and CAA SerialCom libraries
	// are added.
PROGRAM PLC_PRG
VAR
	// Instance of the ERROR enumeration from the CAA File library
	eFileError: FILE.ERROR;

	// Instance of the ERROR enumeration from the CAA SerialCom library
	eComError: COM.ERROR;
END_VAR
```

Back to the ***GEN*** block from the Util library. This block is used to generate various signals: sine waves, cosine waves, sawtooth waves, etc. The signal type is determined by the ***MODE*** input value, which has the ***GEN_MODE*** type – this is an enumeration from the same library.

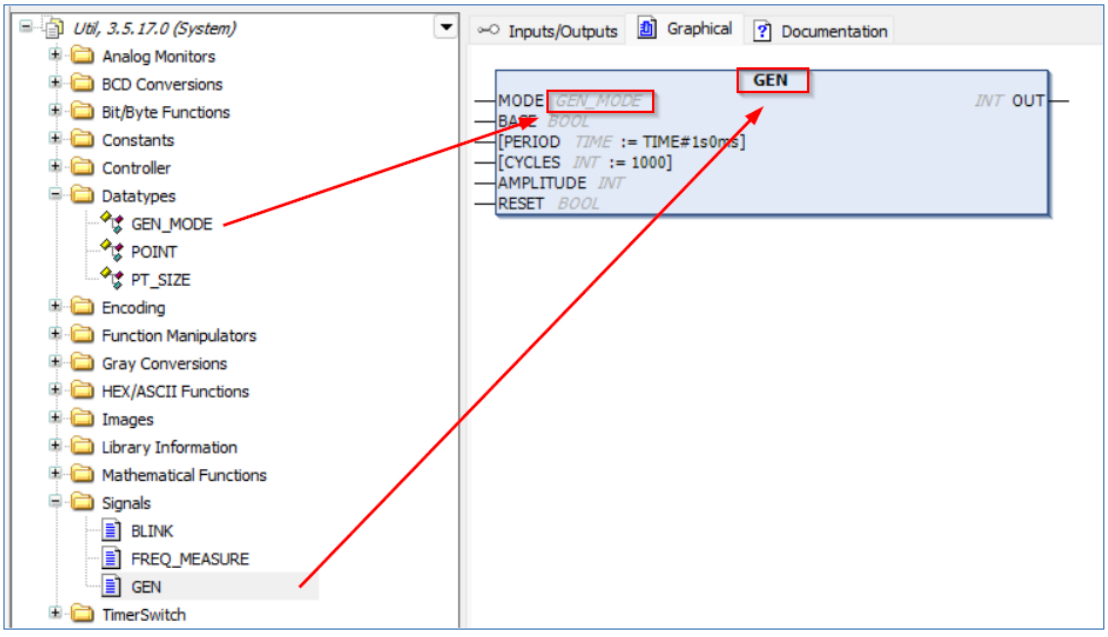


Fig. 5.10.4 Enumeration GEN_MODE in Util library

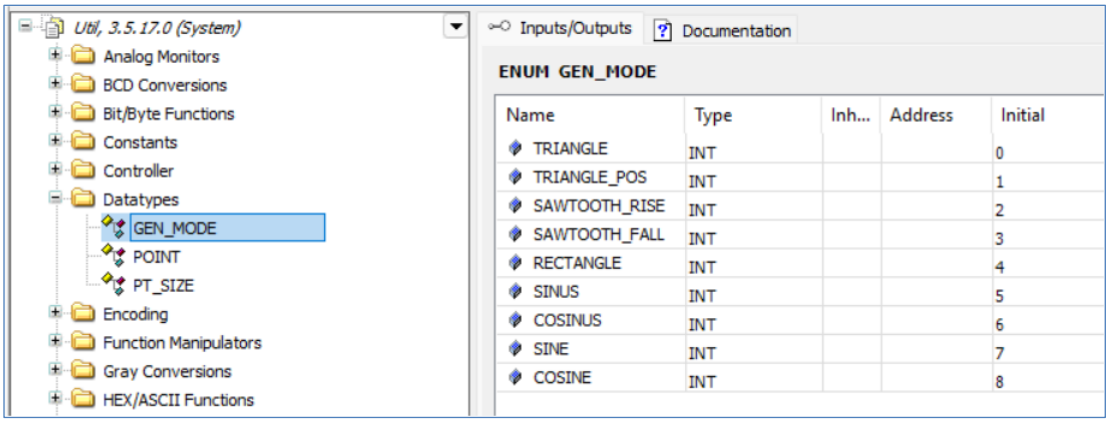


Fig. 5.10.5 Elements of GEN_MODE enumeration

When calling an instance of the ***GEN*** block in the program, the ***MODE*** input is assigned a value as follows:

Example 5.10.3

```
PROGRAM PLC_PRG
VAR
	fbGen: Util.GEN;
END_VAR

// code area

fbGen
(
	MODE := Util.GEN_MODE.SINUS,
	BASE := TRUE,
	PERIOD := T#5S,
	AMPLITUDE := 10
);
```

In this case, when accessing an element of the library enumeration, the library namespace is specified first, followed by the enumeration namespace.

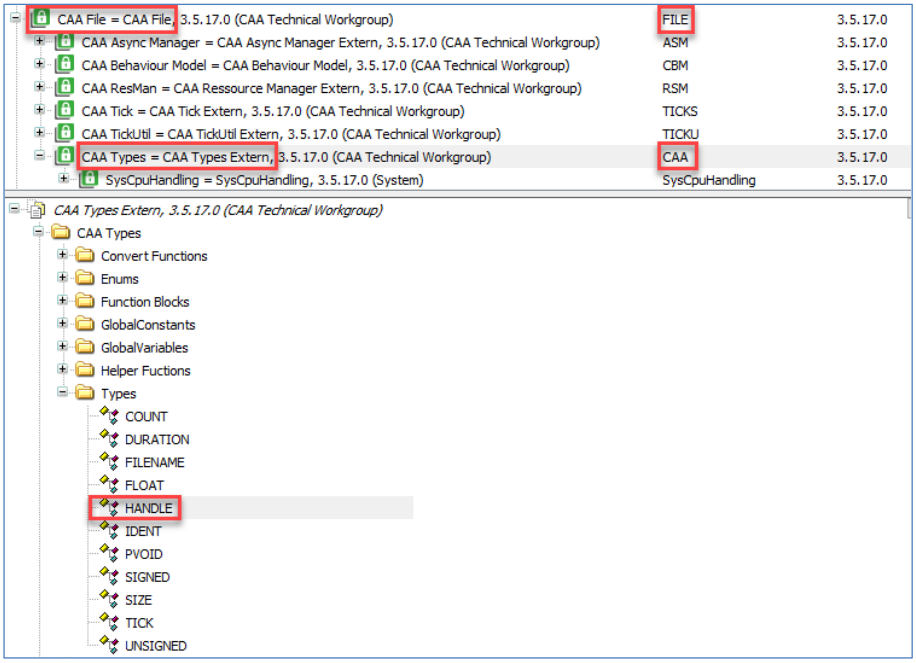


Fig. 5.10.6 Alias HANDLE from CAA Types library, embedded in CAA File library

Libraries can include nested libraries. To access objects in nested libraries, the namespaces of all libraries in the chain must be specified sequentially (Fig. 5.10.6).

Example 5.10.4

```
PROGRAM PLC_PRG
VAR
	hFile: FILE.CAA.HANDLE;
END_VAR
```

Standard library objects (Sect. 5.10.4) are usually used without a namespace, since their names are defined in IEC 61131-3 and it is difficult to confuse these blocks with any others.

5.10.4 Standard library

The most frequently used library is ***Standard*** library. This library contains basic blocks defined in IEC 61131-3 and used in almost any project: edge detectors, counters, timers, etc. Below are examples of how to use the most popular blocks.

1. **Front detectors**

The `R_TRIG` functional block is a rising edge detector. When its `CLK` input changes its value from `FALSE` to `TRUE`, a single pulse is generated at the `Q` output.

The `F_TRIG` block works in a similar way but tracks the signal fall: a single pulse is generated at its output `Q` when the `CLK` input changes its value from `TRUE` to `FALSE`.

These blocks are used in cases where it is necessary to perform certain operations once when the value of a logical variable changes, for example, to send an SMS when an alarm occurs.



Fig. 5.10.7 Inputs and outputs of R_TRIG and F_TRIG blocks

Example 5.10.5

```
PROGRAM PLC_PRG
VAR
	xAlarm: BOOL;
	fbAlarmTrig: R_TRIG;
END_VAR

// code area

fbAlarmTrig(CLK := xAlarm);
IF fbAlarmTrig.Q THEN

	// place for code to send alarm message

END_IF
```

2. **Pulse counters**

Standard library includes three pulse counter blocks:

- ***CTU*** – incremental counter (its value increases by 1 with each pulse)
- ***CTD*** – decrement counter (its value decreases by 1 with each pulse)
- ***CTUD*** – counter that can operate in both increment and decrement modes

Principle of operation of counters on the example of a ***CTU*** unit:

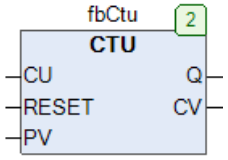


Fig. 5.10.8 CTU block inputs and outputs

`CU` is a count input to which a `BOOL` type variable is linked. On the rising edge of this variable, the `CV` output (`WORD` type), which represents the counter value, increases by one. The `PV` input sets the threshold value: when the value at the `CV` output equals the value at the `PV` input, the `Q` output („threshold reached" signal) will take the value `TRUE`. To reset the counter, set the `RESET` input to `TRUE`.

Example 5.10.6

```
PROGRAM PLC_PRG
VAR
	xDi: BOOL;
	wCounter: WORD;
	fbCtu: CTU;
END_VAR

// code area

fbCtu
(
	CU := xDi,
	CV => wCounter
);
```

The upper limit of the `WORD` type is 65535. Once this limit is reached, the counter will overflow and start counting from zero. If you need a larger range for the signals you are counting, use the `LCTU` block from Standard64 library. Its upper limit is 2^{64} - 1.

3. Timers

Standard library includes three timer blocks:

- `TP` – signal generator with specified duration
- `TON` – switch-on delay timer
- `TOF` – switch-off delay timer

`TON` is the most commonly used timer.

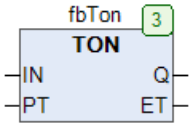


Fig. 5.10.9 TON block inputs and outputs

A time setting is applied to the `PT` input of the `TIME` type. The timer starts on the rising edge at the `IN` input. While the timer is running, the `IN` input must remain `TRUE`, otherwise the block will be stopped, and the time count will restart the next time it is started. The `ET` output returns the time elapsed since the timer was started. When the `ET` output becomes equal to the `PT` input, the `Q` output (timer trigger signal) will take `TRUE`.

Below is an example of using a timer to block contact bounce when pressing a button.

Example 5.10.7

```
PROGRAM PLC_PRG
VAR
	xButton: BOOL;
	fbDelay: TON;
END_VAR

// code area

fbDelay
(
	IN := xButton,
	PT := T#200ms
);
IF fbDelay.Q THEN

	// button click processing code

END_IF
```

Another example of using the **TON** timer is single pulse generation. For this, the inverted value of the timer's output must be fed to its input:

Example 5.10.8

```
PROGRAM PLC_PRG
VAR
	fbPoll: TON;
END_VAR

// code area

fbPoll
(
	IN := NOT(fbPoll.Q),
	PT := T#3s
);
	// A single pulse will be generated at output Q every 3 seconds.
IF fbPoll.Q THEN

	// actions to be performed every 3 seconds

END_IF
```

Timer calls in CODESYS are non-blocking, i.e. while the timer is running, all code placed after its call will continue to be executed every task cycle.

4. **Strings functions**

Standard library includes 9 functions for string (Sect. 5.3.4) processing:

- **CONCAT** – concatenation (merging) of strings
- **DELETE** – deletion of a substring from a string
- **FIND** – search for the first match of a substring in a string
- **INSERT** – insertion of a substring into a string
- **LEFT** – extracting a specified number of characters from the beginning of a string
- **LEN** – determining the length of a string
- **MID** – extracting a specified number of characters from a specified position in a string

- **REPLACE** – replacing a substring in a string
- **RIGHT** – extracting a specified number of characters from the end of a string

All these functions are intended to work with strings of type **STRING** (255 characters). For strings of **WSTRING** type, use similar functions from ***Standard64*** library with the prefix W (**WCONCAT**, **WDELETE**, etc.). To work with strings longer than 255 characters, use functions from ***StringUtils*** library.

Example 5.10.9

```
PROGRAM PLC_PRG
VAR
	sVar1: STRING := 'Hello, ';
	sVar2: STRING := 'world';
	sConcat: STRING;
	iLen: INT;
	sMid: STRING;
END_VAR

// code area

	// sConcat takes the value 'Hello, world'
sConcat := CONCAT(sVar1, sVar2);

	// iLen will receive the value 12
	// (the number of characters in the string ‘Hello, world’)
iLen := LEN(sConcat);

	// sMid will receive the value ‘ello’
	// (copying 4 characters of the string, starting from the 2nd)
sMid := MID(STR := sConcat, LEN := 4, POS := 2);
```

6. Process control project example

We will consider creating a process control project. The complete project, created in this section, is available at the *link*.

6.1 Task definition

The system contains 8 measuring points for certain parameters (temperature, pressure, etc.). It is necessary to organize:

- reading the values of these parameters
- checking whether the parameter values fall within the specified range, generating alarms and activating alarm indicators if the values exceed the range limits
- displaying these values, alarm signals, and alarm history in the visualization
- transmitting the values and alarm signals to the SCADA system
- transmitting the values to akYtec Cloud service

An analog input module MV210-101 is used to obtain analog signals.

A discrete output module MU210-401 is used for alarm indication (signal lamps, audible alarms, etc. can be connected to its outputs).

The module communication protocol is ***Modbus TCP***.

By now, you should have read Section 3 of the “Quick Start” guide and learned the basic steps required when creating a project (add components, connection to the controller, load project, etc.).

6.2 Mx210 module configuration

If you have the MV210-101 and MU210-401 modules used in the example, you need to configure their network settings: IP address, subnet mask, and gateway.

First, download and install the ***akYtecToolPro*** utility, which is used to configure the modules.

Then connect the module to your PC via the MicroUSB interface and launch ***akYtecToolPro***. Click the ***Add Devices*** button, select the interface with a name containing “***Virtual COM Port***”, and select the ***Find Device*** mode (with address 1). Click the ***Find*** button. Once the device is detected, select it with a check mark and click the ***Add Devices*** button at the bottom of the window to open the device settings (Fig. 6.2.1).

In the module settings, go to the section ***Network Settings*** > ***Ethernet Settings*** and enter the values for ***IP address***, ***Subnet mask***, and ***Gateway***. These settings must match the settings of the network interface of the controller through which the modules will be accessed. Assumed, the controller has the following settings:

- IP address: 10.2.11.174
- Subnet mask: 255.255.0.0
- Gateway: 10.2.1.1

The modules can be configured as follows:

- MV210-101 IP address: 10.2.11.120
- MU210-401 IP address: 10.2.11.121
- Subnet mask: 255.255.0.0
- Gateway: 10.2.1.1

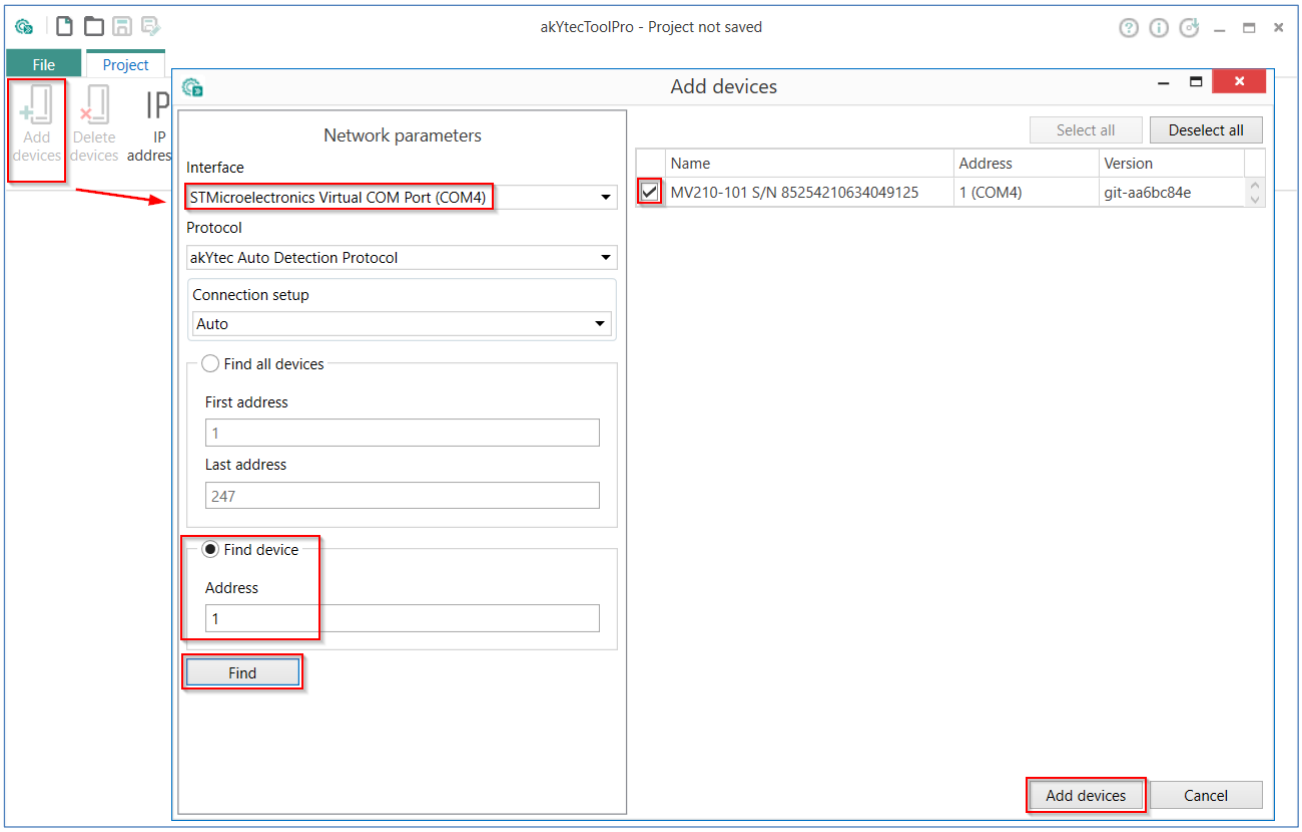


Fig. 6.2.1 Connection to Mx210 module in akYtecToolPro

The subnet mask and gateway for the controller and modules must be the same, and the IP addresses must belong to the same subnet and be unique within it. If you connect the modules directly to the controller rather than to the local network, the gateway address is usually irrelevant.

All other module parameters (e.g., sensor types for analog channels of the MB210-101 module, etc.) do not need to be configured at this stage; you can set them in the CODESYS project when configuring the communication.

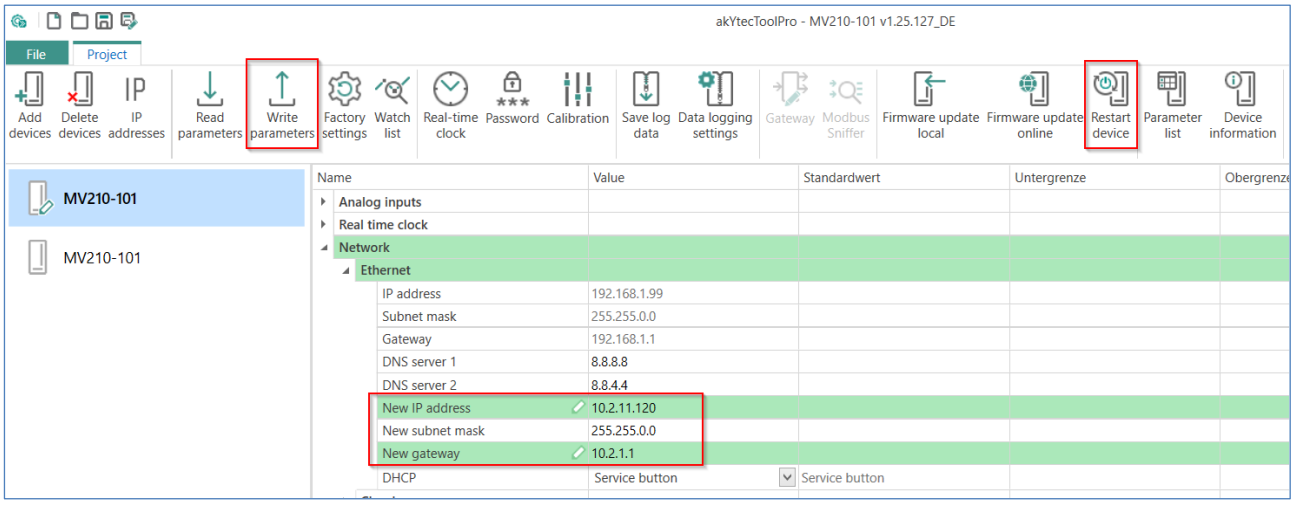


Fig. 6.2.2 Mx210 module configuration in akYtecToolPro

After changing the network settings, the module must be restarted using the ***Restart Device*** button or by powering it down and on again for the new settings to take effect.

6.3 Mx210 module polling configuration using templates

6.3.1 New project creation and connection to controller

Create a new CODESYS project named “MonitoringSystem” based on the template of the SPC210 touch panel controller. If you have a different controller, select its template.

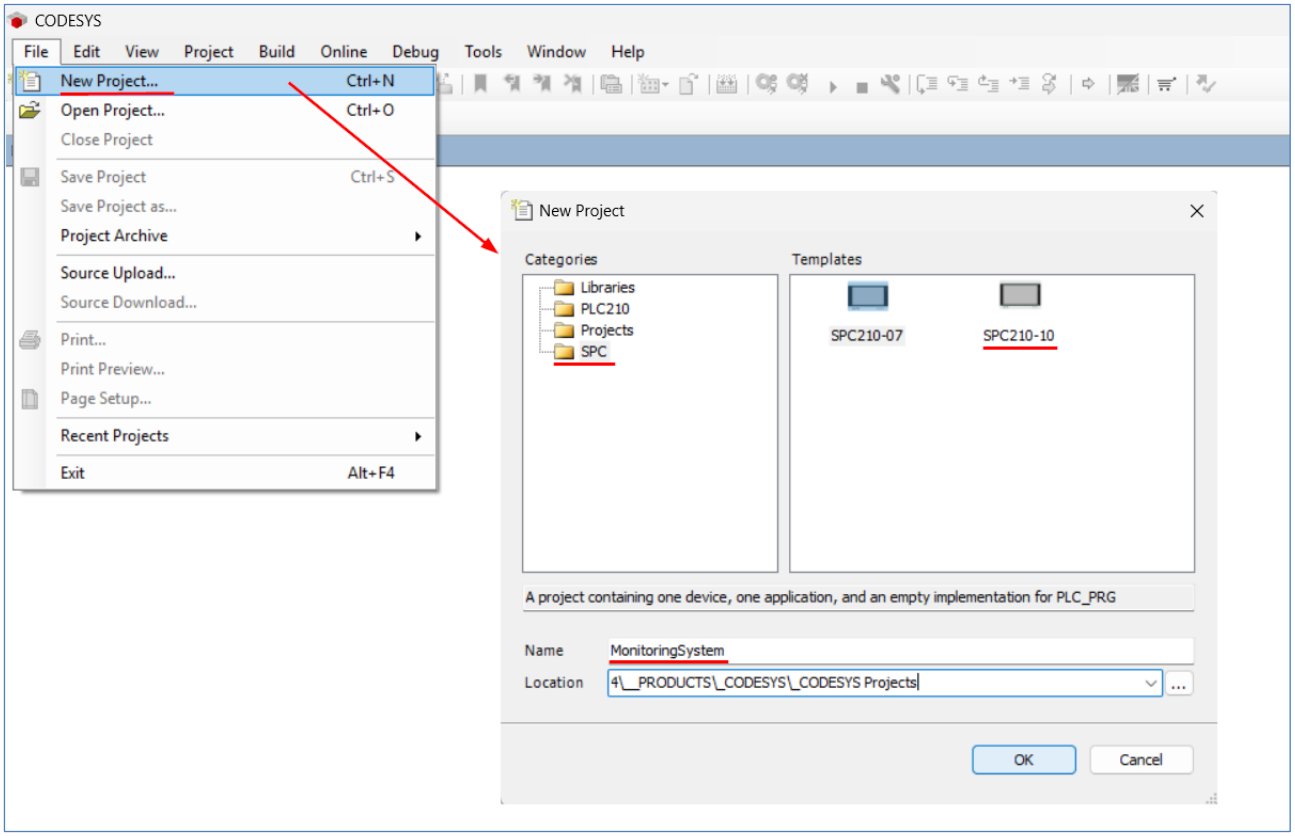


Fig. 6.3.1 New project from template

Remove the ***Modbus COM*** components from the project tree – in our project, the modules will be polled via the Ethernet interface, so there is no need for these components.

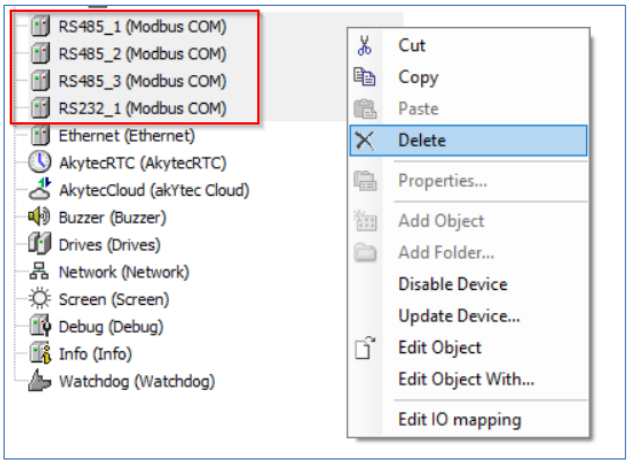


Fig. 6.3.2 Deletion of Modbus COM components

Double-click on the ***Device*** node and go to the ***Communication Settings*** tab. Run the ***Scan Network*** command and select your controller from the list.

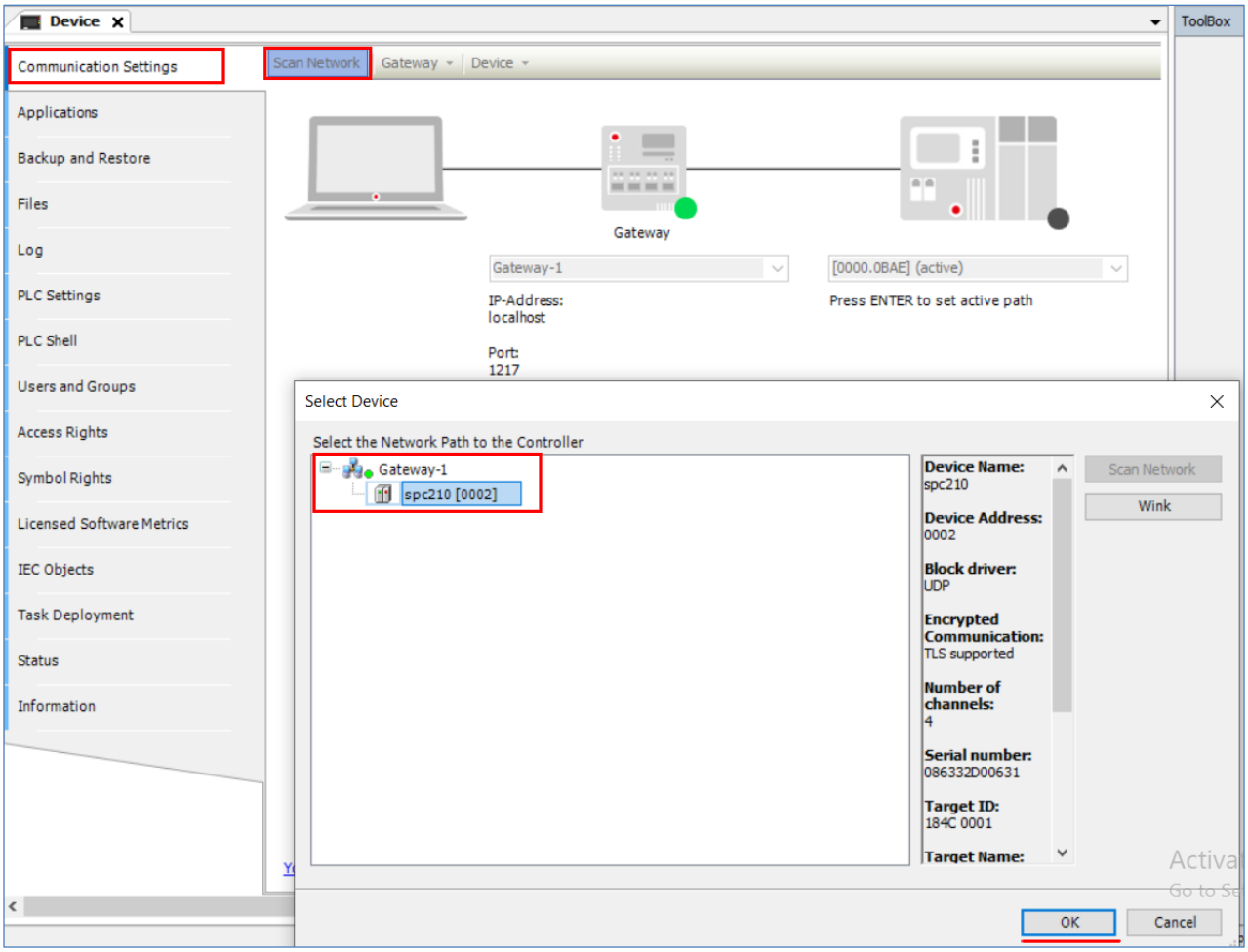


Fig. 6.3.3 Searching for controller using network scan

In some cases, the controller may not be detected during network scanning. E.g., if antivirus software is running on the PC and blocks the transmission of broadcast UDP packets or the reception of packets on ports 1740…1743. In this case, enter the IP address of the controller manually and press ***Enter***. The word "***active***" after the address appears automatically after the controller is detected.

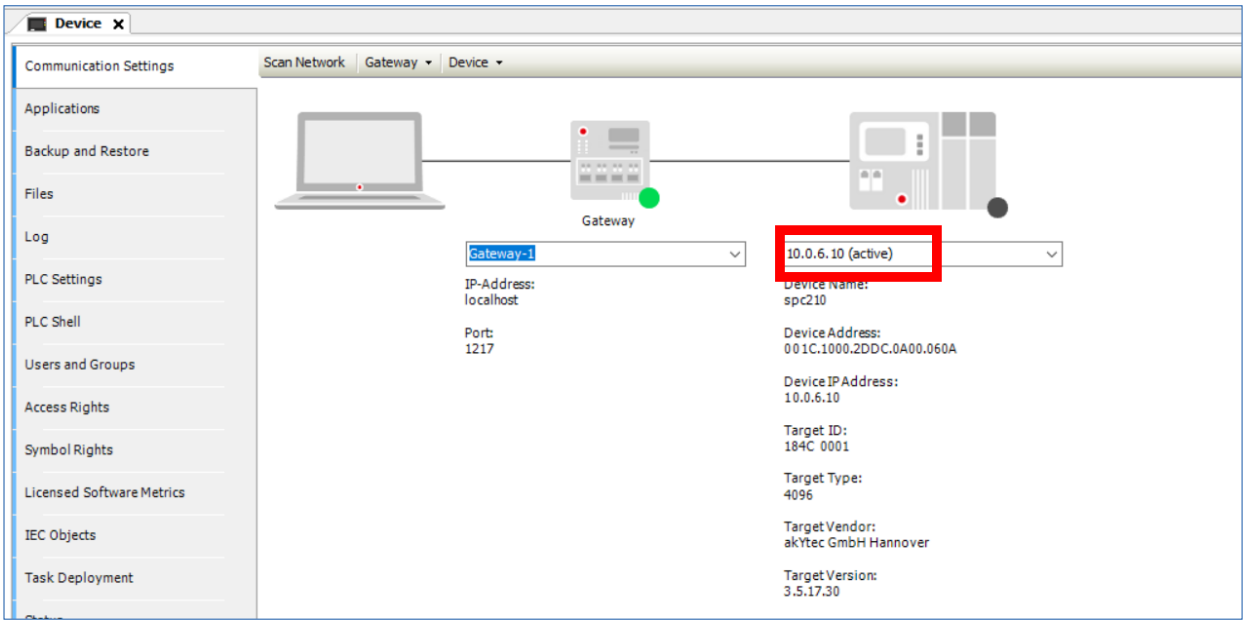


Fig. 6.3.4 Entering IP address of controller manually

If you do not have a real controller available, you can use a virtual controller. To do this, use the ***Update Device*** command from the ***Device*** node context menu and select the ***CODESYS Control Win V3*** target file. The target file version must match the CODESYS version you are using (for example, for environment version V3.5 SP17 Patch 3, use target file version 3.5.17.30).

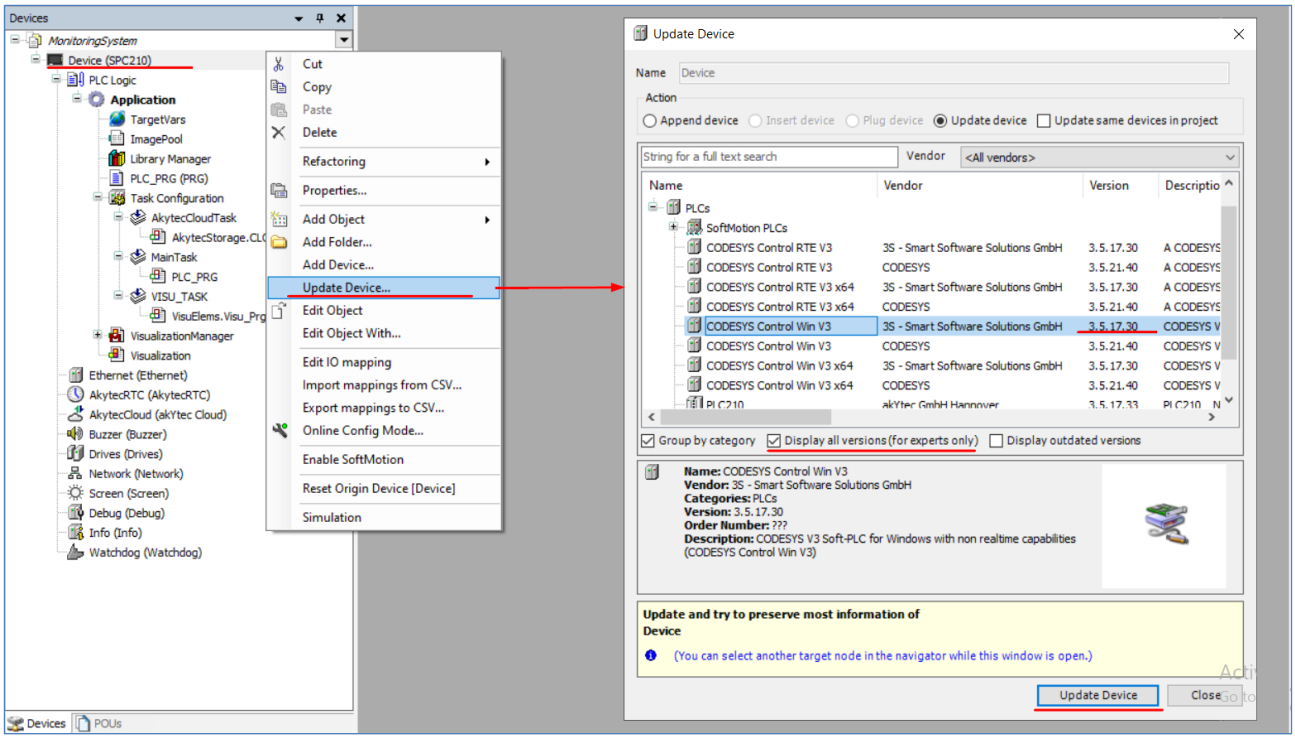


Fig. 6.3.5 Target file selection for CODESYS Control Win V3 virtual controller

After that, delete the ***AkytecCloudTask*** task, because you cannot connect a virtual controller to the ***AkytecCloud*** service. The presence of this task in the virtual controller project will lead to compilation errors.

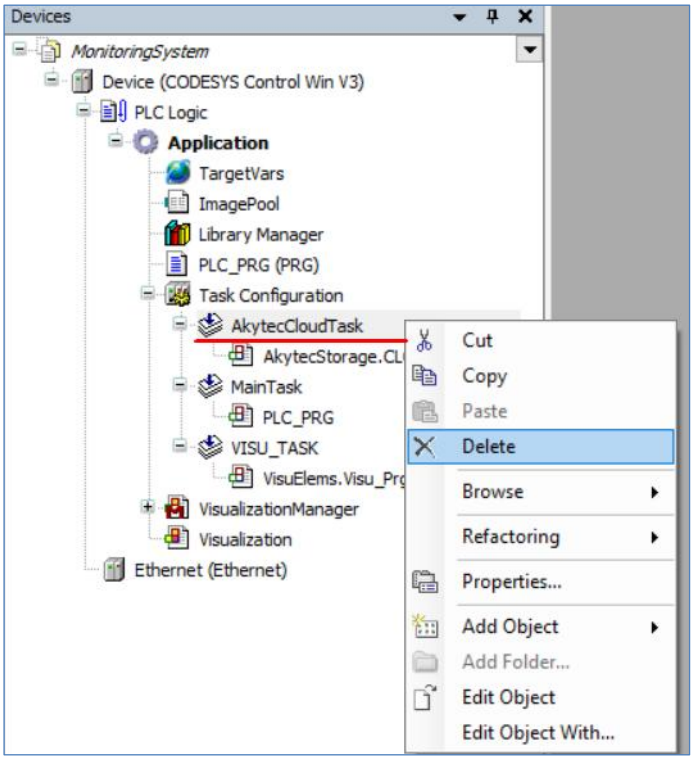


Fig. 6.3.6 Removing AkytecCloudTask task from the virtual controller project

To start the virtual controller, open the Windows system tray and select ***Start PLC*** from the context menu of the virtual controller icon (Fig. 6.3.7).

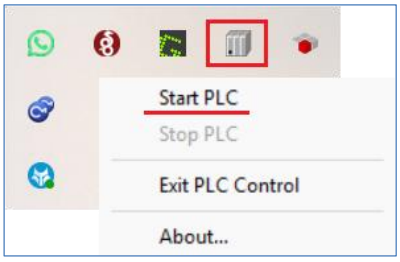


Fig. 6.3.7 Starting the virtual PLC

Now you can establish a connection to the virtual controller by scanning the network (Fig. 6.3.3).

6.3.2 Mx210 template pack installation

We will configure the exchange with modules using ready-made templates, which significantly simplifies the configuration process. You can download the Mx210 template package here.

The package is installed using the ***CODESYS Installer*** utility (***Tools*** > ***CODESYS Installer***) (Sect. 2.2).

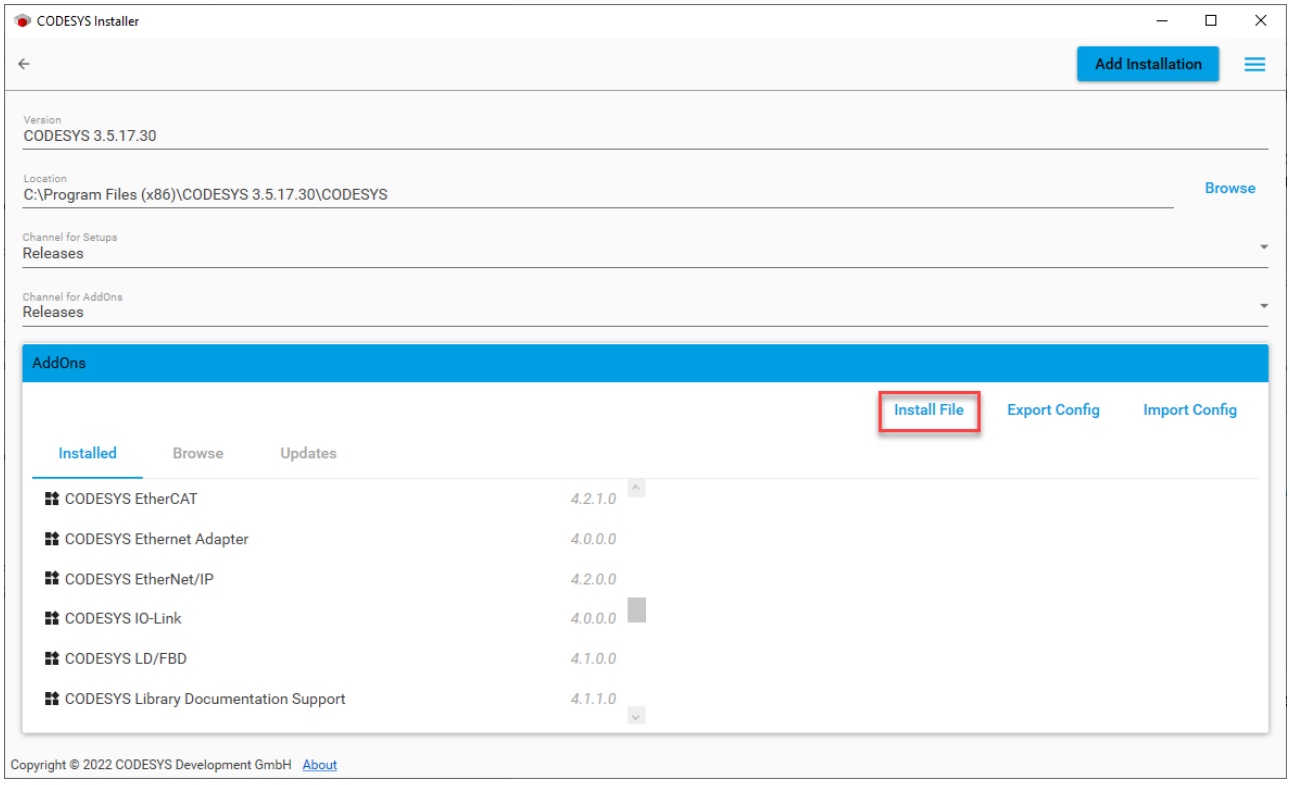


Fig. 6.3.8 Package installation button

6.3.3 Modbus component configuration

Open the Ethernet component and select the controller network interface that will be used to communicate with the Mx210 modules. We have previously established the connection with the controller in order to obtain a list of its interfaces.

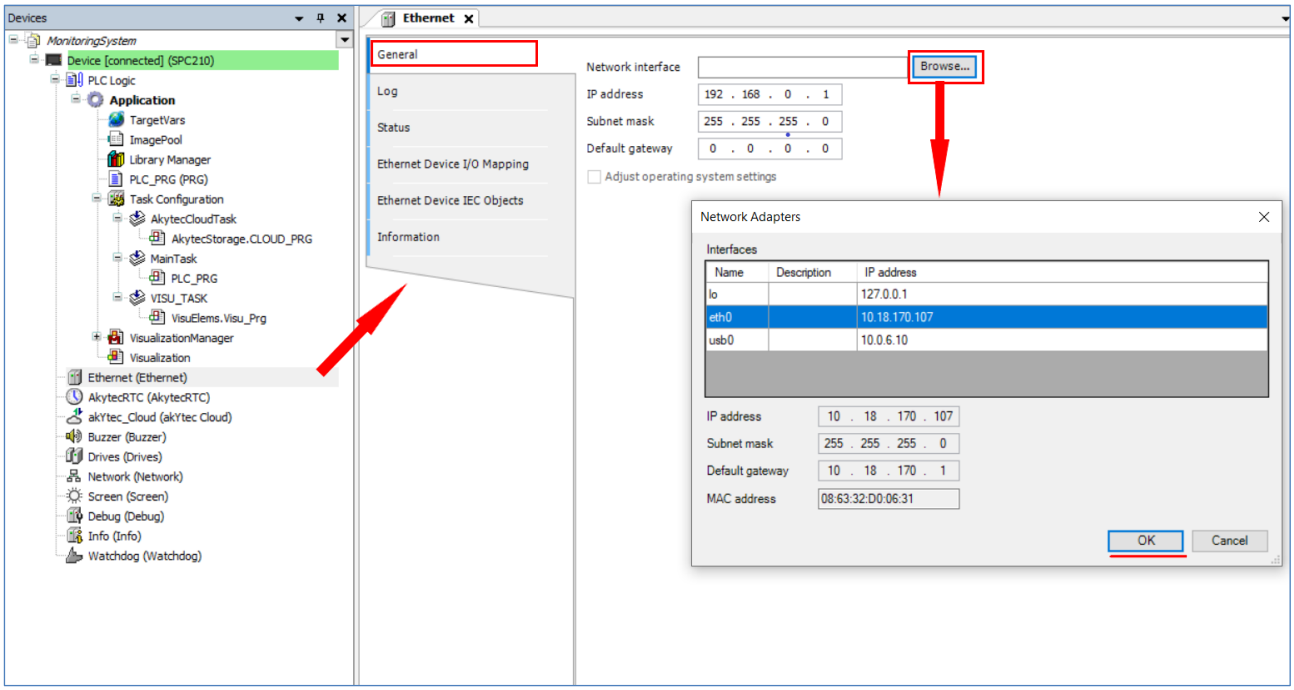


Fig. 6.3.9 Network interface selection in Ethernet component

Add the ***Modbus TCP Master*** component to the project using the ***Add Device*** command from the ***Ethernet*** component context menu.

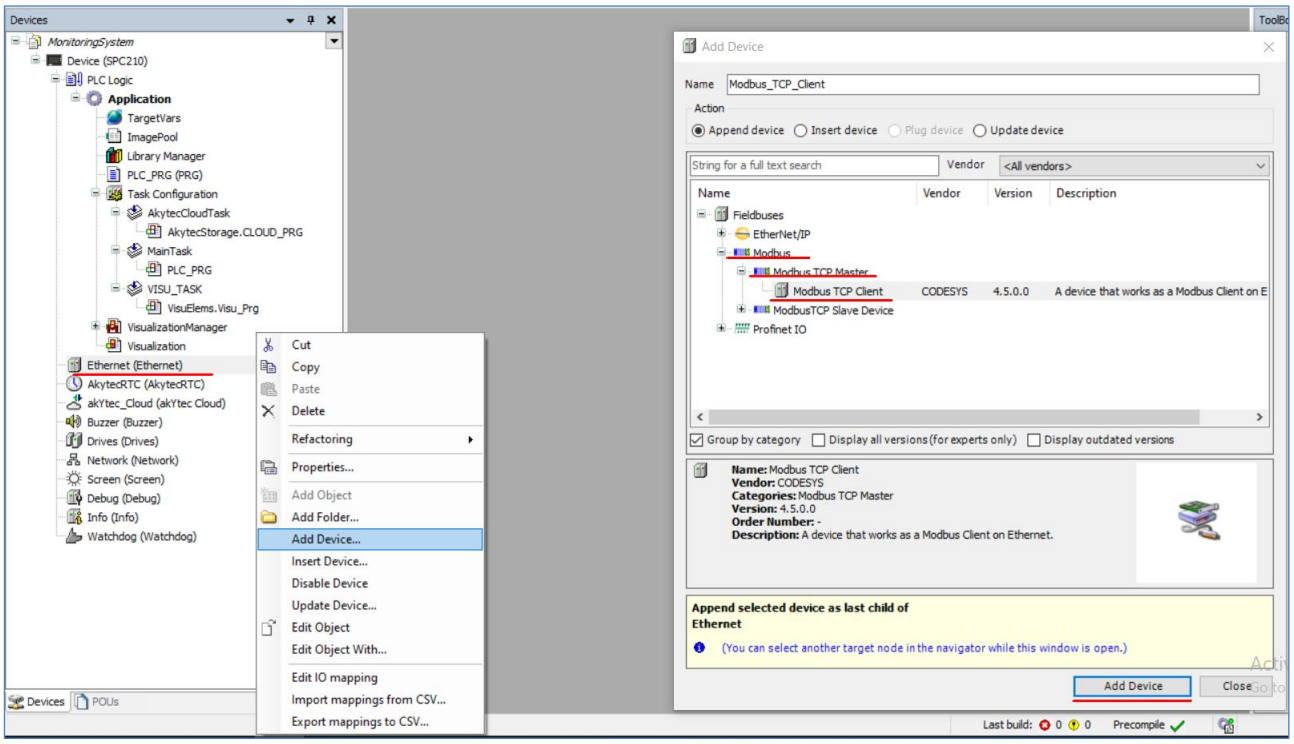


Fig. 6.3.10 Add Modbus TCP Master

In the component settings on the ***General*** tab, mark the ***Auto-reconnect*** checkbox. If it is activated, in the event of communication errors, the system will not stop polling the modules but will attempt to re-establish communication with them.

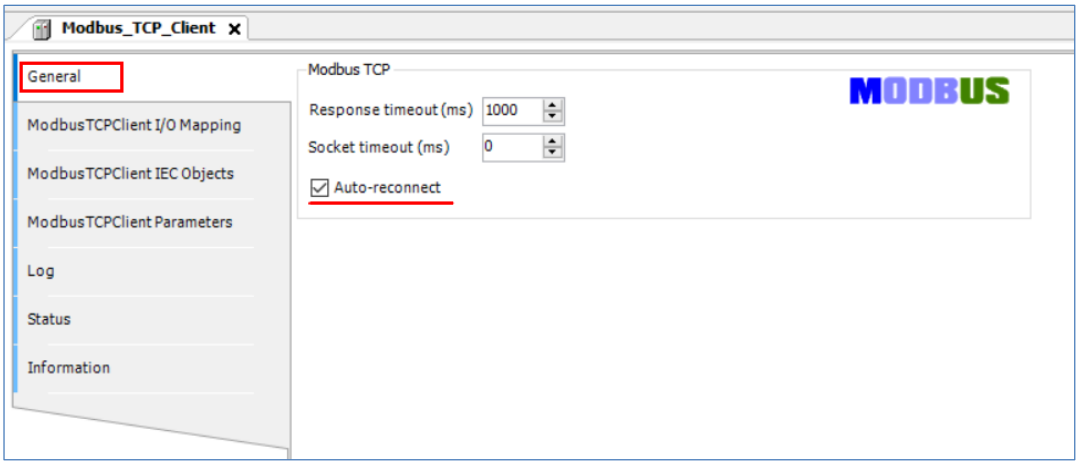


Fig. 6.3.11 Modbus TCP Master settings

To add the MB210-101 and MU210-401 module templates to the project, use the ***Add Device*** command from the ***Modbus TCP Master*** component context menu. If the templates are not displayed, it means that their package has not been installed yet (Fig. 6.3.8).

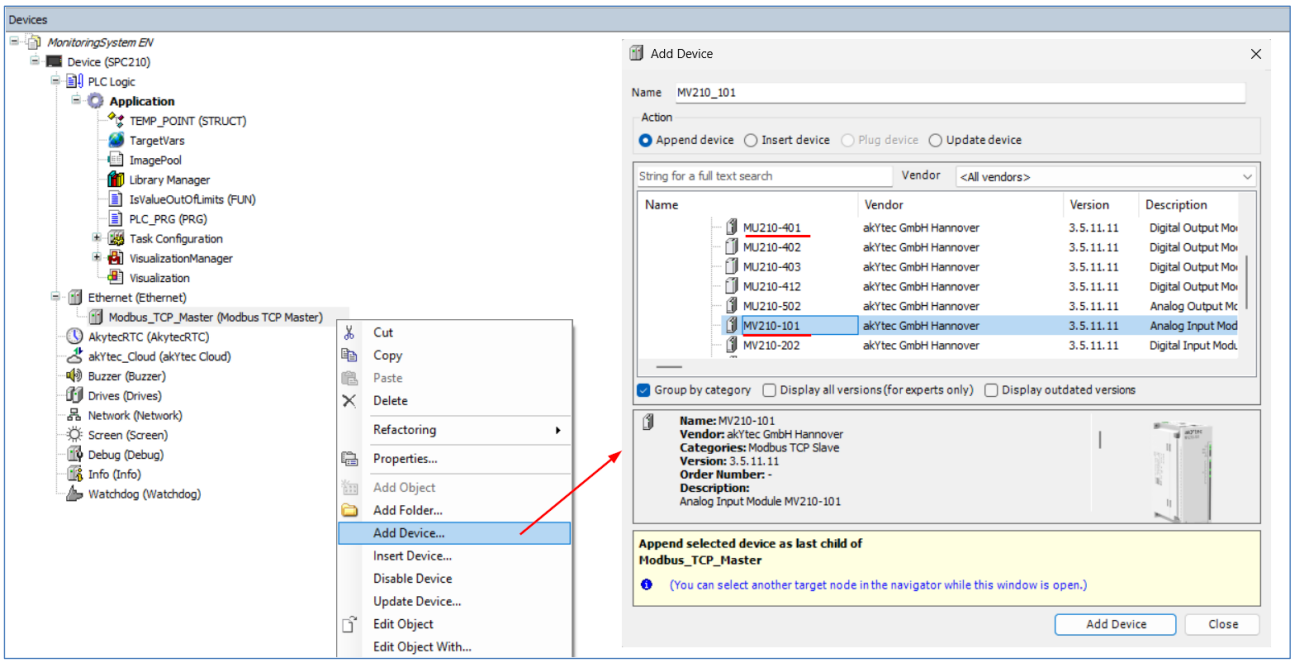


Fig. 6.3.12 Add MV210-101 and MU210-401 templates

In the settings for the added modules, on the General tab, the IP addresses of the modules must be specified. There are several options:

- If you use real modules, specify the IP addresses that you assigned to them in ***akYtecToolPro***.
- If you are emulating modules connected to a real controller, specify the IP address of the PC network interface to which the controller is connected for both modules (Fig. 6.3.13). Note that your PC network interface address is different.
- If you are emulating modules connected to the CODESYS Control Win V3 virtual controller (Sect. 6.3.1), specify the IP address ***127.0.0.1 (localhost)*** for both modules.

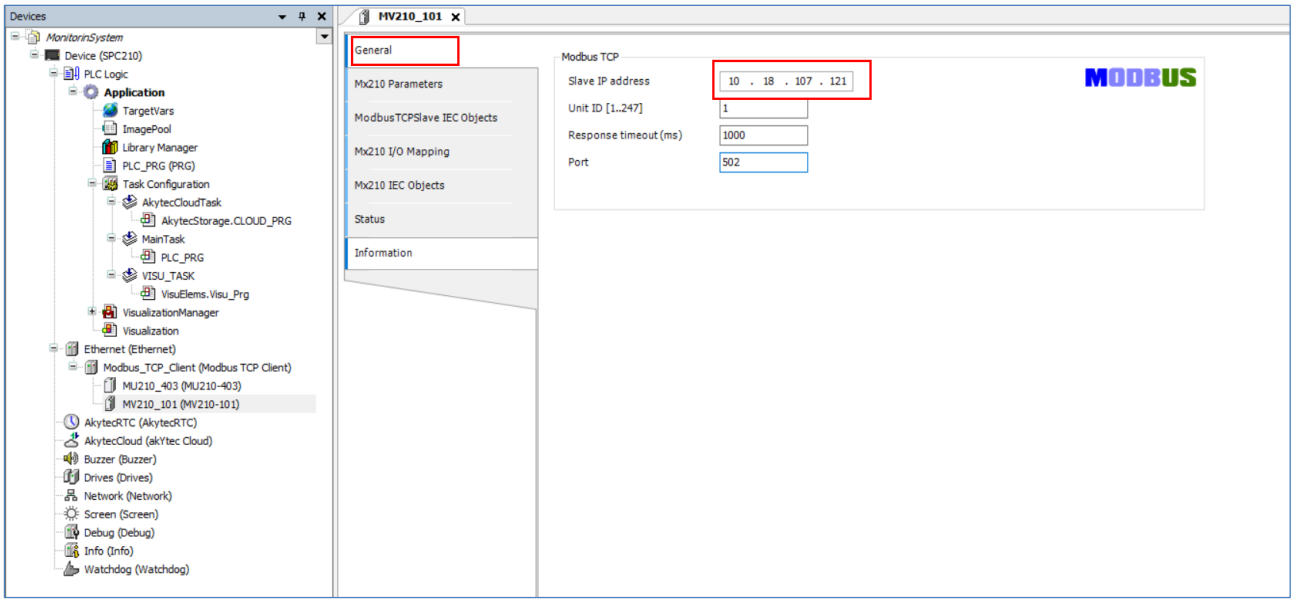


Fig. 6.3.13 IP address of MB210-101

The template contains two tabs with module parameters: ***Parameters*** and ***I/O Mapping***. The ***Parameters*** tab of the MB210-101 template is used to set the module's analog input settings: sensor type, filter band, etc. Since the ***Use configuration settings*** parameter is set to ***TRUE*** by default, the settings specified here will be written to the module when the project is loaded. If you have already specified all these settings using the ***akYtecToolPro*** utility, set the ***Use configuration settings*** parameter to ***FALSE*** so that your settings are not overwritten by the settings from this tab.

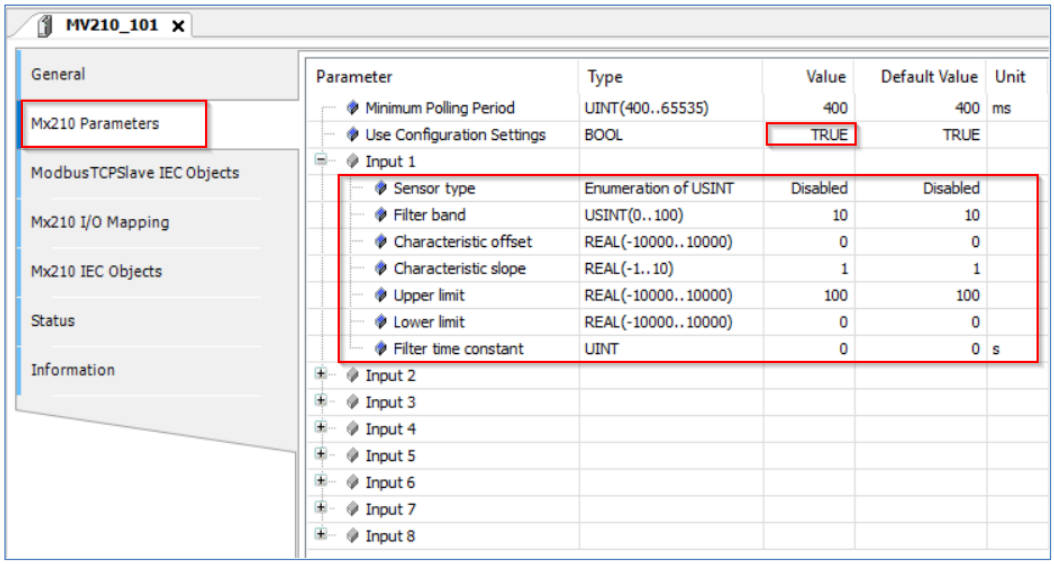


Fig. 6.3.14 "Parameters" tab of MV210-101 module template

The ***I/O Mapping*** tab contains the channel parameters of the module: analog input values, their statuses, etc. Variables used in the project code or visualization are linked to these channels. In our project, the variables have not yet been declared, so the linking will be done in the following steps.

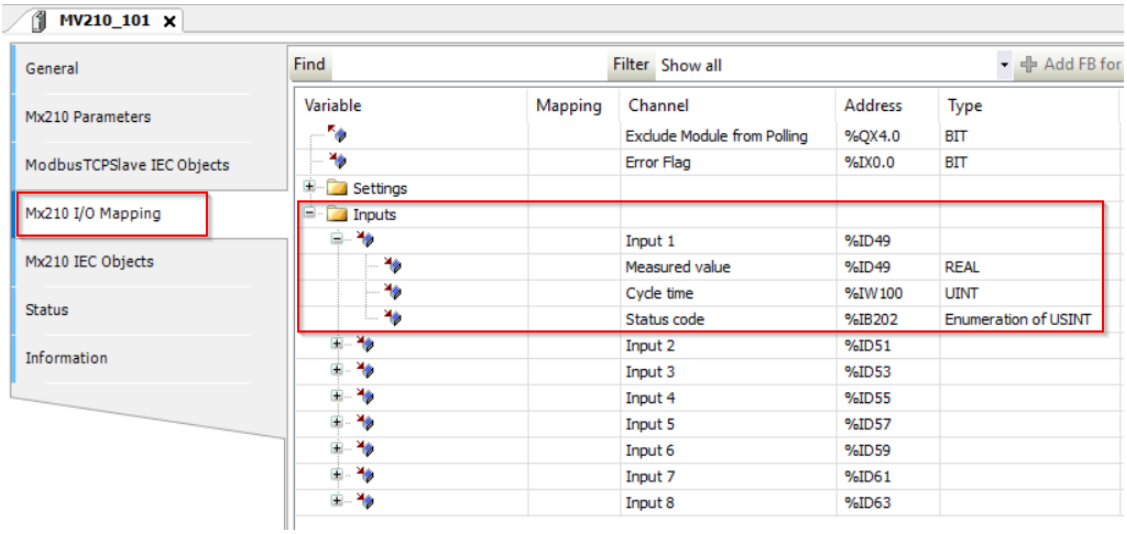


Fig. 6.3.15 "I/O Mapping" tab of MV210-101 module template

However, at this stage, we can already test communication with the modules (Sect. 6.3.4).

6.3.4 Communication test

Download the application to the controller and run it using the ***Online > Login*** and ***Debug > Start*** commands in the menu or the and items on the toolbar respectively (Fig. 6.3.16, 6.3.17).

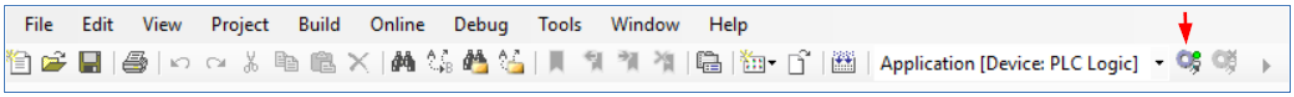


Fig. 6.3.16 Download the application into the controller

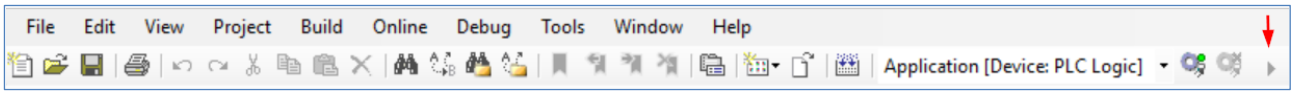


Fig. 6.3.17 Start the application

If the exchange is configured without errors and the physical connection between devices (PLC / I/O modules / PC, depending on the equipment used) is made correctly, the ***Modbus*** component icons in the project tree will light green (Fig. 6.3.18).

If you use real modules, you will receive signals from the sensors connected to the module inputs. If you use an OPC server, you need to enter the values manually.

If you open the ***I/O Mapping*** tab of the MB210-101 module template in CODESYS, you will see the number you entered, which the controller read from the OPC server (Fig. 6.3.19).

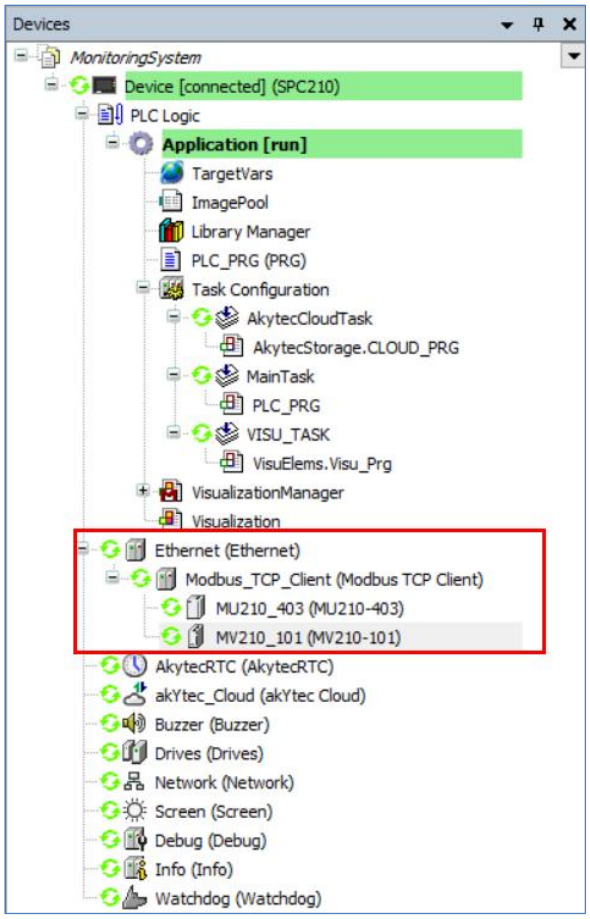


Fig. 6.3.18 Indication of successful data exchange

The ***Status code*** channel of the I/O Mapping tab is used to display sensor errors (e.g., sensor break, no connection to the ADC, etc.). The error code is transmitted in the most significant byte of the measured value. A list of error codes is provided in the module user manual.

When using real modules, you can simulate an error manually (e.g., by disabling the sensor).

6.4 Program development

Now we can write program code to process the values of the MV210-101 analog inputs and form the bitmask of the discrete outputs of the MU210-401 module.

6.4.1 Creating structure

Before starting to write code, the variables should be declared that will be used in this code. The measuring points are main objects in our project. The parameters required to describe a single point are:

- name of the measuring point displayed in the visualization (e.g., “HWS temperature”)
- measured value
- measurement error code
- measurement error text to be displayed in the visualization
- lower and upper limits of the acceptable range
- alarm signal when the value exceeds the acceptable range
- alarm counter

A good solution would be to create a structure that includes variables corresponding to all of the above parameters. Some parameters must be non-volatile – these are limit values (in our project, they will be set by

the operator in the visualization, so they cannot be constants) and the alarm counter. Therefore, it is reasonable to create two structures – one will include energy-dependent parameters (***MEASURE_POINT***), and the other – energy-independent ones (***MEASURE_POINT_MEM***).

To create a structure, select the ***Add Object > DUT > Structure*** command from the ***Application*** node context menu.

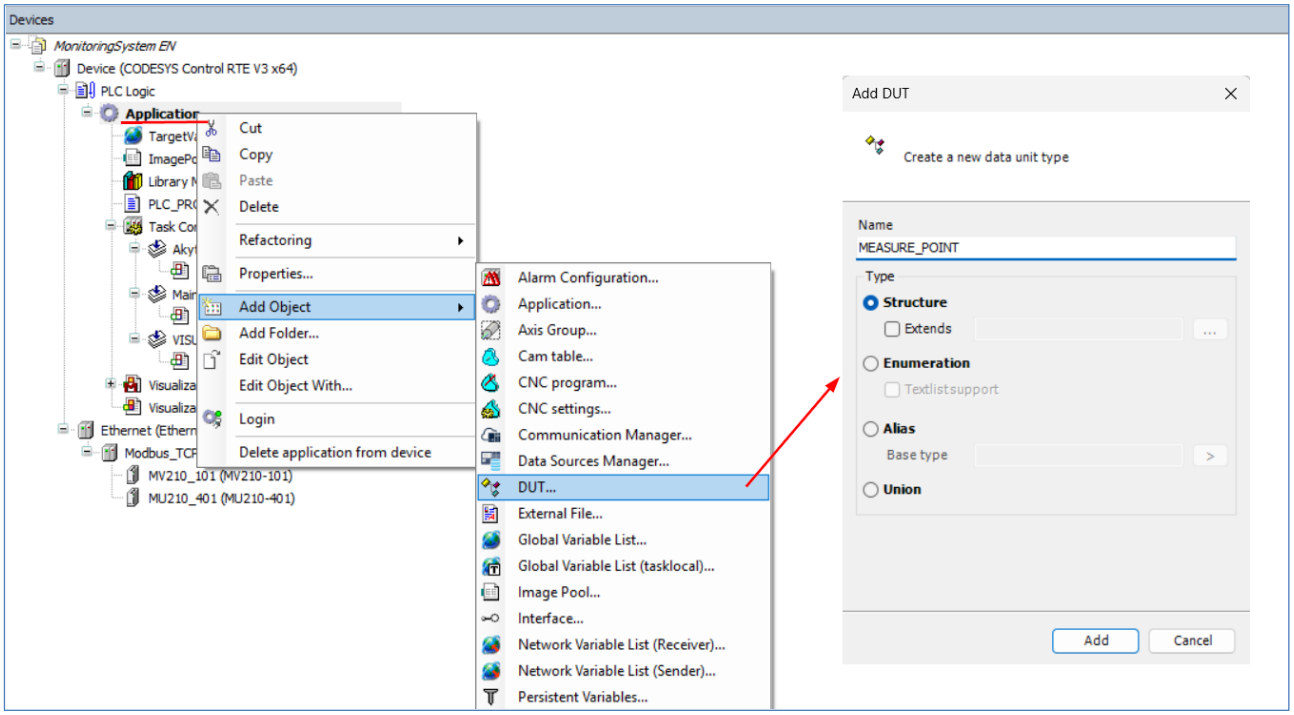


Fig. 6.4.1 Create structure

Example 6.4.1

```
      // Structure of operational parameters of a single measuring point
TYPE MEASURE_POINT :
STRUCT
            // Measuring point name
      wsPointName: WSTRING(20);
            // Current measured value
      rValue: REAL;
            // Analog input error code
      eStatus: Mx210Assistant.ANALOG_SENSOR_ERRORS;
            // Error description
      wsStatus: WSTRING;
            // Out-of-range flag
      xAlarm: BOOL;
END_STRUCT
END_TYPE
```

In the **`MEASURE_POINT`** structure, the **`ANALOG_SENSOR_ERRORS`** enumeration from the **`Mx210Assistant`** library is used for analog input error codes. This library is automatically added to the library manager when the MV210-101 module template is added.

Example 6.4.2

```
      // Structure of non-volatile parameters of a single measuring point
TYPE MEASURE_POINT_MEM :
STRUCT
            // Signal lower limit
```

```
        rLowAlarmValue: REAL := 10.0;
                // Signal upper limit
        rHighAlarmValue: REAL := 30.0;
                // Error counter instance
        fbAlarmCounter: CTU;
END_STRUCT
END_TYPE
```

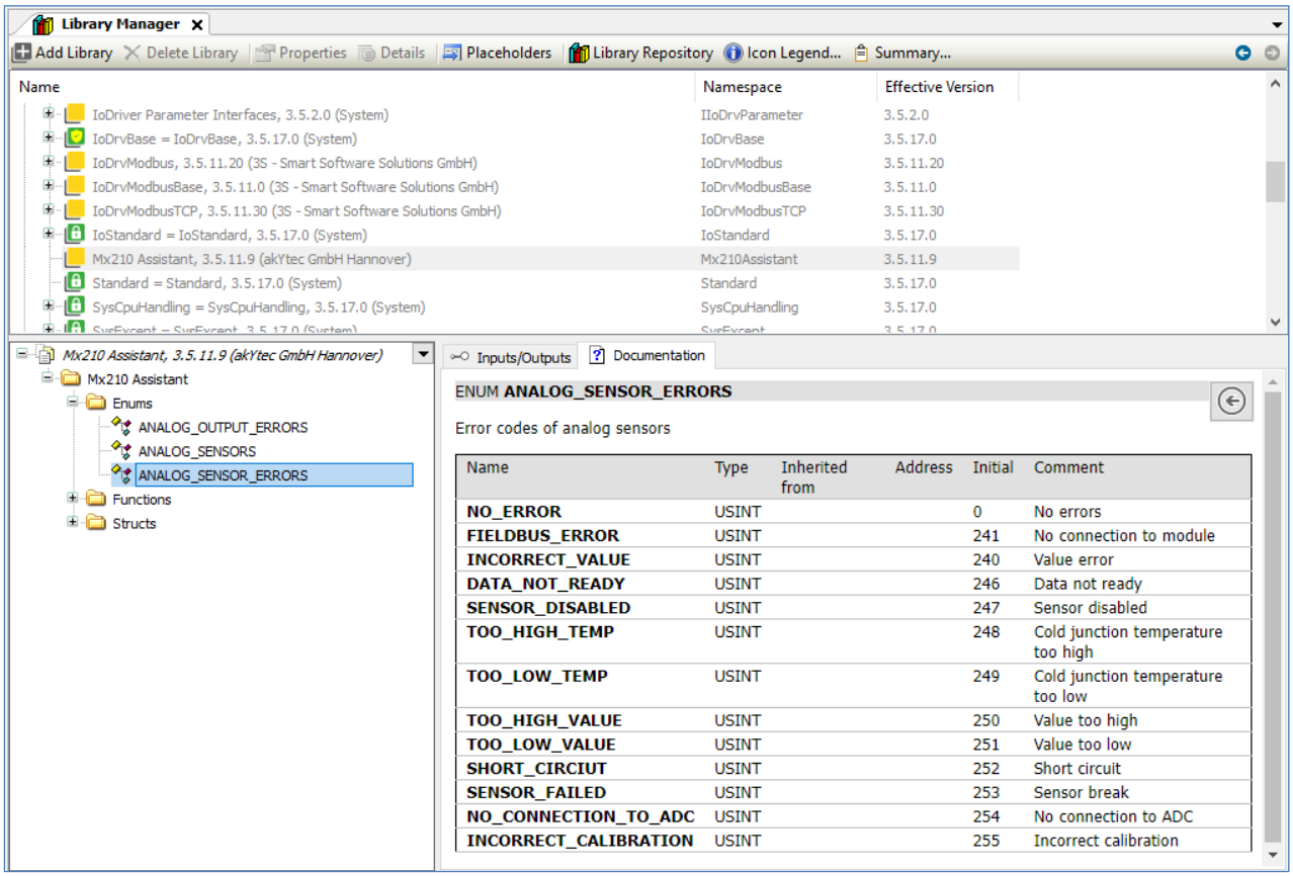


Fig. 6.4.2 List of elements in ANALOG_SENSOR_ERRORS enumeration of Mx210Assistant library

In the structure of non-volatile variables `MEASURE_POINT_MEM`, we declared an instance of the `CTU` function block from the ***Standard*** library. In general, storing instances of function blocks in non-volatile memory is usually quite expensive, since they can contain many variables. It is reasonable to declare in the block only those variables whose values must be saved after the controller is rebooted, then declare “copies” of these variables in non-volatile memory and write code that will synchronize the values of these non-volatile “copies” and the corresponding variables of the function block when the application starts.

However, to simplify our example, we stored the entire block instance in non-volatile memory. In addition, the `CTU` block instance occupies only 16 bytes of memory (this can be determined using the `XSIZEOF` operator (Sect. 5.6.10)) – this is quite small, and this approach is quite applicable.

After creating the structures, we can declare their instances in the existent `PLC_PRG` program.

The number of measuring points in different projects may vary. In our project, there are 8 (the number of inputs of MB210-101).

Therefore, we declare the arrays (Sect. 5.3.7) of our structures, using a constant as the upper limit – so we can easily adapt the project if the number of points changes. We will process the array in a `FOR` loop (Sect. 5.7.3), so we declare the variable `i` as the loop counter.

Example 6.4.3

```
PROGRAM PLC_PRG
VAR
	astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
	i: INT;
END_VAR
VAR RETAIN
	astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
	c_iMeasurePointsCount: INT := 8;
END_VAR
```

6.4.2 Assigning variables to Mx210 templates

The **MEASURE_POINT** structure created in the previous section contains the variables **rValue** (measured value) and **eStatus** (analog input error code), whose values are read from MB210-101. The **xAlarm** variable represents an error signal – these signals must be written to the MU210-401 module to control its discrete outputs (e.g., light and sound alarm devices can be connected to them).

These variables must be assigned to the template channels. Open the ***I/O Mapping*** tab of the MB210-101 template, double-click on the cell in the ***Variable*** column of the desired channel, and use the input wizard to select the corresponding structure variable (***rValue*** for the ***Measured value*** channel, ***eStatus*** for the ***Status Code*** channel):

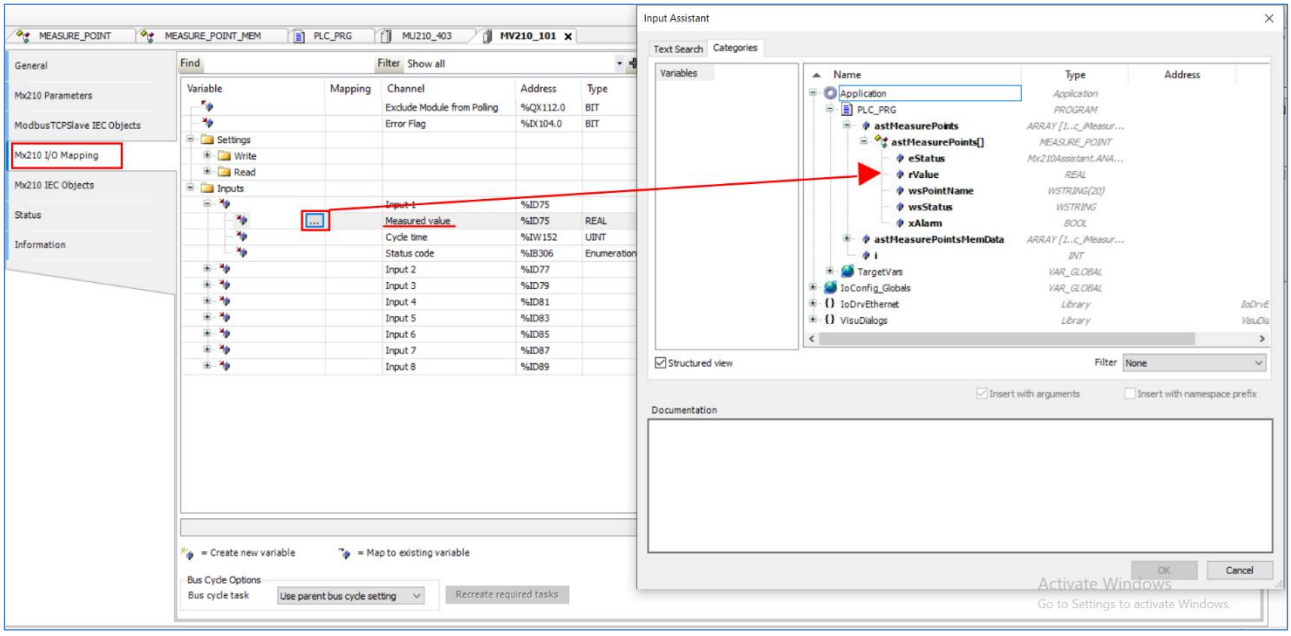


Fig. 6.4.3 Selecting variable to assign to channel

After clicking ***OK***, the path to the variable (including all necessary namespaces) will be written to the cell (Fig. 6.4.4).

Please note that by default, the array index is not specified in parentheses, you must enter it yourself – 1 for the first entry, 2 for the second, etc. (Fig. 6.4.5).

Assign variables to the ***Measured value*** and ***Status code*** channels of all inputs. Fig. 6.4.6 shows the assignment of variables to analog inputs 1 and 2. The only difference for the other inputs will be in the array index.

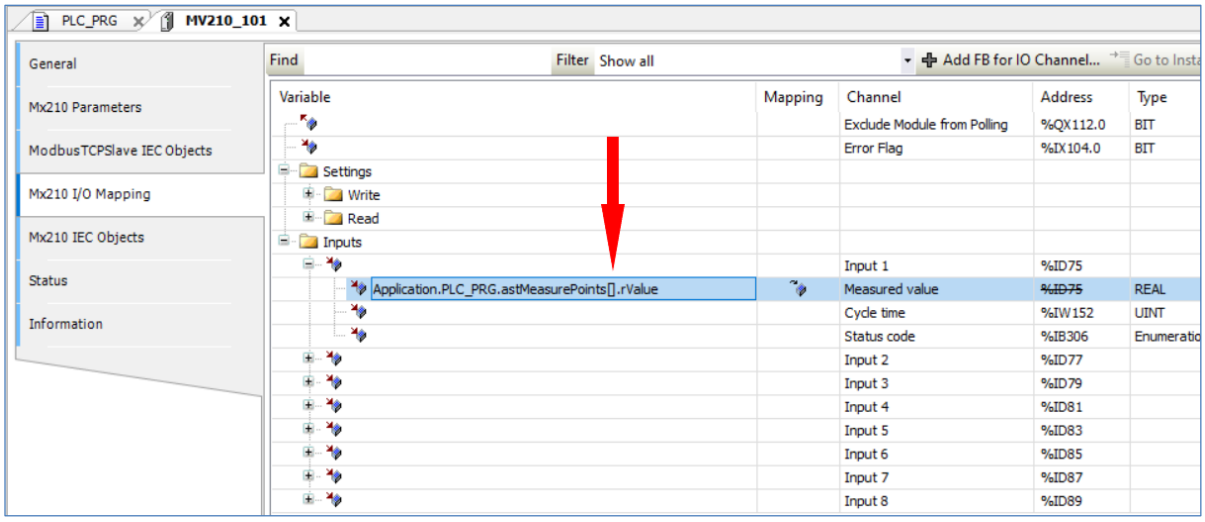


Fig. 6.4.4 Variable assignment. Array index is missing.

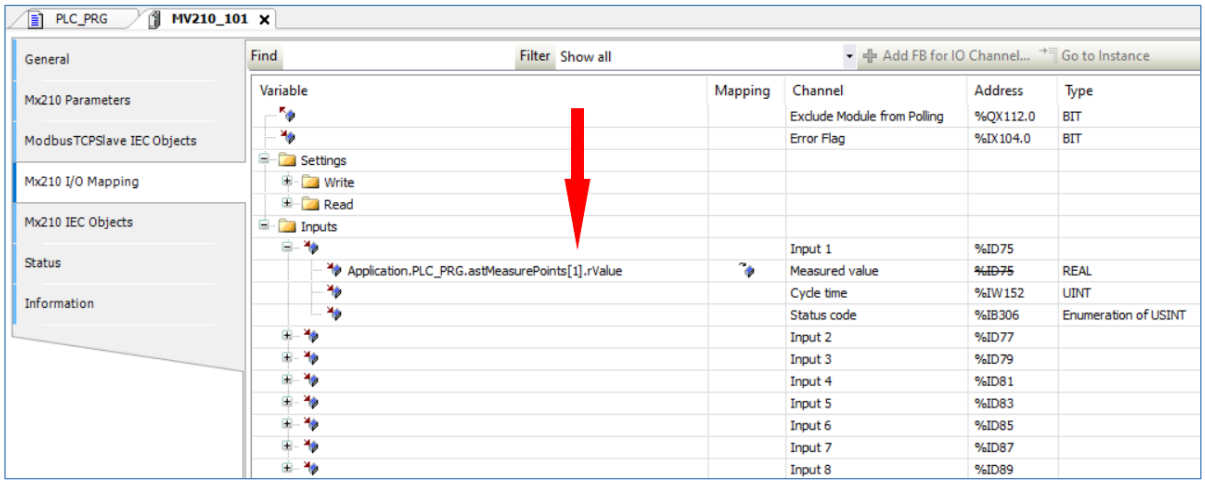


Fig. 6.4.5 Variable assignment. Manual array index input.

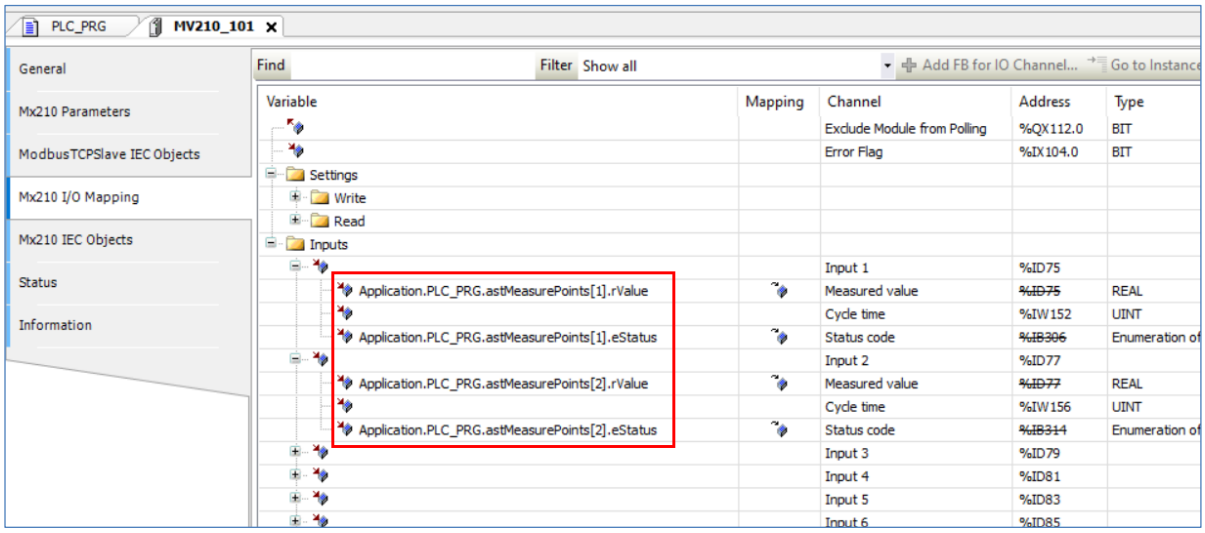


Fig. 6.4.6 Assigning variables to channels in the MB210-101 template

In the template MU210-401, assign the ***xAlarm*** variables to the subchannels of the ***Output bitmask (write)*** channel in the same way.

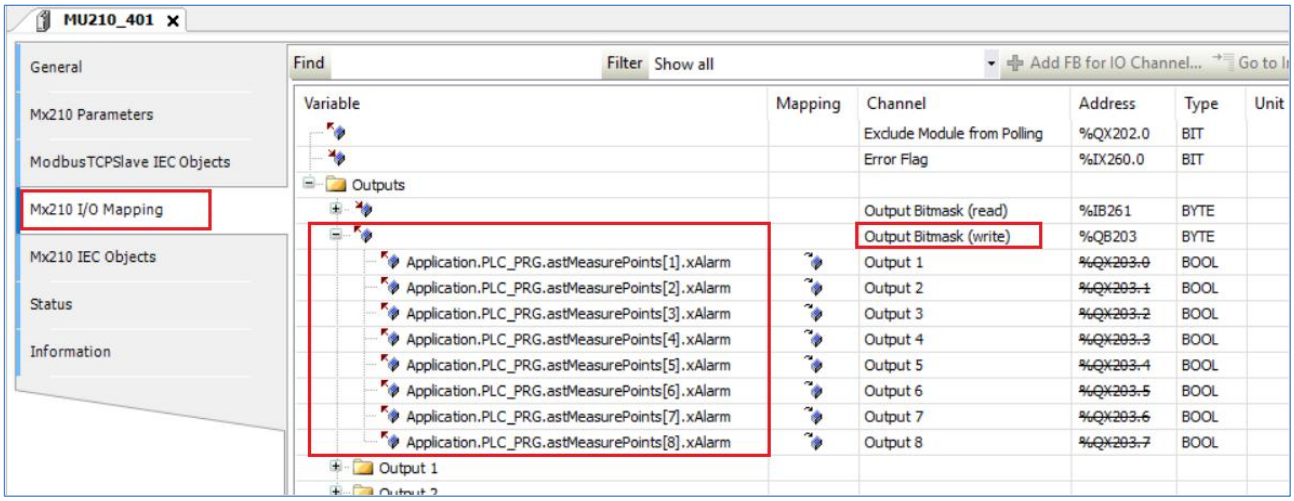


Fig. 6.4.7 Assigning variables to channels in the MB210-401 template

6.4.3 Creating IsValueOutOfLimits function

The project must provide alarm generation in case the measured value exceeds the permissible range. To check whether the value belongs to the specified range, we will create a simple function ***IsValueOutOfLimits***.

To create a function, select the command ***Add Object > POU*** from the context menu of the ***Application*** node. Select the type ***Function*** and the return type ***BOOL***.

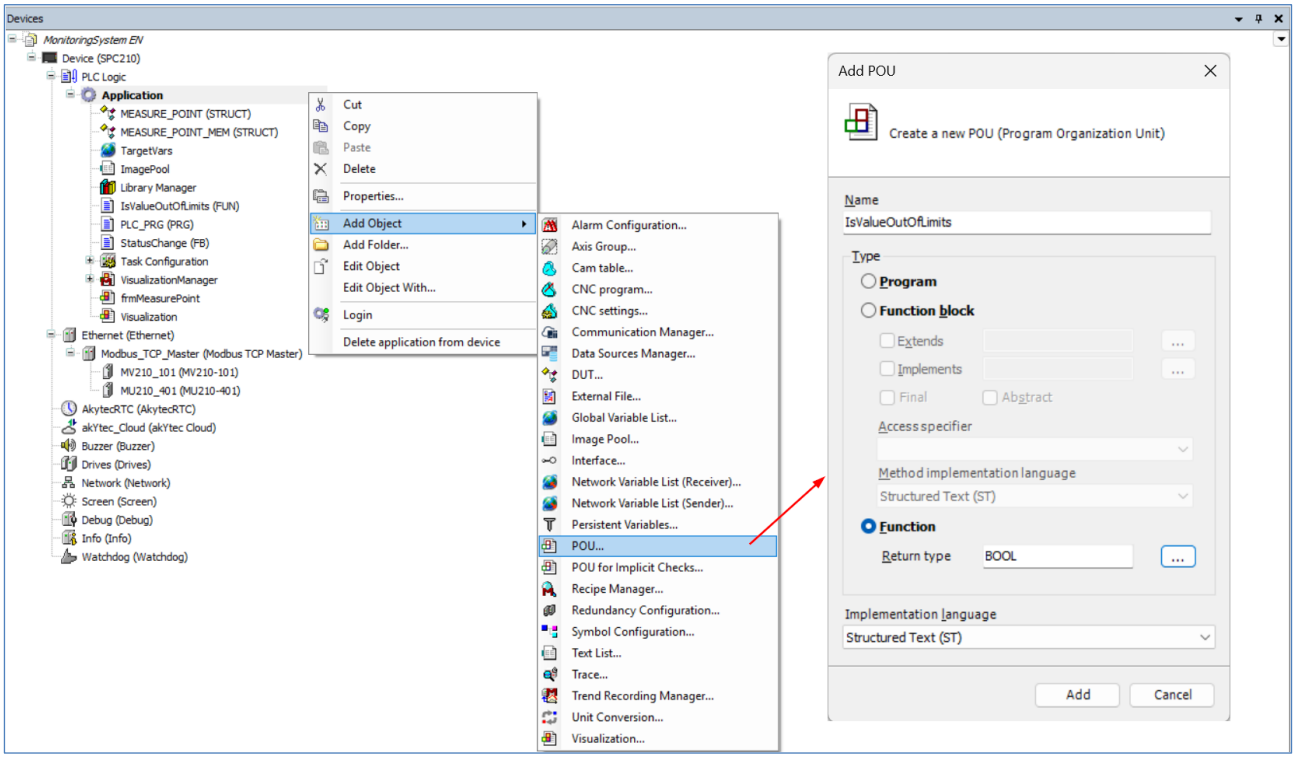


Fig. 6.4.8 Creation of a function

Example 6.4.4

```
FUNCTION IsValueOutOfLimits : BOOL
VAR_INPUT
      rValue: REAL;
```

```
        rLowAlarmValue: REAL;
        rHighAlarmValue: REAL;
END_VAR
VAR
END_VAR

// code area

IsValueOutOfLimits := rValue < rLowAlarmValue OR rValue > rHighAlarmValue;
```

6.4.4 Program code - First version

We added code to the **PLC_PRG** program to generate alarm signals if the measured value exceeds the permissible range and to count the number of such alarms. Furthermore, we implement the conversion of the analog input error code into a string representation to display the text description of the error. All these tasks can be easily processed in a **FOR** loop (Sect. 5.7.3).

Example 6.4.5

```
PROGRAM PLC_PRG
VAR
        astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
        i: INT;
END_VAR
VAR RETAIN
        astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
        c_iMeasurePointsCount: INT := 8;
END_VAR

// code area

FOR i := 1 TO c_iMeasurePointsCount DO
        astMeasurePoints[i].xAlarm := IsValueOutOfLimits(astMeasurePoints[i].rValue,
                astMeasurePointsMemData[i].rLowAlarmValue,
                        astMeasurePointsMemData[i].rHighAlarmValue);
        astMeasurePointsMemData[i].fbAlarmCounter(CU := astMeasurePoints[i].xAlarm);
        astMeasurePoints[i].wsStatus :=
                Mx210Assistant.ANALOG_SENSOR_ERROR_TO_WSTRING(astMeasurePoints[i].eStatus);
END_FOR
```

To convert the error code into a string, we use the **ANALOG_SENSOR_ERROR_TO_WSTRING** function from the **Mx210 Assistant** library.

This version of the project can already be downloaded to the controller and tested to ensure correct operation. When creating the **MEASURE_POINT_MEM** structure, we specified the permissible range between **10** and **30** (Sect. 6.4.1). Using the OPC server, we can set the **AI1** value that exceeds this range and check whether the flag **xAlarm** triggers to **TRUE**.

Do not confuse it with two other variables:

- **eStatus** - sensor error code
- **wsStatus** - text message about the sensor error corresponding to the error code

Consider the following code line:

Example 6.4.6

```
        astMeasurePointsMemData[i].fbAlarmCounter(CU := astMeasurePoints[i].xAlarm);
```

The alarm flag is passed to the `CU` counter input. Ultimately, the counter counts the total number of alarms that have occurred during operation. However, we do not assign any variable to the `CV` counter output to which this value is transmitted, because we will not use the counter value in the code, but will only display it in the visualization. In this case, we can do without an intermediate variable.

The already created code is fully functional, but some aspects could be improved:

1. The `MEASURE_POINT` structure contains a variable `wsPointName`, which defines the name of the measuring point. Currently, the name of each point is an empty string, but initial values can be assigned to these variables.
2. An alarm (`xAlarm`) is generated when the value exceeds the permissible range. However, if the measured value fluctuates at the edge of the range, “chattering” may occur, meaning that the alarm will be generated very frequently. In real life, this may correspond to chaotic flashing of emergency lights, constant switching on/off of audible alarms, etc.
3. The conversion of the status code to a string is performed in each controller cycle. However, it is more reasonable to perform this conversion only when the status code changes, i.e., to switch from cyclic execution to event-driven execution.

In the following sections, we will optimize our program according to the above comments.

6.4.5 Program code - Second version

1. We could assign names to the measuring points when declaring the array of structures, but as mentioned in Sect. 5.3.8, the syntax for initializing an array of structures is quite complex. Instead, we will perform the initialization directly in the code (Example 6.5.5). The changes are highlighted in bold:

Example 6.4.7

```
PROGRAM PLC_PRG
VAR
	astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
	i: INT;
	xInit: BOOL;
END_VAR
VAR RETAIN
	astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
	c_iMeasurePointsCount: INT := 8;
END_VAR

// code area

IF NOT(xInit) THEN
		// The code placed here will be executed once on application startup.
	astMeasurePoints[1].wsPointName := "Measuring point 1";
	astMeasurePoints[2].wsPointName := "Measuring point 2";
	astMeasurePoints[3].wsPointName := "Measuring point 3";
	astMeasurePoints[4].wsPointName := "Measuring point 4";
	astMeasurePoints[5].wsPointName := "Measuring point 5";
	astMeasurePoints[6].wsPointName := "Measuring point 6";
	astMeasurePoints[7].wsPointName := "Measuring point 7";
	astMeasurePoints[8].wsPointName := "Measuring point 8";
	xInit := TRUE;
END_IF

FOR i := 1 TO c_iMeasurePointsCount DO
	astMeasurePoints[i].xAlarm := IsValueOutOfLimits(astMeasurePoints[i].rValue,
		astMeasurePointsMemData[i].rLowAlarmValue,
			astMeasurePointsMemData[i].rHighAlarmValue);
	astMeasurePointsMemData[i].fbAlarmCounter(CU := astMeasurePoints[i].xAlarm);
```

```
        astMeasurePoints[i].wsStatus :=
                Mx210Assistant.ANALOG_SENSOR_ERROR_TO_WSTRING(astMeasurePoints[i].eStatus);
END_FOR
```

2. One way to block chattering is to delay the alarm response, i.e., the signal indicating that the measured value has exceeded the range limits must remain active for a specified period of time before the alarm is triggered.

   Accordingly, we need to split the out-of-range flag and the alarm signal into two separate variables, as well as declare a timer instance. The changes made to the **MEASURE_POINT** structure (Example 6.4.1) are highlighted in bold:

Example 6.4.8

```
        // Structure of operational parameters of a single measuring point
TYPE MEASURE_POINT :
STRUCT
                // Measuring point name
        wsPointName: WSTRING(20);
                // Current measured value
        rValue: REAL;
                // Analog input error code
        eStatus: Mx210Assistant.ANALOG_SENSOR_ERRORS;
                // Error description
        wsStatus: WSTRING;
                // Out-of-range flag
        xOutOfLimits: BOOL;
                // Alarm signal
        xAlarm: BOOL;
                // Timer instance used to delay alarm generation. It filters short-term
                // overshoots of measured values beyond acceptable limits.
        fbAlarmDelay: TON;
END_STRUCT
END_TYPE
```

Now we will add a timer instance call to the program code (Example 6.4.7). Changes are highlighted in bold:

Example 6.4.9

```
PROGRAM PLC_PRG
VAR
        astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
        i: INT;
        xInit: BOOL;
END_VAR
VAR RETAIN
        astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
        c_iMeasurePointsCount: INT := 8;
        c_tAlarmDelay: TIME := T#5S;
END_VAR

// code area

IF NOT(xInit) THEN
                // The code placed here will be executed once on application startup.
        astMeasurePoints[1].wsPointName := "Measuring point 1";
        astMeasurePoints[2].wsPointName := "Measuring point 2";
        astMeasurePoints[3].wsPointName := "Measuring point 3";
        astMeasurePoints[4].wsPointName := "Measuring point 4";
```

```
        astMeasurePoints[5].wsPointName := "Measuring point 5";
        astMeasurePoints[6].wsPointName := "Measuring point 6";
        astMeasurePoints[7].wsPointName := "Measuring point 7";
        astMeasurePoints[8].wsPointName := "Measuring point 8";
        xInit := TRUE;
END_IF

FOR i := 1 TO c_iMeasurePointsCount DO
        astMeasurePoints[i].xOutOfLimits := IsValueOutOfLimits(astMeasurePoints[i].rValue,
                astMeasurePointsMemData[i].rLowAlarmValue,
                        astMeasurePointsMemData[i].rHighAlarmValue);

        astMeasurePoints[i].fbAlarmDelay
        (
                IN := astMeasurePoints[i].xOutOfLimits OR
                        astMeasurePoints[i].eStatus <>
                                Mx210Assistant.ANALOG_SENSOR_ERRORS.NO_ERROR,
                PT := c_tAlarmDelay,
                Q => astMeasurePoints[i].xAlarm
        );

        astMeasurePointsMemData[i].fbAlarmCounter(CU := astMeasurePoints[i].xAlarm);
        astMeasurePoints[i].wsStatus :=
                Mx210Assistant.ANALOG_SENSOR_ERROR_TO_WSTRING(astMeasurePoints[i].eStatus);
END_FOR
```

Now, in the alarm generation logic, we use not only the out-of-range flag `xOutOfLimits`, but also the sensor error code `eStatus`. If the `eStatus` value differs from `NO_ERROR`, this will also trigger an alarm signal.

6.4.6 Creating StatusChange FB

The last thing we planned to optimize (Sect. 6.4.4, improvement 3) was converting the error code to a string not in every cycle, but only when the error code changed. Therefore, we need to determine the moment of change, i.e., compare the status code read in the current task cycle with the status code read in the previous task cycle.

We can do this using a functional block. A function cannot be used here, as they do not have memory, and we need to save the status code between POU calls.

To create an FB, select the command ***Add Object > POU*** from the context menu of the ***Application*** node of the type ***Functional Block*** and with the name ***StatusChange***.

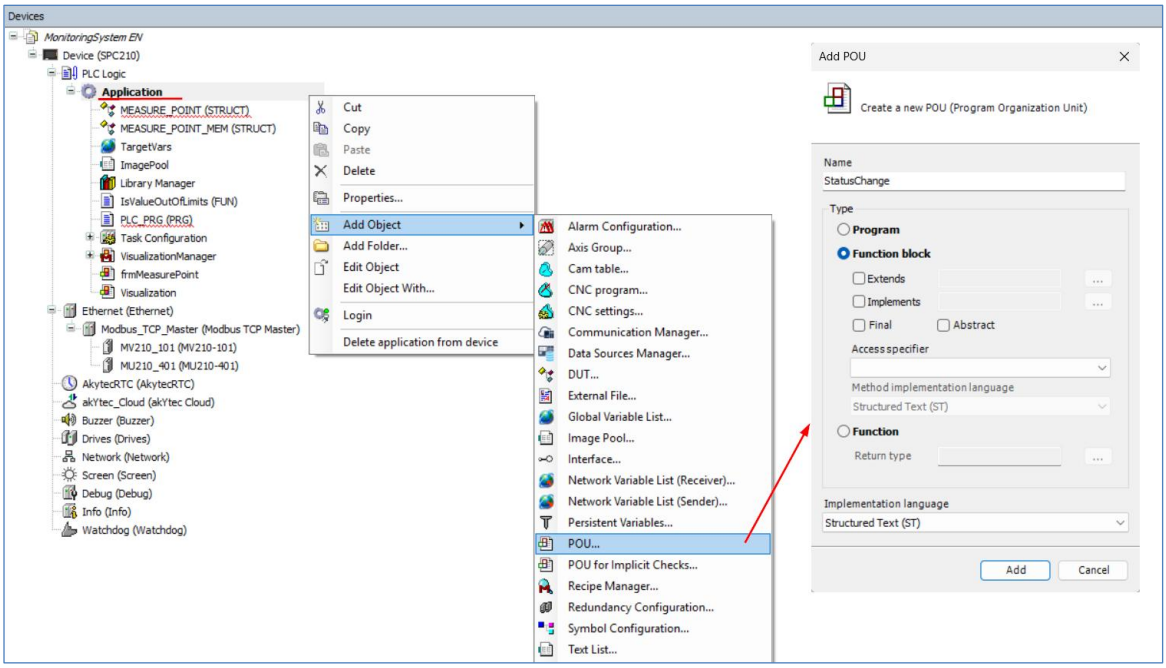


Fig. 6.4.9 FB creation

Example 6.4.10

```
FUNCTION_BLOCK StatusChange
VAR_INPUT
	eStatus: Mx210Assistant.ANALOG_SENSOR_ERRORS;
END_VAR
VAR_OUTPUT
	Q: BOOL;
END_VAR
VAR
	ePrevStatus: Mx210Assistant.ANALOG_SENSOR_ERRORS;
END_VAR

// code area

IF eStatus <> ePrevStatus THEN
	Q := TRUE;
	ePrevStatus := eStatus;
ELSE
	Q := FALSE;
END_IF
```

If the status code has not changed since the previous task cycle, the output **Q** is assigned **FALSE**.

If the status code has changed, the output **Q** is assigned **TRUE**.

Now we add an instance of the block to the **MEASURE_POINT** structure (Example 6.4.8). Changes are highlighted in bold:

Example 6.4.11

```
	// Structure of operational parameters of a single measuring point
TYPE MEASURE_POINT :
STRUCT
		// Measuring point name
	wsPointName: WSTRING(20);
		// Current measured value
	rValue: REAL;
```

```
		// Analog input error code
	eStatus: Mx210Assistant.ANALOG_SENSOR_ERRORS;
		// Error description
	wsStatus: WSTRING;
		// Out-of-range flag
	xOutOfLimits: BOOL;
		// Alarm signal
	xAlarm: BOOL;
		// Timer instance used to delay alarm generation. It filters short-term
		// overshoots of measured values beyond acceptable limits.
	fbAlarmDelay: TON;
		// An instance of the error code change detection block
		// is used to determine at what point a text description of the error code
		// should be generated.
	fbStatusChange: StatusChange;
END_STRUCT
END_TYPE
```

Now we need to call an FB instance in the code (Example 6.4.9):

Example 6.4.12

```
PROGRAM PLC_PRG
VAR
	astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
	i: INT;
	xInit: BOOL;
END_VAR
VAR RETAIN
	astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
	c_iMeasurePointsCount: INT := 8;
	c_tAlarmDelay: TIME := T#5S;
END_VAR

// code area

IF NOT(xInit) THEN
		// The code placed here will be executed once on application startup.
	astMeasurePoints[1].wsPointName := "Measuring point 1";
	astMeasurePoints[2].wsPointName := "Measuring point 2";
	astMeasurePoints[3].wsPointName := "Measuring point 3";
	astMeasurePoints[4].wsPointName := "Measuring point 4";
	astMeasurePoints[5].wsPointName := "Measuring point 5";
	astMeasurePoints[6].wsPointName := "Measuring point 6";
	astMeasurePoints[7].wsPointName := "Measuring point 7";
	astMeasurePoints[8].wsPointName := "Measuring point 8";
	xInit := TRUE;
END_IF

FOR i := 1 TO c_iMeasurePointsCount DO
	astMeasurePoints[i].xOutOfLimits := IsValueOutOfLimits(astMeasurePoints[i].rValue,
		astMeasurePointsMemData[i].rLowAlarmValue,
			astMeasurePointsMemData[i].rHighAlarmValue);

	astMeasurePoints[i].fbAlarmDelay
	(
		IN := astMeasurePoints[i].xOutOfLimits OR
			astMeasurePoints[i].eStatus <>
				Mx210Assistant.ANALOG_SENSOR_ERRORS.NO_ERROR,
		PT := c_tAlarmDelay,
		Q => astMeasurePoints[i].xAlarm
	);
```

```
        astMeasurePointsMemData[i].fbAlarmCounter(CU := astMeasurePoints[i].xAlarm);
        astMeasurePoints[i].wsStatus :=
              Mx210Assistant.ANALOG_SENSOR_ERROR_TO_WSTRING(astMeasurePoints[i].eStatus);

        IF astMeasurePoints[i].fbStatusChange.Q THEN
              astMeasurePoints[i].wsStatus :=
              Mx210Assistant.ANALOG_SENSOR_ERROR_TO_WSTRING(astMeasurePoints[i].eStatus);
        END_IF

END_FOR
```

Optimization of the program is now complete. In the following sections, we will declare a few more variables in it, but the main part of its code will remain unchanged.

Now we can start creating visualization.

6.5 Creating visualization

The project will contain two visualization screens:

- main visualization with information about measuring points
- second visualization with information about alarms

This makes it difficult to fit everything onto one display while maintaining readability. Therefore, we will opt for displaying information from a single measuring point, with the ability to switch between them. We will implement a frame – a visualization template that can be used to display the values of different variables in the same graphic elements.

6.5.1 Creating frame

The template-based project already contains a visualization component named ***Visualization***, which will be used as the main (parent) screen on which the frame will be displayed. We will add a new visualization named ***frmMeasurePoint*** to the project, which we will use to create the frame (Fig. 6.5.1).

In the frame properties, select the ***Use specified visualization size*** option and set the size to 400x250, because the frame will only be part of the main visualization screen. The dimensions of the main visualization are equal to the SPC210 display resolution of 800x480 (Fig. 6.5.2).

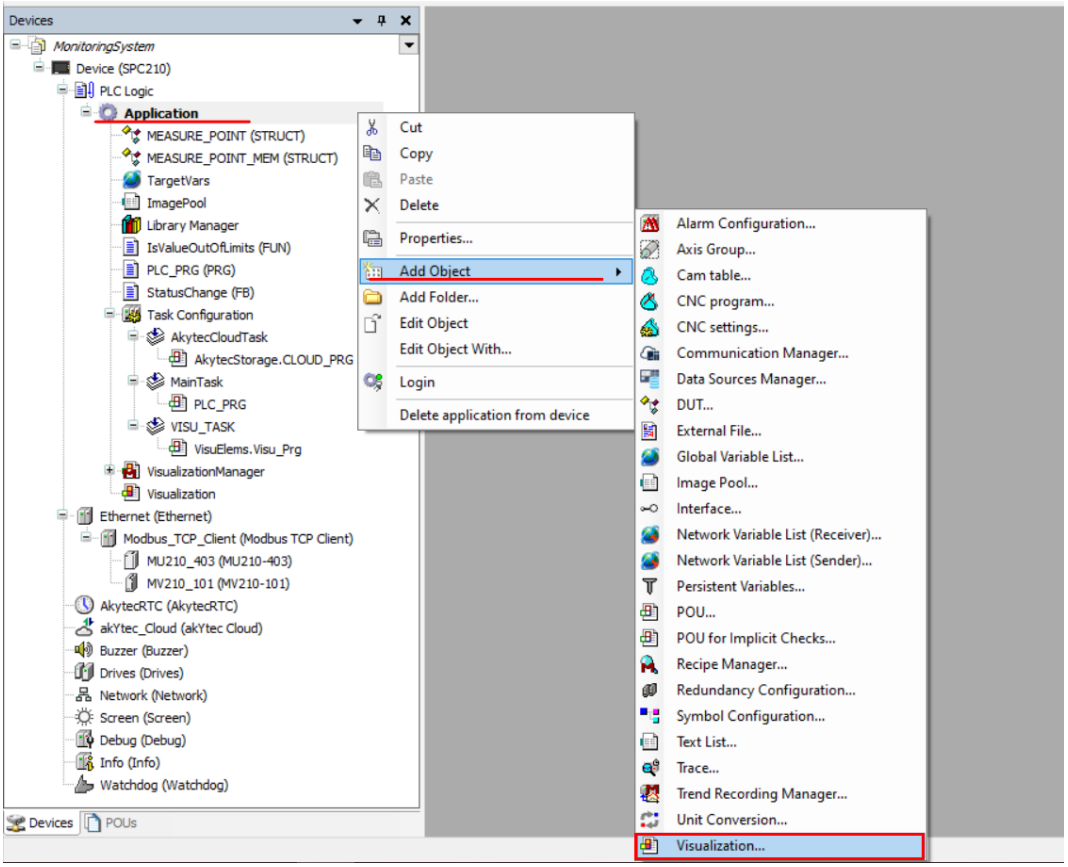


Fig. 6.5.1 Creating visualization

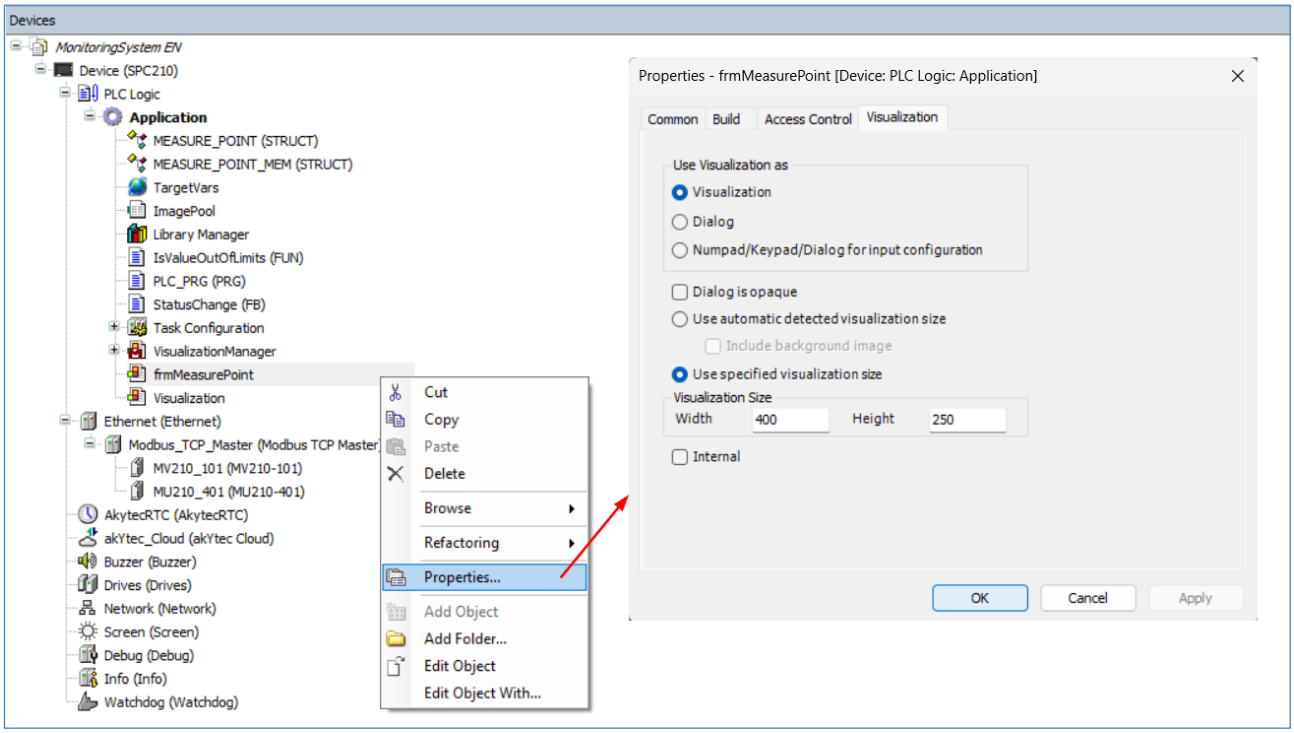


Fig. 6.5.2 Visualization properties

Double-click the ***frMeasurePoint*** component to open it in the visualization editor, click the arrow 1 to open the Interface Editor, and add an input-output area (`VAR_IN_OUT`) to it. In this area, declare instances of the `MEASURE_POINT` and `MEASURE_POINT_MEM` structures (Fig. 6.5.3, Example 6.5.1).

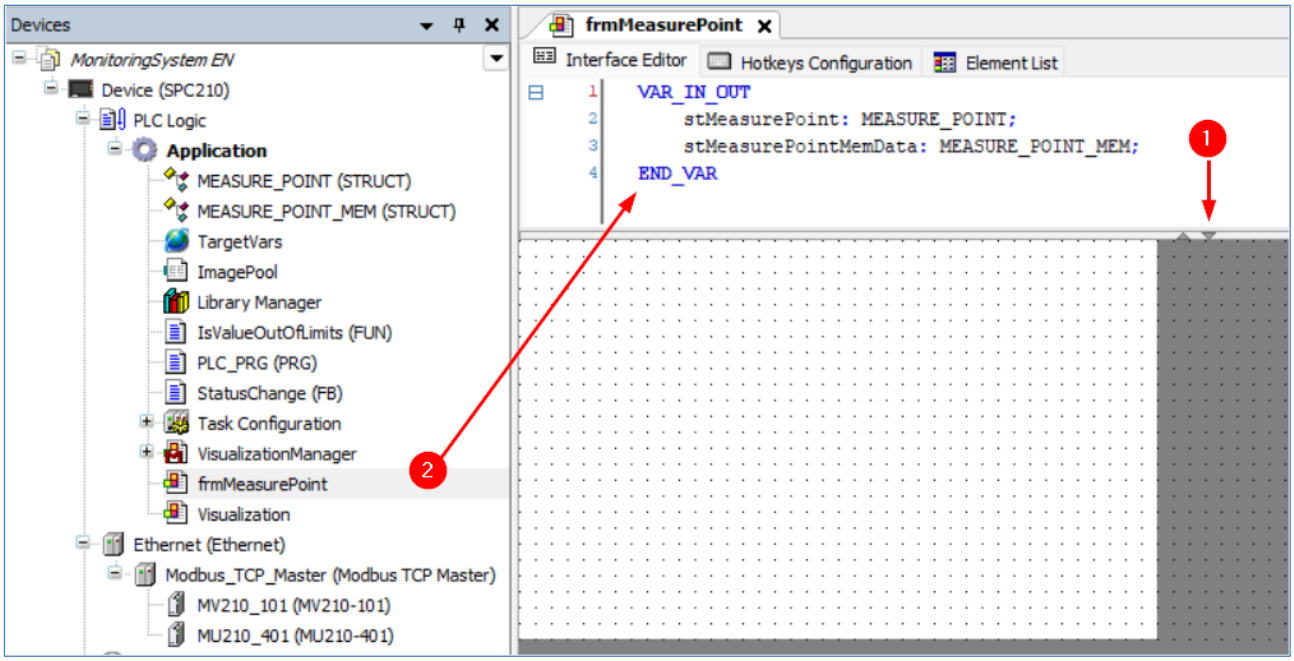


Fig. 6.5.3 Creating frame interface

Example 6.5.1

```
VAR_IN_OUT
      stMeasurePoint: MEASURE_POINT;
      stMeasurePointMemData: MEASURE_POINT_MEM;
END_VAR
```

When variables are declared in `VAR_INPUT` or `VAR_IN_OUT` areas, the visualization screen becomes a frame (visualization template) that cannot be used as a standalone screen. It must be added to one of the visualization screens (parent screen) using the ***Frame*** or ***Tab Control*** graphic element from the ***Toolbox***.

The `VAR_IN_OUT` area is used because some parameters (e.g., measuring point name and measured value) will be transferred from the program to the visualization, while others (e.g., acceptable range limits) will be specified by the operator in the visualization and transferred to the program.

In principle, all values transferred from the visualization are located in the `MEASURE_POINT_MEM` structure, so it would be possible to leave only it in the `VAR_IN_OUT` area and place the `MEASURE_POINT` structure instance in the `VAR_INPUT` area. However, placing both instances in the `VAR_IN_OUT` area is also legitimate.

Now open the visualization toolbar and drag the graphic elements from the ***Toolbox*** onto the frame screen (Fig. 6.5.4):

- 6x ***Rectangle*** (***Basic*** tab)
- 2x ***Lamp*** (***Lamps/Switches/Bitmaps*** tab)
- 1x ***Button*** (***Common Controls*** tab)

Click on an element to open its ***Properties*** pane. To move an element across the screen, drag-and-drop the element to the desired position. To resize an element, drag-and-drop one of its anchor points (displayed as blue squares). You can also move elements around the screen and change their size using the properties of the ***Position*** group (Fig. 6.5.5).

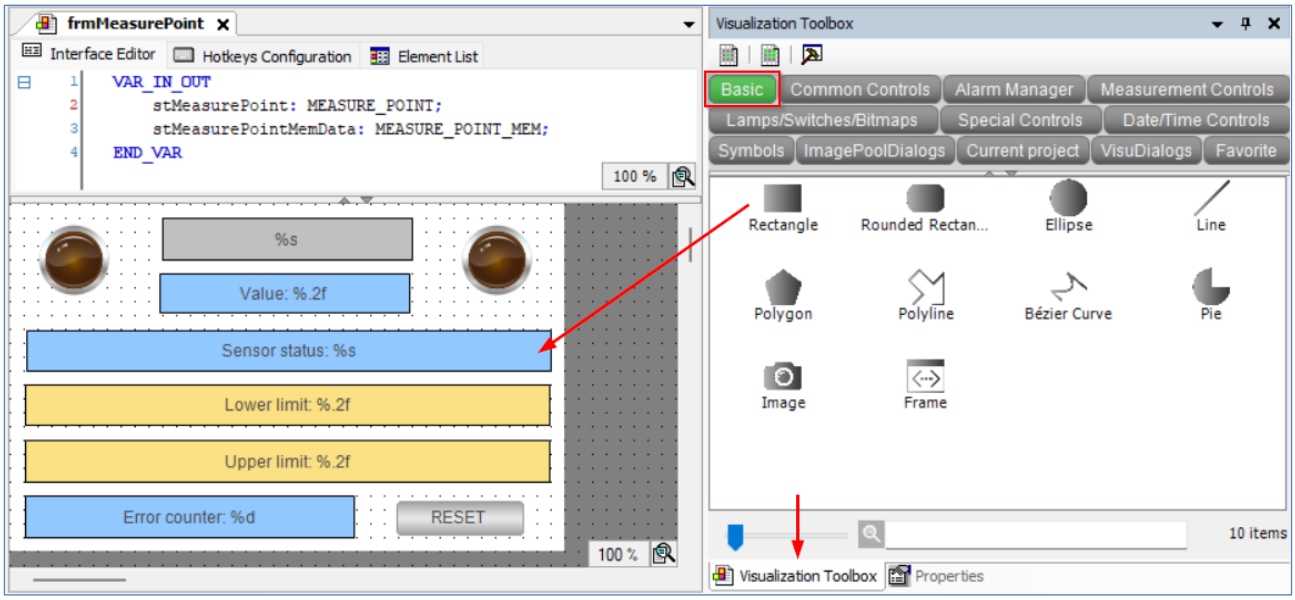


Fig. 6.5.4 Add elements to the frame screen

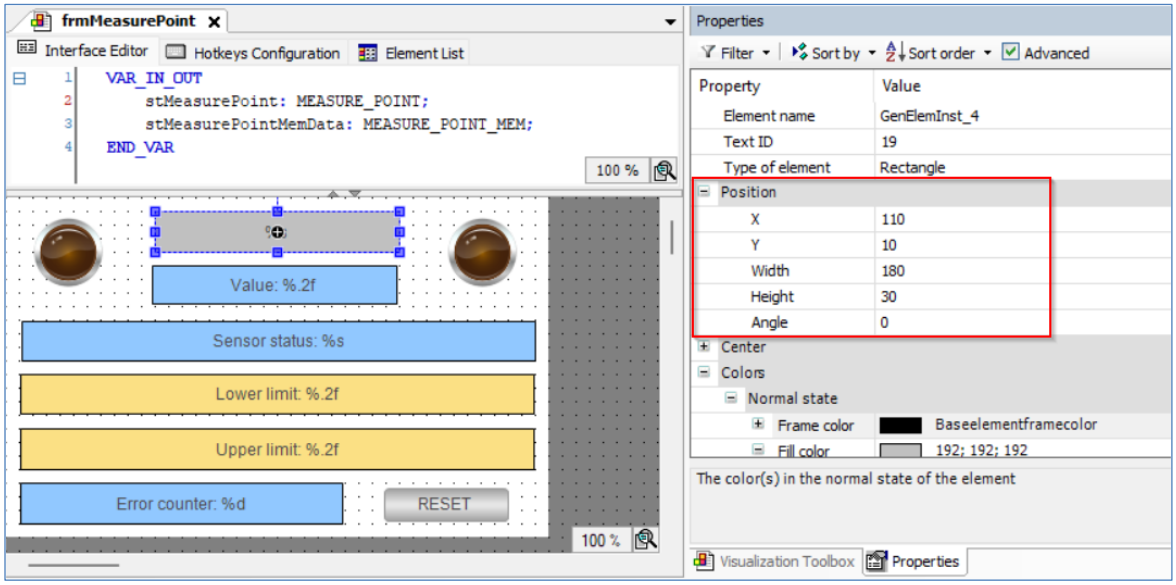


Fig. 6.5.5 Position properties of visualization elements

The colors of elements are set in the property ***Colors*** > ***Normal state*** > ***Fill color***, and texts are set in the parameter ***Texts*** > ***Text*** (Fig. 6.5.6).

Three colors are used in the frame:

- gray – to display static information (measuring point name)
- blue – to display values that cannot be changed by the visualization user (measured value, sensor status text, etc.)
- sand – to display values that can be changed by the visualization user (measurement range limits)

The texts of elements use the following specifiers that define the type of value transmitted to the element:

- ***%s*** – for string variables
- ***%.nf*** – for floating point variables (***n*** is the number of digits displayed after the decimal point)
- ***%d*** – for integer variables

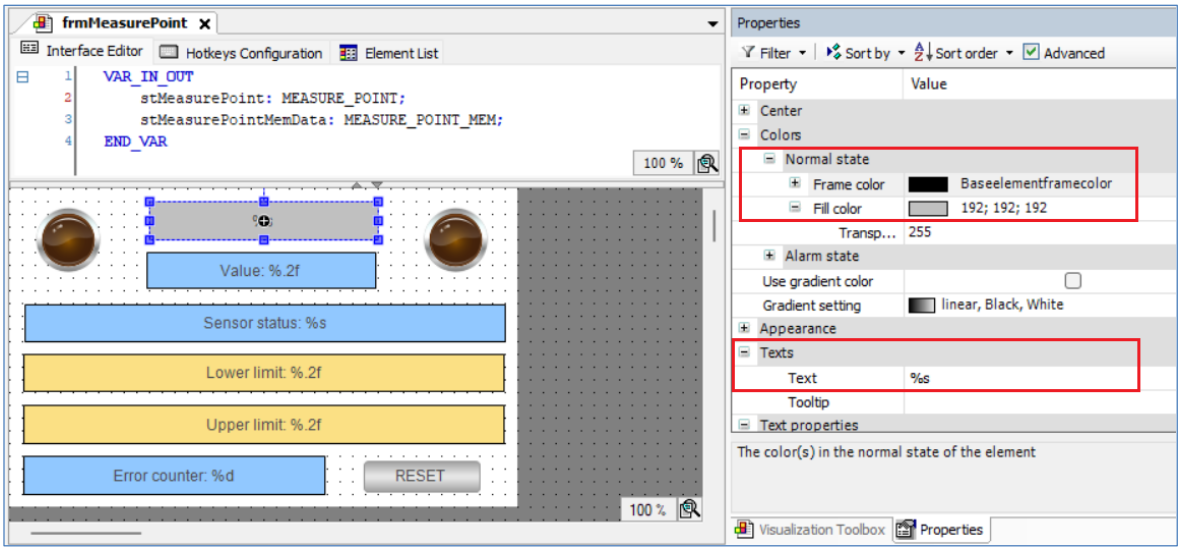


Fig. 6.5.6 Color and text of visualization elements

The variable whose value will be displayed in the element is linked in the property ***Text variables*** > ***Text variable*** (Fig. 6.5.7). In the case of a frame, the variables of structures declared in the frame interface are linked to these properties. The following variables are associated with the rectangles:

- upper rectangle – ***stMeasurePoint.wsPointName***
- Value – ***stMeasurePoint.rValue***
- Sensor status – ***stMeasurePoint.wsStatus***
- Lower limit – ***stMeasurePointMemData.rLowAlarmValue***
- Upper limit – ***stMeasurePointMemData.rHighAlarmValue***
- Error counter – ***stMeasurePointMemData.fbAlarmCounter.CV***

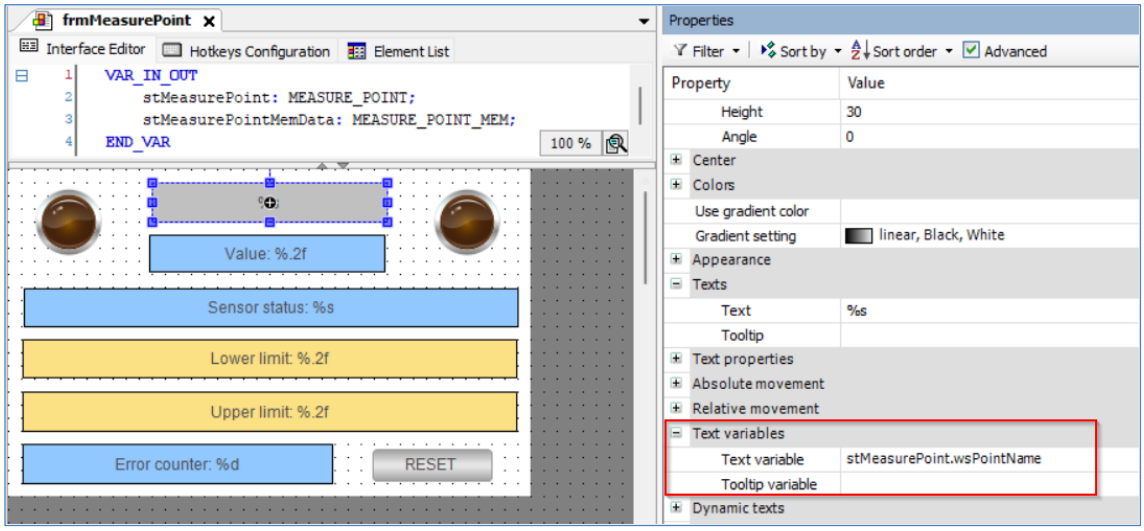


Fig. 6.5.7 Linking a variable to an element

NOTE

The output of the functional block instance is linked to the “Error counter”. This approach is quite acceptable if you do not need to use the block output in the program itself; otherwise, it is better to create an intermediate variable to improve code readability.

Configuration of the parameters ***Lower limit*** and ***Upper limit***, that are written by the visualization user, must be discussed separately (Fig. 6.5.8).

The ***Input Configuration*** section contains a list of events – like clicking or releasing an element – where you can assign a specific set of actions to each event. Select the ***OnMouseClick*** event (clicking and releasing an element) and configure the ***Write Variable*** action for it.

Keep the default option – ***Use text output variable*** – in this case, the recording will be made to the variable linked to the ***Text variable*** property.

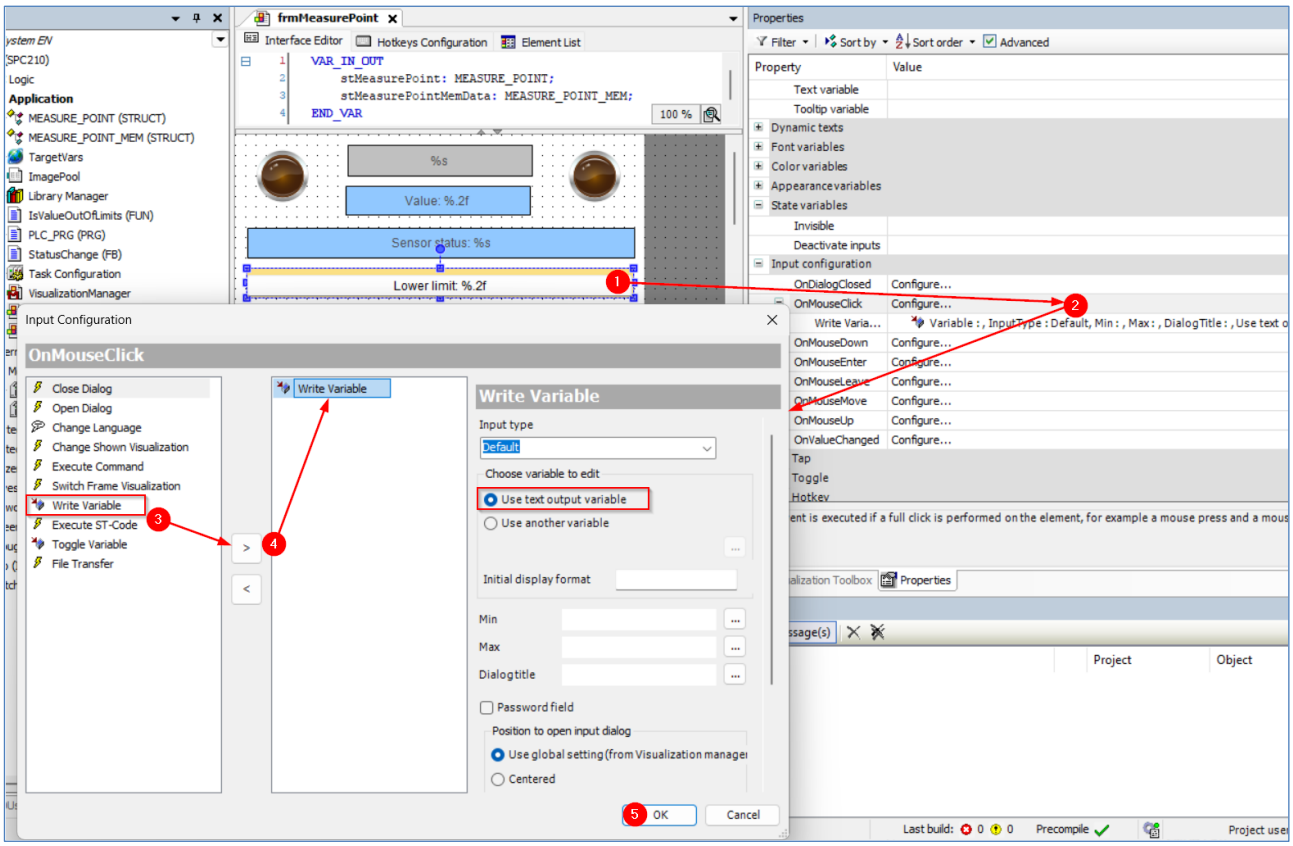


Fig. 6.5.8 Configuration of the input field for writing the entered value

You can select the input type (on-screen or hardware keyboard), the title of the input dialog for the on-screen keyboard, the lower and upper limits of the input value, etc.

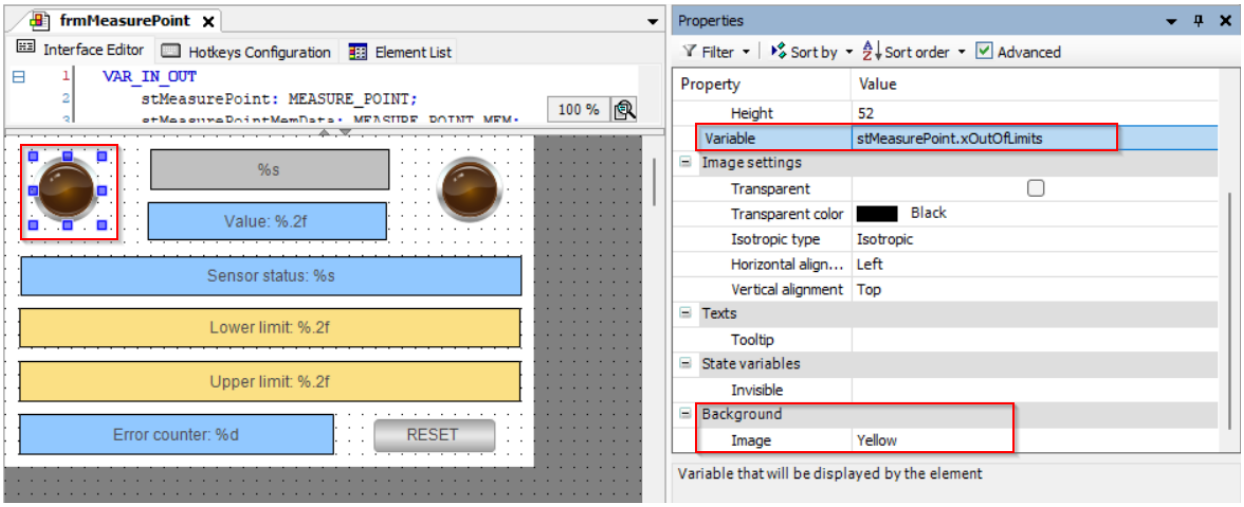


Fig. 6.5.9 Linking a variable to an indicator and setting its color

We will link the variable ***stMeasurePoint.xOutOfLimits*** (out-of-range flag) to the yellow indicator (left), and ***stMeasurePoint.xAlarm*** (alarm signal) to the red indicator (right). The indicator color is selected in the ***Background > Image*** property (Fig. 6.5.9).

To link the variable to the alarm counter reset button, open the ***Input configuration*** property group and link the ***RESET*** counter block input ***stMeasurePointMemData.fbAlarmCounter.RESET*** to the ***Tap > Variable*** property. (Fig. 6.5.10).

The variable linked to this property takes the value ***TRUE*** when the button is pressed. After the button is released, it will be reset to ***FALSE***.

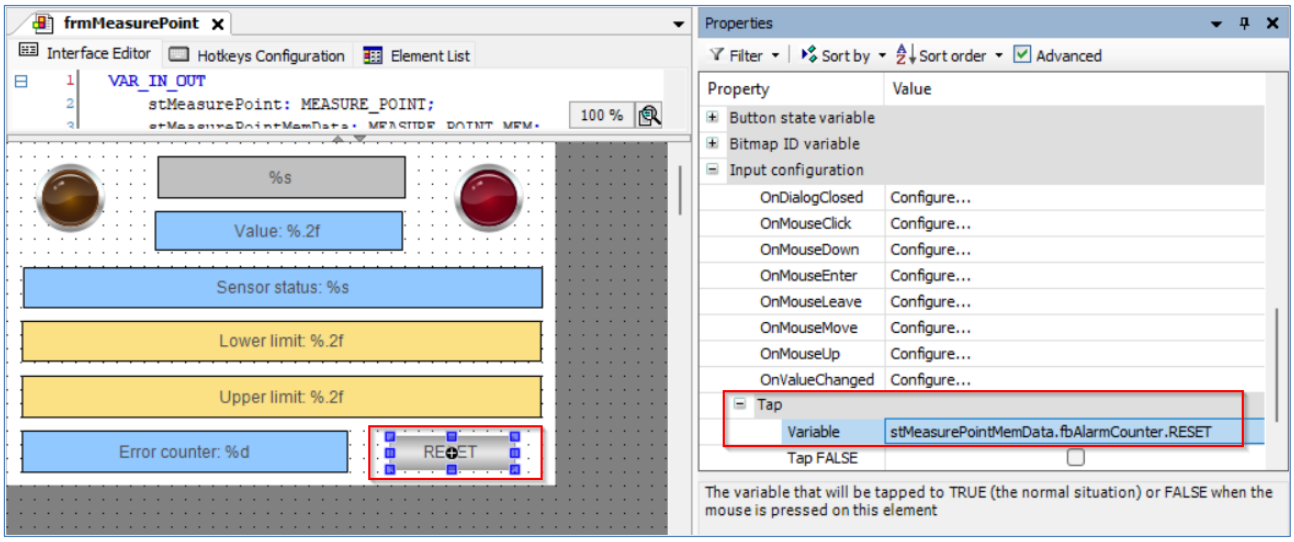


Fig. 6.5.10 Linking a variable to a button

If you need to create a toggle button that changes the value of a variable to its opposite when clicked, use the ***Input configuration > Toggle > Variable*** property.

Our frame is ready, now we can place it on the main visualization screen.

6.5.2 Basic elements - Adding and configuring

Our template-based project already has a screen named ***Visualization***. Double-click its name in the project tree and rename it to ***visuMainScreen***. This will open the renaming and refactoring windows. Click ***Yes*** and ***OK*** in them, respectively (Fig. 6.5.11, Fig. 6.5.12).

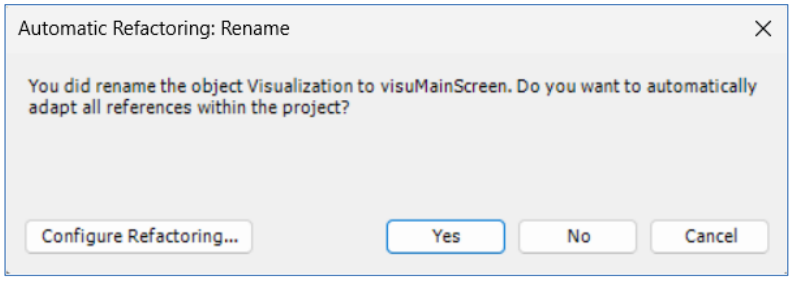


Fig. 6.5.11 "Rename" window

To create a second screen for displaying the alarm table, select ***Copy*** from the context menu of the ***visuMainScreen*** component in the project tree, then use ***Paste*** from the context menu of the ***Application*** node (Fig. 6.5.13).

Rename the copied screen to ***visuAlarmTable***.

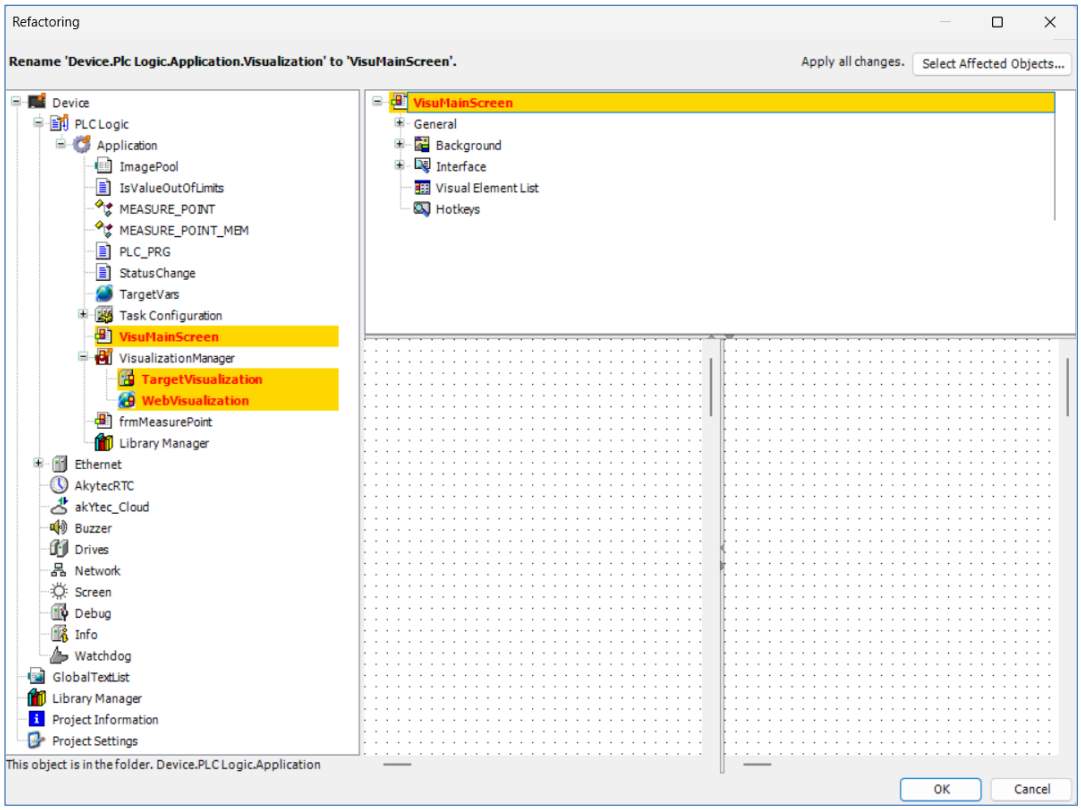


Fig. 6.5.12 Refactoring window

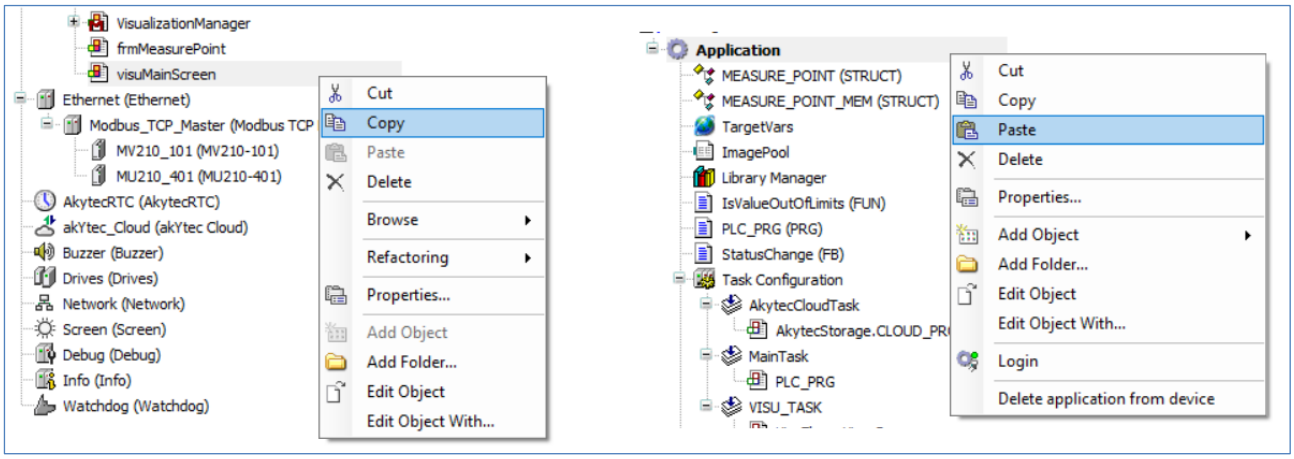


Fig. 6.5.13 Copy visualization

The next step is to lay out the information on the ***visuMainScreen*** screen. Place the following elements on the screen (Fig. 6.5.14):

- 1x ***Label*** element to display the screen name (***Standard Controls*** tab)
- 2x ***Rectangle*** to display the current date and time
- 2x ***Buttons*** to switch screens

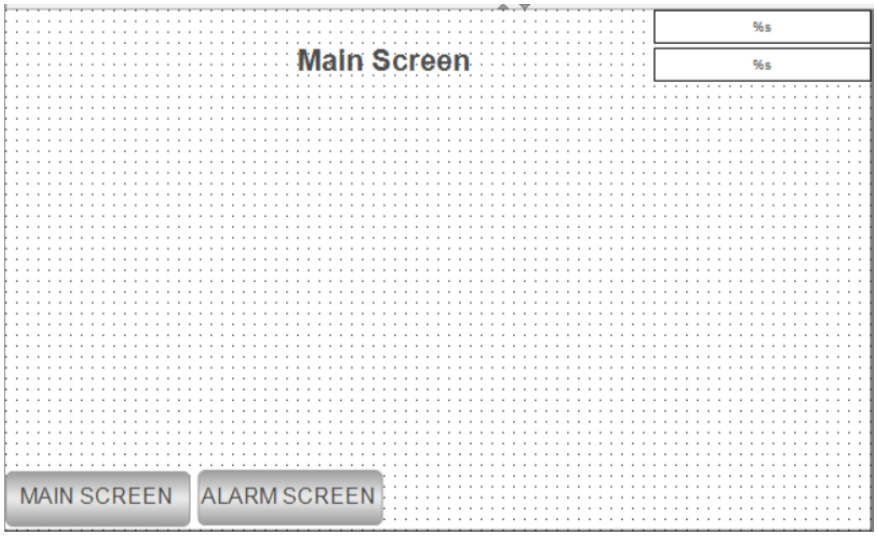


Fig. 6.5.14 visuMainScreen layout

The project tree contains the ***AkytecRTC*** system node, with channels for accessing the system time and a list of global variables ***TargetVars*** containing instances of structures whose variables are already linked to these channels (Fig. 6.5.16).

Link the ***TargetVars.stRtc.sGetFormatDate*** variable to the text variable of the upper rectangle, and the ***TargetVars.stRtc.sGetFormatTime*** variable to the text variable of the lower rectangle.

Since the current date and time must be displayed in string format, the ***%s*** specifier must be used in the ***Texts > Text*** property (Fig. 6.5.15).

NOTE

This functionality cannot be tested on a virtual controller, as it does not have system nodes in the project tree, only akYtec controllers have them.

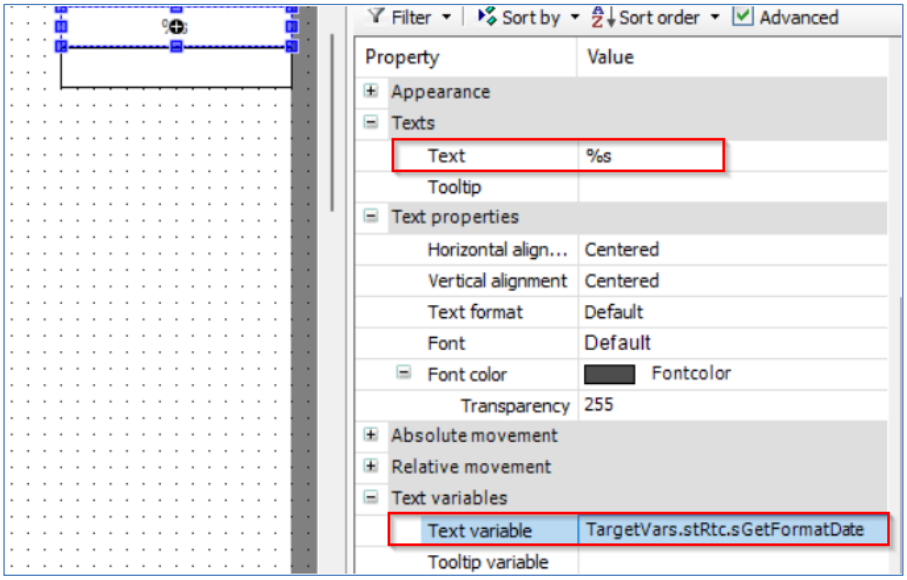


Fig. 6.5.15 System time display settings

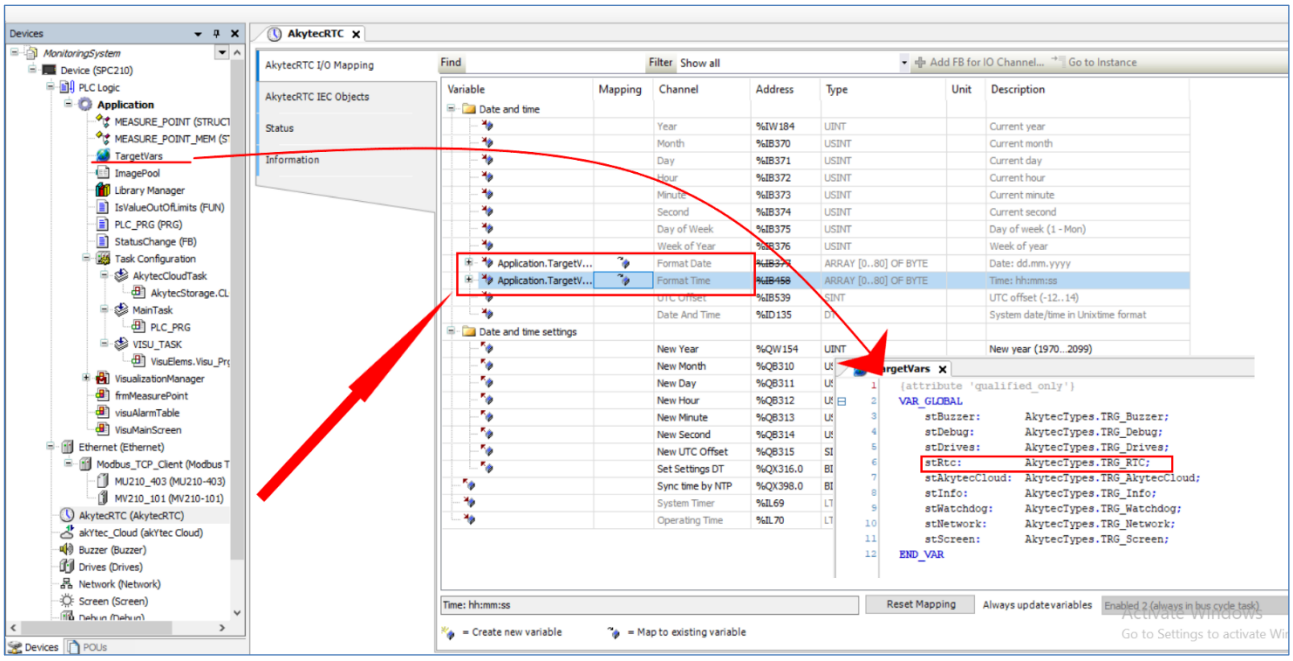


Fig. 6.5.16 Instance variables of structures of the TargetVars global variable list are associated with channels of system components

Now you can configure the button functions for switching between screens. In the ***Input Configuration*** properties group, select the ***OnMouseClick*** event, link the ***Change shown visualization*** action to it, and select the ***visuMainScreen*** visualization for the ***MAIN*** button and the ***visuAlarmTable*** visualization for the ***ALARMS*** button (Fig. 6.5.17).

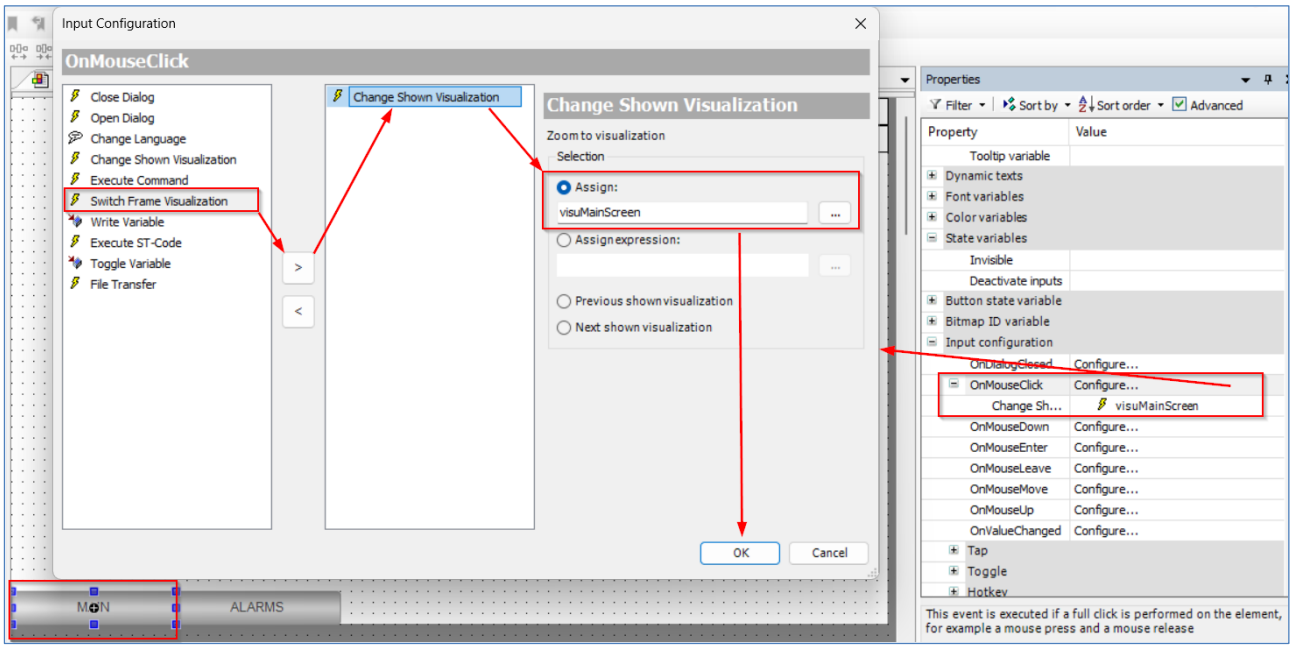


Fig. 6.5.17 Configuring the button functions for switching between screens

Copy all items from the main screen to the alarm screen. Change the screen name and button colors (the button for the currently open screen should be highlighted with a color) (Fig. 6.5.18).

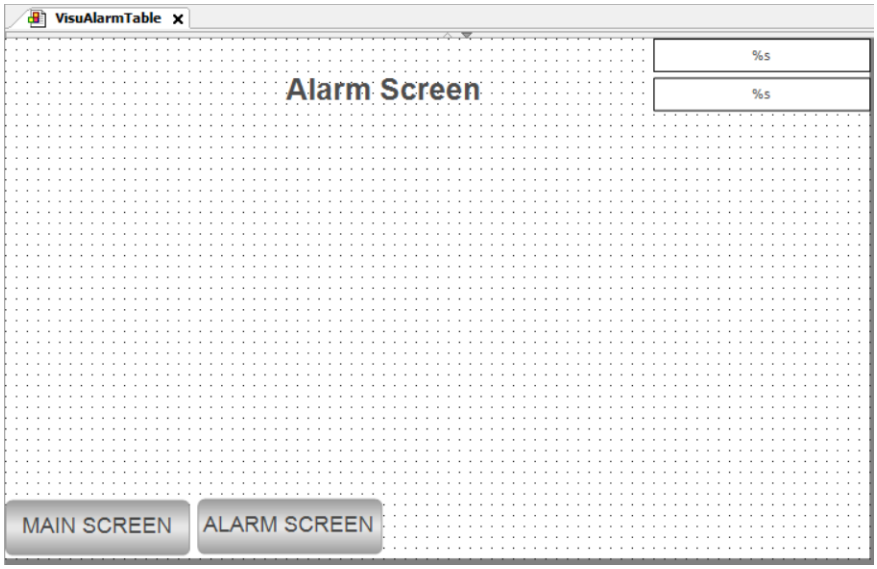


Fig. 6.5.18 visuAlarmTable layout

We have configured the elements common to both screens. Now we will discuss the elements specific to each screen: for the ***visuMainScreen*** screen, this is the frame, and for the ***visuAlarmTable*** screen, this is the alarm table.

6.5.3 visuMainScreen configuration

The frame created in section 6.5.1 need to be placed on the ***visuMainScreen*** screen. To do this, add the ***Frame*** element (***Basic*** tab) to the screen. After adding the element, the ***Frame configuration*** window opens. Select the ***frmMeasurePoint*** frame in the left pane and click the ***Add*** button in the right pane to link its instance to the added ***Frame*** element.

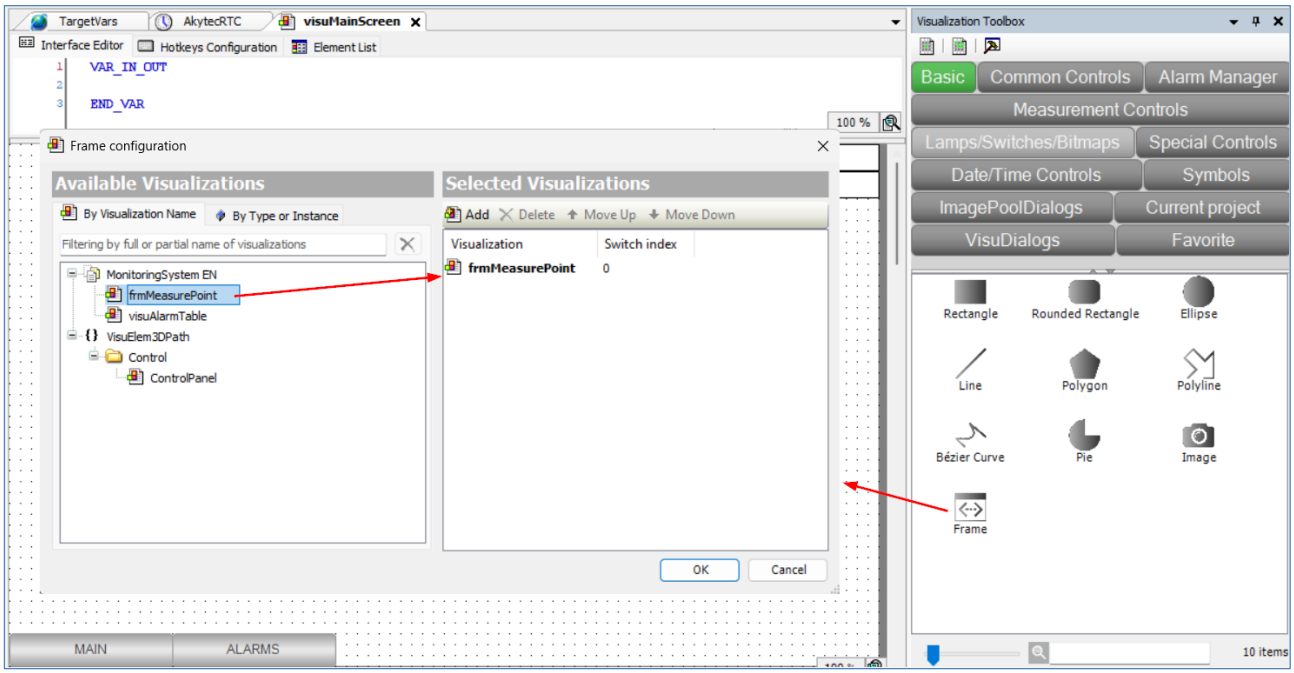


Fig. 6.5.19 Adding Frame to the screen

Why did we link only one frame instance to the ***Frame*** element, given that there are eight measurement points in our project? This is because at any given moment, the visualization user will only view information

about one point at a time. Therefore, we can use a single frame instance and point navigation buttons to substitute the data for the respective point into this instance.

For this, we need a new variable containing the number of the current measurement point displayed in the visualization. We declare it in the ***PLC_PRG*** program and assign it an initial value of 1 (first point) (Example 6.4.12).

Example 6.5.2

```
PROGRAM PLC_PRG
VAR
	astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
	i: INT;
	xInit: BOOL;
	iCurrentFrameIndex: INT := 1;
END_VAR
VAR RETAIN
	astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
	c_iMeasurePointsCount: INT := 8;
	c_tAlarmDelay: TIME := T#5S;
END_VAR
```

Now we need to link the variables you want to display in the ***Frame*** element to that frame. Open the ***References*** property group in the ***Frame*** properties. It contains an instance of the linked frame and its interface variables (Fig. 6.5.20).

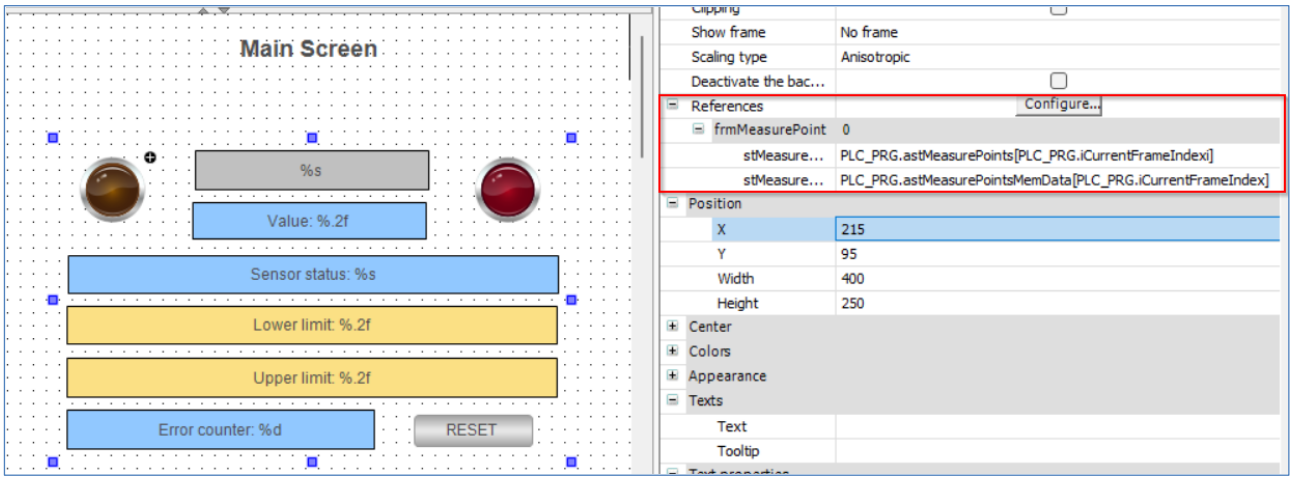


Fig. 6.5.20 Linking variables to the Frame

We link the variable ***PLC_PRG.astMeasurePoints[PLC_PRG.iCurrentFrameIndex]*** to the input/output ***stMeasurePoint***, i.e., we link an element of the array of structures, using the variable as an index. By changing the value of this variable, we will change the element of the array, the value of which is currently displayed in the frame (Fig. 6.5.21).

Then, we link the variable ***PLC_PRG.astMeasurePointsMemData[PLC_PRG.iCurrentFrameIndex]*** to the input-output ***stMeasurePointMemData***.

Now we add the point navigation buttons, which we will use to change the value of the ***iCurrentFrameIndex*** variable (Fig. 6.5.22).

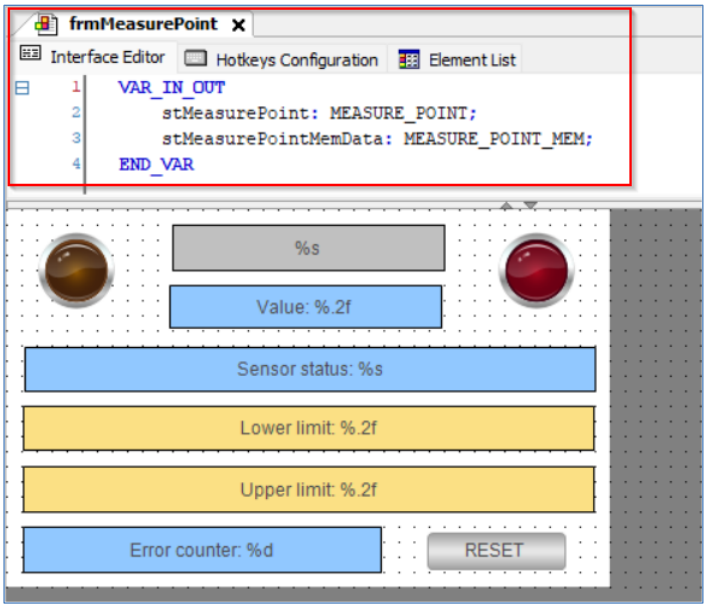


Fig. 6.5.21 Frame element links display the interface variables created in Sect. 6.5.1.

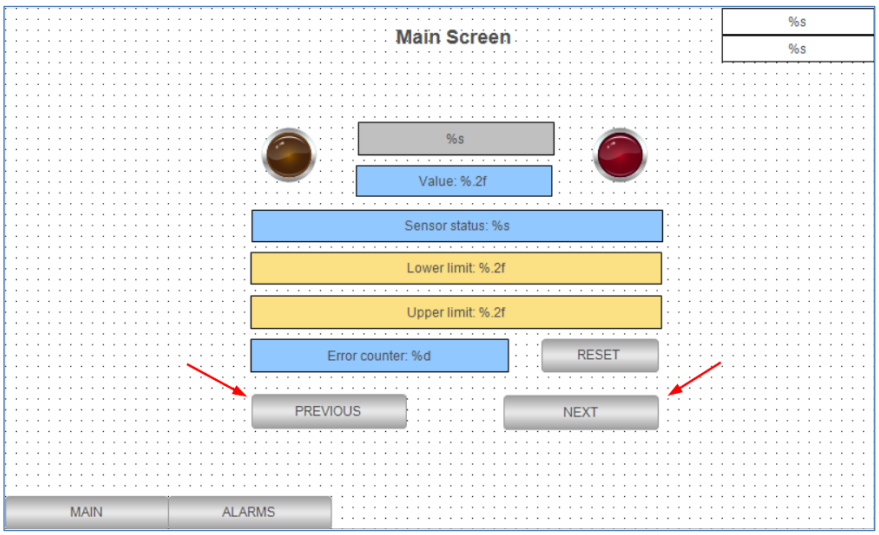


Fig. 6.5.22 Measuring point navigation buttons

In the ***Input Configuration*** properties group, select the ***OnMouseClick*** event and link the ***Execute ST-Code*** action to it.

Please note that when accessing program variables, the program namespace must be specified, i.e., its name ***PLC_PRG***.

Code for the ***PREVIOUS*** button (Example 6.5.3):

Example 6.5.3

```
IF PLC_PRG.iCurrentFrameIndex = 1 THEN
      PLC_PRG.iCurrentFrameIndex := PLC_PRG.c_iMeasurePointsCount;
ELSE
      PLC_PRG.iCurrentFrameIndex := PLC_PRG.iCurrentFrameIndex - 1;
END_IF
```

The code is simple: when the button is pressed, the array index (i.e., the point number) decreases by 1. If the current point is the first point, pressing the button causes a jump to the last point. The number of the last point is determined by the **PLC_PRG.c_iMeasurePointsCount** constant.

The ***NEXT*** button code acts in reverse: each time the button is pressed, the array index increases by 1. If the current point is the last point, pressing the button causes a jump to the first point (Example 6.5.4).

Example 6.5.4

```
IF PLC_PRG.iCurrentFrameIndex = PLC_PRG.c_iMeasurePointsCount THEN
       PLC_PRG.iCurrentFrameIndex := 1;
ELSE
       PLC_PRG.iCurrentFrameIndex := PLC_PRG.iCurrentFrameIndex + 1;
END_IF
```

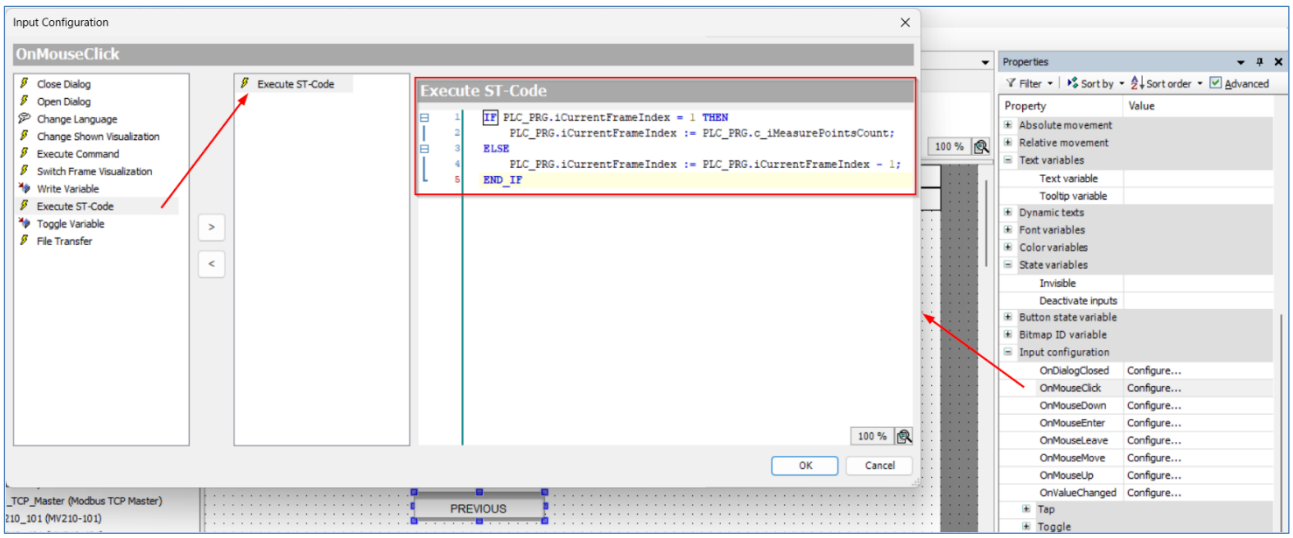


Fig. 6.5.23 Configuring ST code execution when pressing the button

This completes the creation of the ***visuMainScreen*** screen.

Now we need to organize alarm management.

6.5.4 Alarm configuration

To display the alarm table on the ***visuAlarmTable*** screen, alarms must first be configured. Add the ***Alarm Configuration*** object to the project (Fig. 6.5.24). Then, add the ***Alarm Group*** sub-object to the ***Alarm Configuration*** (Fig. 6.5.25).

NOTE
When adding the ***Alarm Configuration*** to the project, the ***AlarmManagerTask*** task was automatically added to the ***Task Configuration***. This task should not be deleted (otherwise it will cause compilation errors) and it is not recommended to edit it (i.e., change its priority or call interval).

The alarm configuration now includes the following objects:

- ***Error***, ***Info***, and ***Warning*** alarm classes – these enable the user to configure the alarm acknowledgment type, the color of the alarm message in the alarm table, etc. Each alarm belongs to a specific class. The user can create alarm classes themselves.
- ***AlarmGroup*** – this group is used to configure the alarms themselves (selection of conditions for their activation, classes, message texts, etc.). The number of alarms in a group and the number of groups are unlimited. A text list is automatically created along with the group, which stores the texts of the messages displayed in the alarm table.

- ***AlarmStorage*** – this object defines the parameters for storing alarm history in the controller memory

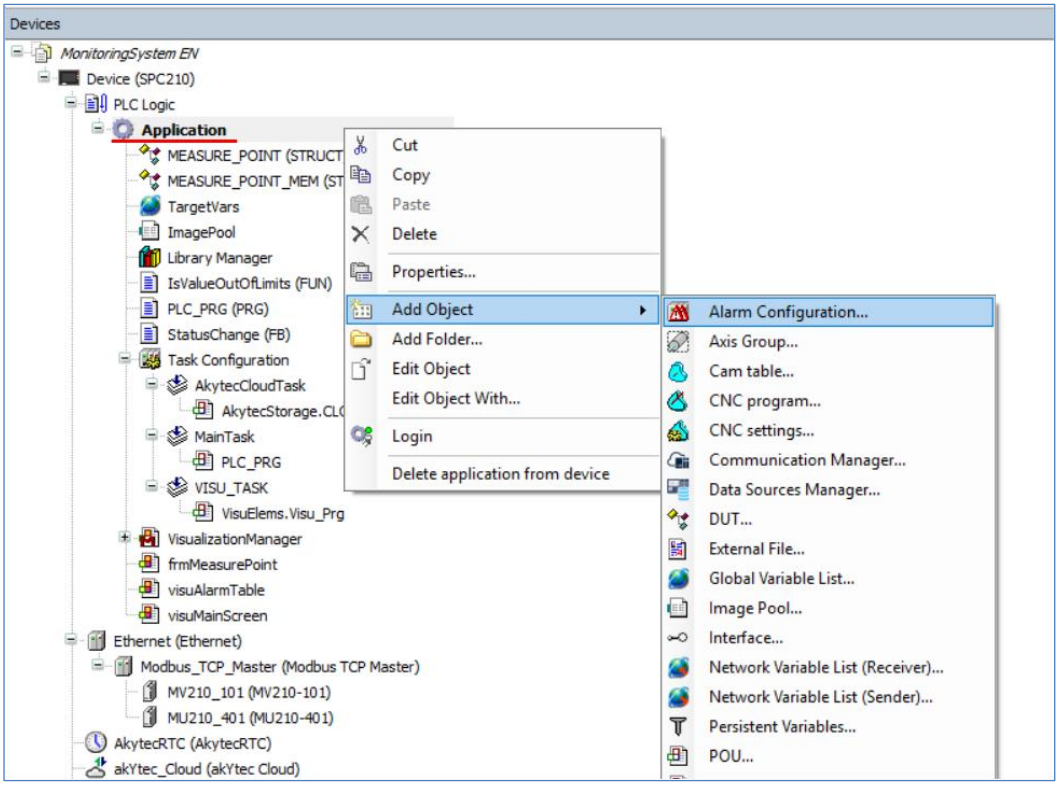


Fig. 6.5.24 Adding Alarm Configuration to the project

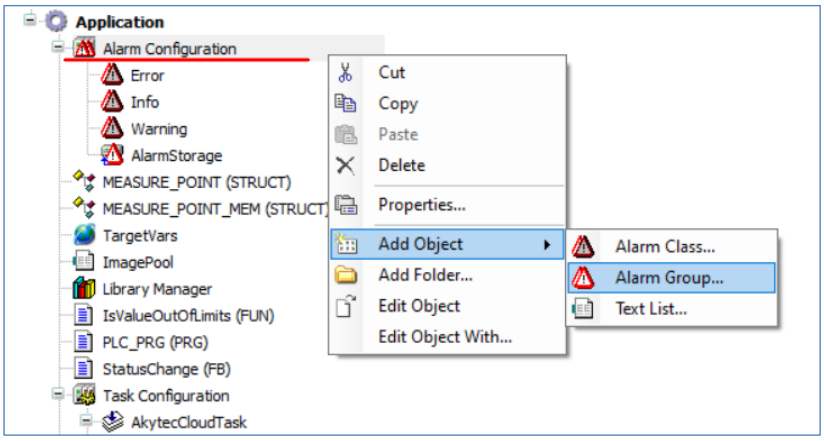


Fig. 6.5.25 Adding Alarm Group

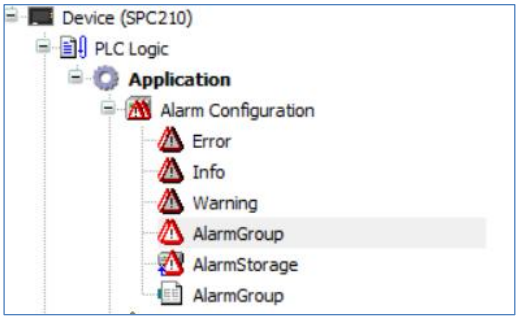


Fig. 6.5.26 Alarm Configuration sub-objects after adding Alarm Group

We will use a single alarm class – ***Error*** – and select the ***REP_ACK*** acknowledgment method – acknowledgment is only possible if the alarm condition no longer exists. Mark the ***Archiving*** checkbox to ensure that alarms are stored in the controller memory and displayed in the alarm history (Fig. 6.5.27).

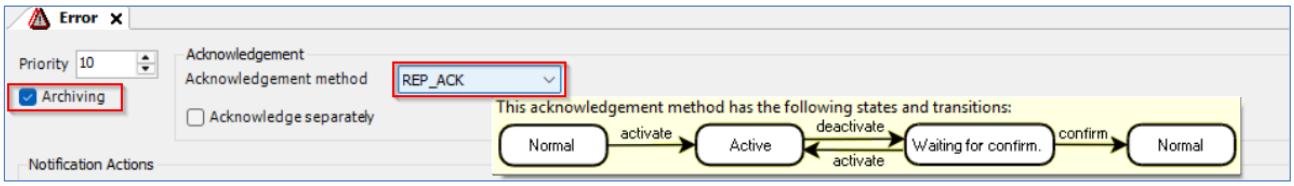


Fig. 6.5.27 Configuring the Error alarm class

In the ***Alarm Group*** settings, create an alarm for each of the 8 inputs using the variable ***xAlarm*** from the ***MEASURE_POINT*** structure which represents the alarm (Fig. 6.5.28).

- In the ***Observation Type*** column, select ***Digital***.
- In the ***Details*** column, specify the ***xAlarm*** variables from the corresponding elements of the structure array as the condition.
- In the ***Class*** column, specify the ***Error*** class for all alarms.
- In the ***Message*** column, specify the message text to be displayed in the alarm table for each alarm.
- In the ***Archiving*** parameter, specify the ***AlarmStorage***.

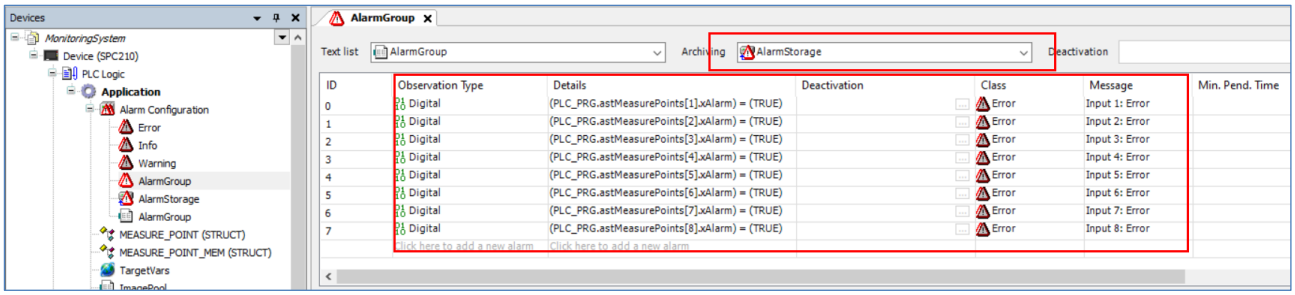


Fig. 6.5.28 AlarmGroup settings

The last visible column in the table (Fig. 6.5.28) is ***Min. Pend. Time*** – here you can set the alarm activation delay, which we implemented in the program code using a timer. In principle, both options are acceptable; the advantage of organizing the delay in the code is that the delay value can be changed if necessary (for example, via visualization). In the ***Min. Pend. Time*** column, the delay values are specified as constants, and you cannot change them while the application is running.

In ***AlarmStorage*** object, set a limit on the size of the alarm file – by number of records or by file size. If the limit is exceeded, the oldest records in the file will be overwritten by new ones (the file is saved in cyclic buffer mode). If you do not set a limit (leave the default option ***No limit***), sooner or later the file will take up all the controller's memory, which will prevent proper operation.

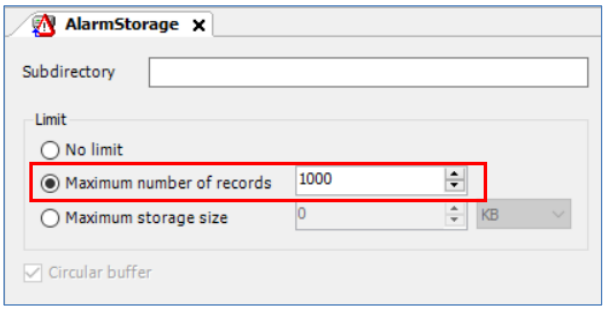


Fig. 6.5.29 AlarmStorage settings

The alarm configuration setup is now complete – you can now proceed to creating the alarm table.

6.5.5 visuAlarmTable screen configuration

Add the ***Alarm Table*** element from the ***Alarm Manager*** tab to the ***visuAlarmTable*** screen (Fig. 6.5.30).

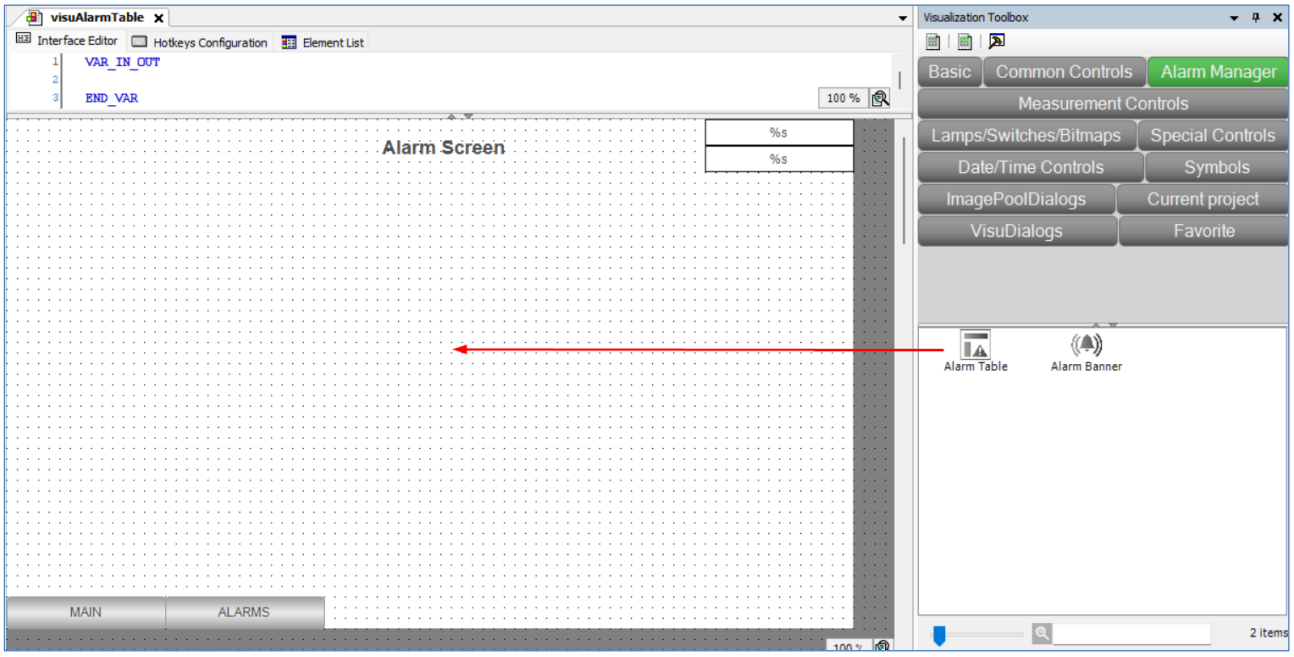


Fig. 6.5.30 Adding Alarm Table element to visuAlarmTable

By default, the table contains two columns. In its properties, open the ***Columns*** property group and click the ***Create new*** button to add a third column (Fig. 6.5.31).

- 1st column will display the alarm timestamp
- 2nd column will display the alarm message text
- 3rd column will display the alarm acknowledgment timestamp

Set the column header, width, and data type (determines what data will be displayed in the column).

> **NOTE**
> It is recommended to set the table width (***Position > Width***) equal to the sum of all column widths including the row header column (***General table configuration > Row header width***), which displays the row numbers. Otherwise, the table will display a scroll bar on the X-axis.

Now we need to add buttons for alarm acknowledgment and alarm history viewing.

Use the ***Insert elements for acknowledging alarms*** command from the table context menu (Fig. 6.5.32). There are 4 elements available:

ACK selected – button to acknowledge the selected alarm

ACK all visible – button to acknowledge all alarms in the table

History – button to switch to history mode

Freeze scrolling position – button to freeze the table in history mode. In this mode, when a new alarm appears, the view jumps to the first row of the table by default, which can be inconvenient. If the view is frozen, the view does not jump to the first row when a new alarm appears.

We will add the first three elements – we will not create a freeze button.

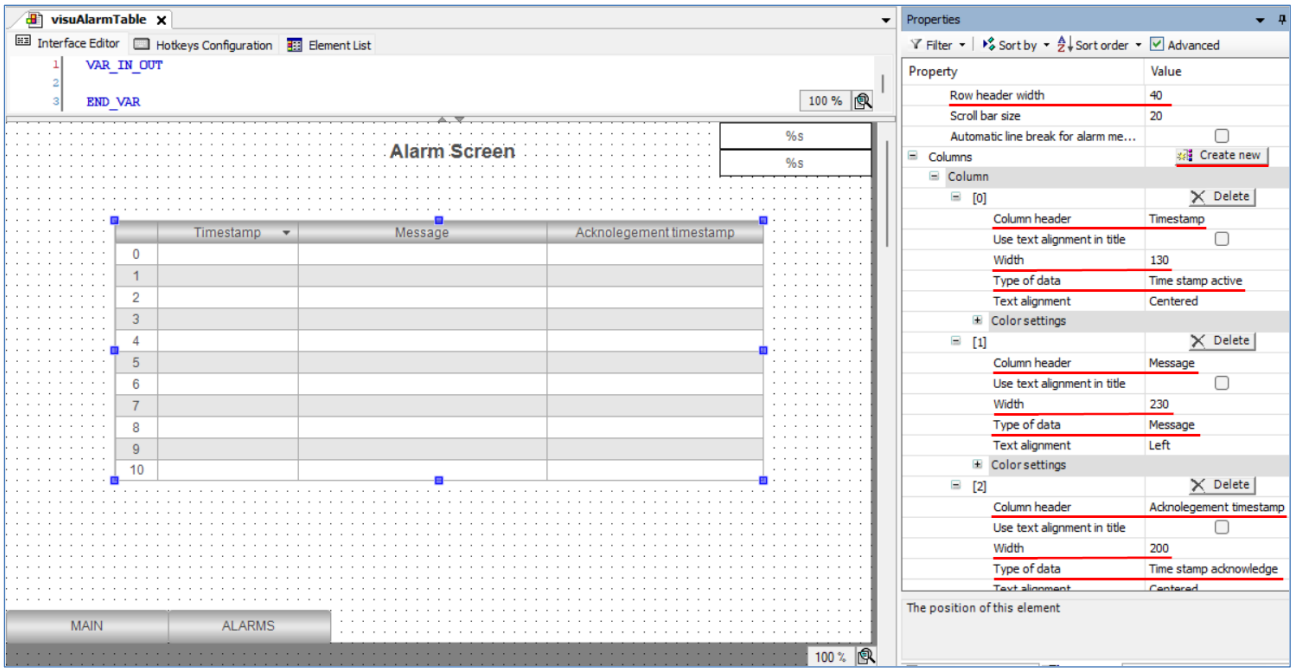


Fig. 6.5.31 Alarm table settings

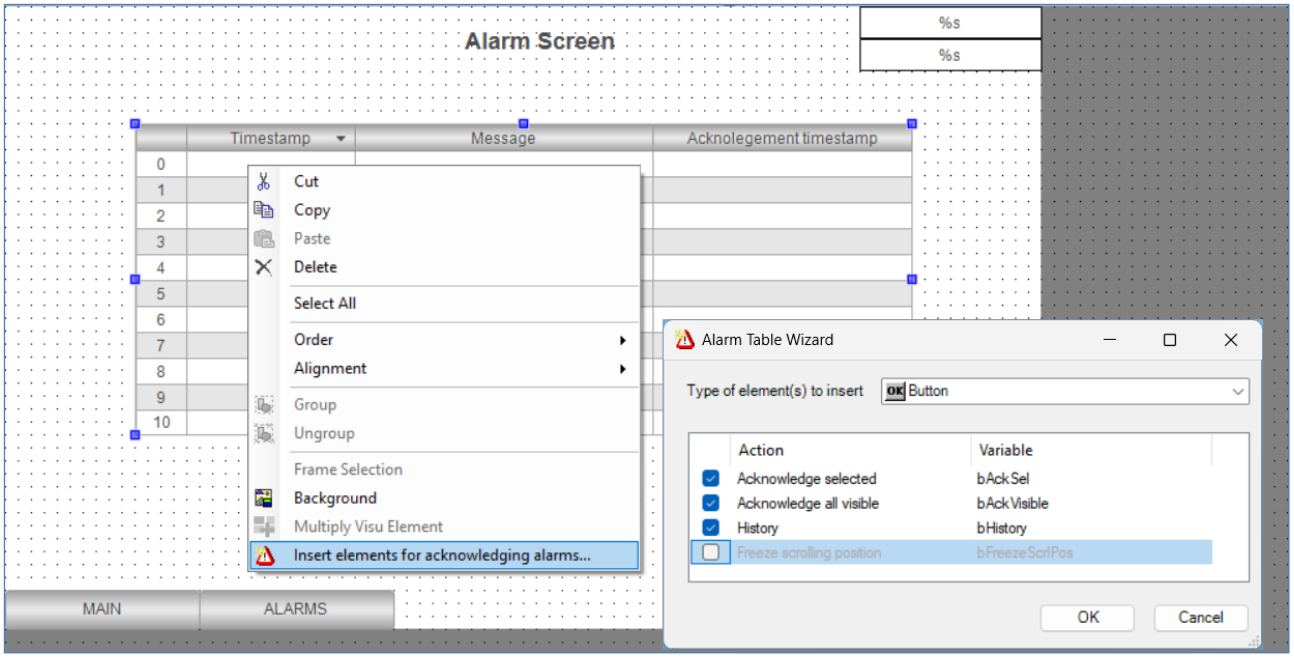


Fig. 6.5.32 Adding acknowledgment buttons

You do not need to configure the buttons. When you create them in the screen interface, local variables will be automatically declared and linked to the buttons and table properties in the ***Control variables*** property group. You can only change the text of the buttons and, if necessary, their appearance (Fig. 6.5.33).

We have already mentioned (Sect. 6.5.1) that when variables are declared, the screen becomes a frame and cannot be used as a normal visualization screen. However, this only applies to input variables (`VAR_INPUT`) and inputs/outputs (`VAR_IN_OUT`). Declaring local variables (`VAR`) does not make the screen a frame.

Local visualization variables can be used in cases where these variables are not needed outside the visualization (i.e., you do not intend to use them in the program), as in the case we discussed with acknowledgment button variables. If necessary, we could declare these variables in the program and link

them to the table properties, e.g., if we needed the ability to acknowledge alarms on command from another controller or operator panel.

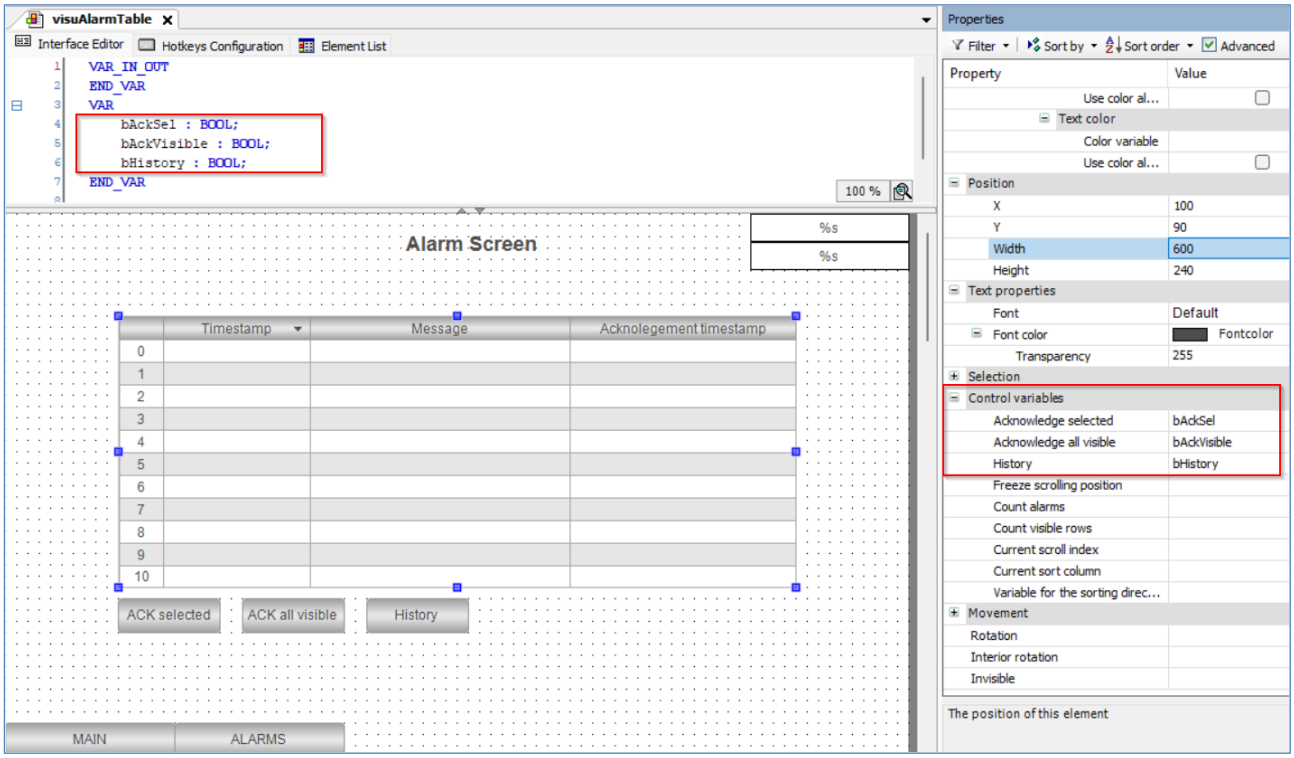


Fig. 6.5.33 Interface variables for acknowledgment buttons

The ***visuAlarmTable*** screen configuration is complete. Now we can test how our example works, in which the program and visual parts are complete.

6.6 Connecting controller to akYtec Cloud service

If you need to organize remote data collection from an akYtec controller, the akYtec Cloud service is a convenient option.

NOTE
Connection to the cloud service is only available for the akYtec controller, but not for controllers from other manufacturers or virtual controllers.

Because the variables for the cloud service are added via Symbol Configuration, this component must be added to the Application (Fig. 6.6.1).

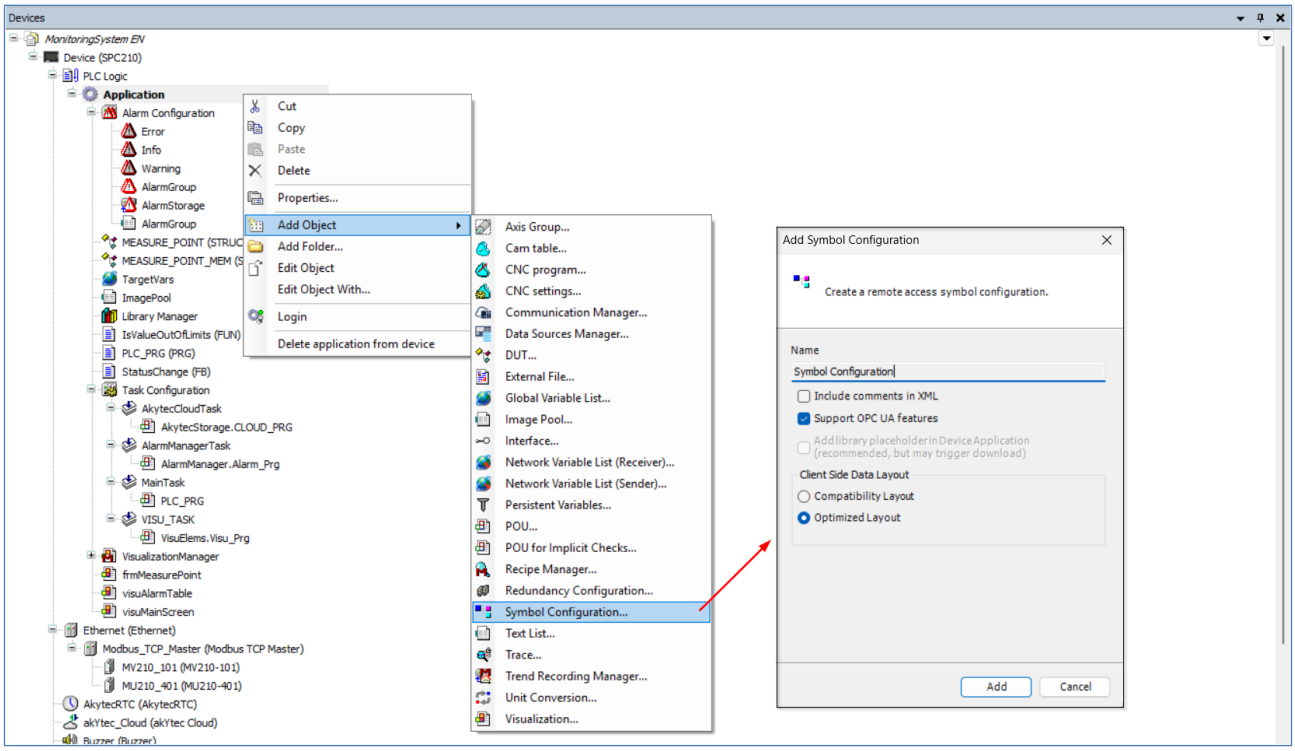


Fig. 6.6.1 Adding Symbol Configuration to the project

The cloud service does not support the import of structures, so additional variables must be declared in the program, and the necessary values are assigned to them (Example 6.6.1).

Example 6.6.1

```
PROGRAM PLC_PRG
VAR
	astMeasurePoints: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT;
	i: INT;
	xInit: BOOL;
	iCurrentFrameIndex: INT := 1;
	xEnableArchiver: BOOL := TRUE;
		// Measuring point 1
	rValuePoint1_akYtecCloud: REAL;
		// Measuring point 2
	rValuePoint2_akYtecCloud: REAL;
		// Measuring point 3
	rValuePoint3_akYtecCloud: REAL;
		// Measuring point 4
	rValuePoint4_akYtecCloud: REAL;
		// Measuring point 5
	rValuePoint5_akYtecCloud: REAL;
		// Measuring point 6
	rValuePoint6_akYtecCloud: REAL;
		// Measuring point 7
	rValuePoint7_akYtecCloud: REAL;
		// Measuring point 8
	rValuePoint8_akYtecCloud: REAL;
END_VAR
VAR_RETAIN
	astMeasurePointsMemData: ARRAY [1..c_iMeasurePointsCount] OF MEASURE_POINT_MEM;
END_VAR
VAR CONSTANT
	c_iMeasurePointsCount: INT := 8;
	c_tAlarmDelay: TIME := T#5S;
```

```
END_VAR

// code area

IF NOT(xInit) THEN
	astMeasurePoints[1].wsPointName := "Measuring point 1";
	astMeasurePoints[2].wsPointName := "Measuring point 2";
	astMeasurePoints[3].wsPointName := "Measuring point 3";
	astMeasurePoints[4].wsPointName := "Measuring point 4";
	astMeasurePoints[5].wsPointName := "Measuring point 5";
	astMeasurePoints[6].wsPointName := "Measuring point 6";
	astMeasurePoints[7].wsPointName := "Measuring point 7";
	astMeasurePoints[8].wsPointName := "Measuring point 8";
	xInit := TRUE;
END_IF

FOR i := 1 TO c_iMeasurePointsCount DO
	astMeasurePoints[i].xOutOfLimits := IsValueOutOfLimits(
		astMeasurePoints[i].rValue,
		astMeasurePointsMemData[i].rLowAlarmValue,
		astMeasurePointsMemData[i].rHighAlarmValue);
	astMeasurePoints[i].fbAlarmDelay(IN := astMeasurePoints[i].xOutOfLimits OR
		astMeasurePoints[i].eStatus <>
Mx210Assistant.ANALOG_SENSOR_ERRORS.NO_ERROR,
		PT := c_tAlarmDelay, Q => astMeasurePoints[i].xAlarm);
	astMeasurePointsMemData[i].fbAlarmCounter(CU := astMeasurePoints[i].xAlarm);

	astMeasurePoints[i].fbStatusChange(eStatus := astMeasurePoints[i].eStatus);
	IF astMeasurePoints[i].fbStatusChange.Q THEN
		astMeasurePoints[i].wsStatus :=
		Mx210Assistant.ANALOG_SENSOR_ERROR_TO_WSTRING(astMeasurePoints[i].eStatus);
	END_IF
END_FOR

rValuePoint1_akYtecCloud := astMeasurePoints[1].rValue;
rValuePoint2_akYtecCloud := astMeasurePoints[2].rValue;
rValuePoint3_akYtecCloud := astMeasurePoints[3].rValue;
rValuePoint4_akYtecCloud := astMeasurePoints[4].rValue;
rValuePoint5_akYtecCloud := astMeasurePoints[5].rValue;
rValuePoint6_akYtecCloud := astMeasurePoints[6].rValue;
rValuePoint7_akYtecCloud := astMeasurePoints[7].rValue;
rValuePoint8_akYtecCloud := astMeasurePoints[8].rValue;
```

To display the added variables in the Symbol Configuration, click the ***Build*** button (Fig. 6.6.2).

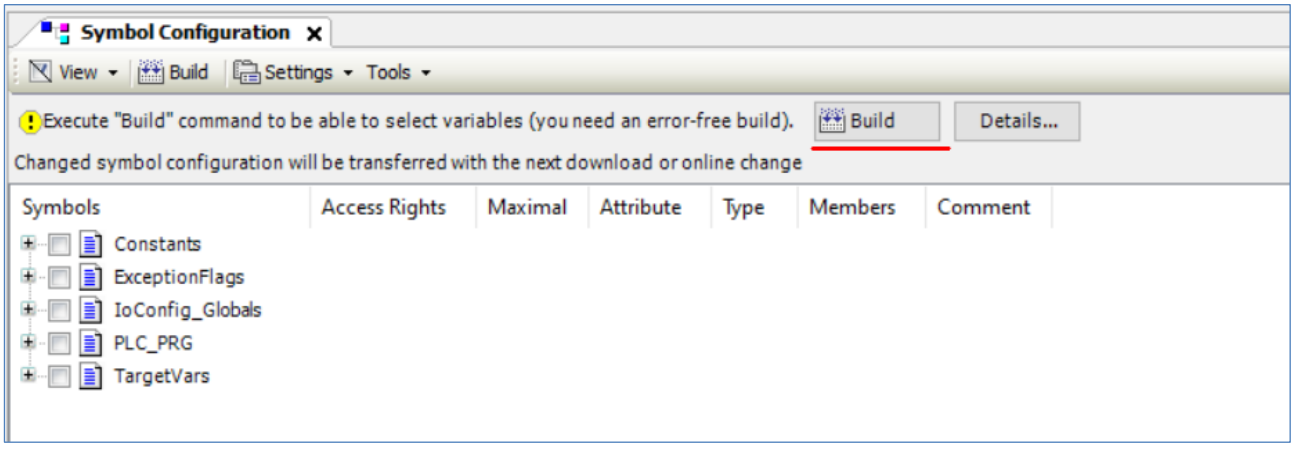


Fig. 6.6.2 Updating the list of variables

A few changes need to be made in the Symbol Configuration.

1. Unmark the variable `astMeasurePoints`. The cloud service does not support arrays or structures, so the presence of variables of these types in the Symbol Configuration will prevent the connection from being established.
2. Mark the `REAL`-type variables we created for processing the measuring point values.

Click the ***Build*** button once more to update the variables (Fig. 6.6.3).

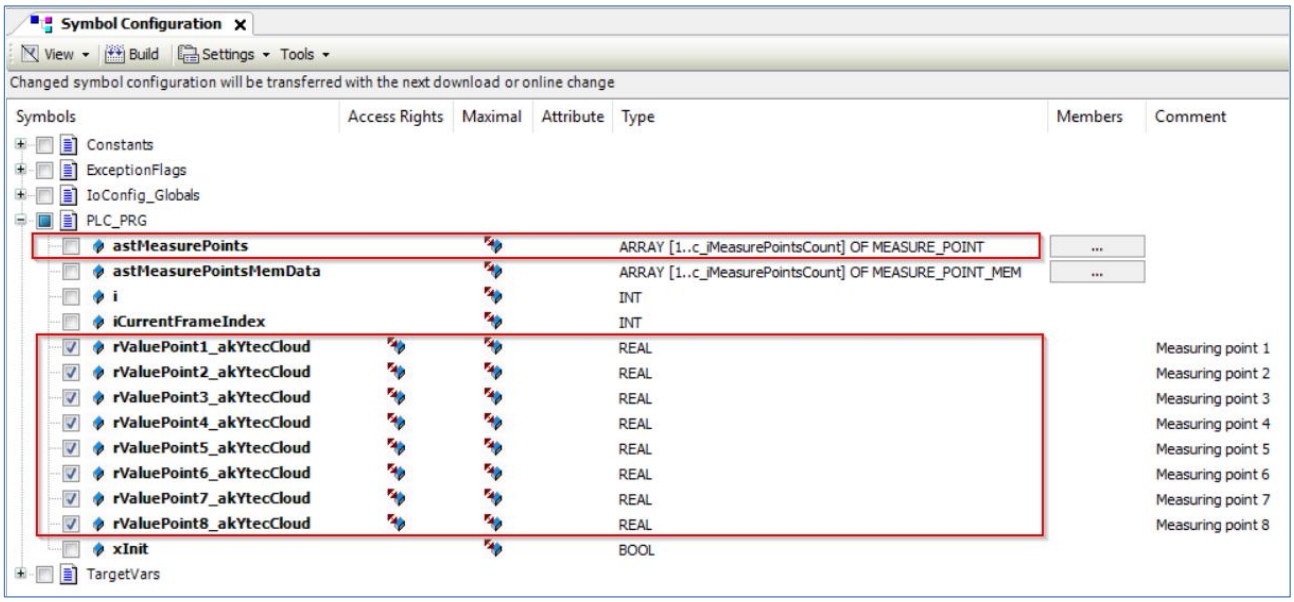


Fig. 6.6.3 Select variables to be transferred to the cloud service

Click the ***Settings*** button and select the ***Configure comments and attributes*** command. In the open window, check all the boxes related to comments and attributes so that variable comments are used as parameter names in the cloud service.

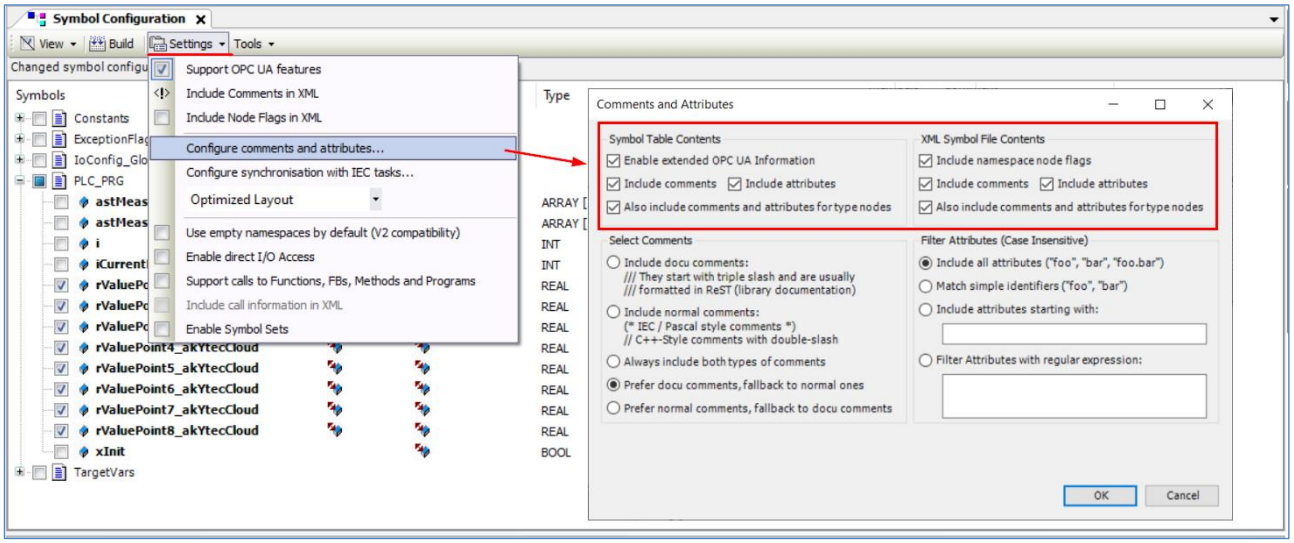


Fig. 6.6.4 Configure comments and attributes

The connection to the cloud service is configured in the ***akYtec Cloud*** system node. The parameter values can be left at their defaults, but please note the ***Password*** parameter – this is the password to enter when adding the controller to the cloud service. In this example, we will leave the default password ***123456*** (Fig. 6.6.5).

The ***I/O Mapping*** tab contains diagnostic channels for communication with the cloud service. If the ***Enable AkYtecCloud service*** channel is set to ***TRUE*** (default), the controller will attempt to establish a connection with the cloud service.

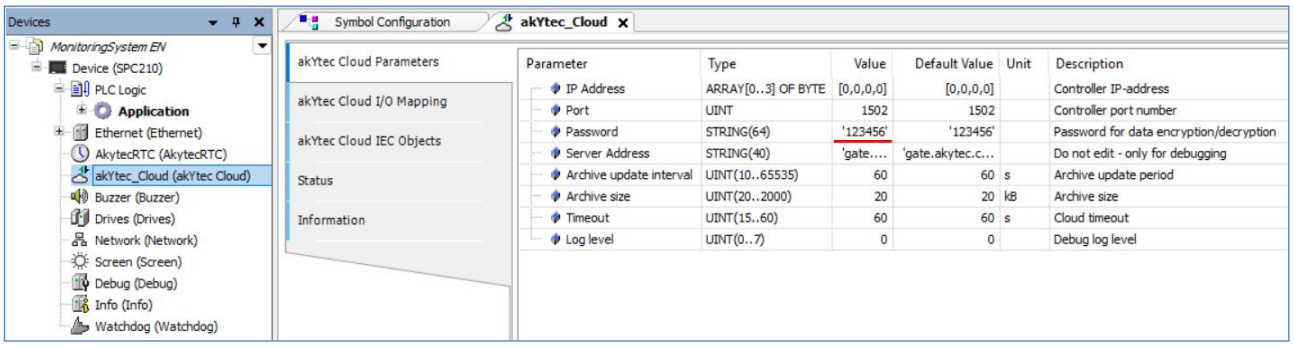


Fig. 6.6.5 akYtec Cloud component configuration parameters

Download the application to the controller and run it using the ***Online > Login*** and ***Debug > Start*** commands in the menu or the and items on the toolbar respectively.

Make sure the controller is connected to a network with internet access. This is a necessary condition for connection to the cloud service. The easiest way to check this is to go to the controller's web configurator and, on the ***Network > Diagnostics*** tab, send a ping request to the server ***gate.akytec.com***.

If the connection is established, you will see the responses. If there are no responses, ensure the network to which the controller is connected has internet access and that the controller itself has the correct settings for that network, specifically, the gateway address (Fig. 6.6.6).

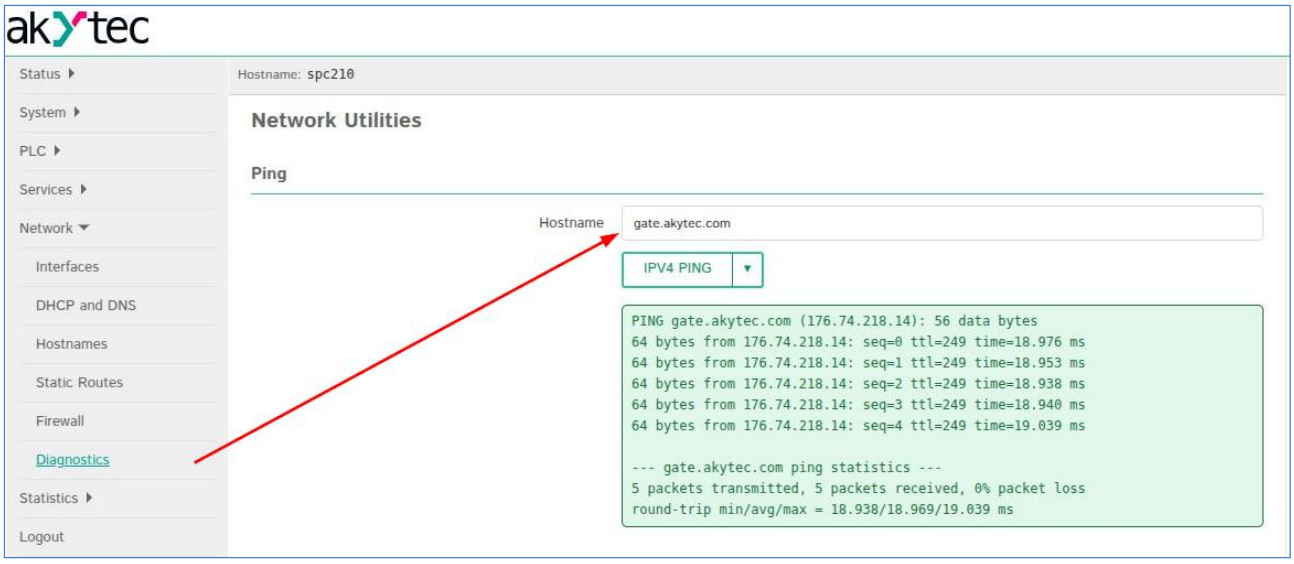


Fig. 6.6.6 Ping test to the cloud service via the controller's web configurator

Enter the address https://cloud.akytec.de in the address bar of your web browser, sign up if you don't have an account yet, and log in. Switch to the ***Administration*** section, and on the ***Devices*** tab, click the ***Add Device*** button (Fig. 6.6.7).

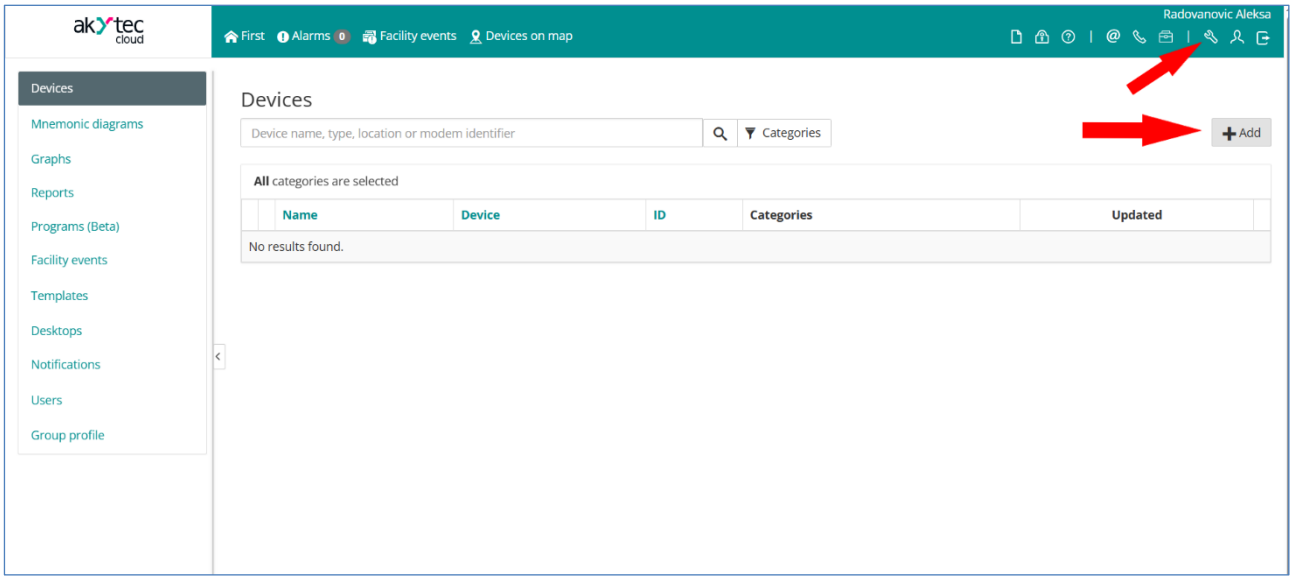


Fig. 6.6.7 Adding a device to the cloud service

In the device addition window, specify:

- ***Device type*** – Programmable controller (with autodetection)
- ***ID*** – the controller's serial number. You can find this in the web configurator (Fig. 6.6.9) or on the device housing.
- ***Device name***
- ***Time zone*** where the device is located

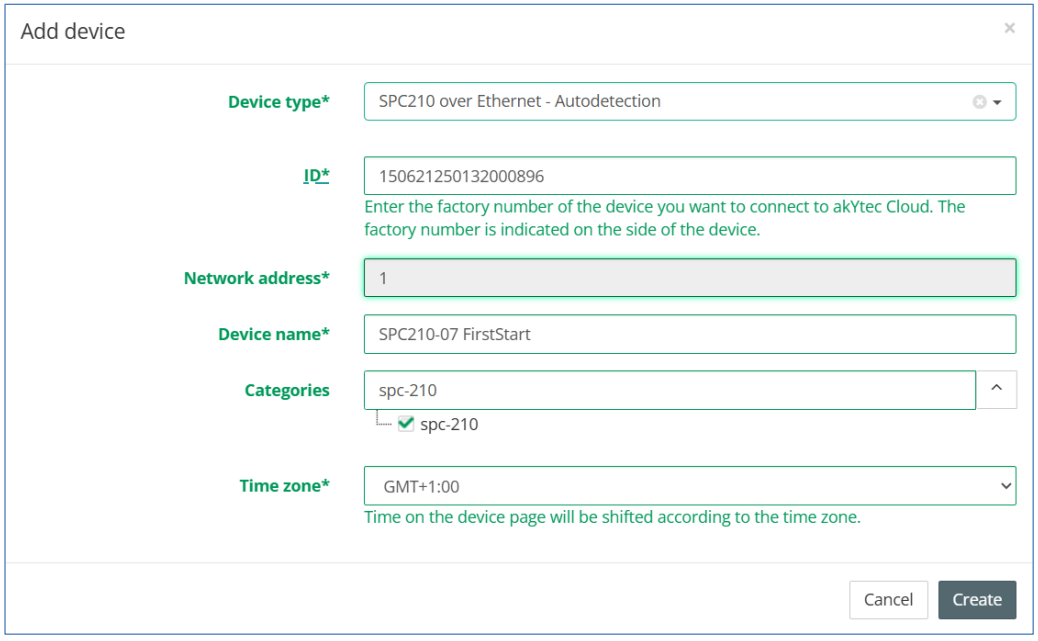


Fig. 6.6.8 New device settings

Finally, click the ***Create*** button. In the open window (Fig. 6.6.10), enter the password specified in the ***akYtec Cloud*** node of the project tree (Fig. 6.6.5) and click the ***Save*** button.

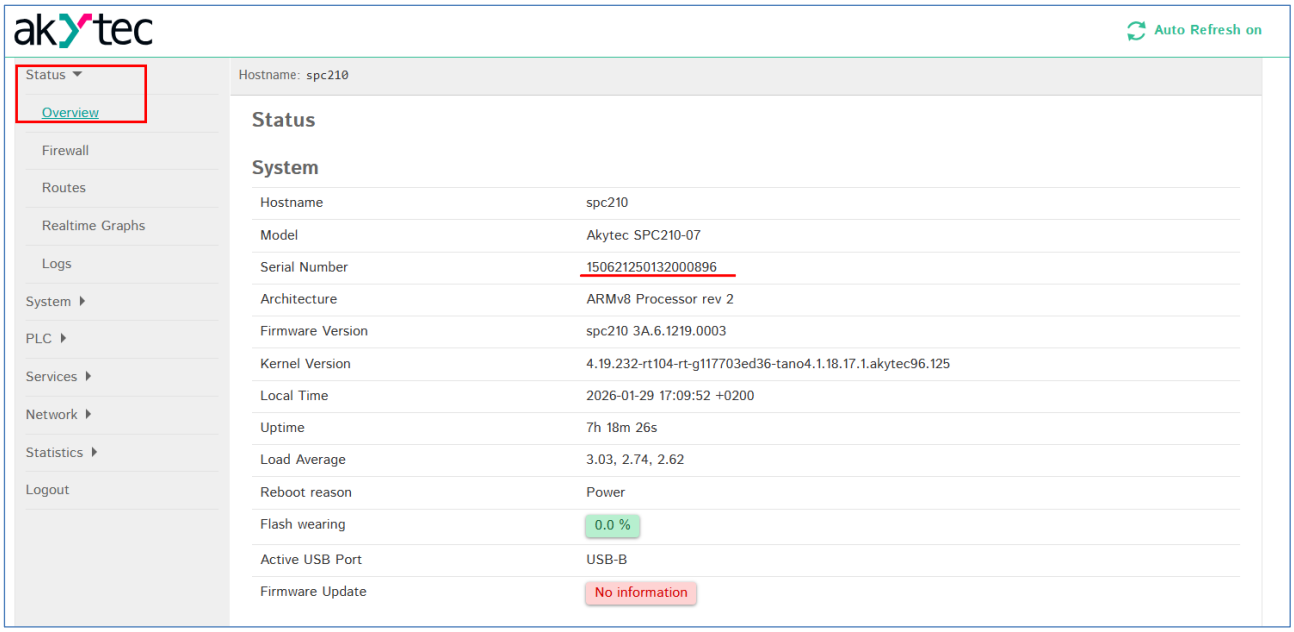


Fig. 6.6.9 ID – serial number of the device in web configurator

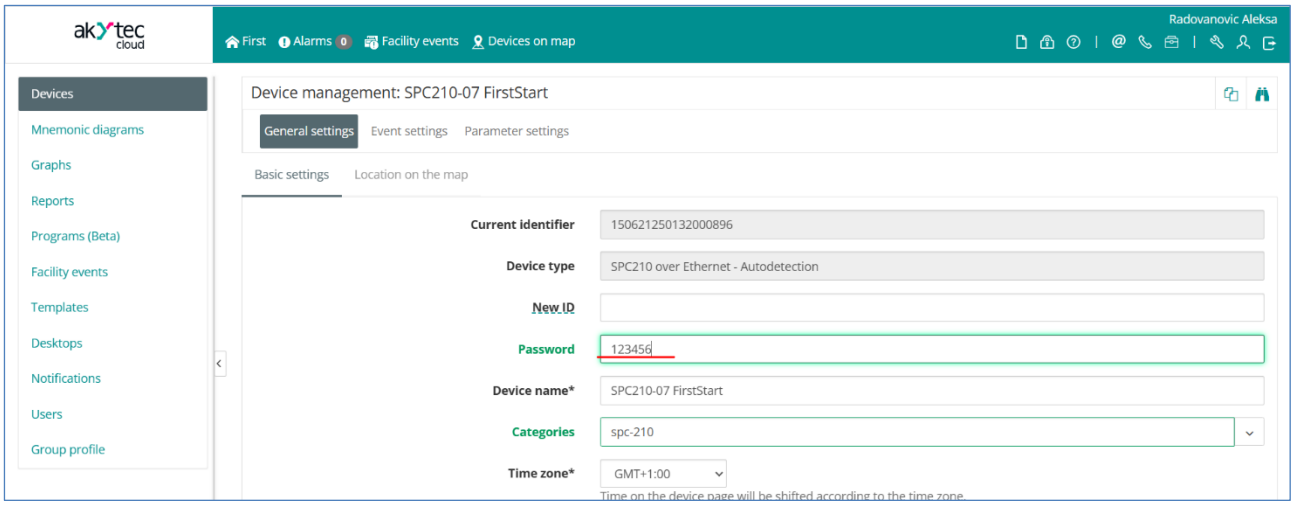


Fig. 6.6.10 Password from the akYtec Cloud node

Click the ***View Device*** button (Fig. 6.6.11).

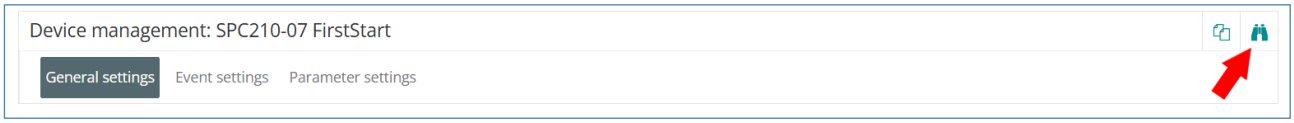


Fig. 6.6.11 View Device

Wait for the cloud service to connect to the controller and for the controller configuration to be uploaded. This may take from 10 seconds to several minutes, depending on the number of configuration variables and the communication channel. If the controller variables do not appear on the page after the ***last updated*** status is displayed, press ***F5*** to refresh the page.

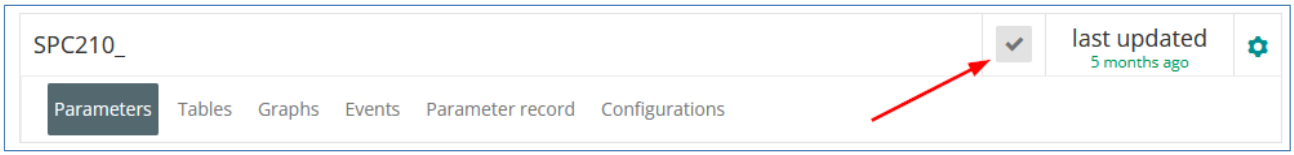


Fig. 6.6.12 Update device page

7. SPC and PLC comparison

In our example, we used the SPK210 panel controller. akYtec also produces the PLC210 series of controllers, also programmed using CODESYS V3.5. The main differences are that the PLCs do not have a display, but they do have built-in inputs and outputs.

The PLC210 project differs from the SPC210 project in the following ways:

1. The project tree contains a node named ***PLC210_XX*** (where ***XX*** indicates the PLC model), which includes the sub-nodes ***LeftSide*** and ***RightSide*** (Fig. 7.1). The node includes diagnostic channels, such as the controller temperature and the status of the ***Start/Stop*** switch located on its housing. If you set this switch to the ***STOP*** position and reboot the controller, the CODESYS application will not start. This feature is useful during the debugging phase, for example, if you have unintentionally configured a cyclic PLC reboot in the program.

   The ***LeftSide*** sub-node is responsible for the inputs and outputs located on the left PCB of the controller, while the ***RightSide*** sub-node is responsible for those on the right PCB. Like other components, these nodes include two tabs: ***Parameters*** and ***I/O Mapping***. The ***Parameters*** tab is used to configure inputs and outputs (input filtering time, switching inputs to encoder signal processing mode, output operating mode, etc.), while the ***I/O Mapping*** tab is used to assign variables to input/output bitmask channels, input counters, etc.

2. The project tree does not contain ***Screen*** and ***Network*** nodes. There is no ***Screen*** node because the PLC210 does not have a display. The ***Network*** node is absent because the PLC210 has several network interfaces that can be combined into various configurations (i.e., bridge), so creating a configuration interface for them in CODESYS would be an extremely difficult task. Instead, the PLC's network interfaces are configured in the web configurator.

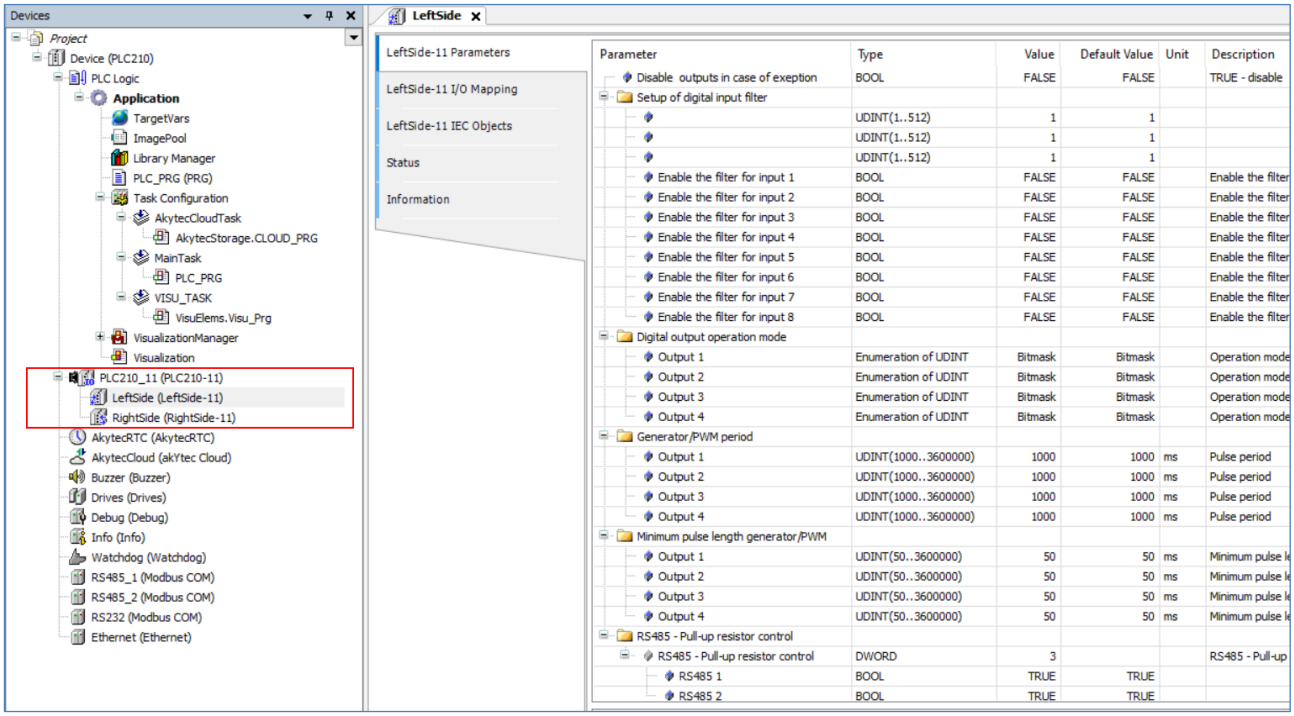


Fig. 7.1 PLC210 project tree

3. The Target Visualization subcomponent cannot be added to the Visualization Manager component, again because the PLC210 does not have a display.

There are no other differences when developing projects for SPC210 systems and PLC210.